xgboost

Release 3.4.0-dev

xgboost developers

Jun 18, 2026

CONTENTS

1 Contents **3**
1.1 Installation Guide 3
1.2 Building From Source 9
1.3 Get Started with XGBoost 16
1.4 XGBoost Tutorials 18
1.5 Frequently Asked Questions 132
1.6 XGBoost GPU Support 134
1.7 XGBoost Parameters 136
1.8 Prediction 149
1.9 Tree Methods 151
1.10 XGBoost Python Package 153
1.11 XGBoost R Package 395
1.12 XGBoost JVM Package 412
1.13 XGBoost.jl 433
1.14 XGBoost C Package 434
1.15 XGBoost C++ API 471
1.16 Security disclosure 471
1.17 Contribute to XGBoost 472
1.18 Release Notes 501

Python Module Index **523**

Index **525**

XGBoost is an optimized distributed gradient boosting library designed to be highly **efficient**, **flexible** and **portable**. It implements machine learning algorithms under the Gradient Boosting framework. XGBoost provides a parallel tree boosting (also known as GBDT, GBM) that solve many data science problems in a fast and accurate way. The same code runs on major distributed environment (Hadoop, SGE, MPI) and can solve problems beyond billions of examples.

CHAPTER

ONE

CONTENTS

1.1 Installation Guide

XGBoost provides binary packages for some language bindings. The binary packages support the GPU algorithm (`device=cuda:0`) on machines with NVIDIA GPUs. Please note that **training with multiple GPUs is only supported for Linux platform**. See *XGBoost GPU Support*. Also we have both stable releases and nightly builds, see below for how to install them. For building from source, visit *this page*.

Contents

- *Installation Guide*
  - *Stable Release*
    - *Python*
      - *Minimal installation (CPU-only)*
      - *Conda*
    - *R*
    - *JVM*
  - *Nightly Build*
    - *Python*
    - *R*
    - *JVM*

1.1.1 Stable Release

Python

Pre-built binary wheels are uploaded to PyPI (Python Package Index) for each release. Supported platforms are Linux (x86_64, aarch64), Windows (x86_64) and MacOS (x86_64, Apple Silicon).

```
# Pip 21.3+ is required
pip install xgboost
```

You might need to run the command with `--user` flag or use `virtualenv` if you run into permission errors.

Note

Parts of the Python package now require glibc 2.28+

Starting from 2.1.0, XGBoost Python package will be distributed in two variants:

- `manylinux_2_28`: for recent Linux distros with glibc 2.28 or newer. This variant comes with all features enabled.
- `manylinux2014`: for old Linux distros with glibc older than 2.28. This variant does not support GPU algorithms or federated learning.

The `pip` package manager will automatically choose the correct variant depending on your system.

Starting from **May 31, 2025**, we will stop distributing the `manylinux2014` variant and exclusively distribute the `manylinux_2_28` variant. We made this decision so that our CI/CD pipeline won't have depend on software components that reached end-of-life (such as CentOS 7). We strongly encourage everyone to migrate to recent Linux distros in order to use future versions of XGBoost.

Note. If you want to use GPU algorithms or federated learning on an older Linux distro, you have two alternatives:

1. Upgrade to a recent Linux distro with glibc 2.28+. OR
2. Build XGBoost from the source.

Note

Windows users need to install Visual C++ Redistributable

XGBoost requires DLLs from Visual C++ Redistributable in order to function, so make sure to install it. Exception: If you have Visual Studio installed, you already have access to necessary libraries and thus don't need to install Visual C++ Redistributable.

Capabilities of binary wheels for each platform:

Platform	GPU	Multi-Node-Multi-GPU
Linux x86_64	✓	✓
Linux aarch64	✓	
MacOS x86_64		
MacOS Apple Silicon		
Windows	✓	

Linux aarch64 wheels now ship with CUDA support, so `pip install xgboost` on modern Jetson or Graviton machines provides the same GPU functionality as the Linux x86_64 wheel. Multi-node and multi-GPU training remain experimental on ARM64 at this time.

Minimal installation (CPU-only)

The default installation with `pip` will install the full XGBoost package, including the support for the GPU algorithms and federated learning.

You may choose to reduce the size of the installed package and save the disk space, by opting to install `xgboost-cpu` instead:

```
pip install xgboost-cpu
```

The xgboost-cpu variant will have drastically smaller disk footprint, but does not provide some features, such as the GPU algorithms and federated learning.

Conda

You may use the Conda packaging manager to install XGBoost:

```
conda install -c conda-forge py-xgboost
```

Conda should be able to detect the existence of a GPU on your machine and install the correct variant of XGBoost. If you run into issues, try indicating the variant explicitly:

```
# CPU variant
conda install -c conda-forge py-xgboost=*=cpu*
# GPU variant
conda install -c conda-forge py-xgboost=*=cuda*
```

To force the installation of the GPU variant on a machine that does not have an NVIDIA GPU, use environment variable CONDA_OVERRIDE_CUDA, as described in "Managing Virtual Packages" in the conda docs.

```
export CONDA_OVERRIDE_CUDA="12.8"
conda install -c conda-forge py-xgboost=*=cuda*
```

You can install Conda from the following link: Download the conda-forge Installer.

R

- From R Universe

```
install.packages('xgboost', repos = c('https://dmlc.r-universe.dev', 'https://cloud.r-
↪project.org'))
```

Note

Using all CPU cores (threads) on Mac OSX

If you are using Mac OSX, you should first install OpenMP library (libomp) by running

```
brew install libomp
```

and then run install.packages("xgboost"). Without OpenMP, XGBoost will only use a single CPU core, leading to suboptimal training speed.

- We also provide **experimental** pre-built binary with GPU support. With this binary, you will be able to use the GPU algorithm without building XGBoost from the source. Download the binary package from the Releases page. The file name will be of the form xgboost_r_gpu_[os]_[version].tar.gz, where [os] is either linux or win64. (We build the binaries for 64-bit Linux and Windows.) Then install XGBoost by running:

```
# Install dependencies
R -q -e "install.packages(c('data.table', 'jsonlite'))"
# Install XGBoost
R CMD INSTALL ./xgboost_r_gpu_linux.tar.gz
```

- From CRAN (outdated):

Warning

We are working on bringing the CRAN version of XGBoost up-to-date, in the meantime, please use packages from the R-universe.

```
install.packages("xgboost")
```

Note

Using all CPU cores (threads) on Mac OSX

If you are using Mac OSX, you should first install OpenMP library (`libomp`) by running

```
brew install libomp
```

and then run `install.packages("xgboost")`. Without OpenMP, XGBoost will only use a single CPU core, leading to suboptimal training speed.

JVM

- XGBoost4j-Spark

Listing 1: Maven

```
<properties>
  ...
  <!-- Specify Scala version in package name -->
  <scala.binary.version>2.12</scala.binary.version>
</properties>

<dependencies>
  ...
  <dependency>
      <groupId>ml.dmlc</groupId>
      <artifactId>xgboost4j-spark_${scala.binary.version}</artifactId>
      <version>latest_version_num</version>
  </dependency>
</dependencies>
```

Listing 2: sbt

```
libraryDependencies ++= Seq(
  "ml.dmlc" %% "xgboost4j-spark" % "latest_version_num"
)
```

- XGBoost4j-Spark-GPU

Listing 3: Maven

```
<properties>
  ...
  <!-- Specify Scala version in package name -->
  <scala.binary.version>2.12</scala.binary.version>
```

(continues on next page)

(continued from previous page)

```
</properties>

<dependencies>
  ...
  <dependency>
      <groupId>ml.dmlc</groupId>
      <artifactId>xgboost4j-spark-gpu_${scala.binary.version}</artifactId>
      <version>latest_version_num</version>
  </dependency>
</dependencies>
```

Listing 4: sbt

```
libraryDependencies ++= Seq(
  "ml.dmlc" %% "xgboost4j-spark-gpu" % "latest_version_num"
)
```

This will check out the latest stable version from the Maven Central.

For the latest release version number, please check release page.

To enable the GPU algorithm (`device='cuda'`), use artifacts `xgboost4j-spark-gpu_2.12` instead (note the `gpu` suffix).

Note

Windows not supported in the JVM package

Currently, XGBoost4J-Spark does not support Windows platform, as the distributed training algorithm is inoperational for Windows. Please use Linux or MacOS.

1.1.2 Nightly Build

Python

Nightly builds are available. You can go to this page, find the wheel with the commit ID you want and install it with pip:

```
pip install <url to the wheel>
```

The capability of Python pre-built wheel is the same as stable release.

R

Other than standard CRAN installation, we also provide *experimental* pre-built binary on with GPU support. You can go to this page, Find the commit ID you want to install and then locate the file `xgboost_r_gpu_[os]_[commit].tar.gz`, where `[os]` is either `linux` or `win64`. (We build the binaries for 64-bit Linux and Windows.) Download it and run the following commands:

```
# Install dependencies
R -q -e "install.packages(c('data.table', 'jsonlite', 'remotes'))"
# Install XGBoost
R CMD INSTALL ./xgboost_r_gpu_linux.tar.gz
```

JVM

- XGBoost4j/XGBoost4j-Spark

Listing 5: Maven

```
<repository>
  <id>XGBoost4J Snapshot Repo</id>
  <name>XGBoost4J Snapshot Repo</name>
  <url>https://s3-us-west-2.amazonaws.com/xgboost-maven-repo/snapshot/</url>
</repository>
```

Listing 6: sbt

```
resolvers += "XGBoost4J Snapshot Repo" at "https://s3-us-west-2.amazonaws.com/xgboost-
↪maven-repo/snapshot/"
```

Then add XGBoost4J-Spark as a dependency:

Listing 7: maven

```
<properties>
  ...
  <!-- Specify Scala version in package name -->
  <scala.binary.version>2.12</scala.binary.version>
</properties>

<dependencies>
  <dependency>
      <groupId>ml.dmlc</groupId>
      <artifactId>xgboost4j-spark_${scala.binary.version}</artifactId>
      <version>latest_version_num-SNAPSHOT</version>
  </dependency>
</dependencies>
```

Listing 8: sbt

```
libraryDependencies ++= Seq(
  "ml.dmlc" %% "xgboost4j-spark" % "latest_version_num-SNAPSHOT"
)
```

- XGBoost4j-Spark-GPU

Listing 9: maven

```
<properties>
  ...
  <!-- Specify Scala version in package name -->
  <scala.binary.version>2.12</scala.binary.version>
</properties>

<dependencies>
  <dependency>
      <groupId>ml.dmlc</groupId>
      <artifactId>xgboost4j-spark-gpu_${scala.binary.version}</artifactId>
```

(continues on next page)

(continued from previous page)

```
      <version>latest_version_num-SNAPSHOT</version>
  </dependency>
</dependencies>
```

Listing 10: sbt

```
libraryDependencies ++= Seq(
  "ml.dmlc" %% "xgboost4j-spark-gpu" % "latest_version_num-SNAPSHOT"
)
```

Look up the `version` field in pom.xml to get the correct version number.

The SNAPSHOT JARs are hosted by the XGBoost project. Every commit in the `master` branch will automatically trigger generation of a new SNAPSHOT JAR. You can control how often Maven should upgrade your SNAPSHOT installation by specifying `updatePolicy`. See here for details.

You can browse the file listing of the Maven repository at https://s3-us-west-2.amazonaws.com/xgboost-maven-repo/list.html.

To enable the GPU algorithm (`device='cuda'`), use artifacts `xgboost4j-gpu_2.12` and `xgboost4j-spark-gpu_2.12` instead (note the `gpu` suffix).

1.2 Building From Source

This page gives instructions on how to build and install XGBoost from the source code on various systems. If the instructions do not work for you, please feel free to ask questions at GitHub.

Note

Pre-built binary is available: now with GPU support

Consider installing XGBoost from a pre-built binary, to avoid the trouble of building XGBoost from the source. Checkout *Installation Guide*.

Contents

- *Obtaining the Source Code*
- *Building the Shared Library*
  - *Running CMake and build*
  - *Building with GPU support*
  - *Federated Learning*
- *Building Python Package from Source*
  - *Building Python Package with Default Toolchains*
- *Building R Package From Source*
  - *Installing the development version (Linux / Mac OSX)*
  - *Building R package with GPU support*

- *Building JVM Packages*
  - *Additional System-dependent Features*
- *Building the Documentation*

1.2.1 Obtaining the Source Code

To obtain the development repository of XGBoost, one needs to use `git`. XGBoost uses Git submodules to manage dependencies. So when you clone the repo, remember to specify `--recursive` option:

```
git clone --recursive https://github.com/dmlc/xgboost
```

1.2.2 Building the Shared Library

This section describes the procedure to build the shared library and CLI interface independently. For building language specific package, see corresponding sections in this document.

- On Linux and other UNIX-like systems, the target library is `libxgboost.so`
- On MacOS, the target library is `libxgboost.dylib`
- On Windows the target library is `xgboost.dll`

This shared library is used by different language bindings (with some additions depending on the binding you choose). The minimal building requirement is

- A recent C++ compiler supporting C++17. We use gcc, clang, and MSVC for daily testing. Mingw is only used for the R package and has limited features.
- CMake 3.18 or higher.

For a list of CMake options like GPU support, see `#-- Options` in CMakeLists.txt on top level of source tree. We use `ninja` for build in this document, specified via the CMake flag `-GNinja`. If you prefer other build tools like `make` or `Visual Studio 17 2022`, please change the corresponding CMake flags. Consult the CMake generator document when needed.

Running CMake and build

After obtaining the source code, one builds XGBoost by running CMake:

```
cd xgboost
cmake -B build -S . -DCMAKE_BUILD_TYPE=RelWithDebInfo -GNinja
cd build && ninja
```

The same command applies for both Unix-like systems and Windows. After running the build, one should see a shared object under the `xgboost/lib` directory.

- Building on MacOS

  On MacOS, one needs to obtain `libomp` from Homebrew first:

  ```
  brew install libomp
  ```

- Visual Studio

  The latest Visual Studio has builtin support for CMake projects. If you prefer using an IDE over the command line, you can use the `open with visual studio` option in the right-click menu under the `xgboost` source directory. Consult the VS document for more info.

Building with GPU support

XGBoost can be built with GPU support for both Linux and Windows using CMake. See *Building R package with GPU support* for special instructions for R.

An up-to-date version of the CUDA toolkit is required.

Note

Checking your compiler version

CUDA is really picky about supported compilers, a table for the compatible compilers for the latest CUDA version on Linux can be seen here.

Some distros package a compatible `gcc` version with CUDA. If you run into compiler errors with `nvcc`, try specifying the correct compiler with `-DCMAKE_CXX_COMPILER=/path/to/correct/g++ -DCMAKE_C_COMPILER=/path/to/correct/gcc`. On Arch Linux, for example, both binaries can be found under `/opt/cuda/bin/`. In addition, the `CMAKE_CUDA_HOST_COMPILER` parameter can be useful.

From the command line on Linux starting from the XGBoost directory, add the `USE_CUDA` flag:

```
cmake -B build -S . -DUSE_CUDA=ON -GNinja
cd build && ninja
```

To speed up compilation, the compute version specific to your GPU could be passed to cmake as, e.g., `-DCMAKE_CUDA_ARCHITECTURES=75`. A quick explanation and numbers for some architectures can be found in this page.

- Faster distributed GPU training with NCCL

  By default, distributed GPU training is enabled with the option `USE_NCCL=ON`. Distributed GPU training depends on NCCL2, available at this link. Since NCCL2 is only available for Linux machines, **Distributed GPU training is available only for Linux**.

  ```
  cmake -B build -S . -DUSE_CUDA=ON -DUSE_NCCL=ON -DNCCL_ROOT=/path/to/nccl2 -GNinja
  cd build && ninja
  ```

  Some additional flags are available for NCCL, `BUILD_WITH_SHARED_NCCL` enables building XGBoost with NCCL as a shared library, while `USE_DLOPEN_NCCL` enables XGBoost to load NCCL at runtime using `dlopen`.

Federated Learning

The federated learning plugin requires `grpc` and `protobuf`. To install grpc, refer to the installation guide from the gRPC website. Alternatively, one can use the `libgrpc` and the `protobuf` package from conda forge if conda is available. After obtaining the required dependencies, enable the flag: `-DPLUGIN_FEDERATED=ON` when running CMake. Please note that only Linux is supported for the federated plugin.

```
cmake -B build -S . -DPLUGIN_FEDERATED=ON -GNinja
cd build && ninja
```

1.2.3 Building Python Package from Source

The Python package is located at `python-package/`.

Building Python Package with Default Toolchains

There are several ways to build and install the package from source:

1. Build C++ core with CMake first

   You can first build C++ library using CMake as described in *Building the Shared Library*. After compilation, a shared library will appear in `lib/` directory. On Linux distributions, the shared library is `lib/libxgboost.so`. The install script `pip install .` will reuse the shared library instead of compiling it from scratch, making it quite fast to run.

```
$ cd python-package/
$ pip install .  # Will re-use lib/libxgboost.so
```

2. Install the Python package directly

   If the shared object is not present, the Python project setup script will try to run the CMake build command automatically. Navigate to the `python-package/` directory and install the Python package by running:

```
$ cd python-package/
$ pip install -v . # Builds the shared object automatically.
```

   which will compile XGBoost's native (C++) code using default CMake flags. To enable additional compilation options, pass corresponding `--config-settings`:

```
$ pip install -v . --config-settings use_cuda=True --config-settings use_
↪nccl=True
```

   Use Pip 22.1 or later to use `--config-settings` option.

   Here are the available options for `--config-settings`:

```
@dataclasses.dataclass
class BuildConfiguration:  # pylint: disable=R0902
    """Configurations use when building libxgboost"""

    # Whether to hide C++ symbols in libxgboost.so
    hide_cxx_symbols: bool = True
    # Whether to enable OpenMP
    use_openmp: bool = True
    # Whether to enable CUDA
    use_cuda: bool = False
    # Whether to enable NCCL
    use_nccl: bool = False
    # Whether to load nccl dynamically
    use_dlopen_nccl: bool = False
    # Whether to enable federated learning
    plugin_federated: bool = False
    # Whether to enable rmm support
    plugin_rmm: bool = False
    # Special option: See explanation below
    use_system_libxgboost: bool = False
```

   `use_system_libxgboost` is a special option. See Item 4 below for detailed description.

> **Note**
>
> Verbose flag recommended
>
> As `pip install .` will build C++ code, it will take a while to complete. To ensure that the build is progressing successfully, we suggest that you add the verbose flag (`-v`) when invoking `pip install`.

3. Editable installation

   To further enable rapid development and iteration, we provide an **editable installation**. In an editable installation, the installed package is simply a symbolic link to your working copy of the XGBoost source code. So every changes you make to your source directory will be immediately visible to the Python interpreter. To install XGBoost as editable installation, first build the shared library as previously described in *Running CMake and build*, then install the Python package with the `-e` flag:

```
# Build shared library libxgboost.so
cmake -B build -S . -GNinja
cd build && ninja
# Install as editable installation
cd ../python-package
pip install -e .
```

4. Reuse the `libxgboost.so` on system path.

   This option is useful for package managers that wish to separately package `libxgboost.so` and the XGBoost Python package. For example, Conda publishes `libxgboost` (for the shared library) and `py-xgboost` (for the Python package).

   To use this option, first make sure that `libxgboost.so` exists in the system library path:

```
import sys
import pathlib
libpath = pathlib.Path(sys.base_prefix).joinpath("lib", "libxgboost.so")
assert libpath.exists()
```

   Then pass `use_system_libxgboost=True` option to `pip install`:

```
cd python-package
pip install . --config-settings use_system_libxgboost=True
```

Note

See *Notes on packaging XGBoost's Python package* for instructions on packaging and distributing XGBoost as Python distributions.

1.2.4 Building R Package From Source

By default, the package installed by running `install.packages` is built from source using the package from CRAN. Here we list some other options for installing development version.

Installing the development version (Linux / Mac OSX)

Make sure you have installed git and a recent C++ compiler supporting C++11 (See above sections for requirements of building C++ core).

Due to the use of git-submodules, `remotes::install_github()` cannot be used to install the latest version of R package. Thus, one has to run git to check out the code first, see *Obtaining the Source Code* on how to initialize the git repository for XGBoost. The simplest way to install the R package after obtaining the source code is:

```
cd R-package
R CMD INSTALL .
```

Use the environment variable `MAKEFLAGS=-j$(nproc)` if you want to speedup the build. As an alternative, the package can also be loaded through `devtools::load_all()` from the same subfolder `R-package` in the repository's root, and by extension, can be installed through RStudio's build panel if one adds that folder `R-package` as an R package project in the RStudio IDE.

```
library(devtools)
devtools::load_all(path = "/path/to/xgboost/R-package")
```

On Linux, if you want to use the CMake build for greater flexibility around compile flags, the earlier snippet can be replaced by:

```
cmake -B build -S . -DR_LIB=ON -GNinja
cd build && ninja install
```

Warning

MSVC is not supported for the R package as it has difficulty handling R C headers. CMake build is not supported either.

Note in this case that `cmake` will not take configurations from your regular `Makevars` file (if you have such a file under `~/.R/Makevars`) - instead, custom configurations such as compilers to use and flags need to be set through CMake variables like `-DCMAKE_CXX_COMPILER`.

Building R package with GPU support

The procedure and requirements are similar as in *Building with GPU support*, so make sure to read it first.

On Linux, starting from the XGBoost directory type:

```
cmake -B build -S . -DUSE_CUDA=ON -DR_LIB=ON
cmake --build build --target install -j$(nproc)
```

When default target is used, an R package shared library would be built in the `build` area. The `install` target, in addition, assembles the package files with this shared library under `build/R-package` and runs `R CMD INSTALL`.

1.2.5 Building JVM Packages

Building XGBoost4J using Maven requires Maven 3 or newer, Java 7+ and CMake 3.18+ for compiling Java code as well as the Java Native Interface (JNI) bindings. In addition, a Python script is used during configuration, make sure the command `python` is available on your system path (some distros use the name `python3` instead of `python`).

Before you install XGBoost4J, you need to define environment variable `JAVA_HOME` as your JDK directory to ensure that your compiler can find `jni.h` correctly, since XGBoost4J relies on JNI to implement the interaction between the JVM and native libraries.

After your `JAVA_HOME` is defined correctly, it is as simple as run `mvn package` under jvm-packages directory to install XGBoost4J. You can also skip the tests by running `mvn -DskipTests=true package`, if you are sure about the correctness of your local setup.

To publish the artifacts to your local maven repository, run

```
mvn install
```

Or, if you would like to skip tests, run

```
mvn -DskipTests install
```

This command will publish the xgboost binaries, the compiled java classes as well as the java sources to your local repository. Then you can use XGBoost4J in your Java projects by including the following dependency in `pom.xml`:

```
<dependency>
  <groupId>ml.dmlc</groupId>
  <artifactId>xgboost4j</artifactId>
  <version>latest_source_version_num</version>
</dependency>
```

For sbt, please add the repository and dependency in build.sbt as following:

```
resolvers += "Local Maven Repository" at "file://"+Path.userHome.absolutePath+"/.m2/
↪repository"

"ml.dmlc" % "xgboost4j" % "latest_source_version_num"
```

If you want to use XGBoost4J-Spark, replace `xgboost4j` with `xgboost4j-spark`.

> **Note**
>
> XGBoost4J-Spark requires Apache Spark 2.3+
>
> XGBoost4J-Spark now requires **Apache Spark 3.4+**. Latest versions of XGBoost4J-Spark uses facilities of *org.apache.spark.ml.param.shared* extensively to provide for a tight integration with Spark MLLIB framework, and these facilities are not fully available on earlier versions of Spark.
>
> Also, make sure to install Spark directly from Apache website. **Upstream XGBoost is not guaranteed to work with third-party distributions of Spark, such as Cloudera Spark.** Consult appropriate third parties to obtain their distribution of XGBoost.

Additional System-dependent Features

- OpenMP on MacOS: See *Running CMake and build* for installing `openmp`. The flag `-mvn -Duse.openmp=OFF` can be used to disable OpenMP support.
- GPU support can be enabled by passing an additional flag to maven `mvn -Duse.cuda=ON install`. See *Building with GPU support* for more info. In addition, `-Dplugin.rmm=ON` can enable the optional RMM support.

1.2.6 Building the Documentation

XGBoost uses Sphinx for documentation. To build it locally, you need a installed XGBoost with all its dependencies along with:

- System dependencies
  - git
  - graphviz
- Python dependencies

  Checkout the `requirements.txt` file under `doc/`

Under `xgboost/doc` directory, run `make <format>` with `<format>` replaced by the format you want. For a list of supported formats, run `make help` under the same directory. This builds a partial document for Python but not other language bindings. To build the full document, see *Documentation and Examples*.

1.3 Get Started with XGBoost

This is a quick start tutorial showing snippets for you to quickly try out XGBoost on the demo dataset on a binary classification task.

1.3.1 Links to Other Helpful Resources

- See *Installation Guide* on how to install XGBoost.
- See *Text Input Format* on using text format for specifying training/testing data.
- See *Tutorials* for tips and tutorials.
- See Learning to use XGBoost by Examples for more code examples.

1.3.2 Python

```
from xgboost import XGBClassifier
# read data
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
data = load_iris()
X_train, X_test, y_train, y_test = train_test_split(data['data'], data['target'], test_
↪size=.2)
# create model instance
bst = XGBClassifier(n_estimators=2, max_depth=2, learning_rate=1, objective=
↪'binary:logistic')
# fit model
bst.fit(X_train, y_train)
# make predictions
preds = bst.predict(X_test)
```

1.3.3 R

```
# load data
data(agaricus.train, package='xgboost')
data(agaricus.test, package='xgboost')
train <- agaricus.train
test <- agaricus.test
# fit model
bst <- xgboost(x = train$data, y = factor(train$label),
               max.depth = 2, eta = 1, nrounds = 2,
               nthread = 2, objective = "binary:logistic")
# predict
pred <- predict(bst, test$data)
```

1.3.4 Julia

```
using XGBoost
# read data
train_X, train_Y = readlibsvm("demo/data/agaricus.txt.train", (6513, 126))
test_X, test_Y = readlibsvm("demo/data/agaricus.txt.test", (1611, 126))
# fit model
num_round = 2
bst = xgboost(train_X, num_round, label=train_Y, eta=1, max_depth=2)
# predict
pred = predict(bst, test_X)
```

1.3.5 Scala

```
import ml.dmlc.xgboost4j.scala.DMatrix
import ml.dmlc.xgboost4j.scala.XGBoost

object XGBoostScalaExample {
  def main(args: Array[String]) {
    // read trainining data, available at xgboost/demo/data
    val trainData =
      new DMatrix("/path/to/agaricus.txt.train")
    // define parameters
    val paramMap = List(
      "eta" -> 0.1,
      "max_depth" -> 2,
      "objective" -> "binary:logistic").toMap
    // number of iterations
    val round = 2
    // train the model
    val model = XGBoost.train(trainData, paramMap, round)
    // run prediction
    val predTrain = model.predict(trainData)
    // save model to the file.
    model.saveModel("/local/path/to/model")
  }
}
```

1.4 XGBoost Tutorials

This section contains official tutorials inside XGBoost package. See Awesome XGBoost for more resources. Also, don't miss the feature introductions in each package.

1.4.1 Introduction to Boosted Trees

XGBoost stands for "Extreme Gradient Boosting", where the term "Gradient Boosting" originates from the paper *Greedy Function Approximation: A Gradient Boosting Machine*, by Friedman.

The term **gradient boosted trees** has been around for a while, and there are a lot of materials on the topic. This tutorial will explain boosted trees in a self-contained and principled way using the elements of supervised learning. We think this explanation is cleaner, more formal, and motivates the model formulation used in XGBoost.

Elements of Supervised Learning

XGBoost is used for supervised learning problems, where we use the training data (with multiple features) x_i to predict a target variable y_i. Before we learn about trees specifically, let us start by reviewing the basic elements in supervised learning.

Model and Parameters

The **model** in supervised learning usually refers to the mathematical structure of by which the prediction y_i is made from the input x_i. A common example is a *linear model*, where the prediction is given as $\hat{y}_i = \sum_j \theta_j x_{ij}$, a linear combination of weighted input features. The prediction value can have different interpretations, depending on the task, i.e., regression or classification. For example, it can be logistic transformed to get the probability of positive class in logistic regression, and it can also be used as a ranking score when we want to rank the outputs.

The **parameters** are the undetermined part that we need to learn from data. In linear regression problems, the parameters are the coefficients θ. Usually we will use θ to denote the parameters (there are many parameters in a model, our definition here is sloppy).

Objective Function: Training Loss + Regularization

With judicious choices for y_i, we may express a variety of tasks, such as regression, classification, and ranking. The task of **training** the model amounts to finding the best parameters θ that best fit the training data x_i and labels y_i. In order to train the model, we need to define the **objective function** to measure how well the model fit the training data.

A salient characteristic of objective functions is that they consist of two parts: **training loss** and **regularization term**:

$$\text{obj}(\theta) = L(\theta) + \Omega(\theta)$$

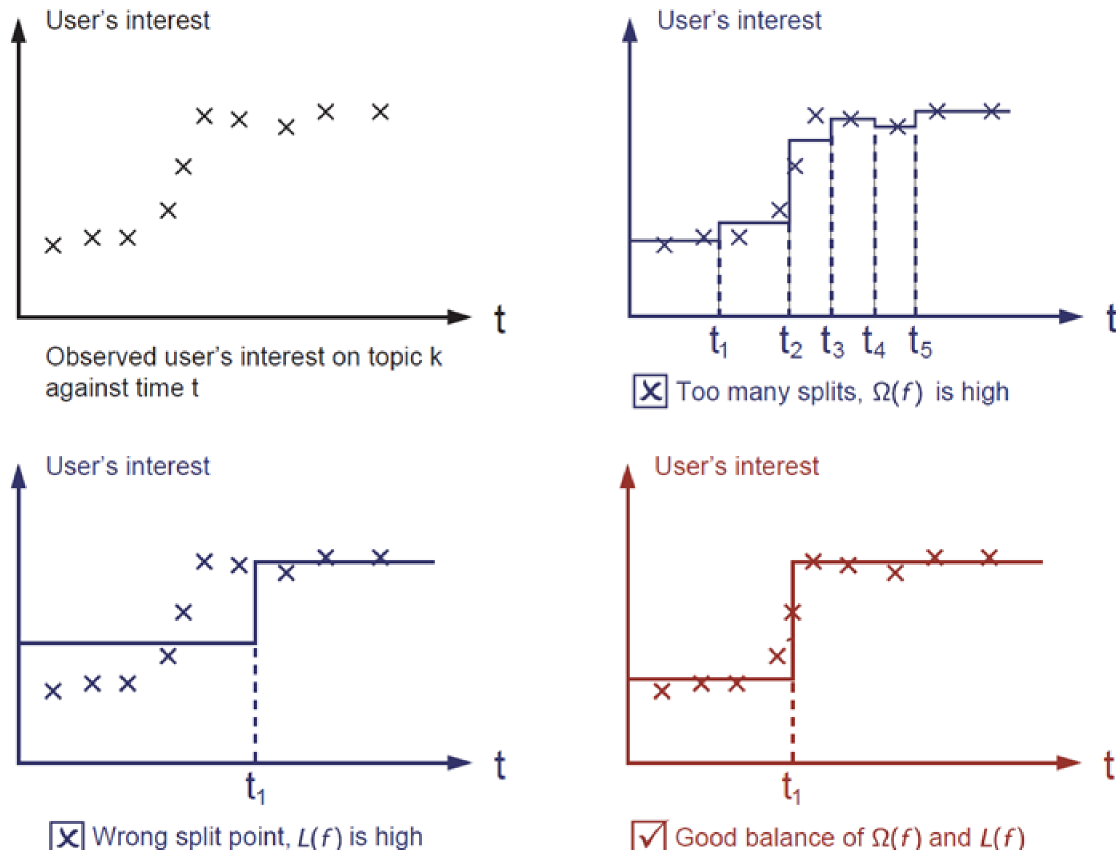
where L is the training loss function, and Ω is the regularization term. The training loss measures how *predictive* our model is with respect to the training data. A common choice of L is the *mean squared error*, which is given by

$$L(\theta) = \sum_i (y_i - \hat{y}_i)^2$$

Another commonly used loss function is logistic loss, to be used for logistic regression:

$$L(\theta) = \sum_i [y_i \ln(1 + e^{-\hat{y}_i}) + (1 - y_i) \ln(1 + e^{\hat{y}_i})]$$

The **regularization term** is what people usually forget to add. The regularization term controls the complexity of the model, which helps us to avoid overfitting. This sounds a bit abstract, so let us consider the following problem in the following picture. You are asked to *fit* visually a step function given the input data points on the upper left corner of the image. Which solution among the three do you think is the best fit?



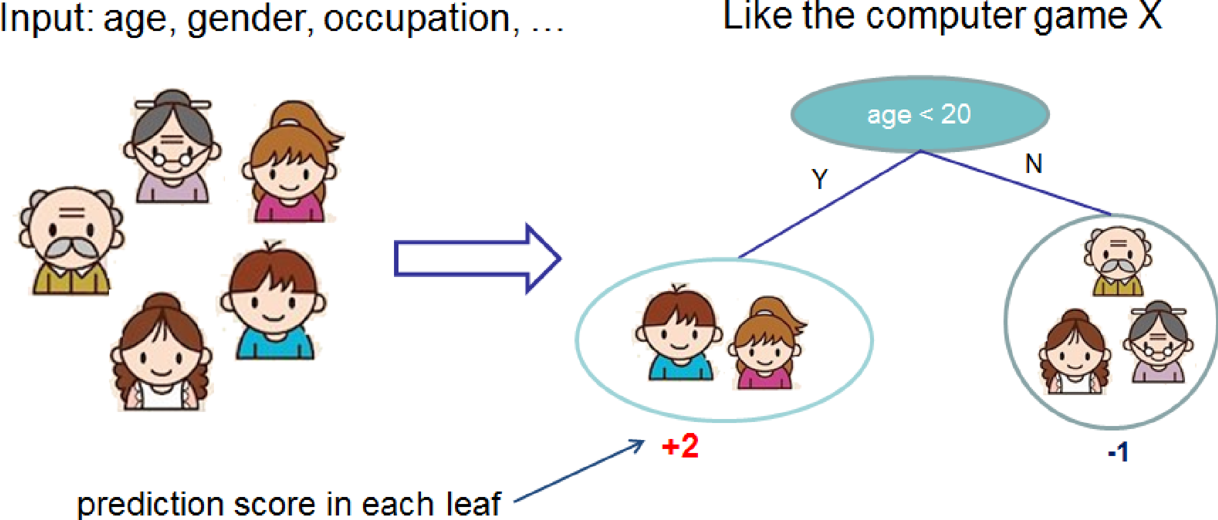
The correct answer is marked in red. Please consider if this visually seems a reasonable fit to you. The general principle is we want both a *simple* and *predictive* model. The tradeoff between the two is also referred as **bias-variance tradeoff** in machine learning.

Why introduce the general principle?

The elements introduced above form the basic elements of supervised learning, and they are natural building blocks of machine learning toolkits. For example, you should be able to describe the differences and commonalities between gradient boosted trees and random forests. Understanding the process in a formalized way also helps us to understand the objective that we are learning and the reason behind the heuristics such as pruning and smoothing.

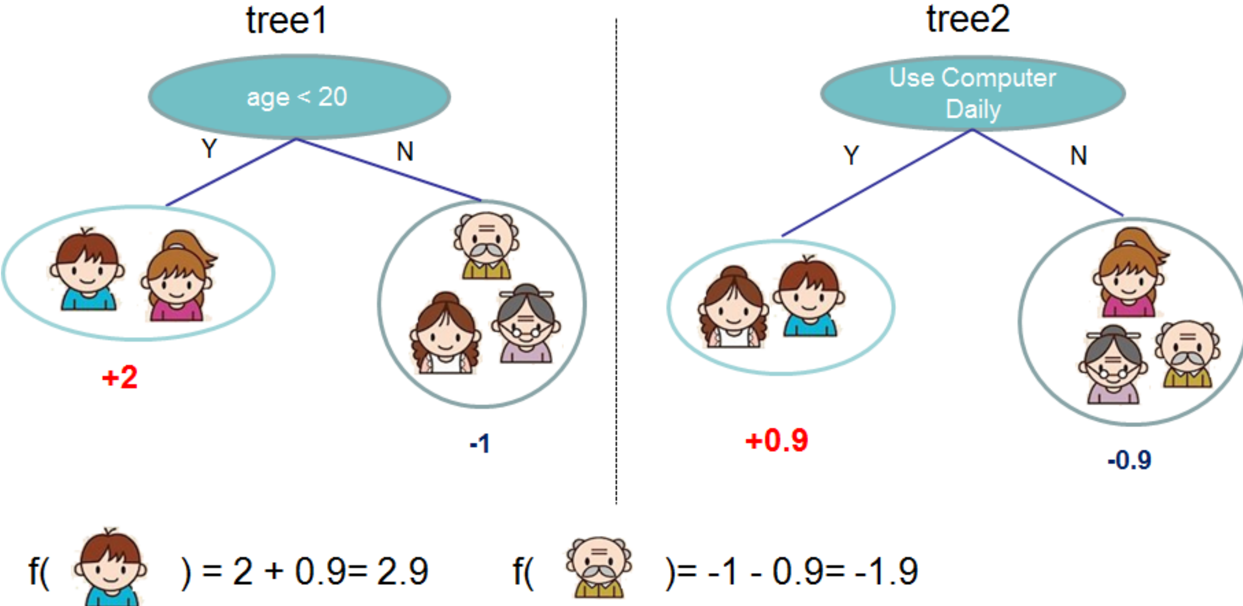
Decision Tree Ensembles

Now that we have introduced the elements of supervised learning, let us get started with real trees. To begin with, let us first learn about the model choice of XGBoost: **decision tree ensembles**. The tree ensemble model consists of a set of classification and regression trees (CART). Here's a simple example of a CART that classifies whether someone will like a hypothetical computer game X.



We classify the members of a family into different leaves, and assign them the score on the corresponding leaf. A CART is a bit different from decision trees, in which the leaf only contains decision values. In CART, a real score is associated with each of the leaves, which gives us richer interpretations that go beyond classification. This also allows for a principled, unified approach to optimization, as we will see in a later part of this tutorial.

Usually, a single tree is not strong enough to be used in practice. What is actually used is the ensemble model, which sums the prediction of multiple trees together.



Here is an example of a tree ensemble of two trees. The prediction scores of each individual tree are summed up to get the final score. If you look at the example, an important fact is that the two trees try to *complement* each other. Mathematically, we can write our model in the form

$$\hat{y}_i = \sum_{k=1}^{K} f_k(x_i), f_k \in \mathcal{F}$$

where K is the number of trees, f_k is a function in the functional space $\mathcal{F}$, and $\mathcal{F}$ is the set of all possible CARTs. The

objective function to be optimized is given by

$$\text{obj}(\theta) = \sum_i^n l(y_i, \hat{y}_i) + \sum_{k=1}^K \omega(f_k)$$

where $\omega(f_k)$ is the complexity of the tree f_k, defined in detail later.

Now here comes a trick question: what is the *model* used in random forests? Tree ensembles! So random forests and boosted trees are really the same models; the difference arises from how we train them. This means that, if you write a predictive service for tree ensembles, you only need to write one and it should work for both random forests and gradient boosted trees. (See Treelite for an actual example.) One example of why elements of supervised learning rock.

Tree Boosting

Now that we introduced the model, let us turn to training: How should we learn the trees? The answer is, as is always for all supervised learning models: *define an objective function and optimize it*!

Let the following be the objective function (remember it always needs to contain training loss and regularization):

$$\text{obj} = \sum_{i=1}^n l(y_i, \hat{y}_i^{(t)}) + \sum_{k=1}^t \omega(f_k)$$

in which t is the number of trees in our ensemble. (Each training step will add one new tree, so that at step t the ensemble contains $K = t$ trees).

Additive Training

The first question we want to ask: what are the **parameters** of trees? You can find that what we need to learn are those functions f_k, each containing the structure of the tree and the leaf scores. Learning tree structure is much harder than traditional optimization problem where you can simply take the gradient. It is intractable to learn all the trees at once. Instead, we use an additive strategy: fix what we have learned, and add one new tree at a time. We write the prediction value at step t as $\hat{y}_i^{(t)}$. Then we have

$$\begin{aligned}
\hat{y}_i^{(0)} &= 0 \\
\hat{y}_i^{(1)} &= f_1(x_i) = \hat{y}_i^{(0)} + f_1(x_i) \\
\hat{y}_i^{(2)} &= f_1(x_i) + f_2(x_i) = \hat{y}_i^{(1)} + f_2(x_i) \\
&\dots \\
\hat{y}_i^{(t)} &= \sum_{k=1}^t f_k(x_i) = \hat{y}_i^{(t-1)} + f_t(x_i)
\end{aligned}$$

It remains to ask: which tree do we want at each step? A natural thing is to add the one that optimizes our objective.

$$\begin{aligned}
\text{obj}^{(t)} &= \sum_{i=1}^n l(y_i, \hat{y}_i^{(t)}) + \sum_{k=1}^t \omega(f_k) \\
&= \sum_{i=1}^n l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \omega(f_t) + \mathrm{constant}
\end{aligned}$$

If we consider using mean squared error (MSE) as our loss function, the objective becomes

$$\begin{aligned}
\text{obj}^{(t)} &= \sum_{i=1}^n (y_i - (\hat{y}_i^{(t-1)} + f_t(x_i)))^2 + \sum_{k=1}^t \omega(f_k) \\
&= \sum_{i=1}^n [2(\hat{y}_i^{(t-1)} - y_i)f_t(x_i) + f_t(x_i)^2] + \omega(f_t) + \mathrm{constant}
\end{aligned}$$

The form of MSE is friendly, with a first order term (usually called the residual) and a quadratic term. For other losses of interest (for example, logistic loss), it is not so easy to get such a nice form. So in the general case, we take the *Taylor expansion of the loss function up to the second order*:

$$\text{obj}^{(t)} = \sum_{i=1}^n [l(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \omega(f_t) + \text{constant}$$

where the g_i and h_i are defined as

$$\begin{split}g_i &= \partial_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)})\\
h_i &= \partial_{\hat{y}_i^{(t-1)}}^2 l(y_i, \hat{y}_i^{(t-1)})\end{split}$$

After we remove all the constants, the specific objective at step t becomes

$$\sum_{i=1}^n [g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \omega(f_t)$$

This becomes our optimization goal for the new tree. One important advantage of this definition is that the value of the objective function only depends on g_i and h_i. This is how XGBoost supports custom loss functions. We can optimize every loss function, including logistic regression and pairwise ranking, using exactly the same solver that takes g_i and h_i as input!

Model Complexity

We have introduced the training step, but wait, there is one important thing, the **regularization term**! We need to define the complexity of the tree $\omega(f)$. In order to do so, let us first refine the definition of the tree $f(x)$ as

$$f_t(x) = w_{q(x)}, w \in R^T, q:R^d\rightarrow \{1,2,\cdots,T\} .$$

Here w is the vector of scores on leaves, q is a function assigning each data point to the corresponding leaf, and T is the number of leaves. In XGBoost, we define the complexity as

$$\omega(f) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^T w_j^2$$

Of course, there is more than one way to define the complexity, but this one works well in practice. The regularization is one part most tree packages treat less carefully, or simply ignore. This was because the traditional treatment of tree learning only emphasized improving impurity, while the complexity control was left to heuristics. By defining it formally, we can get a better idea of what we are learning and obtain models that perform well in the wild.

The Structure Score

Here is the magical part of the derivation. After re-formulating the tree model, we can write the objective value with the t-th tree as:

$$\begin{split}\text{obj}^{(t)} &\approx \sum_{i=1}^n [g_i w_{q(x_i)} + \frac{1}{2} h_i w_{q(x_i)}^2] + \gamma T + \frac{1}{2}\lambda \sum_{j=1}^T w_j^2\\
&= \sum^T_{j=1} [(\sum_{i\in I_j} g_i) w_j + \frac{1}{2} (\sum_{i\in I_j} h_i + \lambda) w_j^2] + \gamma T\end{split}$$

where $I_j = \{i|q(x_i)=j\}$ is the set of indices of data points assigned to the j-th leaf. Notice that in the second line we have changed the index of the summation because all the data points on the same leaf get the same score. We could

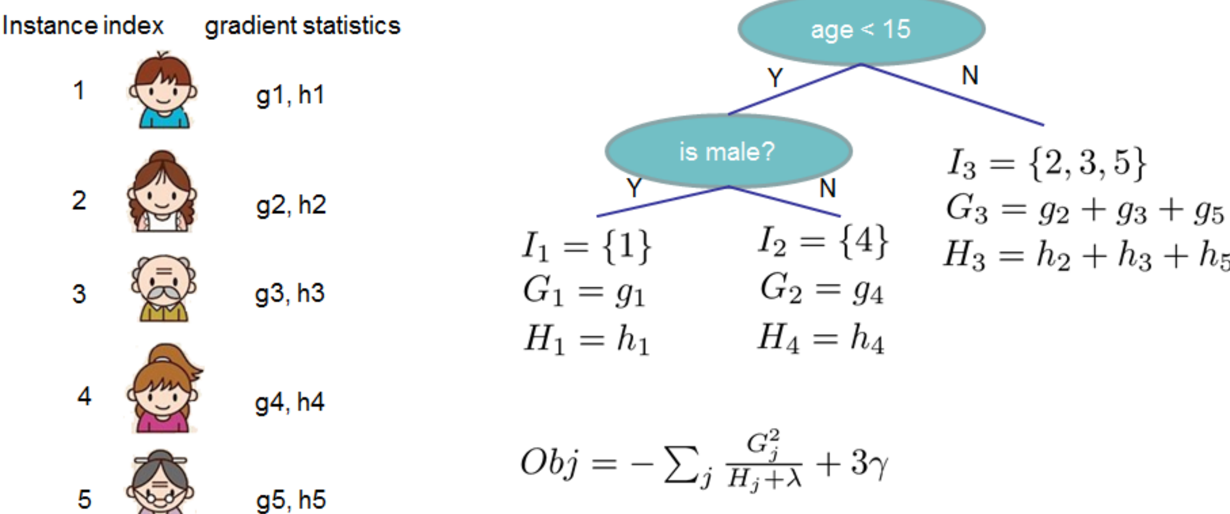
further compress the expression by defining $G_j = \sum_{i\in I_j} g_i$ and $H_j = \sum_{i\in I_j} h_i$:

$$\text{obj}^{(t)} = \sum_{j=1}^{T} [G_j w_j + \frac{1}{2}(H_j + \lambda) w_j^2] + \gamma T$$

In this equation, w_j are independent with respect to each other, the form $G_j w_j + \frac{1}{2}(H_j + \lambda) w_j^2$ is quadratic and the best w_j for a given structure $q(x)$ and the best objective reduction we can get is:

$$w_j^\ast = -\frac{G_j}{H_j+\lambda}$$
$$\text{obj}^\ast = -\frac{1}{2} \sum_{j=1}^T \frac{G_j^2}{H_j+\lambda} + \gamma T$$

The last equation measures *how good* a tree structure $q(x)$ is.



If all this sounds a bit complicated, let's take a look at the picture, and see how the scores can be calculated. Basically, for a given tree structure, we push the statistics g_i and h_i to the leaves they belong to, sum the statistics together, and use the formula to calculate how good the tree is. This score is like the impurity measure in a decision tree, except that it also takes the model complexity into account.

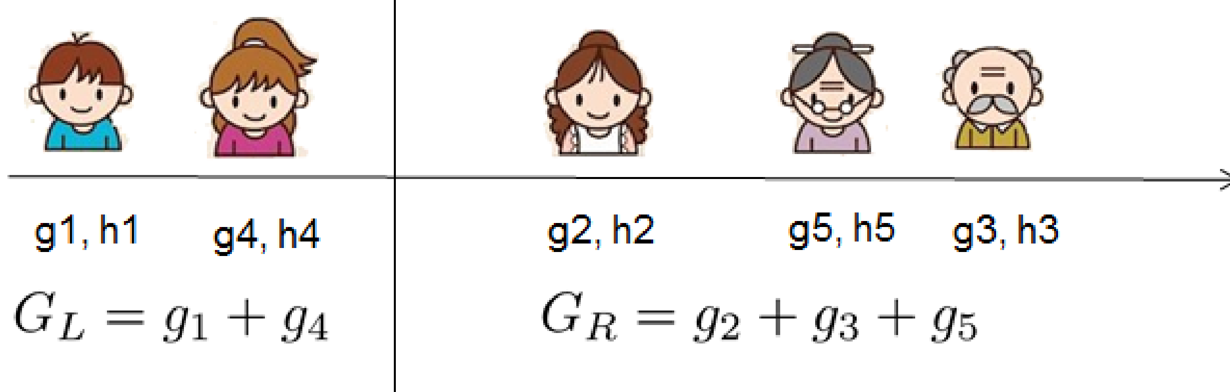
Learn the tree structure

Now that we have a way to measure how good a tree is, ideally we would enumerate all possible trees and pick the best one. In practice this is intractable, so we will try to optimize one level of the tree at a time. Specifically we try to split a leaf into two leaves, and the score it gains is

$$Gain = \frac{1}{2} \left[\frac{G_L^2}{H_L+\lambda}+\frac{G_R^2}{H_R+\lambda}-\frac{(G_L+G_R)^2}{H_L+H_R+\lambda}\right] - \gamma$$

This formula can be decomposed as 1) the score on the new left leaf 2) the score on the new right leaf 3) The score on the original leaf 4) regularization on the additional leaf. We can see an important fact here: if the gain is smaller than γ, we would do better not to add that branch. This is exactly the **pruning** techniques in tree based models! By using the principles of supervised learning, we can naturally come up with the reason these techniques work :)

For real valued data, we usually want to search for an optimal split. To efficiently do so, we place all the instances in sorted order, like the following picture.



A left to right scan is sufficient to calculate the structure score of all possible split solutions, and we can find the best split efficiently.

Note

Limitation of additive tree learning

Since it is intractable to enumerate all possible tree structures, we add one split at a time. This approach works well most of the time, but there are some edge cases that fail due to this approach. For those edge cases, training results in a degenerate model because we consider only one feature dimension at a time. See Can Gradient Boosting Learn Simple Arithmetic? for an example.

Final words on XGBoost

Now that you understand what boosted trees are, you may ask, where is the introduction for XGBoost? XGBoost is exactly a tool motivated by the formal principle introduced in this tutorial! More importantly, it is developed with both deep consideration in terms of **systems optimization** and **principles in machine learning**. The goal of this library is to push the extreme of the computation limits of machines to provide a **scalable**, **portable** and **accurate** library. Make sure you try it out, and most importantly, contribute your piece of wisdom (code, examples, tutorials) to the community!

1.4.2 Introduction to Model IO

Contents

- *A note on backward compatibility of models and memory snapshots*
- *Custom objective and metric*
- *Loading pickled files or RDS files*
- *Saving and Loading the internal parameters configuration*
- *Difference between saving model and dumping model*
- *Categories*
- *Brief History*

Since 2.1.0, the default model format for XGBoost is the UBJSON format, the option is enabled for serializing models to file, serializing models to buffer, and for memory snapshot (pickle and alike).

JSON and UBJSON have the same document structure with different representations, and we will refer them collectively as the JSON format. This tutorial aims to share some basic insights into the JSON serialisation method used in XGBoost. Without explicitly mentioned, the following sections assume you are using the one of the 2 outputs formats, which can be enabled by providing the file name with `.json` (or `.ubj` for binary JSON) as file extension when saving/loading model: `booster.save_model('model.json')`. More details below.

Before we get started, XGBoost is a gradient boosting library with focus on tree models, which means inside XGBoost, there are 2 distinct parts:

1. The model consisting of trees and
2. Hyperparameters and configurations used for building the model.

If you come from the Deep Learning community, then it should be clear to you that there are differences between the neural network structures composed of weights with fixed tensor operations, and the optimizers (like RMSprop) used to train them.

So when one calls `booster.save_model` (`xgb.save` in R), XGBoost saves the trees, some model parameters like number of input columns in trained trees, and the objective function, which combined to represent the concept of "model" in XGBoost. As for why are we saving the objective as part of model, that's because objective controls transformation of global bias (called `base_score` or the intercept in XGBoost) and task-specific information. Users can share this model with others for inference, evaluation or continue the training with a different set of hyper-parameters etc.

However, this is not the end of story. There are cases where we need to save something more than just the model itself. For example, in distributed training, XGBoost performs checkpointing operation. Or for some reasons, your favorite distributed computing framework decide to copy the model from one worker to another and continue the training in there. In such cases, the serialisation output is required to contain enough information to continue previous training without user providing any parameters again. We consider such scenario as **memory snapshot** (or memory based serialisation method) and distinguish it with normal model IO operation. Currently, memory snapshot is used in the following places:

- Python package: when the `Booster` object is pickled with the built-in `pickle` module.
- R package: when the `xgb.Booster` object is persisted with the built-in functions `saveRDS` or `save`.
- JVM packages: when the `Booster` object is serialized with the built-in functions `saveModel`.

To enable JSON format support for model IO (saving only the trees and objective), provide a filename with `.json` or `.ubj` as file extension, the latter is the extension for Universal Binary JSON

Listing 11: Python

```
bst.save_model('model_file_name.json')
```

Listing 12: R

```
xgb.save(bst, 'model_file_name.json')
```

Listing 13: Scala

```
val format = "json"  // or val format = "ubj"
model.write.option("format", format).save("model_directory_path")
```

Note

Only load models from JSON files that were produced by XGBoost. Attempting to load JSON files that were produced by an external source may lead to undefined behaviors and crashes.

When loading the model back, XGBoost recognizes the file extensions `.json` and `.ubj`, and can dispatch accordingly. If the extension is not specified, XGBoost tries to guess the right one.

A note on backward compatibility of models and memory snapshots

We guarantee backward compatibility for models but not for memory snapshots.

Models (trees and objective) use a stable representation, so that models produced in earlier versions of XGBoost are accessible in later versions of XGBoost. **If you'd like to store or archive your model for long-term storage, use** `save_model` (Python) and `xgb.save` (R).

On the other hand, memory snapshot (serialisation) captures many stuff internal to XGBoost, and its format is not stable and is subject to frequent changes. Therefore, memory snapshot is suitable for checkpointing only, where you persist the complete snapshot of the training configurations so that you can recover robustly from possible failures and resume the training process. Loading memory snapshot generated by an earlier version of XGBoost may result in errors or undefined behaviors. **If a model is persisted with** `pickle.dump` (Python) or `saveRDS` (R), **then the model may not be accessible in later versions of XGBoost.**

Custom objective and metric

XGBoost accepts user provided objective, metric, and callback functions as extensions. These functions are not saved in model file as they are language dependent features. With Python, user can pickle the model to include these functions in saved binary. One drawback is that the output from pickle is not a stable serialization format and doesn't work on different Python versions nor XGBoost versions, not to mention different language environments. Another way to workaround this limitation is to provide these functions again after the model is loaded by separating the serialization between the XGBoost built-in model and auxiliary methods. If the customized function is useful, please consider making a PR for implementing it inside XGBoost, this way we can have your functions working with different language bindings. See the next section for more about pickling.

Loading pickled files or RDS files

- From a different XGBoost version

  As noted, pickled model is neither portable nor stable, but in some cases the pickled models are valuable. One way to restore it in the future is to load it back with that specific version of Python and XGBoost, and then export the model by calling *`xgboost.Booster.save_model()`* or `xgboost.XGBModel.save_model()`.

Note

Pickle is not secure

Only load pickled files from a trusted source. The `pickle` Python module is NOT secure. And by extension, `joblib`, `cloudpickle` are also not safe when loading files from unknown sources. See https://docs.python.org/3/library/pickle.html for more information.

A similar procedure may be used to recover the model persisted in an old RDS file. In R, you are able to install an older version of XGBoost using the `remotes` package:

```
library(remotes)
remotes::install_version("xgboost", "0.90.0.1")  # Install version 0.90.0.1
```

Once the desired version is installed, you can load the RDS file with `readRDS` and recover the `xgb.Booster` object. Then call `xgb.save` to export the model using the stable representation. Now you should be able to use the model in the latest version of XGBoost.

Saving and Loading the internal parameters configuration

XGBoost's `C API`, `Python API` and `R API` support saving and loading the internal configuration directly as a JSON string. In Python package:

```
bst = xgboost.train(...)
config = bst.save_config()
print(config)
```

or in R:

```
config <- xgb.config(bst)
print(config)
```

Will print out something similar to (not actual output as it's too long for demonstration):

```
{
  "Learner": {
    "generic_parameter": {
      "device": "cuda:0",
      "gpu_page_size": "0",
      "n_jobs": "0",
      "random_state": "0",
      "seed": "0",
      "seed_per_iteration": "0"
    },
    "gradient_booster": {
      "gbtree_train_param": {
        "num_parallel_tree": "1",
        "process_type": "default",
        "tree_method": "hist",
        "updater": "grow_gpu_hist",
        "updater_seq": "grow_gpu_hist"
      },
      "name": "gbtree",
      "updater": {
        "grow_gpu_hist": {
          "gpu_hist_train_param": {
            "debug_synchronize": "0",
          },
          "train_param": {
            "alpha": "0",
            "cache_opt": "1",
            "colsample_bylevel": "1",
            "colsample_bynode": "1",
            "colsample_bytree": "1",
            "default_direction": "learn",

            ...
```

(continues on next page)

(continued from previous page)

```
            "subsample": "1"
          }
        }
      }
    },
    "learner_train_param": {
      "booster": "gbtree",
      "disable_default_eval_metric": "0",
      "objective": "reg:squarederror"
    },
    "metrics": [],
    "objective": {
      "name": "reg:squarederror",
      "reg_loss_param": {
        "scale_pos_weight": "1"
      }
    }
  },
  "version": [1, 0, 0]
}
```

You can load it back to the model generated by same version of XGBoost by:

```
bst.load_config(config)
```

This way users can study the internal representation more closely. Please note that some JSON generators make use of locale dependent floating point serialization methods, which is not supported by XGBoost.

Difference between saving model and dumping model

XGBoost has a function called `dump_model` in the Booster class, which lets you to export the model in a readable format like `text`, `json` or `dot` (graphviz). The primary use case for it is for model interpretation and visualization, and is not supposed to be loaded back to XGBoost.

Categories

Since 3.1, the categories encoding from a training dataframe is stored in the booster to provide test-time re-coding support, see *Auto-recoding (Data Consistency)* for more info about how the re-coder works. We will briefly explain the JSON format for the serialized category index.

The categories are saved in a JSON object named "cats" under the gbtree model. It contains three keys:

- feature_segments

This is a CSR-like pointer that stores the number of categories for each feature. It starts with zero and ends with the total number of categories from all features. For example:

```
feature_segments = [0, 3, 3, 5]
```

The `feature_segments` list represents a dataset with two categorical features and one numerical feature. The first feature contains three categories, the second feature is numerical and thus has no categories, and the last feature includes two categories.

- sorted_idx

This array stores the sorted indices (*argsort*) of categories across all features, segmented by the `feature_segments`. Given a feature with categories: `["b", "c", "a"]`, the sorted index is `[2, 0, 1]`.

- enc

This is an array with a length equal to the number of features, storing all the categories in the same order as the input dataframe. The storage schema depends on whether the categories are strings (XGBoost also supports numerical categories, such as integers). For string categories, we use a schema similar to the arrow format for a string array. The categories of each feature are represented by two arrays, namely `offsets` and `values`. The format is also similar to a CSR-matrix. The `values` field is a `uint8` array storing characters from all category names. Given a feature with three categories: `["bb", "c", "a"]`, the `values` field is `[98, 98, 99, 97]`. Then the `offsets` segments the `values` array similar to a CSR pointer: `[0, 2, 3, 4]`. We chose to not store the `values` as a JSON string to avoid handling special characters and string encoding. The string names are stored exactly as given by the dataframe.

As for numerical categories, the `enc` contains two keys: `type` and `values`. The `type` field is an integer ID that identifies the type of the categories, such as 64-bit integers and 32-bit floating points (note that they are all f32 inside a decision tree). The exact mapping between the type to the integer ID is internal but stable. The `values` is an array storing all categories in a feature.

Brief History

- The JSON format was introduced in 1.0, aiming to replace the now removed old binary internal format with an open format that can be easily reused
- Later in XGBoost 1.6.0, additional support for Universal Binary JSON was introduced as an optimization for more efficient model IO.
- UBJSON has been set to default in 2.1.
- The old binary format was removed in 3.1.
- The JSON schema file is no longer maintained and has been removed in 3.2. The underlying schema of the model is not changed.

1.4.3 Slicing Models

Slice tree model

When `booster` is set to `gbtree` or `dart`, XGBoost builds a tree model, which is a list of trees and can be sliced into multiple sub-models.

Python

R

```
import xgboost as xgb
from sklearn.datasets import make_classification
num_classes = 3
X, y = make_classification(n_samples=1000, n_informative=5,
                           n_classes=num_classes)
dtrain = xgb.DMatrix(data=X, label=y)
num_parallel_tree = 4
num_boost_round = 16
# total number of built trees is num_parallel_tree * num_classes * num_boost_round

# We build a boosted random forest for classification here.
booster = xgb.train({
    'num_parallel_tree': 4, 'subsample': 0.5, 'num_class': 3},
                    num_boost_round=num_boost_round, dtrain=dtrain)
```

(continues on next page)

(continued from previous page)

```
# This is the sliced model, containing [3, 7) forests
# step is also supported with some limitations like negative step is invalid.
sliced: xgb.Booster = booster[3:7]

# Access individual tree layer
trees = [_ for _ in booster]
assert len(trees) == num_boost_round
```

```
library(xgboost)
data(agaricus.train, package = "xgboost")
dm <- xgb.DMatrix(agaricus.train$data, label = agaricus.train$label)

model <- xgb.train(
  params = xgb.params(objective = "binary:logistic", max_depth = 4),
  data = dm,
  nrounds = 20
)
sliced <- model[seq(3, 7)]
##### xgb.Booster
# of features: 126
# of rounds:  5
```

The sliced model is a copy of selected trees, that means the model itself is immutable during slicing. This feature is the basis of `save_best` option in early stopping callback. See *Demo for prediction using individual trees and model slices* for a worked example on how to combine prediction with sliced trees.

Note

The returned model slice doesn't contain attributes like *best_iteration* and *best_score*.

1.4.4 Learning to Rank

Contents

- *Overview*
- *Training with the Pairwise Objective*
- *Position Bias*
- *Loss*
- *Constructing Pairs*
- *Obtaining Good Result*
- *Distributed Training*
- *Comparing Results with Version 1.7*
- *Reproducible Result*
- *References*

Overview

Often in the context of information retrieval, learning-to-rank aims to train a model that arranges a set of query results into an ordered list *[1]*. For supervised learning-to-rank, the predictors are sample documents encoded as feature matrix, and the labels are relevance degree for each sample. Relevance degree can be multi-level (graded) or binary (relevant or not). The training samples are often grouped by their query index with each query group containing multiple query results.

XGBoost implements learning to rank through a set of objective functions and performance metrics. The default objective is `rank:ndcg` based on the `LambdaMART` *[2]* algorithm, which in turn is an adaptation of the `LambdaRank` *[3]* framework to gradient boosting trees. For a history and a summary of the algorithm, see *[5]*. The implementation in XGBoost features deterministic GPU computation, distributed training, position debiasing and two different pair construction strategies.

Training with the Pairwise Objective

`LambdaMART` is a pairwise ranking model, meaning that it compares the relevance degree for every pair of samples in a query group and calculate a proxy gradient for each pair. The default objective `rank:ndcg` is using the surrogate gradient derived from the `ndcg` metric. To train a XGBoost model, we need an additional sorted array called `qid` for specifying the query group of input samples. An example input would look like this:

QID	Label	Features
1	0	x_1
1	1	x_2
1	0	x_3
2	0	x_4
2	1	x_5
2	1	x_6
2	1	x_7

Notice that the samples are sorted based on their query index in a non-decreasing order. In the above example, the first three samples belong to the first query and the next four samples belong to the second. For the sake of simplicity, we will use a synthetic binary learning-to-rank dataset in the following code snippets, with binary labels representing whether the result is relevant or not, and randomly assign the query group index to each sample. For an example that uses a real world dataset, please see *Getting started with learning to rank*.

```
from sklearn.datasets import make_classification
import numpy as np

import xgboost as xgb

# Make a synthetic ranking dataset for demonstration
seed = 1994
X, y = make_classification(random_state=seed)
rng = np.random.default_rng(seed)
n_query_groups = 3
qid = rng.integers(0, n_query_groups, size=X.shape[0])

# Sort the inputs based on query index
sorted_idx = np.argsort(qid)
X = X[sorted_idx, :]
y = y[sorted_idx]
qid = qid[sorted_idx]
```

The simplest way to train a ranking model is by using the scikit-learn estimator interface. Continuing the previous snippet, we can train a simple ranking model without tuning:

```
ranker = xgb.XGBRanker(tree_method="hist", lambdarank_num_pair_per_sample=8, objective=
↪"rank:ndcg", lambdarank_pair_method="topk")
ranker.fit(X, y, qid=qid)
```

Please note that, as of writing, there's no learning-to-rank interface in scikit-learn. As a result, the *xgboost.XGBRanker* class does not fully conform the scikit-learn estimator guideline and can not be directly used with some of its utility functions. For instances, the `auc_score` and `ndcg_score` in scikit-learn don't consider query group information nor the pairwise loss. Most of the metrics are implemented as part of XGBoost, but to use scikit-learn utilities like `sklearn.model_selection.cross_validation()`, we need to make some adjustments in order to pass the `qid` as an additional parameter for *xgboost.XGBRanker.score()*. Given a data frame `X` (either pandas or cuDF), add the column `qid` as follows:

```
import pandas as pd

# `X`, `qid`, and `y` are from the previous snippet, they are all sorted by the `sorted_
↪idx`.
df = pd.DataFrame(X, columns=[str(i) for i in range(X.shape[1])])
df["qid"] = qid

ranker.fit(df, y)  # No need to pass qid as a separate argument

from sklearn.model_selection import StratifiedGroupKFold, cross_val_score
# Works with cv in scikit-learn, along with HPO utilities like GridSearchCV
kfold = StratifiedGroupKFold(shuffle=False)
cross_val_score(ranker, df, y, cv=kfold, groups=df.qid)
```

The above snippets build a model using `LambdaMART` with the `NDCG@8` metric. The outputs of a ranker are relevance scores:

```
scores = ranker.predict(X)
sorted_idx = np.argsort(scores)[::-1]
# Sort the relevance scores from most relevant to least relevant
scores = scores[sorted_idx]
```

Position Bias

Added in version 2.0.0.

Note

The feature is considered experimental. This is a heated research area, and your input is much appreciated!

Obtaining real relevance degrees for query results is an expensive and strenuous, requiring human labelers to label all results one by one. When such labeling task is infeasible, we might want to train the learning-to-rank model on user click data instead, as it is relatively easy to collect. Another advantage of using click data directly is that it can reflect the most up-to-date user preferences *[1]*. However, user clicks are often biased, as users tend to choose results that are displayed in higher positions. User clicks are also noisy, where users might accidentally click on irrelevant documents. To ameliorate these issues, XGBoost implements the `Unbiased LambdaMART` *[4]* algorithm to debias the position-dependent click data. The feature can be enabled by the `lambdarank_unbiased` parameter; see *Parameters for learning to rank (rank:ndcg, rank:map, rank:pairwise)* for related options and *Getting started with learning to rank* for a worked example with simulated user clicks.

Loss

XGBoost implements different `LambdaMART` objectives based on different metrics. We list them here as a reference. Other than those used as objective function, XGBoost also implements metrics like `pre` (for precision) for evaluation. See *parameters* for available options and the following sections for how to choose these objectives based of the amount of effective pairs.

- NDCG

Normalized Discounted Cumulative Gain `NDCG` can be used with both binary relevance and multi-level relevance. If you are not sure about your data, this metric can be used as the default. The name for the objective is `rank:ndcg`.

- MAP

Mean average precision `MAP` is a binary measure. It can be used when the relevance label is 0 or 1. The name for the objective is `rank:map`.

- Pairwise

The *LambdaMART* algorithm scales the logistic loss with learning to rank metrics like `NDCG` in the hope of including ranking information into the loss function. The `rank:pairwise` loss is the original version of the pairwise loss, also known as the *RankNet loss* *[7]* or the *pairwise logistic loss*. Unlike the `rank:map` and the `rank:ndcg`, no scaling is applied ($|\Delta Z_{ij}| = 1$).

Whether scaling with a LTR metric is actually more effective is still up for debate; *[8]* provides a theoretical foundation for general lambda loss functions and some insights into the framework.

Constructing Pairs

There are two implemented strategies for constructing document pairs for λ-gradient calculation. The first one is the `mean` method, another one is the `topk` method. The preferred strategy can be specified by the `lambdarank_pair_method` parameter.

For the `mean` strategy, XGBoost samples `lambdarank_num_pair_per_sample` pairs for each document in a query list. For example, given a list of 3 documents and `lambdarank_num_pair_per_sample` is set to 2, XGBoost will randomly sample 6 pairs, assuming the labels for these documents are different. On the other hand, if the pair method is set to `topk`, XGBoost constructs about $k \times |query|$ number of pairs with $|query|$ pairs for each sample at the top $k = lambdarank_num_pair$ position. The number of pairs counted here is an approximation since we skip pairs that have the same label.

Obtaining Good Result

Learning to rank is a sophisticated task and an active research area. It's not trivial to train a model that generalizes well. There are multiple loss functions available in XGBoost along with a set of hyperparameters. This section contains some hints for how to choose hyperparameters as a starting point. One can further optimize the model by tuning these hyperparameters.

The first question would be how to choose an objective that matches the task at hand. If your input data has multi-level relevance degrees, then either `rank:ndcg` or `rank:pairwise` should be used. However, when the input has binary labels, we have multiple options based on the target metric. *[6]* provides some guidelines on this topic and users are encouraged to see the analysis done in their work. The choice should be based on the number of *effective pairs*, which refers to the number of pairs that can generate non-zero gradient and contribute to training. *LambdaMART* with `MRR` has the least amount of effective pairs as the λ-gradient is only non-zero when the pair contains a non-relevant document ranked higher than the top relevant document. As a result, it's not implemented in XGBoost. Since `NDCG` is a multi-level metric, it usually generate more effective pairs than `MAP`.

However, when there are sufficiently many effective pairs, it's shown in *[6]* that matching the target metric with the objective is of significance. When the target metric is `MAP` and you are using a large dataset that can provide a sufficient amount of effective pairs, `rank:map` can in theory yield higher `MAP` value than `rank:ndcg`.

The consideration of effective pairs also applies to the choice of pair method (`lambdarank_pair_method`) and the number of pairs for each sample (`lambdarank_num_pair_per_sample`). For example, the mean-`NDCG` considers more pairs than `NDCG@10`, so the former generates more effective pairs and provides more granularity than the latter. Also, using the `mean` strategy can help the model generalize with random sampling. However, one might want to focus the training on the top k documents instead of using all pairs, to better fit their real-world application.

When using the mean strategy for generating pairs, where the target metric (like `NDCG`) is computed over the whole query list, users can specify how many pairs should be generated per each document, by setting the `lambdarank_num_pair_per_sample`. XGBoost will randomly sample `lambdarank_num_pair_per_sample` pairs for each element in the query group ($|pairs| = |query| \times num_pairsample$). Often, setting it to 1 can produce reasonable results. In cases where performance is inadequate due to insufficient number of effective pairs being generated, set `lambdarank_num_pair_per_sample` to a higher value. As more document pairs are generated, more effective pairs will be generated as well.

On the other hand, if you are prioritizing the top k documents, the `lambdarank_num_pair_per_sample` should be set slightly higher than k (with a few more documents) to obtain a good training result. Lastly, XGBoost employs additional regularization for learning to rank objectives, which can be disabled by setting the `lambdarank_normalization` to `False`.

Summary If you have large amount of training data:

- Use the target-matching objective.
- Choose the `topk` strategy for generating document pairs (if it's appropriate for your application).

On the other hand, if you have comparatively small amount of training data:

- Select `NDCG` or the RankNet loss (`rank:pairwise`).
- Choose the `mean` strategy for generating document pairs, to obtain more effective pairs.

For any method chosen, you can modify `lambdarank_num_pair_per_sample` to control the amount of pairs generated.

Distributed Training

XGBoost implements distributed learning-to-rank with integration of multiple frameworks including *Dask*, *Spark*, and *PySpark*. The interface is similar to the single-node counterpart. Please refer to document of the respective XGBoost interface for details.

Warning

Position-debiasing is not yet supported for existing distributed interfaces.

XGBoost works with collective operations, which means data is scattered to multiple workers. We can divide the data partitions by query group and ensure no query group is split among workers. However, this requires a costly sort and groupby operation and might only be necessary for selected use cases. Splitting and scattering a query group to multiple workers is theoretically sound but can affect the model's accuracy. If there are only a small number of groups sitting at the boundaries of workers, the small discrepancy is not an issue, as the amount of training data is usually large when distributed training is used.

For a longer explanation, assuming the pairwise ranking method is used, we calculate the gradient based on relevance degree by constructing pairs within a query group. If a single query group is split among workers and we use worker-local data for gradient calculation, then we are simply sampling pairs from a smaller group for each worker to calculate the gradient and the evaluation metric. The comparison between each pair doesn't change because a group is split into sub-groups, what changes is the number of total and effective pairs and normalizers like *IDCG*. One can generate more pairs from a large group than it's from two smaller subgroups. As a result, the obtained gradient is still valid from a theoretical standpoint but might not be optimal. As long as each data partitions within a worker are correctly sorted

by query IDs, XGBoost can aggregate sample gradients accordingly. And both the (Py)Spark interface and the Dask interface can sort the data according to query ID, please see respected tutorials for more information.

However, it's possible that a distributed framework shuffles the data during map reduce and splits every query group into multiple workers. In that case, the performance would be disastrous. As a result, it depends on the data and the framework for whether a sorted groupby is needed.

Comparing Results with Version 1.7

The learning to rank implementation has been significantly updated in 2.0 with added hyper-parameters and training strategies. To obtain similar result as the 1.7 *xgboost.XGBRanker*, following parameter should be used:

```
params = {
    # 1.7 only supports sampling, while 2.0 and later use top-k as the default.
    # See above sections for the trade-off.
    "lambdarank_pair_method": "mean",
    # 1.7 uses the ranknet loss while later versions use the NDCG weighted loss
    "objective": "rank:pairwise",
    # 1.7 doesn't have this normalization.
    "lambdarank_score_normalization": False,
    "base_score": 0.5,
    # The default tree method has been changed from approx to hist.
    "tree_method": "approx",
    # The default for `mean` pair method is one pair each sample, which is the default in␣
↪1.7 as well.
    # You can leave it as unset.
    "lambdarank_num_pair_per_sample": 1,
}
```

The result still differs due to the change of random seed. But the overall training strategy would be the same for `rank:pairwise`.

Reproducible Result

Like any other tasks, XGBoost should generate reproducible results given the same hardware and software environments (and data partitions, if distributed interface is used). Even when the underlying environment has changed, the result should still be consistent. However, when the `lambdarank_pair_method` is set to `mean`, XGBoost uses random sampling, and results may differ depending on the platform used. The random number generator used on Windows (Microsoft Visual C++) is different from the ones used on other platforms like Linux (GCC, Clang)[1], so the output varies significantly between these platforms.

References

[1] Tie-Yan Liu. 2009. "Learning to Rank for Information Retrieval". Found. Trends Inf. Retr. 3, 3 (March 2009), 225–331.

[2] Christopher J. C. Burges, Robert Ragno, and Quoc Viet Le. 2006. "Learning to rank with nonsmooth cost functions". In Proceedings of the 19th International Conference on Neural Information Processing Systems (NIPS'06). MIT Press, Cambridge, MA, USA, 193–200.

[3] Wu, Q., Burges, C.J.C., Svore, K.M. et al. "Adapting boosting for information retrieval measures". Inf Retrieval 13, 254–270 (2010).

[4] Ziniu Hu, Yang Wang, Qu Peng, Hang Li. "Unbiased LambdaMART: An Unbiased Pairwise Learning-to-Rank Algorithm". Proceedings of the 2019 World Wide Web Conference.

[1] *minstd_rand* implementation is different on MSVC. The implementations from GCC and Thrust produce the same output.

[5] Burges, Chris J.C. "From RankNet to LambdaRank to LambdaMART: An Overview". MSR-TR-2010-82

[6] Pinar Donmez, Krysta M. Svore, and Christopher J.C. Burges. 2009. "On the local optimality of LambdaRank". In Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval (SIGIR '09). Association for Computing Machinery, New York, NY, USA, 460–467.

[7] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. 2005. "Learning to rank using gradient descent". In Proceedings of the 22nd international conference on Machine learning (ICML '05). Association for Computing Machinery, New York, NY, USA, 89–96.

[8] Xuanhui Wang and Cheng Li and Nadav Golbandi and Mike Bendersky and Marc Najork. 2018. "The LambdaLoss Framework for Ranking Metric Optimization". Proceedings of The 27th ACM International Conference on Information and Knowledge Management (CIKM '18).

1.4.5 DART

XGBoost mostly combines a huge number of regression trees with a small learning rate. In this situation, trees added early are significant and trees added late are unimportant.

Vinayak and Gilad-Bachrach proposed a new method to add dropout techniques from the deep neural net community to boosted trees, and reported better results in some situations.

This is a instruction of the dropout mode for tree models. Dropout is controlled by parameters like `rate_drop`. The legacy `dart` booster name remains available for compatibility.

Original paper

Rashmi Korlakai Vinayak, Ran Gilad-Bachrach. "DART: Dropouts meet Multiple Additive Regression Trees." [PMLR, arXiv].

Features

- Drop trees in order to solve the over-fitting.
  - Trivial trees (to correct trivial errors) may be prevented.

Because of the randomness introduced in the training, expect the following few differences:

- Training can be slower than `gbtree` because the random dropout prevents usage of the prediction buffer.
- The early stop might not be stable, due to the randomness.

How it works

- In m-th training round, suppose k trees are selected to be dropped.
- Let $D = \sum_{i \in \mathbf{K}} F_i$ be the leaf scores of dropped trees and $F_m = \eta \tilde{F}_m$ be the leaf scores of a new tree.
- The objective function is as follows:

$$\mathrm{Obj} = \sum_{j=1}^{n} L\left(y_j, \hat{y}_j^{m-1} - D_j + \tilde{F}_m\right) + \Omega\left(\tilde{F}_m\right).$$

- D and F_m are overshooting, so using scale factor

$$\hat{y}_j^m = \sum_{i \notin \mathbf{K}} F_i + a\left(\sum_{i \in \mathbf{K}} F_i + bF_m\right).$$

Parameters

Dropout uses the same tree parameters as `gbtree`, such as `eta`, `gamma`, `max_depth`, and others.

Additional parameters are noted below:

- `sample_type`: type of sampling algorithm.
  - `uniform`: (default) dropped trees are selected uniformly.
  - `weighted`: dropped trees are selected in proportion to weight.
- `normalize_type`: type of normalization algorithm.
  - `tree`: (default) New trees have the same weight of each of dropped trees.

$$
\begin{aligned}
a\left(\sum_{i\in\mathbf{K}} F_i + \frac{1}{k}F_m\right) &= a\left(\sum_{i\in\mathbf{K}} F_i + \frac{\eta}{k}\tilde{F}_m\right)\\
&\sim a\left(1+\frac{\eta}{k}\right)D\\
&= a\frac{k+\eta}{k}D = D,\\
a &= \frac{k}{k+\eta}
\end{aligned}
$$

  - `forest`: New trees have the same weight of sum of dropped trees (forest).

$$
\begin{aligned}
a\left(\sum_{i\in\mathbf{K}} F_i + F_m\right) &= a\left(\sum_{i\in\mathbf{K}} F_i + \eta\tilde{F}_m\right)\\
&\sim a\left(1+\eta\right)D\\
&= a(1+\eta)D = D,\\
a &= \frac{1}{1+\eta}.
\end{aligned}
$$

- `rate_drop`: dropout rate.
  - range: [0.0, 1.0]
- `skip_drop`: probability of skipping dropout.
  - If a dropout is skipped, new trees are added in the same manner as gbtree.
  - range: [0.0, 1.0]

Sample Script

```python
import xgboost as xgb
# read in data
dtrain = xgb.DMatrix('demo/data/agaricus.txt.train?format=libsvm')
dtest = xgb.DMatrix('demo/data/agaricus.txt.test?format=libsvm')
# specify parameters via map
param = {'max_depth': 5, 'learning_rate': 0.1,
         'objective': 'binary:logistic',
         'sample_type': 'uniform',
         'normalize_type': 'tree',
         'rate_drop': 0.1,
         'skip_drop': 0.5}
num_round = 50
```

(continues on next page)

(continued from previous page)

```
bst = xgb.train(param, dtrain, num_round)
preds = bst.predict(dtest)
```

1.4.6 Monotonic Constraints

It is often the case in a modeling problem or project that the functional form of an acceptable model is constrained in some way. This may happen due to business considerations, or because of the type of scientific question being investigated. In some cases, where there is a very strong prior belief that the true relationship has some quality, constraints can be used to improve the predictive performance of the model.

A common type of constraint in this situation is that certain features bear a **monotonic** relationship to the predicted response:

$$f(x_1, x_2, \ldots, x, \ldots, x_{n-1}, x_n) \leq f(x_1, x_2, \ldots, x', \ldots, x_{n-1}, x_n)$$

whenever $x \leq x'$ is an **increasing constraint**; or

$$f(x_1, x_2, \ldots, x, \ldots, x_{n-1}, x_n) \geq f(x_1, x_2, \ldots, x', \ldots, x_{n-1}, x_n)$$

whenever $x \leq x'$ is a **decreasing constraint**.

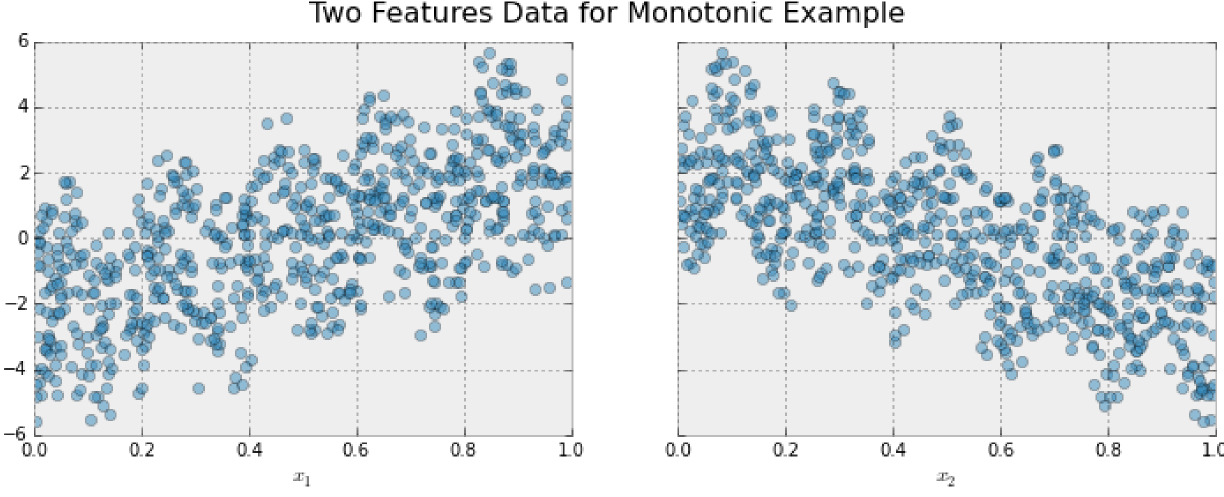
XGBoost has the ability to enforce monotonicity constraints on any features used in a boosted model.

A Simple Example

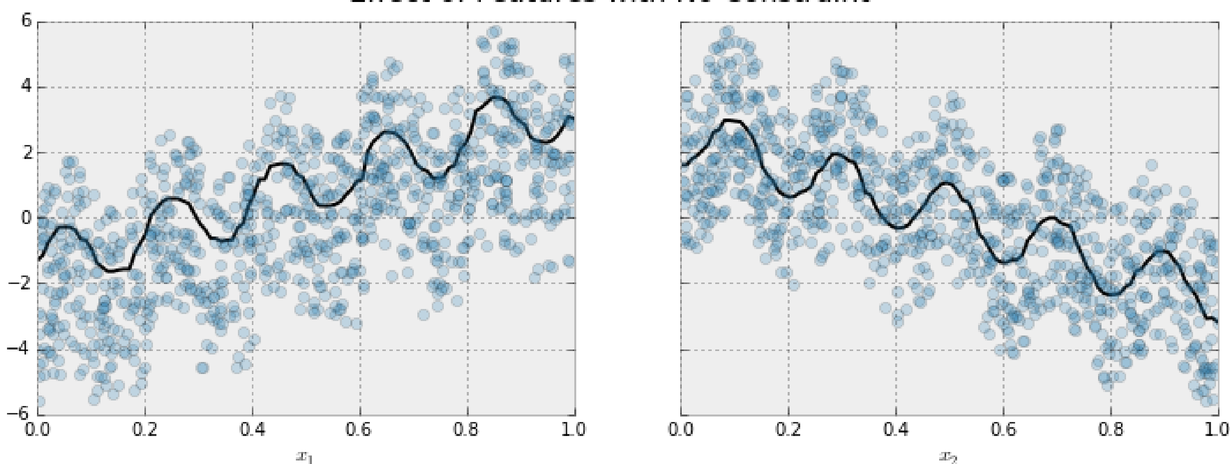
To illustrate, let's create some simulated data with two features and a response according to the following scheme

$$y = 5x_1 + \sin(10\pi x_1) - 5x_2 - \cos(10\pi x_2) + N(0, 0.01)x_1, x_2 \in [0, 1]$$

The response generally increases with respect to the x_1 feature, but a sinusoidal variation has been superimposed, resulting in the true effect being non-monotonic. For the x_2 feature the variation is decreasing with a sinusoidal variation.

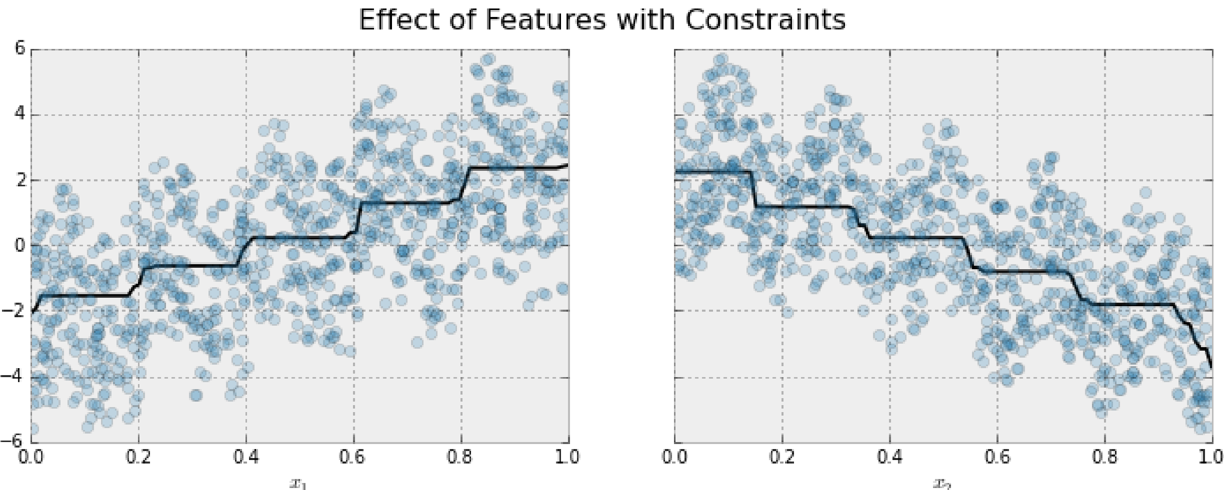


Let's fit a boosted tree model to this data without imposing any monotonic constraints:



The black curve shows the trend inferred from the model for each feature. To make these plots the distinguished feature x_i is fed to the model over a one-dimensional grid of values, while all the other features (in this case only one other feature) are set to their average values. We see that the model does a good job of capturing the general trend with the oscillatory wave superimposed.

Here is the same model, but fit with monotonicity constraints:



We see the effect of the constraint. For each variable the general direction of the trend is still evident, but the oscillatory behaviour no longer remains as it would violate our imposed constraints.

Enforcing Monotonic Constraints in XGBoost

It is very simple to enforce monotonicity constraints in XGBoost. Here we will give an example using Python, but the same general idea generalizes to other platforms.

Suppose the following code fits your model without monotonicity constraints

```
model_no_constraints = xgb.train(params, dtrain,
                                 num_boost_round = 1000, evals = evallist,
                                 early_stopping_rounds = 10)
```

Then fitting with monotonicity constraints only requires adding a single parameter

```
params_constrained = params.copy()
params_constrained['monotone_constraints'] = (1,-1)

model_with_constraints = xgb.train(params_constrained, dtrain,
                                   num_boost_round = 1000, evals = evallist,
                                   early_stopping_rounds = 10)
```

In this example the training data `X` has two columns, and by using the parameter values `(1,-1)` we are telling XGBoost to impose an increasing constraint on the first predictor and a decreasing constraint on the second.

Some other examples:

- `(1,0)`: An increasing constraint on the first predictor and no constraint on the second.
- `(0,-1)`: No constraint on the first predictor and a decreasing constraint on the second.

Note

Note for the 'hist' tree construction algorithm. If `tree_method` is set to either `hist` or `approx`, enabling monotonic constraints may produce unnecessarily shallow trees. This is because the `hist` method reduces the number of candidate splits to be considered at each split. Monotonic constraints may wipe out all available split candidates, in which case no split is made. To reduce the effect, you may want to increase the `max_bin` parameter to consider more split candidates.

Using feature names

XGBoost's Python and R packages support using feature names instead of feature indices for specifying the constraints. Given a data frame with columns `["f0", "f1", "f2"]`, the monotonic constraint can be specified as `{"f0": 1, "f2": -1}` (Python) or as `list(f0=1, f2=-1)` (R, when using 'xgboost()', but not 'xgb.train'), and `"f1"` will default to `0` (no constraint).

1.4.7 Feature Interaction Constraints

The decision tree is a powerful tool to discover interaction among independent variables (features). Variables that appear together in a traversal path are interacting with one another, since the condition of a child node is predicated on the condition of the parent node. For example, the highlighted red path in the diagram below contains three variables: x_1, x_7, and x_{10}, so the highlighted prediction (at the highlighted leaf node) is the product of interaction between x_1, x_7, and x_{10}.

When the tree depth is larger than one, many variables interact on the sole basis of minimizing training loss, and the resulting decision tree may capture a spurious relationship (noise) rather than a legitimate relationship that generalizes across different datasets. **Feature interaction constraints** allow users to decide which variables are allowed to interact and which are not.

Potential benefits include:

- Better predictive performance from focusing on interactions that work – whether through domain specific knowledge or algorithms that rank interactions
- Less noise in predictions; better generalization
- More control to the user on what the model can fit. For example, the user may want to exclude some interactions even if they perform well due to regulatory constraints.

A Simple Example

Feature interaction constraints are expressed in terms of groups of variables that are allowed to interact. For example, the constraint `[0, 1]` indicates that variables x_0 and x_1 are allowed to interact with each other but with no other variable. Similarly, `[2, 3, 4]` indicates that x_2, x_3, and x_4 are allowed to interact with one another but with no other variable. A set of feature interaction constraints is expressed as a nested list, e.g. `[[0, 1], [2, 3, 4]]`, where each inner list is a group of indices of features that are allowed to interact with each other.

In the following diagram, the left decision tree is in violation of the first constraint (`[0, 1]`), whereas the right decision tree complies with both the first and second constraints (`[0, 1]`, `[2, 3, 4]`).

forbidden	allowed

Enforcing Feature Interaction Constraints in XGBoost

It is very simple to enforce feature interaction constraints in XGBoost. Here we will give an example using Python, but the same general idea generalizes to other platforms.

Suppose the following code fits your model without feature interaction constraints:

```
model_no_constraints = xgb.train(params, dtrain,
                                 num_boost_round = 1000, evals = evallist,
                                 early_stopping_rounds = 10)
```

Then fitting with feature interaction constraints only requires adding a single parameter:

```
params_constrained = params.copy()
# Use nested list to define feature interaction constraints
params_constrained['interaction_constraints'] = '[[0, 2], [1, 3, 4], [5, 6]]'
# Features 0 and 2 are allowed to interact with each other but with no other feature
# Features 1, 3, 4 are allowed to interact with one another but with no other feature
# Features 5 and 6 are allowed to interact with each other but with no other feature

model_with_constraints = xgb.train(params_constrained, dtrain,
                                   num_boost_round = 1000, evals = evallist,
                                   early_stopping_rounds = 10)
```

Using feature name instead

XGBoost's Python and R packages support using feature names instead of feature index for specifying the constraints. Given a data frame with columns `["f0", "f1", "f2"]`, the feature interaction constraint can be specified as `[["f0", "f2"]]` (Python) or `list(c("f0", "f2"))` (R, when passing them to function `xgboost()`).

Advanced topic

The intuition behind interaction constraints is simple. Users may have prior knowledge about relations between different features, and encode it as constraints during model construction. But there are also some subtleties around specifying constraints. Take the constraint `[[1, 2], [2, 3, 4]]` as an example. The second feature appears in two different interaction sets, `[1, 2]` and `[2, 3, 4]`. So the union set of features allowed to interact with `2` is `{1, 3, 4}`. In the following diagram, the root splits at feature `2`. Because all its descendants should be able to interact with it, all 4 features are legitimate split candidates at the second layer. At first sight, this might look like disregarding the specified constraint sets, but it is not.

This has lead to some interesting implications of feature interaction constraints. Take `[[0, 1], [0, 1, 2], [1, 2]]` as another example. Assuming we have only 3 available features in our training datasets for presentation purpose,

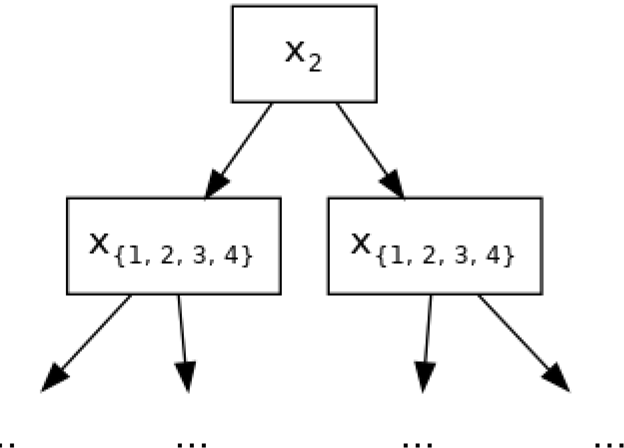


Fig. 1: `{1, 2, 3, 4}` represents the sets of legitimate split features.

careful readers might have found out that the above constraint is the same as simply `[[0, 1, 2]]`. Since no matter which feature is chosen for split in the root node, all its descendants are allowed to include every feature as legitimate split candidates without violating interaction constraints.

For one last example, we use `[[0, 1], [1, 3, 4]]` and choose feature `0` as split for the root node. At the second layer of the built tree, `1` is the only legitimate split candidate except for `0` itself, since they belong to the same constraint set. Following the grow path of our example tree below, the node at the second layer splits at feature `1`. But due to the fact that `1` also belongs to second constraint set `[1, 3, 4]`, at the third layer, we are allowed to include all features as split candidates and still comply with the interaction constraints of its ascendants.

1.4.8 Survival Analysis with Accelerated Failure Time

- *What is survival analysis?*
- *Accelerated Failure Time model*
- *How to use*

What is survival analysis?

Survival analysis (regression) models **time to an event of interest**. Survival analysis is a special kind of regression and differs from the conventional regression task as follows:

- The label is always positive, since you cannot wait a negative amount of time until the event occurs.
- The label may not be fully known, or **censored**, because "it takes time to measure time."

The second bullet point is crucial and we should dwell on it more. As you may have guessed from the name, one of the earliest applications of survival analysis is to model mortality of a given population. Let's take NCCTG Lung Cancer Dataset as an example. The first 8 columns represent features and the last column, Time to death, represents the label.

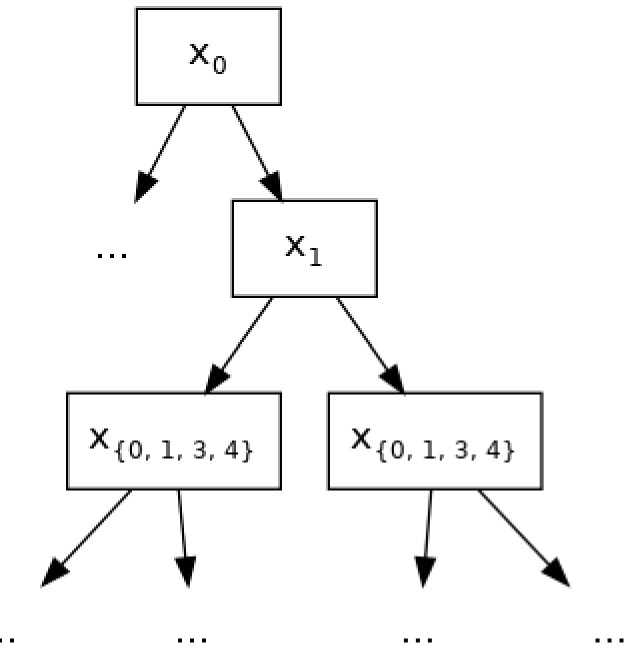


Fig. 2: `{0, 1, 3, 4}` represents the sets of legitimate split features.

Inst	Age	Sex	ph.ecog	ph.karno	pat.karno	meal.cal	wt.loss	**Time to death (days)**
3	74	1	1	90	100	1175	N/A	306
3	68	1	0	90	90	1225	15	455
3	56	1	0	90	90	N/A	15	$[1010, +\infty)$
5	57	1	1	90	60	1150	11	210
1	60	1	0	100	90	N/A	0	883
12	74	1	1	50	80	513	0	$[1022, +\infty)$
7	68	2	2	70	60	384	10	310

Take a close look at the label for the third patient. **His label is a range, not a single number.** The third patient's label is said to be **censored**, because for some reason the experimenters could not get a complete measurement for that label. One possible scenario: the patient survived the first 1010 days and walked out of the clinic on the 1011th day, so his death was not directly observed. Another possibility: The experiment was cut short (since you cannot run it forever) before his death could be observed. In any case, his label is $[1010, +\infty)$, meaning his time to death can be any number that's higher than 1010, e.g. 2000, 3000, or 10000.

There are four kinds of censoring:

- **Uncensored**: the label is not censored and given as a single number.
- **Right-censored**: the label is of form $[a, +\infty)$, where a is the lower bound.
- **Left-censored**: the label is of form $[0, b]$, where b is the upper bound.
- **Interval-censored**: the label is of form $[a, b]$, where a and b are the lower and upper bounds, respectively.

Right-censoring is the most commonly used.

Accelerated Failure Time model

Accelerated Failure Time (AFT) model is one of the most commonly used models in survival analysis. The model is of the following form:

$$\ln Y = \langle \mathbf{w}, \mathbf{x} \rangle + \sigma Z$$

where

- $\mathbf{x}$ is a vector in $\mathbb{R}^d$ representing the features.
- $\mathbf{w}$ is a vector consisting of d coefficients, each corresponding to a feature.
- $\langle \cdot, \cdot \rangle$ is the usual dot product in $\mathbb{R}^d$.
- $\ln{(\cdot)}$ is the natural logarithm.
- Y and Z are random variables.
  - Y is the output label.
  - Z is a random variable of a known probability distribution. Common choices are the normal distribution, the logistic distribution, and the extreme distribution. Intuitively, Z represents the "noise" that pulls the prediction $\langle \mathbf{w}, \mathbf{x} \rangle$ away from the true log label $\ln Y$.
- σ is a parameter that scales the size of Z.

Note that this model is a generalized form of a linear regression model $Y = \langle \mathbf{w}, \mathbf{x} \rangle$. In order to make AFT work with gradient boosting, we revise the model as follows:

$$\ln Y = \mathcal{T}(\mathbf{x}) + \sigma Z$$

where $\mathcal{T}(\mathbf{x})$ represents the output from a decision tree ensemble, given input $\mathbf{x}$. Since Z is a random variable, we have a likelihood defined for the expression $\ln Y = \mathcal{T}(\mathbf{x}) + \sigma Z$. So the goal for XGBoost is to maximize the (log) likelihood by fitting a good tree ensemble $\mathcal{T}(\mathbf{x})$.

How to use

The first step is to express the labels in the form of a range, so that **every data point has two numbers associated with it, namely the lower and upper bounds for the label.** For uncensored labels, use a degenerate interval of form $[a, a]$.

Censoring type	Interval form	Lower bound finite?	Upper bound finite?
Uncensored	$[a, a]$	✓	✓
Right-censored	$[a, +\infty)$	✓	
Left-censored	$[0, b]$	✓	✓
Interval-censored	$[a, b]$	✓	✓

Collect the lower bound numbers in one array (let's call it `y_lower_bound`) and the upper bound number in another array (call it `y_upper_bound`). The ranged labels are associated with a data matrix object via calls to *xgboost.DMatrix.set_float_info()*:

Listing 14: Python

```
import numpy as np
import xgboost as xgb

# 4-by-2 Data matrix
X = np.array([[1, -1], [-1, 1], [0, 1], [1, 0]])
```

(continues on next page)

(continued from previous page)

```
dtrain = xgb.DMatrix(X)

# Associate ranged labels with the data matrix.
# This example shows each kind of censored labels.
#                         uncensored    right     left  interval
y_lower_bound = np.array([      2.0,      3.0,     0.0,     4.0])
y_upper_bound = np.array([      2.0, +np.inf,     4.0,     5.0])
dtrain.set_float_info('label_lower_bound', y_lower_bound)
dtrain.set_float_info('label_upper_bound', y_upper_bound)
```

Listing 15: R

```
library(xgboost)

# 4-by-2 Data matrix
X <- matrix(c(1., -1., -1., 1., 0., 1., 1., 0.),
            nrow=4, ncol=2, byrow=TRUE)
dtrain <- xgb.DMatrix(X)

# Associate ranged labels with the data matrix.
# This example shows each kind of censored labels.
#                   uncensored  right  left  interval
y_lower_bound <- c(        2.,    3.,   0.,       4.)
y_upper_bound <- c(        2.,  +Inf,   4.,       5.)
setinfo(dtrain, 'label_lower_bound', y_lower_bound)
setinfo(dtrain, 'label_upper_bound', y_upper_bound)
```

Now we are ready to invoke the training API:

Listing 16: Python

```
params = {'objective': 'survival:aft',
          'eval_metric': 'aft-nloglik',
          'aft_loss_distribution': 'normal',
          'aft_loss_distribution_scale': 1.20,
          'tree_method': 'hist', 'learning_rate': 0.05, 'max_depth': 2}
bst = xgb.train(params, dtrain, num_boost_round=5,
                evals=[(dtrain, 'train')])
```

Listing 17: R

```
params <- list(objective='survival:aft',
               eval_metric='aft-nloglik',
               aft_loss_distribution='normal',
               aft_loss_distribution_scale=1.20,
               tree_method='hist',
               learning_rate=0.05,
               max_depth=2)
watchlist <- list(train = dtrain)
bst <- xgb.train(params, dtrain, nrounds=5, watchlist)
```

We set `objective` parameter to `survival:aft` and `eval_metric` to `aft-nloglik`, so that the log likelihood for the AFT model would be maximized. (XGBoost will actually minimize the negative log likelihood, hence the name

aft-nloglik.)

The parameter aft_loss_distribution corresponds to the distribution of the Z term in the AFT model, and aft_loss_distribution_scale corresponds to the scaling factor σ.

Currently, you can choose from three probability distributions for aft_loss_distribution:

aft_loss_distribution	Probability Density Function (PDF)
normal	$\frac{\exp{(-z^2/2)}}{\sqrt{2\pi}}$
logistic	$\frac{e^z}{(1+e^z)^2}$
extreme	$e^z e^{-\exp{z}}$

Note that it is not yet possible to set the ranged label using the scikit-learn interface (e.g. *xgboost.XGBRegressor*). For now, you should use *xgboost.train* with *xgboost.DMatrix*. For a collection of Python examples, see *Survival Analysis Walkthrough*

1.4.9 Categorical Data

Contents

- *Training with scikit-learn Interface*
- *Optimal Partitioning*
- *Using native interface*
- *Auto-recoding (Data Consistency)*
- *Miscellaneous*
- *References*

Since version 1.5, XGBoost has support for categorical data. For numerical data, the split condition is defined as $value < threshold$, while for categorical data the split is defined depending on whether partitioning or onehot encoding is used. For partition-based splits, the splits are specified as $value \in categories$, where categories is the set of categories in one feature. If onehot encoding is used instead, then the split is defined as $value == category$. More advanced categorical split strategy is planned for future releases and this tutorial details how to inform XGBoost about the data type.

Training with scikit-learn Interface

The easiest way to pass categorical data into XGBoost is using dataframe and the scikit-learn interface like *XGBClassifier*. For preparing the data, users need to specify the data type of input predictor as category. For pandas/cudf Dataframe, this can be achieved by

```
X["cat_feature"].astype("category")
```

for all columns that represent categorical features. After which, users can tell XGBoost to enable training with categorical data. Assuming that you are using the *XGBClassifier* for classification problem, specify the parameter enable_categorical:

```
# Supported tree methods are `approx` and `hist`.
clf = xgb.XGBClassifier(tree_method="hist", enable_categorical=True, device="cuda")
# X is the dataframe we created in previous snippet
clf.fit(X, y)
# Must use JSON/UBJSON for serialization, otherwise the information is lost.
clf.save_model("categorical-model.json")
```

Once training is finished, most of other features can utilize the model. For instance one can plot the model and calculate the global feature importance:

```
# Get a graph
graph = xgb.to_graphviz(clf, num_trees=1)
# Or get a matplotlib axis
ax = xgb.plot_tree(clf, num_trees=1)
# Get feature importances
clf.feature_importances_
```

The `scikit-learn` interface from dask is similar to single node version. The basic idea is create dataframe with category feature type, and tell XGBoost to use it by setting the `enable_categorical` parameter. See *Getting started with categorical data* for a worked example of using categorical data with `scikit-learn` interface with one-hot encoding. A comparison between using one-hot encoded data and XGBoost's categorical data support can be found *Train XGBoost with cat_in_the_dat dataset*.

Added in version 3.0: Support for the R package using `factor`.

Optimal Partitioning

Added in version 1.6.

Optimal partitioning is a technique for partitioning the categorical predictors for each node split, the proof of optimality for numerical output was first introduced by *[1]*. The algorithm is used in decision trees *[2]*, later LightGBM *[3]* brought it to the context of gradient boosting trees and now is also adopted in XGBoost as an optional feature for handling categorical splits. More specifically, the proof by Fisher *[1]* states that, when trying to partition a set of discrete values into groups based on the distances between a measure of these values, one only needs to look at sorted partitions instead of enumerating all possible permutations. In the context of decision trees, the discrete values are categories, and the measure is the output leaf value. Intuitively, we want to group the categories that output similar leaf values. During split finding, we first sort the gradient histogram to prepare the contiguous partitions then enumerate the splits according to these sorted values. One of the related parameters for XGBoost is `max_cat_to_onehot`, which controls whether one-hot encoding or partitioning should be used for each feature, see *Parameters for Categorical Feature* for details.

Using native interface

The `scikit-learn` interface is user friendly, but lacks some features that are only available in native interface. For instance users cannot compute SHAP value directly. Also native interface supports more data types. To use the native interface with categorical data, we need to pass the similar parameter to `DMatrix` or `QuantileDMatrix` and the `train` function. For dataframe input:

```
# X is a dataframe we created in previous snippet
Xy = xgb.DMatrix(X, y, enable_categorical=True)
booster = xgb.train({"tree_method": "hist", "max_cat_to_onehot": 5}, Xy)
# Must use JSON for serialization, otherwise the information is lost
booster.save_model("categorical-model.json")
```

SHAP value computation:

```
SHAP = booster.predict(Xy, pred_interactions=True)

# categorical features are listed as "c"
print(booster.feature_types)
```

For other types of input, like `numpy array`, we can tell XGBoost about the feature types by using the `feature_types` parameter in *DMatrix*:

```
# "q" is numerical feature, while "c" is categorical feature
ft = ["q", "c", "c"]
X: np.ndarray = load_my_data()
assert X.shape[1] == 3
Xy = xgb.DMatrix(X, y, feature_types=ft, enable_categorical=True)
```

For numerical data, the feature type can be `"q"` or `"float"`, while for categorical feature it's specified as `"c"`. The Dask module in XGBoost has the same interface so `dask.Array` can also be used for categorical data. Lastly, the sklearn interface *XGBRegressor* has the same parameter.

Auto-recoding (Data Consistency)

Changed in version 3.1: Starting with XGBoost 3.1, the **Python** interface can perform automatic re-coding for new inputs.

XGBoost accepts parameters to indicate which feature is considered categorical, either through the `dtypes` of a dataframe or through the `feature_types` parameter. However, except for the Python interface, XGBoost doesn't store the information about how categories are encoded in the first place. For instance, given an encoding schema that maps music genres to integer codes:

```
{"acoustic": 0, "indie": 1, "blues": 2, "country": 3}
```

Aside from the Python interface (R/Java/C, etc), XGBoost doesn't know this mapping from the input and hence cannot store it in the model. The mapping usually happens in the users' data engineering pipeline. To ensure the correct result from XGBoost, users need to keep the pipeline for transforming data consistent across training and testing data.

Starting with 3.1, the **Python** interface can remember the encoding and perform recoding during inference and training continuation when the input is a dataframe (*pandas*, *cuDF*, *polars*, *pyarrow*, *modin*). The feature support focuses on basic usage. It has some restrictions on the types of inputs that can be accepted. First, category names must have one of the following types:

- string
- integer, from 8-bit to 64-bit, both signed and unsigned are supported.
- 32-bit or 64-bit floating point

Other category types are not supported. Second, the input types must be strictly consistent. For example, XGBoost will raise an error if the categorical columns in the training set are unsigned integers whereas the test dataset has signed integer columns. If you have categories that are not one of the supported types, you need to perform the re-coding using a pre-processing data transformer like the `sklearn.preprocessing.OrdinalEncoder`. See *Feature engineering pipeline for categorical data* for a worked example using an ordinal encoder. To clarify, the type here refers to the type of the name of categories (called `Index` in pandas):

```
# string type
{"acoustic": 0, "indie": 1, "blues": 2, "country": 3}
# integer type
{-1: 0, 1: 1, 3: 2, 7: 3}
```

(continues on next page)

(continued from previous page)

```
# depending on the dataframe implementation, it can be signed or unsigned.
{5: 0, 1: 1, 3: 2, 7: 3}
# floating point type, both 32-bit and 64-bit are supported.
{-1.0: 0, 1.0: 1, 3.0: 2, 7.0: 3}
```

Internally, XGBoost attempts to extract the categories from the dataframe inputs. For inference (predict), the re-coding happens on the fly and there's no data copy (baring some internal transformations performed by the dataframe itself). For training continuation however, re-coding requires some extra steps if you are using the native interface. The sklearn interface and the Dask interface can handle training continuation automatically. Last, please note that using the re-coder with the native interface is still experimental. It's ready for testing, but we want to observe the feature usage for a period of time and might make some breaking changes if needed. The following is a snippet of using the native interface:

```
import pandas as pd

X = pd.DataFrame()
Xy = xgboost.QuantileDMatrix(X, y, enable_categorical=True)
booster = xgboost.train({}, Xy)

# XGBoost can handle re-coding for inference without user intervention
X_new = pd.DataFrame()
booster.inplace_predict(X_new)

# Get categories saved in the model for training continuation
categories = booster.get_categories()
# Use saved categories as a reference for re-coding.
# Training continuation requires a re-coded DMatrix, pass the categories as feature_types
Xy_new = xgboost.QuantileDMatrix(
  X_new, y_new, feature_types=categories, enable_categorical=True, ref=Xy
)
booster_1 = xgboost.train({}, Xy_new, xgb_model=booster)
```

No extra step is required for using the scikit-learn interface as long as the inputs are dataframes. During training continuation, XGBoost will either extract the categories from the previous model or use the categories from the new training dataset if the input model doesn't have the information. As a side note, users can inspect the content of the categories by exporting it to arrow arrays. This interface is still experimental:

```
categories = booster.get_categories(export_to_arrow=True)
print(categories.to_arrow())
```

In addition to the notes above, there's a blog post about using XGBoost with Polars for categorical features with various examples.

The re-coder handles missing categories at inference time. However, if there's a new category during inference that's unseen during training (missing during training), a re-coder doesn't help as it doesn't know what would be a valid code. There are various heuristics for handling unseen categories during inference. The best and simplest approach is to re-train the model since a new category represents a new type of data. The type of a categorical feature is defined by the set of discrete values. If the set is changed, then the type is considered to be different. In addition, one might add an "unknown" category during training and synthesize some samples with this category as missing values. Lastly, you might consider the new category similar to an existing one based on your domain knowledge, and map to that category during ETL.

For **R**, the auto-recoding is not yet supported as of 3.1. To provide an example:

```
> f0 = factor(c("a", "b", "c"))
> as.numeric(f0)
[1] 1 2 3
> f0
[1] a b c
Levels: a b c
```

In the above snippet, we have the mapping: `a -> 1, b -> 2, c -> 3`. Assuming the above is the training data, and the next snippet is the test data:

```
> f1 = factor(c("a", "c"))
> as.numeric(f1)
[1] 1 2
> f1
[1] a c
Levels: a c
```

Now, we have `a -> 1, c -> 2` because `b` is missing, and the R factor encodes the data differently, resulting in invalid test-time encoding. XGBoost cannot remember the original encoding for the R package. You will have to encode the data explicitly during inference:

```
> f1 = factor(c("a", "c"), levels = c("a", "b", "c"))
> f1
[1] a c
Levels: a b c
> as.numeric(f1)
  [1] 1 3
```

Miscellaneous

By default, XGBoost assumes input category codes are integers starting from 0 till the number of categories $[0, n_categories)$. However, user might provide inputs with invalid values due to mistakes or missing values in training dataset. It can be negative value, integer values that can not be accurately represented by 32-bit floating point, or values that are larger than actual number of unique categories. During training this is validated but for prediction it's treated as the same as not-chosen category for performance reasons.

References

[1] Walter D. Fisher. "On Grouping for Maximum Homogeneity". Journal of the American Statistical Association. Vol. 53, No. 284 (Dec., 1958), pp. 789-798.

[2] Trevor Hastie, Robert Tibshirani, Jerome Friedman. "The Elements of Statistical Learning". Springer Series in Statistics Springer New York Inc. (2001).

[3] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, Tie-Yan Liu. "LightGBM: A Highly Efficient Gradient Boosting Decision Tree." Advances in Neural Information Processing Systems 30 (NIPS 2017), pp. 3149-3157.

1.4.10 Multiple Outputs

Contents

- *Training with One-Model-Per-Target*
- *Training with Vector Leaf*
- *Using Reduced Gradient (Sketch Boost)*
- *References*

Added in version 1.6.

Starting from version 1.6, XGBoost has experimental support for multi-output regression and multi-label classification with Python package. Multi-label classification usually refers to targets that have multiple non-exclusive class labels. For instance, a movie can be simultaneously classified as both sci-fi and comedy. For detailed explanation of terminologies related to different multi-output models please refer to the scikit-learn user guide.

Note

As of XGBoost 3.0, the feature is experimental and has limited features. Only the Python package is tested. In addition, `glinear` is not supported.

Training with One-Model-Per-Target

By default, XGBoost builds one model for each target similar to sklearn meta estimators, with the added benefit of reusing data and other integrated features like SHAP. For a worked example of regression, see *A demo for multi-output regression*. For multi-label classification, the binary relevance strategy is used. Input `y` should be of shape `(n_samples, n_classes)` with each column having a value of 0 or 1 to specify whether the sample is labeled as positive for respective class. Given a sample with 3 output classes and 2 labels, the corresponding *y* should be encoded as `[1, 0, 1]` with the second class labeled as negative and the rest labeled as positive. At the moment XGBoost supports only dense matrix for labels.

```
from sklearn.datasets import make_multilabel_classification
import numpy as np

X, y = make_multilabel_classification(
    n_samples=32, n_classes=5, n_labels=3, random_state=0
)
clf = xgb.XGBClassifier(tree_method="hist")
clf.fit(X, y)
np.testing.assert_allclose(clf.predict(X), y)
```

The feature is still under development with limited support from objectives and metrics.

Training with Vector Leaf

Added in version 2.0.0.

Note

This is still working-in-progress, and most features are missing.

XGBoost can optionally build multi-output trees with the size of leaf equals to the number of targets when the tree method *hist* is used. The behavior can be controlled by the `multi_strategy` training parameter, which can take the

value *one_output_per_tree* (the default) for building one model per-target or *multi_output_tree* for building multi-output trees.

```
clf = xgb.XGBClassifier(tree_method="hist", multi_strategy="multi_output_tree")
```

See *A demo for multi-output regression* for a worked example with regression.

Using Reduced Gradient (Sketch Boost)

Added in version 3.2.0.

Note

This is still working-in-progress, and most features are missing. It is documented here for early testers to provide feedback. Related interface might change without notice.

When the number of targets is large, training a gradient boosting tree model using the full gradient matrix becomes challenging. The training procedure may run out of memory for storing the histogram, or run extremely slowly due to the amount of computation needed. As an optimization, XGBoost implements an interface for using two types of gradients based on the concepts from *Sketch Boost* [1].

The key insight is that we can use different gradients for two distinct purposes:

- **Split gradient**: A reduced-dimension gradient used to determine the tree structure.
- **Value gradient**: The full gradient used to calculate the final leaf values for accurate predictions.

This separation allows the expensive histogram building and split finding to operate on a smaller gradient matrix, while still producing valid predictions using the full loss function for leaf values. The *Sketch Boost* paper proposes using dimensionality reduction on the gradient matrix. In practice, one can also define a different but related loss with a small gradient matrix for finding the tree structure.

To access this feature, create a custom objective that inherits from `TreeObjective` and implement the `split_grad` method.

```
from xgboost.objective import TreeObjective
from cuml.decomposition import TruncatedSVD

import cupy as cp

class LsObj(TreeObjective):
    def __call__(self, iteration: int, y_pred, dtrain):
        """Least squared error."""
        y_true = dtrain.get_label()
        grad = y_pred - y_true
        hess = cp.ones(grad.shape)
        return cp.array(grad), cp.array(hess)

    def split_grad(self, iteration: int, grad, hess):
        svd_params = {"algorithm": "jacobi", "n_components": 2, "n_iter": 8}
        svd = TruncatedSVD(output_type="cupy", **svd_params)
        svd.fit(grad)
        grad = svd.transform(grad)
        hess = svd.transform(hess)
        hess = cp.clip(hess, 0.01, None)
```

(continues on next page)

(continued from previous page)

```
    return grad, hess
```

See *A demo for multi-output regression using reduced gradient* for a complete worked example. The feature supports only the `multi_strategy=multi_output_tree`.

References

[1] Leonid Iosipoi, Anton Vakhrushev. "Fast Gradient Boosted Decision Tree for Multioutput Problems". NeurIPS 2022, pp 25422 - 25435.

1.4.11 Random Forests(TM) in XGBoost

XGBoost is normally used to train gradient-boosted decision trees and other gradient boosted models. Random Forests use the same model representation and inference, as gradient-boosted decision trees, but a different training algorithm. One can use XGBoost to train a standalone random forest or use random forest as a base model for gradient boosting. Here we focus on training standalone random forest.

We have native APIs for training random forests since the early days, and a new Scikit-Learn wrapper after 0.82 (not included in 0.82). Please note that the new Scikit-Learn wrapper is still **experimental**, which means we might change the interface whenever needed.

Standalone Random Forest With XGBoost API

The following parameters must be set to enable random forest training.

- `booster` should be set to `gbtree`, as we are training forests. Note that as this is the default, this parameter needn't be set explicitly.
- `subsample` must be set to a value less than 1 to enable random selection of training cases (rows).
- One of `colsample_by*` parameters must be set to a value less than 1 to enable random selection of columns. Normally, `colsample_bynode` would be set to a value less than 1 to randomly sample columns at each tree split.
- `num_parallel_tree` should be set to the size of the forest being trained.
- `num_boost_round` should be set to 1 to prevent XGBoost from boosting multiple random forests. Note that this is a keyword argument to `train()`, and is not part of the parameter dictionary.
- `eta` (alias: `learning_rate`) must be set to 1 when training random forest regression.
- `random_state` can be used to seed the random number generator.

Other parameters should be set in a similar way they are set for gradient boosting. For instance, `objective` will typically be `reg:squarederror` for regression and `binary:logistic` for classification, `lambda` should be set according to a desired regularization weight, etc.

If both `num_parallel_tree` and `num_boost_round` are greater than 1, training will use a combination of random forest and gradient boosting strategy. It will perform `num_boost_round` rounds, boosting a random forest of `num_parallel_tree` trees at each round. If early stopping is not enabled, the final model will consist of `num_parallel_tree` * `num_boost_round` trees.

Here is a sample parameter dictionary for training a random forest on a GPU using xgboost:

```
params = {
  "colsample_bynode": 0.8,
  "learning_rate": 1,
  "max_depth": 5,
```

(continues on next page)

(continued from previous page)

```
  "num_parallel_tree": 100,
  "objective": "binary:logistic",
  "subsample": 0.8,
  "tree_method": "hist",
  "device": "cuda",
}
```

A random forest model can then be trained as follows:

```
bst = train(params, dmatrix, num_boost_round=1)
```

Standalone Random Forest With Scikit-Learn-Like API

XGBRFClassifier and XGBRFRegressor are SKL-like classes that provide random forest functionality. They are basically versions of XGBClassifier and XGBRegressor that train random forest instead of gradient boosting, and have default values and meaning of some of the parameters adjusted accordingly. In particular:

- n_estimators specifies the size of the forest to be trained; it is converted to num_parallel_tree, instead of the number of boosting rounds
- learning_rate is set to 1 by default
- colsample_bynode and subsample are set to 0.8 by default
- booster is always gbtree

For a simple example, you can train a random forest regressor with:

```
from sklearn.model_selection import KFold

# Your code ...

kf = KFold(n_splits=2)
for train_index, test_index in kf.split(X, y):
    xgb_model = xgb.XGBRFRegressor(random_state=42).fit(
    X[train_index], y[train_index])
```

Note that these classes have a smaller selection of parameters compared to using train(). In particular, it is impossible to combine random forests with gradient boosting using this API.

Caveats

- XGBoost uses 2nd order approximation to the objective function. This can lead to results that differ from a random forest implementation that uses the exact value of the objective function.
- XGBoost does not perform replacement when subsampling training cases. Each training case can occur in a subsampled set either 0 or 1 time.

1.4.12 Distributed XGBoost on Kubernetes

Distributed XGBoost training on Kubernetes is supported via Kubeflow Trainer. Kubeflow Trainer provides a built-in XGBoost runtime that manages the scheduling, distributed coordination, and lifecycle of XGBoost training jobs on Kubernetes clusters.

This tutorial covers the end-to-end workflow: from setting up prerequisites, through writing distributed training code, to launching and monitoring multi-node XGBoost jobs.

- *Overview*
  - *Architecture*
  - *Environment Variables*
  - *Worker Count Calculation*
- *Prerequisites*
  - *Verify the Installation*
- *XGBoost ClusterTrainingRuntime*
- *Example: Distributed XGBoost Training*
  - *Using the Python SDK*
    - *Step 1: Define the Training Function*
    - *Step 2: Submit the Training Job*
    - *Step 3: Monitor the Training Job*
    - *Step 4: Clean Up*
  - *Using kubectl with YAML*
    - *CPU Training Example*
    - *GPU Training Example*
    - *Monitoring with kubectl*
- *How It Works*
  - *XGBoost Runtime Plugin*
  - *Tracker Discovery*
- *Best Practices*
  - *Use QuantileDMatrix for Memory Efficiency*
  - *Early Stopping*
  - *Logging in Distributed Mode*
  - *Setting verbose_eval for Production*
  - *Checkpointing*
  - *Data Partitioning*
  - *Rank-Specific Logic*
- *Common Issues and Edge Cases*
  - *Reserved Environment Variables*
  - *No Environment Injection When Trainer Is Nil*
  - *Resource Precedence: TrainJob Overrides Runtime*
  - *GPU Device Ordinal in Distributed Mode*
  - *Data Matrices Must Be Inside CommunicatorContext*

  - *Single-Node Defaults*
  - *CPU Over-Subscription*
- *Support*

Overview

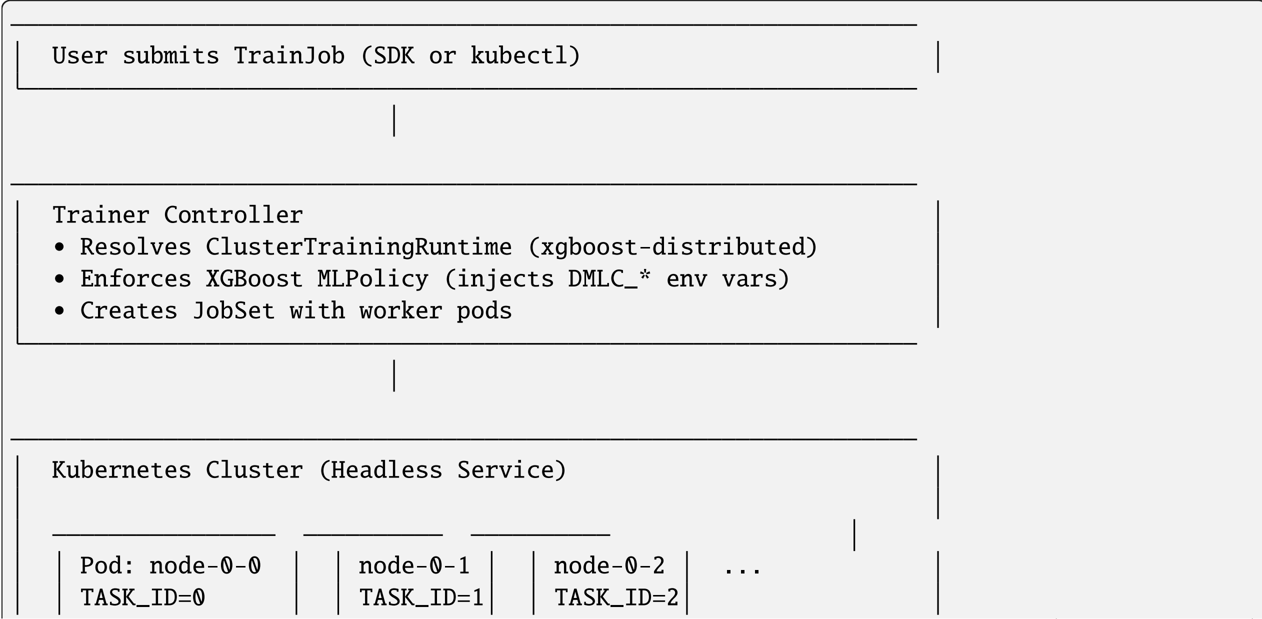
XGBoost supports distributed training through the **Collective** communication protocol (historically known as Rabit). In a distributed setting, multiple worker processes each operate on a shard of the data and synchronize histogram bin statistics via AllReduce to agree on the best tree splits. Kubeflow Trainer's XGBoost runtime automates the orchestration of this process on Kubernetes by:

- Deploying worker pods as a JobSet
- Automatically injecting the `DMLC_*` environment variables required by XGBoost's Collective communication layer
- Providing the rank-0 pod with the tracker address so user code can start a `RabitTracker` for worker coordination
- Supporting both CPU and GPU training workloads

Architecture

The distributed XGBoost training architecture on Kubernetes consists of the following components:

1. **TrainJob**: A Kubernetes custom resource that declares the training job configuration (number of nodes, resources per node, training code).
2. **ClusterTrainingRuntime**: A cluster-scoped resource that defines the XGBoost runtime template (container image, ML policy, default settings). The built-in runtime is named `xgboost-distributed`.
3. **Trainer Controller**: Resolves the `TrainJob` against the referenced runtime, enforces the XGBoost ML policy (injects environment variables), and creates the underlying `JobSet`.
4. **Worker Pods**: Each pod runs the same training script. The user's training code on the rank-0 pod is responsible for starting a `RabitTracker` for coordination.



(continues on next page)

(continued from previous page)

```
│  │ (Tracker)        │  │ (Worker) │  │ (Worker) │                  │
│  └────────────────     └──────────   └──────────               │
│                                                                 │
│          │             │              │                         │
│          └──── Collective Protocol ───────                     │
└─────────────────────────────────────────────────────────────────┘
```

Environment Variables

The XGBoost runtime plugin automatically injects the following environment variables into each worker pod. These are native to XGBoost's Collective protocol:

Variable	Description	Example Value
`DMLC_TRACKER_URI`	DNS address of the rank-0 pod running the tracker	`myjob-node-0-0.myjob`
`DMLC_TRACKER_PORT`	Port for tracker communication	`29500`
`DMLC_TASK_ID`	Worker rank (derived from pod completion index)	`0`, `1`, `2`, `...`
`DMLC_NUM_WORKER`	Total number of workers across all nodes	`4`

These environment variables are **reserved** and cannot be manually set by the user in the `TrainJob` spec. The runtime plugin validates this and rejects any `TrainJob` that attempts to override them.

Worker Count Calculation

The total number of workers (`DMLC_NUM_WORKER`) is calculated as:

```
DMLC_NUM_WORKER = numNodes × workersPerNode
```

Where `workersPerNode` is determined by:

- **CPU training**: 1 worker per node. XGBoost does **not** spawn multiple worker processes for CPU training. Instead, a single worker process uses OpenMP to parallelize tree building across all available CPU cores on the node. This means if a pod has 8 CPU cores, 1 XGBoost worker will use all 8 cores for intra-process parallelism (histogram construction, split evaluation, etc.).

  The number of threads can be controlled with the `nthread` Booster parameter:

  ```
  # By default, XGBoost uses all available CPU cores.
  # Set nthread to limit the number of OpenMP threads per worker.
  params = {
      "objective": "binary:logistic",
      "nthread": 4,          # Use only 4 of the available cores
      "tree_method": "hist",
  }
  ```

  The `nthread` parameter in the DMatrix constructor controls parallelism during data loading, while `nthread` in the Booster parameters controls parallelism during training. If not set, both default to the maximum number of threads available on the machine.

  > **Tip**
  >
  > When setting `resourcesPerNode` CPU requests in your `TrainJob`, align the `nthread` parameter with the CPU requests to avoid over-subscription. For example, if you request `cpu: "4"`, set `"nthread": 4` in

your training parameters.

- **GPU training**: 1 worker per GPU. The GPU count is derived from the `resourcesPerNode` limits in the `TrainJob` or runtime template. In distributed environments, use `device="cuda"` (not `"cuda:<ordinal>"`); GPU ordinal selection is handled by the distributed framework, and specifying an ordinal will result in an error.

Configuration	numNodes	workersPerNode	DMLC_NUM_WORKER
4 nodes, CPU-only	4	1	4
2 nodes, 4 GPUs each	2	4	8
1 node, 8 GPUs	1	8	8

Prerequisites

Before running distributed XGBoost jobs on Kubernetes, ensure the following:

1. **Kubernetes Cluster**: A running Kubernetes cluster (v1.27+). You can use kind, minikube, or a managed Kubernetes service (GKE, EKS, AKS).
2. **kubectl**: The Kubernetes CLI tool, configured to communicate with your cluster. See the kubectl installation guide.
3. **Kubeflow Trainer**: Install Kubeflow Trainer and its dependencies (JobSet) on your cluster. Follow the Kubeflow Trainer installation guide:

```
# Install the Kubeflow Trainer control plane (includes JobSet).
kubectl apply --server-side -k "github.com/kubeflow/trainer/manifests/overlays/
↪standalone"
```

4. **Kubeflow Python SDK** (optional, for programmatic job submission):

```
pip install kubeflow
```

5. **GPU Support** (optional, for GPU training): Ensure the NVIDIA GPU Operator or equivalent device plugin is installed on your cluster.

Verify the Installation

After installing Kubeflow Trainer, verify that the XGBoost runtime is available:

```
kubectl get clustertrainingruntime
```

You should see the `xgboost-distributed` runtime listed:

```
NAME                   AGE
xgboost-distributed    1m
```

XGBoost ClusterTrainingRuntime

The `xgboost-distributed` `ClusterTrainingRuntime` is deployed as part of the Kubeflow Trainer installation. It defines the default XGBoost runtime template:

```
apiVersion: trainer.kubeflow.org/v1alpha1
kind: ClusterTrainingRuntime
```

(continues on next page)

(continued from previous page)

```yaml
metadata:
  name: xgboost-distributed
  labels:
    trainer.kubeflow.org/framework: xgboost
spec:
  mlPolicy:
    numNodes: 1
    xgboost: {}
  template:
    spec:
      replicatedJobs:
        - name: node
          template:
            metadata:
              labels:
                trainer.kubeflow.org/trainjob-ancestor-step: trainer
            spec:
              template:
                spec:
                  containers:
                    - name: node
                      image: ghcr.io/kubeflow/trainer/xgboost-runtime:latest
```

Key points:

- `mlPolicy.xgboost: {}` activates the XGBoost runtime plugin, which handles injection of `DMLC_*` environment variables.
- `numNodes` defaults to 1 and can be overridden per `TrainJob`.
- The container image `ghcr.io/kubeflow/trainer/xgboost-runtime:latest` is based on `nvidia/cuda:12.4.0-runtime-ubuntu22.04` and includes XGBoost 3.0.2 with CUDA 12 support, NumPy, and scikit-learn.

Example: Distributed XGBoost Training

This section demonstrates two approaches for running distributed XGBoost training: using the Python SDK (recommended for interactive use) and using `kubectl` with YAML manifests.

Using the Python SDK

The Kubeflow Python SDK provides a `TrainerClient` that simplifies submitting and managing training jobs programmatically.

Step 1: Define the Training Function

Write the training function that will be serialized and executed on each worker node. The `DMLC_*` environment variables are automatically injected by the runtime.

```python
def xgboost_train_classification():
    """
    Distributed XGBoost training function using the Collective API.

    DMLC_* env vars are injected by the Kubeflow Trainer XGBoost plugin:
```

(continues on next page)

(continued from previous page)

```
  - DMLC_TRACKER_URI:  DNS name of the rank-0 pod running the tracker
  - DMLC_TRACKER_PORT: Port for tracker communication (default: 29500)
  - DMLC_TASK_ID:      Worker rank (0, 1, 2, ...)
  - DMLC_NUM_WORKER:   Total number of workers
"""
import os
import xgboost as xgb
from xgboost import collective as coll
from xgboost.tracker import RabitTracker
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Read injected environment variables.
rank = int(os.environ["DMLC_TASK_ID"])
world_size = int(os.environ["DMLC_NUM_WORKER"])
tracker_uri = os.environ["DMLC_TRACKER_URI"]
tracker_port = int(os.environ["DMLC_TRACKER_PORT"])

# Rank 0 starts the Rabit tracker (required for coordination).
tracker = None
if rank == 0:
    tracker = RabitTracker(
        host_ip="0.0.0.0", n_workers=world_size, port=tracker_port
    )
    tracker.start()

# All workers connect to the tracker via the Collective communicator.
with coll.CommunicatorContext(
    dmlc_tracker_uri=tracker_uri,
    dmlc_tracker_port=tracker_port,
    dmlc_task_id=str(rank),
):
    # Generate synthetic classification data.
    # In practice, each worker would load its own data shard.
    X, y = make_classification(
        n_samples=10000, n_features=20, n_informative=10,
        n_classes=2, random_state=42 + rank,
    )
    X_train, X_valid, y_train, y_valid = train_test_split(
        X, y, test_size=0.2, random_state=42,
    )

    # NOTE: DMatrix construction MUST be inside the communicator context
    # because it involves cross-worker synchronization for quantization.
    #
    # Use QuantileDMatrix instead of DMatrix for the hist tree method
    # (the default). QuantileDMatrix quantizes data on-the-fly, avoiding
    # an intermediate dense copy and significantly reducing memory usage.
    dtrain = xgb.QuantileDMatrix(X_train, label=y_train)
    # Validation QuantileDMatrix must reference the training matrix
    # so that the same quantile bins are reused.
```

(continues on next page)

(continued from previous page)

```
        dvalid = xgb.QuantileDMatrix(X_valid, label=y_valid, ref=dtrain)

        # Training parameters.
        params = {
            "objective": "binary:logistic",
            "max_depth": 6,
            "eta": 0.1,
            "eval_metric": "logloss",
        }

        # Distributed training - workers synchronize histogram stats via collective ops.
        # early_stopping_rounds activates early stopping based on the validation metric.
        # verbose_eval=10 prints evaluation results every 10 rounds (rank 0 only).
        model = xgb.train(
            params, dtrain,
            num_boost_round=100,
            evals=[(dvalid, "validation")],
            early_stopping_rounds=10,
            verbose_eval=10,
        )

        # Note: early_stopping_rounds returns the *last* model, not the best.
        # Use bst.best_iteration to slice the model to the best round.
        if hasattr(model, "best_iteration"):
            model = model[: model.best_iteration + 1]

        # Evaluate on validation set.
        preds = model.predict(dvalid)
        predictions = [1 if p > 0.5 else 0 for p in preds]
        accuracy = accuracy_score(y_valid, predictions)

        # Only perform logging and model saving from rank 0
        # to avoid duplicate output and file write conflicts.
        if coll.get_rank() == 0:
            print(f"Validation Accuracy: {accuracy:.4f}")
            model.save_model("/workspace/xgboost_model.json")
            print("Model saved to /workspace/xgboost_model.json")

    # Wait for tracker to finish (rank 0 only).
    if tracker is not None:
        tracker.wait_for()
```

Step 2: Submit the Training Job

Use the `TrainerClient` to submit the training function as a distributed job:

```
from kubeflow.trainer import CustomTrainer, TrainerClient

client = TrainerClient()

# Submit a distributed XGBoost training job on 3 nodes.
job_name = client.train(
```

(continues on next page)

(continued from previous page)

```
    trainer=CustomTrainer(
        func=xgboost_train_classification,
        num_nodes=3,
        resources_per_node={"cpu": 3},
    ),
    runtime="xgboost-distributed",
)

print(f"TrainJob '{job_name}' submitted")
```

For GPU training, include GPU resources:

```
job_name = client.train(
    trainer=CustomTrainer(
        func=xgboost_train_classification,
        num_nodes=2,
        resources_per_node={
            "cpu": 4,
            "gpu": 4,  # 4 GPUs per node → 8 total workers
        },
    ),
    runtime="xgboost-distributed",
)
```

Note

For GPU training, add `"device": "cuda"` to the XGBoost `params` dictionary in your training function.

Step 3: Monitor the Training Job

Check the job status and view logs:

```
# Wait for the job to start running.
client.wait_for_job_status(name=job_name, status={"Running"})

# Check the steps (one per worker node).
for step in client.get_job(name=job_name).steps:
    print(f"Step: {step.name}, Status: {step.status}")

# Stream logs from each worker node.
num_nodes = 3
for i in range(num_nodes):
    logs = client.get_job_logs(name=job_name, follow=True, step=f"node-{i}")
    print(f"\n=== Node {i} ===")
    print("\n".join(logs))
```

Step 4: Clean Up

Delete the training job when it is finished:

```
client.delete_job(job_name)
```

Using kubectl with YAML

You can also create `TrainJob` resources directly using `kubectl`.

CPU Training Example

The following YAML creates a distributed XGBoost training job with 4 worker nodes:

```yaml
apiVersion: trainer.kubeflow.org/v1alpha1
kind: TrainJob
metadata:
  name: xgboost-cpu-example
spec:
  runtimeRef:
    name: xgboost-distributed
  trainer:
    image: ghcr.io/kubeflow/trainer/xgboost-runtime:latest
    command:
      - python
      - train.py
    numNodes: 4
    resourcesPerNode:
      requests:
        cpu: "4"
        memory: "8Gi"
```

Apply the manifest:

```
kubectl apply -f xgboost-cpu-trainjob.yaml
```

GPU Training Example

For multi-node GPU training, specify GPU resources via `resourcesPerNode`:

```yaml
apiVersion: trainer.kubeflow.org/v1alpha1
kind: TrainJob
metadata:
  name: xgboost-gpu-example
spec:
  runtimeRef:
    name: xgboost-distributed
  trainer:
    image: ghcr.io/kubeflow/trainer/xgboost-runtime:latest
    command:
      - python
      - train.py
    numNodes: 2
    resourcesPerNode:
      limits:
        nvidia.com/gpu: "4"
```

(continues on next page)

(continued from previous page)

```
      requests:
        cpu: "4"
        memory: "16Gi"
```

With this configuration, the runtime calculates `DMLC_NUM_WORKER = 2 nodes × 4 GPUs = 8`. Each GPU runs one XGBoost worker process.

Monitoring with kubectl

```
# Check TrainJob status.
kubectl get trainjob xgboost-cpu-example

# View logs from a specific worker pod.
kubectl logs xgboost-cpu-example-node-0-0

# Delete the TrainJob.
kubectl delete trainjob xgboost-cpu-example
```

How It Works

This section provides additional implementation details for users who want to understand the runtime plugin internals.

XGBoost Runtime Plugin

The XGBoost runtime is implemented as a Go plugin in the Kubeflow Trainer controller (see `pkg/runtime/framework/plugins/xgboost/` in the Trainer repository). It implements two interfaces:

- `EnforceMLPolicyPlugin`: Injects the `DMLC_*` environment variables (described in *Environment Variables*) and exposes container port `29500`.
- `CustomValidationPlugin`: Rejects any `TrainJob` that manually sets reserved `DMLC_*` environment variables.

Tracker Discovery

Workers discover the `RabitTracker` on rank-0 via a Kubernetes headless service. The `DMLC_TRACKER_URI` is constructed as:

```
<trainjob-name>-node-0-0.<trainjob-name>
```

For example, a `TrainJob` named `myjob` with 4 nodes creates pods:

```
myjob-node-0-0   DMLC_TASK_ID=0   (Tracker + Worker)
myjob-node-0-1   DMLC_TASK_ID=1   (Worker)
myjob-node-0-2   DMLC_TASK_ID=2   (Worker)
myjob-node-0-3   DMLC_TASK_ID=3   (Worker)
```

> **Note**
>
> Starting the tracker is the **user's responsibility**. The runtime injects the environment variables, but the training code on rank-0 must call `RabitTracker(...).start()` before other workers can connect.

Best Practices

This section covers practical tips for getting the most out of distributed XGBoost on Kubernetes.

Use QuantileDMatrix for Memory Efficiency

The default tree method is `hist` (`tree_method="auto"` resolves to `hist`). When using `hist`, prefer *xgboost.QuantileDMatrix* over *xgboost.DMatrix*. `QuantileDMatrix` generates quantilized data directly from input, skipping the intermediate dense representation and significantly reducing memory consumption:

```
# Standard DMatrix - loads data then quantizes (higher peak memory)
dtrain = xgb.DMatrix(X_train, label=y_train)

# QuantileDMatrix - quantizes on-the-fly (lower peak memory)
dtrain = xgb.QuantileDMatrix(X_train, label=y_train)
```

When constructing a validation `QuantileDMatrix`, always pass the training matrix as `ref` so XGBoost reuses the same quantile bins. Omitting `ref` for validation data may lead to inconsistent quantization and degraded model quality:

```
dtrain = xgb.QuantileDMatrix(X_train, label=y_train)
dvalid = xgb.QuantileDMatrix(X_valid, label=y_valid, ref=dtrain)  # correct
```

> **Note**
>
> `QuantileDMatrix` was added in XGBoost 1.7.0. No explicit `tree_method` parameter is needed — the default `auto` already uses `hist`.

Early Stopping

Early stopping is activated by passing `early_stopping_rounds` to *xgboost.train()*. It requires at least one validation set in `evals`. Training stops if the validation metric does not improve for the specified number of consecutive rounds:

```
model = xgb.train(
    params, dtrain,
    num_boost_round=500,
    evals=[(dvalid, "validation")],
    early_stopping_rounds=10,
)
```

Early stopping works correctly in distributed mode — evaluation metrics are already synchronized across workers via the collective protocol.

Important: `xgb.train` with `early_stopping_rounds` returns the **last** model, not the best one. To get the best model, use model slicing:

```
# After training, slice to keep only rounds up to the best iteration.
if hasattr(model, "best_iteration"):
    model = model[: model.best_iteration + 1]
```

Alternatively, use the *xgboost.callback.EarlyStopping* callback directly with `save_best=True` to automatically keep only the best model:

```
from xgboost.callback import EarlyStopping

model = xgb.train(
    params, dtrain,
    num_boost_round=500,
    evals=[(dvalid, "validation")],
    callbacks=[EarlyStopping(rounds=10, save_best=True)],
)
# model now contains only the rounds up to the best iteration
```

When multiple evaluation datasets are provided in `evals`, the **last** entry is used for early stopping. When multiple `eval_metric` values are specified, the **last** metric is used.

Logging in Distributed Mode

In distributed training, `print()` executes on every worker, producing duplicate log lines. To log from a single worker, guard with a rank check:

```
from xgboost import collective as coll

with coll.CommunicatorContext(...):
    # Print only from rank 0.
    if coll.get_rank() == 0:
        print(f"Training complete, best score: {model.best_score}")
```

`xgboost.collective.communicator_print()` is an alternative that routes messages through the tracker rather than stdout. Note that it does **not** filter by rank — any worker that calls it will have its message printed by the tracker. It is primarily used internally (e.g., by `verbose_eval`, which adds its own rank-0 guard via *xgboost.callback.EvaluationMonitor*).

Setting verbose_eval for Production

In distributed Kubernetes jobs, set `verbose_eval` to an integer rather than `True` to reduce log volume:

```
model = xgb.train(
    params, dtrain,
    num_boost_round=500,
    evals=[(dvalid, "validation")],
    verbose_eval=50,  # print every 50 rounds instead of every round
)
```

Checkpointing

XGBoost provides a *xgboost.callback.TrainingCheckPoint* callback that periodically saves model snapshots during training. The callback automatically saves only from rank 0 to avoid multiple workers writing to the same path:

```
from xgboost.callback import TrainingCheckPoint

model = xgb.train(
    params, dtrain,
    num_boost_round=500,
    evals=[(dvalid, "validation")],
    callbacks=[
```

(continues on next page)

(continued from previous page)

```
        TrainingCheckPoint(
            directory="/workspace/checkpoints",
            name="xgb_model",
            interval=50,  # save every 50 rounds
        ),
    ],
)
```

> **Warning**
>
> XGBoost does not handle distributed file systems. The `directory` path must be writable from the rank-0 pod — for example, a Kubernetes PersistentVolumeClaim mounted into the pod.

To resume training from a checkpoint, pass the saved model file via `xgb_model`:

```
model = xgb.train(
    params, dtrain,
    num_boost_round=500,
    xgb_model="/workspace/checkpoints/xgb_model_200.ubj",  # resume from round 200
    evals=[(dvalid, "validation")],
)
```

Data Partitioning

By default, each worker in a distributed XGBoost job holds a different subset of **rows** (horizontal partitioning). This is controlled by the `data_split_mode` parameter (default: `DataSplitMode.ROW`). In this mode, each worker loads its own shard of the data:

```
with coll.CommunicatorContext(...):
    # Each worker loads a different data shard based on its rank.
    rank = coll.get_rank()
    X_shard, y_shard = load_data_shard(rank)
    dtrain = xgb.QuantileDMatrix(X_shard, label=y_shard)
```

Column-wise splitting (`DataSplitMode.COL`) is also supported, where each worker holds a different subset of features. This is typically used for vertical federated learning scenarios and is not the common distributed training pattern.

Rank-Specific Logic

Use *`xgboost.collective.get_rank()`* and *`xgboost.collective.get_world_size()`* for rank-specific operations inside the communicator context:

```
with coll.CommunicatorContext(...):
    if coll.get_rank() == 0:
        model.save_model("/workspace/model.json")
        # Broadcast results to all workers if needed
        results = coll.broadcast(results, root=0)
```

`xgboost.collective.broadcast()` can broadcast any picklable Python object from one worker to all others. This is useful for sharing preprocessed metadata (e.g., label encoders, feature name lists) computed on rank 0.

Common Issues and Edge Cases

Reserved Environment Variables

The runtime plugin rejects any `TrainJob` that manually sets the reserved `DMLC_*` environment variables (`DMLC_TRACKER_URI`, `DMLC_TRACKER_PORT`, `DMLC_TASK_ID`, `DMLC_NUM_WORKER`). If you set any of these in `spec.trainer.env`, the webhook will return a `Forbidden` error:

```
spec.trainer.env[0]: Forbidden: DMLC_TRACKER_URI is reserved for the XGBoost runtime
```

Remove the reserved variables from your `TrainJob` spec and let the runtime inject them automatically.

No Environment Injection When Trainer Is Nil

If the `TrainJob` does not include a `spec.trainer` section, the XGBoost plugin skips environment variable injection entirely. The `DMLC_*` variables are only injected when `spec.trainer` is present and the runtime can locate the `node` container in the pod template. Ensure your `TrainJob` includes the `trainer` field.

Resource Precedence: TrainJob Overrides Runtime

When GPU resources are specified in both the `ClusterTrainingRuntime` template and the `TrainJob.spec.trainer.resourcesPerNode`, the **TrainJob value takes precedence**. This affects the `workersPerNode` calculation:

```
Runtime template: nvidia.com/gpu: 1  →  workersPerNode = 1
TrainJob override: nvidia.com/gpu: 3  →  workersPerNode = 3  (this wins)
```

If neither specifies GPU resources, `workersPerNode` defaults to 1 (CPU mode).

GPU Device Ordinal in Distributed Mode

In distributed training, do **not** use `device="cuda:0"` or any specific GPU ordinal in your XGBoost parameters. GPU device assignment is handled by the Kubernetes device plugin and the distributed framework. Use `device="cuda"` instead:

```
# Correct
params = {"device": "cuda", "tree_method": "hist"}

# Wrong - will raise an error in distributed mode
params = {"device": "cuda:0", "tree_method": "hist"}
```

Data Matrices Must Be Inside CommunicatorContext

Constructing `xgb.DMatrix` or `xgb.QuantileDMatrix` outside the `CommunicatorContext` may appear to work with dense data, but the behavior is undefined. The constructor performs cross-worker synchronization for data shape validation and quantile sketching (needed by `tree_method="hist"`). Always construct data matrices inside the context:

```
# Wrong - data matrix outside context
dtrain = xgb.QuantileDMatrix(X_train, label=y_train)
with coll.CommunicatorContext(...):
    model = xgb.train(params, dtrain, ...)  # Undefined behavior

# Correct - data matrix inside context
with coll.CommunicatorContext(...):
```

(continues on next page)

(continued from previous page)

```
    dtrain = xgb.QuantileDMatrix(X_train, label=y_train)
    model = xgb.train(params, dtrain, ...)
```

Single-Node Defaults

If `numNodes` is not specified in the `TrainJob`, the runtime uses the default from the `ClusterTrainingRuntime` (1 for the `xgboost-distributed` runtime). A single-node job still goes through the full runtime pipeline — the `RabitTracker` is started on rank-0 (which is the only pod), and `DMLC_NUM_WORKER` is set to 1. This is useful for testing your training function locally before scaling up.

CPU Over-Subscription

By default, XGBoost uses all available CPU cores via OpenMP. In a Kubernetes pod, "available cores" is determined by cgroup limits set by the container runtime. If your pod specifies only CPU **requests** (no **limits**), the cgroup may not cap CPU usage, and XGBoost may attempt to use all cores on the node, causing contention with other pods.

To avoid this, either:

- Set `nthread` in your XGBoost parameters to match your CPU request
- Set CPU `limits` (not just `requests`) in `resourcesPerNode` so the container runtime enforces a cgroup ceiling

```
# Setting both requests and limits ensures XGBoost sees the correct core count
resourcesPerNode:
  requests:
    cpu: "4"
  limits:
    cpu: "4"
```

Support

- For issues related to the Kubeflow Trainer XGBoost runtime, open an issue on the Kubeflow Trainer repository.
- For XGBoost-specific questions, see the XGBoost documentation.
- The complete example notebook is available in the Kubeflow Trainer examples.

1.4.13 Distributed XGBoost with Dask

Dask is a parallel computing library built on Python. Dask allows easy management of distributed workers and excels at handling large distributed data science workflows. The implementation in XGBoost originates from dask-xgboost with some extended functionalities and a different interface. The tutorial here focuses on basic usage of dask with CPU tree algorithms. For an overview of GPU based training and internal workings, see A New, Official Dask API for XGBoost.

Note

The integration is not tested with Windows.

Contents

- *Requirements*
- *Overview*
- *Running prediction*
- *Scikit-Learn Estimator Interface*
- *GPU acceleration*
- *Working with other clusters*
- *Threads*
- *Working with asyncio*
- *Evaluation and Early Stopping*
- *Other customization*
- *Hyper-parameter tuning*
- *Learning to Rank*
- *Troubleshooting*
- *IPv6 Support*
- *Logging the evaluation results*
- *Why is the initialization of `DaskDMatrix` so slow and throws weird errors*
- *Reproducible Result*
- *Memory Usage*

Requirements

Dask can be installed using either pip or conda (see the dask installation documentation for more information). For accelerating XGBoost with GPUs, dask-cuda is recommended for creating GPU clusters.

Overview

A dask cluster consists of three different components: a centralized scheduler, one or more workers, and one or more clients which act as the user-facing entry point for submitting tasks to the cluster. When using XGBoost with dask, one needs to call the XGBoost dask interface from the client side. Below is a small example which illustrates basic usage of running XGBoost on a dask cluster:

```
from xgboost import dask as dxgb

import dask.array as da
import dask.distributed

if __name__ == "__main__":
    cluster = dask.distributed.LocalCluster()
    client = dask.distributed.Client(cluster)

    # X and y must be Dask dataframes or arrays
    num_obs = 1e5
    num_features = 20
```

(continues on next page)

(continued from previous page)

```
    X = da.random.random(size=(num_obs, num_features), chunks=(1000, num_features))
    y = da.random.random(size=(num_obs, 1), chunks=(1000, 1))

    dtrain = dxgb.DaskDMatrix(client, X, y)
    # or
    # dtrain = dxgb.DaskQuantileDMatrix(client, X, y)

    output = dxgb.train(
        client,
        {"verbosity": 2, "tree_method": "hist", "objective": "reg:squarederror"},
        dtrain,
        num_boost_round=4,
        evals=[(dtrain, "train")],
    )
```

Here we first create a cluster in single-node mode with `distributed.LocalCluster`, then connect a `distributed.Client` to this cluster, setting up an environment for later computation. Notice that the cluster construction is guarded by `__name__ == "__main__"`, which is necessary otherwise there might be obscure errors.

We then create a `xgboost.dask.DaskDMatrix` object and pass it to `xgboost.dask.train()`, along with some other parameters, much like XGBoost's normal, non-dask interface. Unlike that interface, `data` and `label` must be either `Dask DataFrame` or `Dask Array` instances.

The primary difference with XGBoost's dask interface is we pass our dask client as an additional argument for carrying out the computation. Note that if client is set to `None`, XGBoost will use the default client returned by dask.

There are two sets of APIs implemented in XGBoost. The first set is functional API illustrated in above example. Given the data and a set of parameters, the `train` function returns a model and the computation history as a Python dictionary:

```
{
  "booster": Booster,
  "history": dict,
}
```

For prediction, pass the `output` returned by `train` into `xgboost.dask.predict()`:

```
prediction = dxgb.predict(client, output, dtrain)
# Or equivalently, pass ``output['booster']``:
prediction = dxgb.predict(client, output['booster'], dtrain)
```

Eliminating the construction of DaskDMatrix is also possible, this can make the computation a bit faster when meta information like `base_margin` is not needed:

```
prediction = dxgb.predict(client, output, X)
# Use inplace version.
prediction = dxgb.inplace_predict(client, output, X)
```

Here `prediction` is a dask `Array` object containing predictions from model if input is a `DaskDMatrix` or `da.Array`. When putting dask collection directly into the `predict` function or using `xgboost.dask.inplace_predict()`, the output type depends on input data. See next section for details.

Alternatively, XGBoost also implements the Scikit-Learn interface with `DaskXGBClassifier`, `DaskXGBRegressor`, `DaskXGBRanker` and 2 random forest variances. This wrapper is similar to the single node Scikit-Learn interface in

xgboost, with dask collection as inputs and has an additional `client` attribute. See following sections and *XGBoost Dask Feature Walkthrough* for more examples.

Running prediction

In previous example we used `DaskDMatrix` as input to `predict` function. In practice, it's also possible to call `predict` function directly on dask collections like `Array` and `DataFrame` and might have better prediction performance. When `DataFrame` is used as prediction input, the result is a dask `Series` instead of array. Also, there's in-place predict support on dask interface, which can help reducing both memory usage and prediction time.

```
# dtrain is the DaskDMatrix defined above.
prediction = dxgb.predict(client, booster, dtrain)
```

or equivalently:

```
# where X is a dask DataFrame or dask Array.
prediction = dxgb.predict(client, booster, X)
```

Also for inplace prediction:

```
# where X is a dask DataFrame or dask Array backed by cupy or cuDF.
booster.set_param({"device": "cuda"})
prediction = dxgb.inplace_predict(client, booster, X)
```

When input is `da.Array` object, output is always `da.Array`. However, if the input type is `dd.DataFrame`, output can be `dd.Series`, `dd.DataFrame` or `da.Array`, depending on output shape. For example, when SHAP-based prediction is used, the return value can have 3 or 4 dimensions , in such cases an `Array` is always returned.

The performance of running prediction, either using `predict` or `inplace_predict`, is sensitive to number of blocks. Internally, it's implemented using `da.map_blocks` and `dd.map_partitions`. When number of partitions is large and each of them have only small amount of data, the overhead of calling predict becomes visible. On the other hand, if not using GPU, the number of threads used for prediction on each block matters. Right now, xgboost uses single thread for each partition. If the number of blocks on each workers is smaller than number of cores, then the CPU workers might not be fully utilized.

One simple optimization for running consecutive predictions is using `distributed.Future`:

```
dataset = [X_0, X_1, X_2]
booster_f = client.scatter(booster, broadcast=True)
futures = []
for X in dataset:
    # Here we pass in a future instead of concrete booster
    shap_f = dxgb.predict(client, booster_f, X, pred_contribs=True)
    futures.append(shap_f)

results = client.gather(futures)
```

This is only available on functional interface, as the Scikit-Learn wrapper doesn't know how to maintain a valid future for booster. To obtain the booster object from Scikit-Learn wrapper object:

```
cls = dxgb.DaskXGBClassifier()
cls.fit(X, y)

booster = cls.get_booster()
```

Scikit-Learn Estimator Interface

As mentioned previously, there's another interface that mimics the scikit-learn estimators with higher level of of abstraction. The interface is easier to use compared to the functional interface but with more constraints. It's worth mentioning that, although the interface mimics scikit-learn estimators, it doesn't work with normal scikit-learn utilities like `GridSearchCV` as scikit-learn doesn't understand distributed dask data collection.

```
from distributed import LocalCluster, Client
from xgboost import dask as dxgb


def main(client: Client) -> None:
    X, y = load_data()
    clf = dxgb.DaskXGBClassifier(n_estimators=100, tree_method="hist")
    clf.client = client  # assign the client
    clf.fit(X, y, eval_set=[(X, y)])
    proba = clf.predict_proba(X)


if __name__ == "__main__":
    with LocalCluster() as cluster:
        with Client(cluster) as client:
            main(client)
```

GPU acceleration

For most of the use cases with GPUs, the Dask-CUDA project should be used to create the cluster, which automatically configures the correct device ordinal for worker processes. As a result, users should NOT specify the ordinal (good: `device=cuda`, bad: `device=cuda:1`). See *Example of training with Dask on GPU* and *Use scikit-learn regressor interface with GPU histogram tree method* for worked examples.

Working with other clusters

Using Dask's `LocalCluster` is convenient for getting started quickly on a local machine. Once you're ready to scale your work, though, there are a number of ways to deploy Dask on a distributed cluster. You can use Dask-CUDA, for example, for GPUs and you can use Dask Cloud Provider to deploy Dask clusters in the cloud. See the Dask documentation for a more comprehensive list.

In the example below, a `KubeCluster` is used for deploying Dask on Kubernetes:

```
from dask_kubernetes.operator import KubeCluster  # Need to install the ``dask-
↪kubernetes`` package
from dask_kubernetes.operator.kubecluster.kubecluster import CreateMode

from dask.distributed import Client
from xgboost import dask as dxgb
import dask.array as da


def main():
  '''Connect to a remote kube cluster with GPU nodes and run training on it.'''
    m = 1000
    n = 10
    kWorkers = 2                # assuming you have 2 GPU nodes on that cluster.
    # You need to work out the worker-spec yourself.  See document in dask_kubernetes for
```

(continues on next page)

(continued from previous page)

```
    # its usage.  Here we just want to show that XGBoost works on various clusters.

    # See notes below for why we use pre-allocated cluster.
    with KubeCluster(
        name="xgboost-test",
        image="my-image-name:latest",
        n_workers=kWorkers,
        create_mode=CreateMode.CONNECT_ONLY,
        shutdown_on_close=False,
    ) as cluster:
        with Client(cluster) as client:
            X = da.random.random(size=(m, n), chunks=100)
            y = X.sum(axis=1)

            regressor = dxgb.DaskXGBRegressor(n_estimators=10, missing=0.0)
            regressor.client = client
            regressor.set_params(tree_method='hist', device="cuda")
            regressor.fit(X, y, eval_set=[(X, y)])


if __name__ == '__main__':
    # Launch the kube cluster on somewhere like GKE, then run this as client process.
    # main function will connect to that cluster and start training xgboost model.
    main()
```

Different cluster classes might have subtle differences like network configuration, or specific cluster implementation might contains bugs that we are not aware of. Open an issue if such case is found and there's no documentation on how to resolve it in that cluster implementation.

An interesting aspect of the Kubernetes cluster is that the pods may become available after the Dask workflow has begun, which can cause issues with distributed XGBoost since XGBoost expects the nodes used by input data to remain unchanged during training. To use Kubernetes clusters, it is necessary to wait for all the pods to be online before submitting XGBoost tasks. One can either create a wait function in Python or simply pre-allocate a cluster with k8s tools (like `kubectl`) before running dask workflows. To pre-allocate a cluster, we can first generate the cluster spec using dask kubernetes:

```
import json

from dask_kubernetes.operator import make_cluster_spec

spec = make_cluster_spec(name="xgboost-test", image="my-image-name:latest", n_workers=16)
with open("cluster-spec.json", "w") as fd:
    json.dump(spec, fd, indent=2)
```

```
kubectl apply -f ./cluster-spec.json
```

Check whether the pods are available:

```
kubectl get pods
```

Once all pods have been initialized, the Dask XGBoost workflow can be run, as in the previous example. It is important to ensure that the cluster sets the parameter `create_mode=CreateMode.CONNECT_ONLY` and optionally `shutdown_on_close=False` if you do not want to shut down the cluster after a single job.

Threads

XGBoost has built in support for parallel computation through threads by the setting `nthread` parameter (`n_jobs` for scikit-learn). If these parameters are set, they will override the configuration in Dask. For example:

```
with dask.distributed.LocalCluster(n_workers=7, threads_per_worker=4) as cluster:
```

There are 4 threads allocated for each dask worker. Then by default XGBoost will use 4 threads in each process for training. But if `nthread` parameter is set:

```
output = dxgb.train(
    client,
    {"verbosity": 1, "nthread": 8, "tree_method": "hist"},
    dtrain,
    num_boost_round=4,
    evals=[(dtrain, "train")],
)
```

XGBoost will use 8 threads in each training process.

Working with asyncio

Added in version 1.2.0.

XGBoost's dask interface supports the new `asyncio` in Python and can be integrated into asynchronous workflows. For using dask with asynchronous operations, please refer to this dask example and document in distributed. To use XGBoost's Dask interface asynchronously, the `client` which is passed as an argument for training and prediction must be operating in asynchronous mode by specifying `asynchronous=True` when the `client` is created (example below). All functions (including `DaskDMatrix`) provided by the functional interface will then return coroutines which can then be awaited to retrieve their result. Please note that XGBoost is a compute-bounded application, where parallelism is more important than concurrency. The support for *asyncio* is more about compatibility instead of performance gain.

Functional interface:

```
async with dask.distributed.Client(scheduler_address, asynchronous=True) as client:
    X, y = generate_array()
    m = await dxgb.DaskDMatrix(client, X, y)
    output = await dxgb.train(client, {}, dtrain=m)

    with_m = await dxgb.predict(client, output, m)
    with_X = await dxgb.predict(client, output, X)
    inplace = await dxgb.inplace_predict(client, output, X)

    # Use ``client.compute`` instead of the ``compute`` method from dask collection
    print(await client.compute(with_m))
```

While for the Scikit-Learn interface, trivial methods like `set_params` and accessing class attributes like `evals_result()` do not require `await`. Other methods involving actual computation will return a coroutine and hence require awaiting:

```
async with dask.distributed.Client(scheduler_address, asynchronous=True) as client:
    X, y = generate_array()
    regressor = await dxgb.DaskXGBRegressor(verbosity=1, n_estimators=2)
    regressor.set_params(tree_method='hist')  # trivial method, synchronous operation
    regressor.client = client  #  accessing attribute, synchronous operation
    regressor = await regressor.fit(X, y, eval_set=[(X, y)])
```

(continues on next page)

(continued from previous page)

```
    prediction = await regressor.predict(X)

    # Use `client.compute` instead of the `compute` method from dask collection
    print(await client.compute(prediction))
```

Evaluation and Early Stopping

Added in version 1.3.0.

The Dask interface allows the use of validation sets that are stored in distributed collections (Dask DataFrame or Dask Array). These can be used for evaluation and early stopping.

To enable early stopping, pass one or more validation sets containing `DaskDMatrix` objects.

```
import dask.array as da
from xgboost import dask as dxgb

num_rows = 1e6
num_features = 100
num_partitions = 10
rows_per_chunk = num_rows / num_partitions

data = da.random.random(
    size=(num_rows, num_features),
    chunks=(rows_per_chunk, num_features)
)

labels = da.random.random(
    size=(num_rows, 1),
    chunks=(rows_per_chunk, 1)
)

X_eval = da.random.random(
    size=(num_rows, num_features),
    chunks=(rows_per_chunk, num_features)
)

y_eval = da.random.random(
    size=(num_rows, 1),
    chunks=(rows_per_chunk, 1)
)

dtrain = dxgb.DaskDMatrix(
    client=client,
    data=data,
    label=labels
)

dvalid = dxgb.DaskDMatrix(
    client=client,
    data=X_eval,
    label=y_eval
)
```

(continues on next page)

(continued from previous page)

```
result = dxgb.train(
    client=client,
    params={
        "objective": "reg:squarederror",
    },
    dtrain=dtrain,
    num_boost_round=10,
    evals=[(dvalid, "valid1")],
    early_stopping_rounds=3
)
```

When validation sets are provided to `xgboost.dask.train()` in this way, the model object returned by `xgboost.dask.train()` contains a history of evaluation metrics for each validation set, across all boosting rounds.

```
print(result["history"])
# {'valid1': OrderedDict([('rmse', [0.28857, 0.28858, 0.288592, 0.288598])])}
```

If early stopping is enabled by also passing `early_stopping_rounds`, you can check the best iteration in the returned booster.

```
booster = result["booster"]
print(booster.best_iteration)
best_model = booster[: booster.best_iteration]
```

Other customization

XGBoost dask interface accepts other advanced features found in single node Python interface, including callback functions, custom evaluation metric and objective:

```
def eval_error_metric(predt, dtrain: xgb.DMatrix):
    label = dtrain.get_label()
    r = np.zeros(predt.shape)
    gt = predt > 0.5
    r[gt] = 1 - label[gt]
    le = predt <= 0.5
    r[le] = label[le]
    return 'CustomErr', np.sum(r)

# custom callback
early_stop = xgb.callback.EarlyStopping(
    rounds=early_stopping_rounds,
    metric_name="CustomErr",
    data_name="Train",
    save_best=True,
)

booster = dxgb.train(
    client,
    params={
        "objective": "binary:logistic",
        "eval_metric": ["error", "rmse"],
```

(continues on next page)

(continued from previous page)

```
        "tree_method": "hist",
    },
    dtrain=D_train,
    evals=[(D_train, "Train"), (D_valid, "Valid")],
    feval=eval_error_metric,  # custom evaluation metric
    num_boost_round=100,
    callbacks=[early_stop],
)
```

Hyper-parameter tuning

See https://github.com/coiled/dask-xgboost-nyctaxi for a set of examples of using XGBoost with dask and optuna.

Learning to Rank

Added in version 3.0.0.

> **Note**
>
> Position debiasing is not yet supported.

There are two operation modes in the Dask learning to rank for performance reasons. The difference is whether a distributed global sort is needed. Please see *Distributed Training* for how ranking works with distributed training in general. Below we will discuss some of the Dask-specific features.

First, if you use the `DaskQuantileDMatrix` interface or the `DaskXGBRanker` with `allow_group_split` set to `True`, XGBoost will try to sort and group the samples for each worker based on the query ID. This mode tries to skip the global sort and sort only worker-local data, and hence no inter-worker data shuffle. Please note that even worker-local sort is costly, particularly in terms of memory usage as there's no spilling when `sort_values()` is used, and we need to concatenate the data. XGBoost first checks whether the QID is already sorted before actually performing the sorting operation. One can choose this if the query groups are relatively consecutive, meaning most of the samples within a query group are close to each other and are likely to be resided to the same worker. Don't use this if you have performed a random shuffle on your data.

If the input data is random, then there's no way we can guarantee most of data within the same group being in the same worker. For large query groups, this might not be an issue. But for small query groups, it's possible that each worker gets only one or two samples from their group for all groups, which can lead to disastrous performance. In that case, we can partition the data according to query group, which is the default behavior of the `DaskXGBRanker` unless the `allow_group_split` is set to `True`. This mode performs a sort and a groupby on the entire dataset in addition to an encoding operation for the query group IDs. Along with partition fragmentation, this option can lead to slow performance. See *Learning to rank with the Dask Interface* for a worked example.

Troubleshooting

- In some environments XGBoost might fail to resolve the IP address of the scheduler, a symptom is user receiving `OSError: [Errno 99] Cannot assign requested address` error during training. A quick workaround is to specify the address explicitly. To do that the collective `Config` is used:

  Added in version 3.0.0.

```
import dask
from distributed import Client
from xgboost import dask as dxgb
```

(continues on next page)

(continued from previous page)

```
from xgboost.collective import Config

# let xgboost know the scheduler address
coll_cfg = Config(retry=1, timeout=20, tracker_host_ip="10.23.170.98", tracker_port=0)

with Client(scheduler_file="sched.json") as client:
    reg = dxgb.DaskXGBRegressor(coll_cfg=coll_cfg)
```

- Please note that XGBoost requires a different port than dask. By default, on a unix-like system XGBoost uses the port 0 to find available ports, which may fail if a user is running in a restricted docker environment. In this case, please open additional ports in the container and specify it as in the above snippet.
- If you encounter a NCCL system error while training with GPU enabled, which usually includes the error message *NCCL failure: unhandled system error*, you can specify its network configuration using one of the environment variables listed in the NCCL document such as the `NCCL_SOCKET_IFNAME`. In addition, you can use `NCCL_DEBUG` to obtain debug logs.
- If NCCL fails to initialize in a container environment, it might be caused by limited system shared memory. With docker, one can try the flag: *–shm-size=4g*.
- MIG (Multi-Instance GPU) is not yet supported by NCCL. You will receive an error message that includes *Multiple processes within a communication group …* upon initialization.
- Starting from version 2.1.0, to reduce the size of the binary wheel, the XGBoost package (installed using pip) loads NCCL from the environment instead of bundling it directly. This means that if you encounter an error message like "Failed to load nccl …", it indicates that NCCL is not installed or properly configured in your environment.

  To resolve this issue, you can install NCCL using pip:

  ```
  pip install nvidia-nccl-cu12 # (or with any compatible CUDA version)
  ```

  The default conda installation of XGBoost should not encounter this error. If you are using a customized XGBoost, please make sure one of the followings is true:

  - XGBoost is NOT compiled with the *USE_DLOPEN_NCCL* flag.
  - The *dmlc_nccl_path* parameter is set to full NCCL path when initializing the collective.

  Here are some additional tips for troubleshooting NCCL dependency issues:

  - Check the NCCL installation path and verify that it's installed correctly. We try to find NCCL by using `from nvidia.nccl import lib` in Python when XGBoost is installed using pip.
  - Ensure that you have the correct CUDA version installed. NCCL requires a compatible CUDA version to function properly.
  - If you are not using distributed training with XGBoost and yet see this error, please open an issue on GitHub.
  - If you continue to encounter NCCL dependency issues, please open an issue on GitHub.

IPv6 Support

Added in version 1.7.0.

XGBoost has initial IPv6 support for the dask interface on Linux. Due to most of the cluster support for IPv6 is partial (dual stack instead of IPv6 only), we require additional user configuration similar to *Troubleshooting* to help XGBoost obtain the correct address information:

```python
import dask
from distributed import Client
from xgboost import dask as dxgb
# let xgboost know the scheduler address, use the same bracket format as dask.
with dask.config.set({"xgboost.scheduler_address": "[fd20:b6f:f759:9800::]"}):
    with Client("[fd20:b6f:f759:9800::]") as client:
        reg = dxgb.DaskXGBRegressor(tree_method="hist")
```

When GPU is used, XGBoost employs NCCL as the underlying communication framework, which may require some additional configuration via environment variable depending on the setting of the cluster. Please note that IPv6 support is Unix only.

Logging the evaluation results

By default, the Dask interface prints evaluation results in the scheduler process. This makes it difficult for a user to monitor training progress. We can define custom evaluation monitors using callback functions. See *Example of forwarding evaluation logs to the client* for a worked example on how to forward the logs to the client process. In the example, there are two potential solutions using Dask builtin methods, including `distributed.Client.forward_logging()` and `distributed.print()`. Both of them have some caveats but can be a good starting point for developing more sophisticated methods like writing to files.

Why is the initialization of `DaskDMatrix` so slow and throws weird errors

The dask API in XGBoost requires construction of `DaskDMatrix`. With the Scikit-Learn interface, `DaskDMatrix` is implicitly constructed for all input data during the `fit` or `predict` steps. You might have observed that `DaskDMatrix` construction can take large amounts of time, and sometimes throws errors that don't seem to be relevant to `DaskDMatrix`. Here is a brief explanation for why. By default most dask computations are lazily evaluated, which means that computation is not carried out until you explicitly ask for a result by, for example, calling `compute()`. See the previous link for details in dask, and this wiki for information on the general concept of lazy evaluation. The `DaskDMatrix` constructor forces lazy computations to be evaluated, which means it's where all your earlier computation actually being carried out, including operations like `dd.read_csv()`. To isolate the computation in `DaskDMatrix` from other lazy computations, one can explicitly wait for results of input data before constructing a `DaskDMatrix`. Also dask's diagnostics dashboard can be used to monitor what operations are currently being performed.

Reproducible Result

In a single node mode, we can always expect the same training result between runs as along as the underlying platforms are the same. However, it's difficult to obtain reproducible result in a distributed environment, since the tasks might get different machine allocation or have different amount of available resources during different sessions. There are heuristics and guidelines on how to achieve it but no proven method for guaranteeing such deterministic behavior. The Dask interface in XGBoost tries to provide reproducible result with best effort. This section highlights some known criteria and try to share some insights into the issue.

There are primarily two different tasks for XGBoost the carry out, training and inference. Inference is reproducible given the same software and hardware along with the same run-time configurations. The remaining of this section will focus on training.

Many of the challenges come from the fact that we are using approximation algorithms, The sketching algorithm used to find histogram bins is an approximation to the exact quantile algorithm, the *AUC* metric in a distributed environment is an approximation to the exact *AUC* score, and floating-point number is an approximation to real number. Floating-point is an issue as its summation is not associative, meaning $(a+b)+c$ does not necessarily equal to $a+(b+c)$, even though this property holds true for real number. As a result, whenever we change the order of a summation, the result can differ. This imposes the requirement that, in order to have reproducible output from XGBoost, the entire pipeline needs to be reproducible.

- The software stack is the same for each runs. This goes without saying. XGBoost might generate different outputs between different versions. This is expected as we might change the default value of hyper-parameter, or the parallel strategy that generates different floating-point result. We guarantee the correctness the algorithms, but there are lots of wiggle room for the final output. The situation is similar for many dependencies, for instance, the random number generator might differ from platform to platform.
- The hardware stack is the same for each runs. This includes the number of workers, and the amount of available resources on each worker. XGBoost can generate different results using different number of workers. This is caused by the approximation issue mentioned previously.
- Similar to the hardware constraint, the network topology is also a factor in final output. If we change topology the workers might be ordered differently, leading to different ordering of floating-point operations.
- The random seed used in various place of the pipeline.
- The partitioning of data needs to be reproducible. This is related to the available resources on each worker. Dask might partition the data differently for each run according to its own scheduling policy. For instance, if there are some additional tasks in the cluster while you are running the second training session for XGBoost, some of the workers might have constrained memory and Dask may not push the training data for XGBoost to that worker. This change in data partitioning can lead to different output models. If you are using a shared Dask cluster, then the result is likely to vary between runs.
- The operations performed on dataframes need to be reproducible. There are some operations like *DataFrame.merge* not being deterministic on parallel hardwares like GPU where the order of the index might differ from run to run.

It's expected to have different results when training the model in a distributed environment than training the model using a single node due to aforementioned criteria.

Memory Usage

Here are some practices on reducing memory usage with dask and xgboost.

- In a distributed work flow, data is best loaded by dask collections directly instead of loaded by client process. When loading with client process is unavoidable, use `client.scatter` to distribute data from client process to workers. See [2] for a nice summary.
- When using GPU input, like dataframe loaded by `dask_cudf`, you can try `xgboost.dask.DaskQuantileDMatrix` as a drop in replacement for `DaskDMatrix` to reduce overall memory usage. See *Example of training with Dask on GPU* for an example.
- Use in-place prediction when possible.

References:

1. https://github.com/dask/dask/issues/6833
2. https://stackoverflow.com/questions/45941528/how-to-efficiently-send-a-large-numpy-array-to-the-cluster-with-dask-array

1.4.14 Distributed XGBoost with PySpark

Starting from version 1.7.0, xgboost supports pyspark estimator APIs.

Note

The integration is only tested on Linux distributions.

- *XGBoost PySpark Estimator*
  - *SparkXGBRegressor*
  - *SparkXGBClassifier*
- *XGBoost PySpark GPU support*
  - *Prepare the necessary packages*
  - *Write your PySpark application*
  - *Submit the PySpark application*
  - *Model Persistence*
  - *Accelerate the whole pipeline for xgboost pyspark*
  - *Advanced Usage*

XGBoost PySpark Estimator

SparkXGBRegressor

SparkXGBRegressor is a PySpark ML estimator. It implements the XGBoost classification algorithm based on XGBoost python library, and it can be used in PySpark Pipeline and PySpark ML meta algorithms like CrossValidator/TrainValidationSplit/OneVsRest.

We can create a `SparkXGBRegressor` estimator like:

```
from xgboost.spark import SparkXGBRegressor
xgb_regressor = SparkXGBRegressor(
  features_col="features",
  label_col="label",
  num_workers=2,
)
```

The above snippet creates a spark estimator which can fit on a spark dataset, and return a spark model that can transform a spark dataset and generate dataset with prediction column. We can set almost all of xgboost sklearn estimator parameters as `SparkXGBRegressor` parameters, but some parameter such as `nthread` is forbidden in spark estimator, and some parameters are replaced with pyspark specific parameters such as `weight_col`, `validation_indicator_col`, for details please see `SparkXGBRegressor` doc.

The following code snippet shows how to train a spark xgboost regressor model, first we need to prepare a training dataset as a spark dataframe contains "label" column and "features" column(s), the "features" column(s) must be `pyspark.ml.linalg.Vector` type or spark array type or a list of feature column names.

```
xgb_regressor_model = xgb_regressor.fit(train_spark_dataframe)
```

The following code snippet shows how to predict test data using a spark xgboost regressor model, first we need to prepare a test dataset as a spark dataframe contains "features" and "label" column, the "features" column must be `pyspark.ml.linalg.Vector` type or spark array type.

```
transformed_test_spark_dataframe = xgb_regressor_model.transform(test_spark_dataframe)
```

The above snippet code returns a `transformed_test_spark_dataframe` that contains the input dataset columns and an appended column "prediction" representing the prediction results.

SparkXGBClassifier

SparkXGBClassifier estimator has similar API with SparkXGBRegressor, but it has some pyspark classifier specific params, e.g. raw_prediction_col and probability_col parameters. Correspondingly, by default, SparkXGBClassifierModel transforming test dataset will generate result dataset with 3 new columns:

- "prediction": represents the predicted label.
- "raw_prediction": represents the output margin values.
- "probability": represents the prediction probability on each label.

XGBoost PySpark GPU support

XGBoost PySpark fully supports GPU acceleration. Users are not only able to enable efficient training but also utilize their GPUs for the whole PySpark pipeline including ETL and inference. In below sections, we will walk through an example of training on a Spark standalone cluster with GPU support. To get started, first we need to install some additional packages, then we can set the device parameter to cuda or gpu.

Prepare the necessary packages

Aside from the PySpark and XGBoost modules, we also need the cuDF package for handling Spark dataframe. We recommend using either Conda or Virtualenv to manage python dependencies for PySpark jobs. Please refer to How to Manage Python Dependencies in PySpark for more details on PySpark dependency management.

In short, to create a Python environment that can be sent to a remote cluster using virtualenv and pip:

```
python -m venv xgboost_env
source xgboost_env/bin/activate
pip install pyarrow pandas venv-pack xgboost
# https://docs.rapids.ai/install#pip-install
pip install cudf-cu11 --extra-index-url=https://pypi.nvidia.com
venv-pack -o xgboost_env.tar.gz
```

With Conda:

```
conda create -y -n xgboost_env -c conda-forge conda-pack python=3.9
conda activate xgboost_env
# use conda when the supported version of xgboost (1.7) is released on conda-forge
pip install xgboost
conda install cudf pyarrow pandas -c rapids -c nvidia -c conda-forge
conda pack -f -o xgboost_env.tar.gz
```

Write your PySpark application

Below snippet is a small example for training xgboost model with PySpark. Notice that we are using a list of feature names instead of vector type as the input. The parameter "device=cuda" specifically indicates that the training will be performed on a GPU.

```
from xgboost.spark import SparkXGBRegressor
spark = SparkSession.builder.getOrCreate()

# read data into spark dataframe
train_data_path = "xxxx/train"
train_df = spark.read.parquet(data_path)
```

(continues on next page)

(continued from previous page)

```
test_data_path = "xxxx/test"
test_df = spark.read.parquet(test_data_path)

# assume the label column is named "class"
label_name = "class"

# get a list with feature column names
feature_names = [x.name for x in train_df.schema if x.name != label_name]

# create a xgboost pyspark regressor estimator and set device="cuda"
regressor = SparkXGBRegressor(
  features_col=feature_names,
  label_col=label_name,
  num_workers=2,
  device="cuda",
)

# train and return the model
model = regressor.fit(train_df)

# predict on test data
predict_df = model.transform(test_df)
predict_df.show()
```

Like other distributed interfaces, the `device` parameter doesn't support specifying ordinal as GPUs are managed by Spark instead of XGBoost (good: `device=cuda`, bad: `device=cuda:0`).

Submit the PySpark application

Assuming you have configured the Spark standalone cluster with GPU support. Otherwise, please refer to spark standalone configuration with GPU support.

Starting from XGBoost 2.0.1, stage-level scheduling is automatically enabled. Therefore, if you are using Spark standalone cluster version 3.4.0 or higher, we strongly recommend configuring the `"spark.task.resource.gpu.amount"` as a fractional value. This will enable running multiple tasks in parallel during the ETL phase. An example configuration would be `"spark.task.resource.gpu.amount=1/spark.executor.cores"`. However, if you are using a XGBoost version earlier than 2.0.1 or a Spark standalone cluster version below 3.4.0, you still need to set `"spark.task.resource.gpu.amount"` equal to `"spark.executor.resource.gpu.amount"`.

> **Note**
>
> As of now, the stage-level scheduling feature in XGBoost is limited to the Spark standalone cluster mode. However, we have plans to expand its compatibility to YARN and Kubernetes once Spark 3.5.1 is officially released.

```
export PYSPARK_DRIVER_PYTHON=python
export PYSPARK_PYTHON=./environment/bin/python

spark-submit \
  --master spark://<master-ip>:7077 \
  --conf spark.executor.cores=12 \
  --conf spark.task.cpus=1 \
```

(continues on next page)

(continued from previous page)

```
  --conf spark.executor.resource.gpu.amount=1 \
  --conf spark.task.resource.gpu.amount=0.08 \
  --archives xgboost_env.tar.gz#environment \
  xgboost_app.py
```

The above command submits the xgboost pyspark application with the python environment created by pip or conda, specifying a request for 1 GPU and 12 CPUs per executor. So you can see, a total of 12 tasks per executor will be executed concurrently during the ETL phase.

Model Persistence

Similar to standard PySpark ml estimators, one can persist and reuse the model with `save` and `load` methods:

```
regressor = SparkXGBRegressor()
model = regressor.fit(train_df)
# save the model
model.save("/tmp/xgboost-pyspark-model")
# load the model
model2 = SparkXGBRankerModel.load("/tmp/xgboost-pyspark-model")
```

To export the underlying booster model used by XGBoost:

```
regressor = SparkXGBRegressor()
model = regressor.fit(train_df)
# the same booster object returned by xgboost.train
booster: xgb.Booster = model.get_booster()
booster.predict(...)
booster.save_model("model.json") # or model.ubj, depending on your choice of format.
```

This booster is not only shared by other Python interfaces but also used by all the XGBoost bindings including the C, Java, and the R package. Lastly, one can extract the booster file directly from a saved spark estimator without going through the getter:

```
import xgboost as xgb
bst = xgb.Booster()
# Loading the model saved in previous snippet
bst.load_model("/tmp/xgboost-pyspark-model/model/part-00000")
```

Accelerate the whole pipeline for xgboost pyspark

With RAPIDS Accelerator for Apache Spark, you can leverage GPUs to accelerate the whole pipeline (ETL, Train, Transform) for xgboost pyspark without the need for any code modifications. Likewise, you have the option to configure the `"spark.task.resource.gpu.amount"` setting as a fractional value, enabling a higher number of tasks to be executed in parallel during the ETL phase. please refer to *Submit the PySpark application* for more details.

An example submit command is shown below with additional spark configurations and dependencies:

```
export PYSPARK_DRIVER_PYTHON=python
export PYSPARK_PYTHON=./environment/bin/python

spark-submit \
  --master spark://<master-ip>:7077 \
  --conf spark.executor.cores=12 \
```

(continues on next page)

(continued from previous page)

```
  --conf spark.task.cpus=1 \
  --conf spark.executor.resource.gpu.amount=1 \
  --conf spark.task.resource.gpu.amount=0.08 \
  --packages com.nvidia:rapids-4-spark_2.12:24.04.1 \
  --conf spark.plugins=com.nvidia.spark.SQLPlugin \
  --conf spark.sql.execution.arrow.maxRecordsPerBatch=1000000 \
  --archives xgboost_env.tar.gz#environment \
  xgboost_app.py
```

When rapids plugin is enabled, both of the JVM rapids plugin and the cuDF Python package are required. More configuration options can be found in the RAPIDS link above along with details on the plugin.

Advanced Usage

XGBoost needs to repartition the input dataset to the num_workers to ensure there will be num_workers training tasks running at the same time. However, repartition is a costly operation.

If there is a scenario where reading the data from source and directly fitting it to XGBoost without introducing the shuffle stage, users can avoid the need for repartitioning by setting the Spark configuration parameters `spark.sql.files.maxPartitionNum` and `spark.sql.files.minPartitionNum` to num_workers. This tells Spark to automatically partition the dataset into the desired number of partitions.

However, if the input dataset is skewed (i.e. the data is not evenly distributed), setting the partition number to num_workers may not be efficient. In this case, users can set the `force_repartition=true` option to explicitly force XGBoost to repartition the dataset, even if the partition number is already equal to num_workers. This ensures the data is evenly distributed across the workers.

1.4.15 Distributed XGBoost with Ray

Ray is a general purpose distributed execution framework. Ray can be used to scale computations from a single node to a cluster of hundreds of nodes without changing any code.

The Python bindings of Ray come with a collection of well maintained machine learning libraries for hyperparameter optimization and model serving.

The XGBoost-Ray project provides an interface to run XGBoost training and prediction jobs on a Ray cluster. It allows to utilize distributed data representations, such as Modin dataframes, as well as distributed loading from cloud storage (e.g. Parquet files).

XGBoost-Ray integrates well with hyperparameter optimization library Ray Tune, and implements advanced fault tolerance handling mechanisms. With Ray you can scale your training jobs to hundreds of nodes just by adding new nodes to a cluster. You can also use Ray to leverage multi GPU XGBoost training.

Installing and starting Ray

Ray can be installed from PyPI like this:

```
pip install ray
```

If you're using Ray on a single machine, you don't need to do anything else - XGBoost-Ray will automatically start a local Ray cluster when used.

If you want to use Ray on a cluster, you can use the Ray cluster launcher.

Installing XGBoost-Ray

XGBoost-Ray is also available via PyPI:

```
pip install xgboost_ray
```

This will install all dependencies needed to run XGBoost on Ray, including Ray itself if it hasn't been installed before.

Using XGBoost-Ray for training and prediction

XGBoost-Ray uses the same API as core XGBoost. There are only two differences:

1. Instead of using a `xgboost.DMatrix`, you'll use a `xgboost_ray.RayDMatrix` object
2. There is an additional `xgboost_ray.RayParams` parameter that you can use to configure distributed training.

Simple training example

To run this simple example, you'll need to install scikit-learn (with `pip install sklearn`).

In this example, we will load the breast cancer dataset and train a binary classifier using two actors.

```
from xgboost_ray import RayDMatrix, RayParams, train
from sklearn.datasets import load_breast_cancer

train_x, train_y = load_breast_cancer(return_X_y=True)
train_set = RayDMatrix(train_x, train_y)

evals_result = {}
bst = train(
    {
        "objective": "binary:logistic",
        "eval_metric": ["logloss", "error"],
    },
    train_set,
    evals_result=evals_result,
    evals=[(train_set, "train")],
    verbose_eval=False,
    ray_params=RayParams(num_actors=2, cpus_per_actor=1))

bst.save_model("model.xgb")
print("Final training error: {:.4f}".format(
    evals_result["train"]["error"][-1]))
```

The only differences compared to the non-distributed API are the import statement (`xgboost_ray` instead of `xgboost`), using the `RayDMatrix` instead of the `DMatrix`, and passing a `xgboost_ray.RayParams` object.

The return object is a regular `xgboost.Booster` instance.

Simple prediction example

```
from xgboost_ray import RayDMatrix, RayParams, predict
from sklearn.datasets import load_breast_cancer
import xgboost as xgb

data, labels = load_breast_cancer(return_X_y=True)
```

(continues on next page)

(continued from previous page)

```
dpred = RayDMatrix(data, labels)

bst = xgb.Booster(model_file="model.xgb")
pred_ray = predict(bst, dpred, ray_params=RayParams(num_actors=2))

print(pred_ray)
```

In this example, the data will be split across two actors. The result array will integrate this data in the correct order.

The RayParams object

The RayParams object is used to configure various settings relating to the distributed training.

Multi GPU training

Ray automatically detects GPUs on cluster nodes. In order to start training on multiple GPUs, all you have to do is to set the gpus_per_actor parameter of the RayParams object, as well as the num_actors parameter for multiple GPUs:

```
ray_params = RayParams(
    num_actors=4,
    gpus_per_actor=1,
)
```

This will train on four GPUs in parallel.

Note that it usually does not make sense to allocate more than one GPU per actor, as XGBoost relies on distributed libraries such as Dask or Ray to utilize multi GPU training.

Setting the number of CPUs per actor

XGBoost natively utilizes multi threading to speed up computations. Thus if your are training on CPUs only, there is likely no benefit in using more than one actor per node. In that case, assuming you have a cluster of homogeneous nodes, set the number of CPUs per actor to the number of CPUs available on each node, and the number of actors to the number of nodes.

If you are using multi GPU training on a single node, divide the number of available CPUs evenly across all actors. For instance, if you have 16 CPUs and 4 GPUs available, each actor should access 1 GPU and 4 CPUs.

If you are using a cluster of heterogeneous nodes (with different amounts of CPUs), you might just want to use the greatest common divisor for the number of CPUs per actor. E.g. if you have a cluster of three nodes with 4, 8, and 12 CPUs, respectively, you'd start 6 actors with 4 CPUs each for maximum CPU utilization.

Fault tolerance

XGBoost-Ray supports two fault tolerance modes. In **non-elastic training**, whenever a training actor dies (e.g. because the node goes down), the training job will stop, XGBoost-Ray will wait for the actor (or its resources) to become available again (this might be on a different node), and then continue training once all actors are back.

In **elastic-training**, whenever a training actor dies, the rest of the actors continue training without the dead actor. If the actor comes back, it will be re-integrated into training again.

Please note that in elastic-training this means that you will train on fewer data for some time. The benefit is that you can continue training even if a node goes away for the remainder of the training run, and don't have to wait until it is back up again. In practice this usually leads to a very minor decrease in accuracy but a much shorter training time compared to non-elastic training.

Both training modes can be configured using the respective `xgboost_ray.RayParams` parameters.

Hyperparameter optimization

XGBoost-Ray integrates well with hyperparameter optimization framework Ray Tune. Ray Tune uses Ray to start multiple distributed trials with different hyperparameter configurations. If used with XGBoost-Ray, these trials will then start their own distributed training jobs.

XGBoost-Ray automatically reports evaluation results back to Ray Tune. There's only a few things you need to do:

1. Put your XGBoost-Ray training call into a function accepting parameter configurations (`train_model` in the example below).
2. Create a `xgboost_ray.RayParams` object (`ray_params` in the example below).
3. Define the parameter search space (`config` dict in the example below).
4. **Call `tune.run()`:**
    - The `metric` parameter should contain the metric you'd like to optimize. Usually this consists of the prefix passed to the `evals` argument of `xgboost_ray.train()`, and an `eval_metric` passed in the XGBoost parameters (`train-error` in the example below).
    - The `mode` should either be `min` or `max`, depending on whether you'd like to minimize or maximize the metric
    - The `resources_per_actor` should be set using `ray_params.get_tune_resources()`. This will make sure that each trial has the necessary resources available to start their distributed training jobs.

```python
from xgboost_ray import RayDMatrix, RayParams, train
from sklearn.datasets import load_breast_cancer

num_actors = 4
num_cpus_per_actor = 1

ray_params = RayParams(
    num_actors=num_actors, cpus_per_actor=num_cpus_per_actor)

def train_model(config):
    train_x, train_y = load_breast_cancer(return_X_y=True)
    train_set = RayDMatrix(train_x, train_y)

    evals_result = {}
    bst = train(
        params=config,
        dtrain=train_set,
        evals_result=evals_result,
        evals=[(train_set, "train")],
        verbose_eval=False,
        ray_params=ray_params)
    bst.save_model("model.xgb")

from ray import tune

# Specify the hyperparameter search space.
config = {
    "tree_method": "approx",
    "objective": "binary:logistic",
```

(continues on next page)

(continued from previous page)

```
    "eval_metric": ["logloss", "error"],
    "eta": tune.loguniform(1e-4, 1e-1),
    "subsample": tune.uniform(0.5, 1.0),
    "max_depth": tune.randint(1, 9)
}

# Make sure to use the `get_tune_resources` method to set the `resources_per_trial`
analysis = tune.run(
    train_model,
    config=config,
    metric="train-error",
    mode="min",
    num_samples=4,
    resources_per_trial=ray_params.get_tune_resources())
print("Best hyperparameters", analysis.best_config)
```

Ray Tune supports various search algorithms and libraries (e.g. BayesOpt, Tree-Parzen estimators), smart schedulers like successive halving, and other features. Please refer to the Ray Tune documentation for more information.

Additional resources

- XGBoost-Ray repository
- XGBoost-Ray documentation
- Ray core documentation
- Ray Tune documentation

1.4.16 Using XGBoost External Memory Version

Contents

- *Overview*
- *Data Iterator*
- *GPU Version (GPU Hist tree method)*
  - *Using CUDA Async Pool*
  - *Using RMM Pool*
  - *NVLink-C2C*
  - *Adaptive Cache*
  - *Non-Uniform Memory Access (NUMA)*
- *Distributed Training*
- *Best Practices*
- *Remarks*
- *Compared to the QuantileDMatrix*
- *Brief History*

Overview

When working with large datasets, training XGBoost models can be challenging as the entire dataset needs to be loaded into the main memory. This can be costly and sometimes infeasible.

External memory training is sometimes called out-of-core training. It refers to the capability that XGBoost can optionally cache data in a location external to the main processor, be it CPU or GPU. XGBoost doesn't support network file systems by itself. As a result, for CPU, the external memory usually refers to a harddrive. And for GPU, it refers to either the host memory or a harddrive.

Users can define a custom iterator to load data in chunks for running XGBoost algorithms. External memory can be used for training and prediction, but training is the primary use case and it will be our focus in this tutorial. For prediction and evaluation, users can iterate through the data themselves, whereas training requires the entire dataset to be loaded into the memory. During model training, XGBoost fetches the cache in batches to construct the decision trees, hence avoiding loading the entire dataset into the main memory and achieve better vertical scaling (scaling within the same node).

Significant progress was made in the 3.0 release for the GPU implementation. We will introduce the difference between CPU and GPU in the following sections.

Note

Training on data from external memory is not supported by the `exact` tree method. We recommend using the default `hist` tree method for performance reasons.

Note

The feature is considered experimental but ready for public testing in 3.0. Vector-leaf is not yet supported.

The external memory support has undergone multiple development iterations. See below sections for a brief history.

Data Iterator

To start using the external memory, users need define a data iterator. The data iterator interface was added to the Python and C interfaces in 1.5, and to the R interface in 3.0.0. Like the *QuantileDMatrix* with *DataIter*, XGBoost loads data batch-by-batch using the custom iterator supplied by the user. However, unlike the *QuantileDMatrix*, external memory does not concatenate the batches. Instead, it caches all batches in the external memory and fetch them on-demand. Go to the end of the document to see a comparison between *QuantileDMatrix* and the external memory version of *ExtMemQuantileDMatrix*.

Some examples are in the `demo` directory for a quick start. To enable external memory training, the custom data iterator needs to have two class methods: `next` and `reset`.

```
import os
from typing import List, Callable

import numpy as np
import xgboost

class Iterator(xgboost.DataIter):
    """A custom iterator for loading files in batches."""

    def __init__(
        self, device: Literal["cpu", "cuda"], file_paths: List[Tuple[str, str]]
```

(continues on next page)

(continued from previous page)

```
    ) -> None:
        self.device = device

        self._file_paths = file_paths
        self._it = 0
        # XGBoost will generate some cache files under the current directory with the
        # prefix "cache"
        super().__init__(cache_prefix=os.path.join(".", "cache"))

    def load_file(self) -> Tuple[np.ndarray, np.ndarray]:
        """Load a single batch of data."""
        X_path, y_path = self._file_paths[self._it]
        # When the `ExtMemQuantileDMatrix` is used, the device must match. GPU cannot
        # consume CPU input data and vice-versa.
        if self.device == "cpu":
            X = np.load(X_path)
            y = np.load(y_path)
        else:
            import cupy as cp

            X = cp.load(X_path)
            y = cp.load(y_path)

        assert X.shape[0] == y.shape[0]
        return X, y

    def next(self, input_data: Callable) -> bool:
        """Advance the iterator by 1 step and pass the data to XGBoost.  This function
        is called by XGBoost during the construction of ``DMatrix``

        """
        if self._it == len(self._file_paths):
            # return False to let XGBoost know this is the end of iteration
            return False

        # input_data is a keyword-only function passed in by XGBoost and has the similar
        # signature to the ``DMatrix`` constructor.
        X, y = self.load_file()
        input_data(data=X, label=y)
        self._it += 1
        return True

    def reset(self) -> None:
        """Reset the iterator to its beginning"""
        self._it = 0
```

After defining the iterator, we can to pass it into the *DMatrix* or the *ExtMemQuantileDMatrix* constructor:

```
it = Iterator(device="cpu", file_paths=["file_0.npy", "file_1.npy", "file_2.npy"])

# Use the ``ExtMemQuantileDMatrix`` for the hist tree method, recommended.
Xy = xgboost.ExtMemQuantileDMatrix(it)
```

(continues on next page)

(continued from previous page)

```
booster = xgboost.train({"tree_method": "hist"}, Xy)

# The ``approx`` tree method also works, but with lower performance and cannot be used
# with the quantile DMatrix.
Xy = xgboost.DMatrix(it)
booster = xgboost.train({"tree_method": "approx"}, Xy)
```

The above snippet is a simplified version of *Experimental support for external memory*. For an example in C, please see `demo/c-api/external-memory/`. The iterator is the common interface for using external memory with XGBoost, you can pass the resulting `DMatrix` object for training, prediction, and evaluation.

The `ExtMemQuantileDMatrix` is an external memory version of the `QuantileDMatrix`. These two classes are specifically designed for the `hist` tree method for reduced memory usage and data loading overhead. See respective references for more info.

It is important to set the batch size based on the memory available. A good starting point for CPU is to set the batch size to 10GB per batch if you have 64GB of memory. It is *not* recommended to set small batch sizes like 32 samples per batch, as this can severely hurt performance in gradient boosting. See below sections for information about the GPU version and other best practices.

GPU Version (GPU Hist tree method)

External memory is supported by GPU algorithms (i.e., when `device` is set to `cuda`). Starting with 3.0, the default GPU implementation is similar to what the CPU version does. It also supports the use of `ExtMemQuantileDMatrix` when the `hist` tree method is employed (default). For a GPU device, the main memory is the device memory, whereas the external memory can be either a disk or the CPU memory. XGBoost stages the cache on CPU memory by default. Users can change the backing storage to disk by specifying the `on_host` parameter in the `DataIter`. However, using the disk is not recommended as it's likely to make the GPU slower than the CPU. The option is here for experimentation purposes only. In addition, `ExtMemQuantileDMatrix` parameters `min_cache_page_bytes`, and `max_quantile_batches` can help control the data placement and memory usage.

Inputs to the `ExtMemQuantileDMatrix` (through the iterator) must be on the GPU. It's crucial to use an asynchronous memory pool for all memory allocations when training with external memory. XGBoost relies on the asynchronous memory pool to reduce the overhead of data fetching. There are two options for setting up the memory pool:

- **CUDA Async Pool**: Uses the CUDA driver's built-in async memory pool. This option doesn't require any additional dependencies. It's the same as using the *CudaAsyncMemoryResource* from RMM (see below).
- **RMM Pool**: Uses RAPIDS Memory Manager (RMM) with an asynchronous memory resource. This option requires RMM to be installed and XGBoost to be built with RMM support.

Choose the one that best fits your use case.

Using CUDA Async Pool

The CUDA async pool uses the driver's default memory pool with a configured release threshold. See *Global Configuration* for the parameter *use_cuda_async_pool*.

Added in version 3.2.0.

Warning

This is an experimental feature and is subject to change without notice. Windows is not supported yet.

```
import cupy as cp
import cuda.bindings.driver as driver
import cuda.bindings.runtime as cudart
from cupy.cuda import MemoryAsyncPool

# Get the default memory pool and configure the release threshold
status, dft_pool = cudart.cudaDeviceGetDefaultMemPool(0)
# Set the release threshold to 90% of total device memory
status, free, total = cudart.cudaMemGetInfo()
v = driver.cuuint64_t(int(total * 0.9))
cudart.cudaMemPoolSetAttribute(
    dft_pool,
    cudart.cudaMemPoolAttr.cudaMemPoolAttrReleaseThreshold,
    v,
)
# Set the allocator for cupy as well.
cp.cuda.set_allocator(MemoryAsyncPool().malloc)

# Make sure XGBoost is using the CUDA async pool for all allocations.
with xgboost.config_context(use_cuda_async_pool=True):
    # Construct the iterators for ExtMemQuantileDMatrix
    # ...
    # Build the ExtMemQuantileDMatrix and start training
    Xy_train = xgboost.ExtMemQuantileDMatrix(it_train, max_bin=n_bins)
    # Use the training DMatrix as a reference
    Xy_valid = xgboost.ExtMemQuantileDMatrix(it_valid, max_bin=n_bins, ref=Xy_train)
    booster = xgboost.train(
        {
            "tree_method": "hist",
            "max_bin": n_bins,
            "device": device,
        },
        Xy_train,
        num_boost_round=n_rounds,
        evals=[(Xy_train, "Train"), (Xy_valid, "Valid")]
    )
```

Using RMM Pool

Alternatively, you can use RMM with an asynchronous memory resource. If XGBoost is not built with RMM support, a warning will be raised:

```
import cupy as cp
import rmm
from rmm.allocators.cupy import rmm_cupy_allocator

# We use the pool memory resource here for simplicity, you can also try the
# `ArenaMemoryResource` for improved memory fragmentation handling.
```

(continues on next page)

(continued from previous page)

```
mr = rmm.mr.PoolMemoryResource(rmm.mr.CudaAsyncMemoryResource())
rmm.mr.set_current_device_resource(mr)
# Set the allocator for cupy as well.
cp.cuda.set_allocator(rmm_cupy_allocator)

# Make sure XGBoost is using RMM for all allocations.
with xgboost.config_context(use_rmm=True):
    # Construct the iterators for ExtMemQuantileDMatrix
    # ...
    # Build the ExtMemQuantileDMatrix and start training
    Xy_train = xgboost.ExtMemQuantileDMatrix(it_train, max_bin=n_bins)
    # Use the training DMatrix as a reference
    Xy_valid = xgboost.ExtMemQuantileDMatrix(it_valid, max_bin=n_bins, ref=Xy_train)
    booster = xgboost.train(
        {
            "tree_method": "hist",
            "max_depth": 6,
            "max_bin": n_bins,
            "device": device,
        },
        Xy_train,
        num_boost_round=n_rounds,
        evals=[(Xy_train, "Train"), (Xy_valid, "Valid")]
    )
```

In addition, the open source NVIDIA Linux driver is required for `Heterogeneous memory management (HMM)` support. Usually, users need not to change *ExtMemQuantileDMatrix* parameters like `min_cache_page_bytes`, they are automatically configured based on the device and don't change model accuracy. However, the `max_quantile_batches` can be useful if *ExtMemQuantileDMatrix* is running out of device memory during construction, see *QuantileDMatrix* and the following sections for more info. Currently, we focus on devices with `NVLink-C2C` support for GPU-based external memory support.

NVLink-C2C

The newer NVIDIA platforms like Grace-Hopper use NVLink-C2C, which facilitates a fast interconnect between the CPU and the GPU. With the host memory serving as the data cache, XGBoost can retrieve data with significantly lower overhead. When the input data is dense, there's minimal to no performance loss for training, except for the initial construction of the *ExtMemQuantileDMatrix*. The initial construction iterates through the input data twice, as a result, the most significant overhead compared to in-core training is one additional data read when the data is dense. Please note that there are multiple variants of the platform and they come with different C2C bandwidths. During initial development of the feature, we used the LPDDR5 480G version, which has about 350GB/s bandwidth for host to device transfer. When choosing the variant for training XGBoost models, one should pay extra attention to the C2C bandwidth.

Here we provide a simple example as a starting point for training with external memory. We used this example for one of the benchmarks. To train a model with *2 ^ 29* 32-bit floating point samples, *512* features (total 1TB) on a GH200 (a H200 GPU connected to a Grace CPU by a chip-to-chip link) system. One can start with: - Evenly divide the data into 128 batches with 8GB per batch. - Define a custom iterator as previously described. - Set the *max_quantile_batches* parameter of the *ExtMemQuantileDMatrix* to 32 (256GB per sub-stream for quantization). Load the data. - Start training with `device=cuda`.

To run experiments on these platforms, the open source NVIDIA Linux driver with version `>=565.47` is required, it should come with CTK 12.7 and later versions. Lastly, there's a known issue with Linux 6.11 that can lead to CUDA host memory allocation failure with an `invalid argument` error.

Adaptive Cache

Starting with 3.1, XGBoost introduces an adaptive cache for GPU-based external memory training. The feature helps split the data cache into a host cache and a device cache. By keeping a portion of the cache on the GPU, we can reduce the amount of data transfer during training when there's sufficient amount of GPU memory. The feature can be controlled by the `cache_host_ratio` parameter of the *xgboost.ExtMemQuantileDMatrix*. Unless explicitly specified, the ratio is automatically estimated based on device memory size and the size of the dataset.

However, this parameter increases memory fragmentation as XGBoost needs large memory pages with irregular sizes. As a result, you might see out of memory error after the construction of the `DMatrix` but before the actual training begins.

For reference, we tested the adaptive cache with a 128GB (512 features) dense 32bit floating dataset using a NVIDIA A6000 GPU, which comes with 48GB device memory. The `cache_host_ratio` was estimated to be about 0.3, meaning about 30 percent of the quantized cache was on the host and rest of 70 percent was actually in-core. Given this ratio, the overhead is minimal. However, the estimated ratio increases as the data size grows.

Non-Uniform Memory Access (NUMA)

On multi-socket systems, NUMA helps optimize data access by prioritizing memory that is local to each socket. On these systems, it's essential to set the correct affinity to reduce the overhead of cross-socket data access. Since the out of core training stages the data cache on the host and trains the model using a GPU, the training performance is particularly sensitive to the data read bandwidth. To provide some context, on a GB200 machine, accessing the wrong NUMA node from a GPU can reduce the C2C bandwidth by half. Even if you are not using distributed training, you should still pay attention to NUMA control since there's no guarantee that your process will have the correct configuration.

To configure the NUMA binding from command line on Linux, one can use the `numactl` or the `hwloc-bind`:

```
numactl --membind=${NODEID} --cpunodebind=${NODEID} ./myapp
```

To obtain the node ID, you can check the machine topology via `nvidia-smi`:

```
nvidia-smi topo -m
```

The column `NUMA Affinity` lists the NUMA node ID for each GPU. In the example output shown below, the *GPU0* is associated with the *0* node ID:

```
        GPU0    GPU1    NIC0    NIC1    NIC2    NIC3    CPU Affinity    NUMA Affinity   ␣
↪GPU NUMA ID
GPU0     X      NV18    NODE    NODE    NODE    SYS     0-71            0               2
GPU1    NV18     X      SYS     SYS     SYS     NODE    72-143          1               ␣
↪10
NIC0    NODE    SYS      X      PIX     NODE    SYS
NIC1    NODE    SYS     PIX      X      NODE    SYS
NIC2    NODE    SYS     NODE    NODE     X      SYS
NIC3    SYS     NODE    SYS     SYS     SYS      X
```

Alternatively, one can also use the `hwloc` command line interface, please make sure the strict flag is used:

```
hwloc-bind --strict --membind node:${NODEID} --cpubind node:${NODEID} ./myapp
```

Both projects provide a programming interface for configuring NUMA bindings within applications. See *Experimental support for distributed training with external memory* for a complete example of using `pyhwloc` in a distributed training setting.

Distributed Training

Distributed training is similar to in-core learning, but the work for framework integration is still on-going. See *Experimental support for distributed training with external memory* for an example for using the communicator to build a simple pipeline. Since users can define their custom data loader, it's unlikely that existing distributed frameworks interface in XGBoost can meet all the use cases, the example can be a starting point for users who have custom infrastructure.

Best Practices

In previous sections, we demonstrated how to train a tree-based model with data residing on an external memory. In addition, we made some recommendations for batch size and NUMA. Here are some other configurations we find useful. The external memory feature involves iterating through data batches stored in a cache during tree construction. For optimal performance, we recommend using the `grow_policy=depthwise` setting, which allows XGBoost to build an entire layer of tree nodes with only a few batch iterations. Conversely, using the `lossguide` policy requires XGBoost to iterate over the data set for each tree node, resulting in significantly slower performance (tree size is exponential to the depth).

In addition, the `hist` tree method should be preferred over the `approx` tree method as the former doesn't recreate the histogram bins for every iteration. Creating the histogram bins requires loading the raw input data, which is prohibitively expensive. The `ExtMemQuantileDMatrix` designed for the `hist` tree method can speed up the initial data construction and the evaluation significantly for external memory.

Since the external memory implementation focuses on training where XGBoost needs to access the entire dataset, only the `X` is divided into batches while everything else is concatenated. As a result, it's recommended for users to define their own management code to iterate through the data for inference, especially for SHAP value computation. The size of SHAP matrix can be larger than the feature matrix `X`, making external memory in XGBoost less effective.

When external memory is used, the performance of CPU training is limited by disk IO (input/output) speed. This means that the disk IO speed primarily determines the training speed. Similarly, PCIe bandwidth limits the GPU performance, assuming the CPU memory is used as a cache and address translation services (ATS) is unavailable. During development, we observed that typical data transfer in XGBoost with PCIe4x16 has about 24GB/s bandwidth and about 42GB/s with PCIe5, which is significantly lower than the GPU processing performance. Whereas with a C2C-enabled machine, the performance of data transfer and processing in training are close to each other.

Running inference is much less computation-intensive than training and, hence, much faster. As a result, the performance bottleneck of inference is back to data transfer. For GPU, the time it takes to read the data from host to device completely determines the time it takes to run inference, even if a C2C link is available.

```
Xy_train = xgboost.ExtMemQuantileDMatrix(it_train, max_bin=n_bins)
Xy_valid = xgboost.ExtMemQuantileDMatrix(it_valid, max_bin=n_bins, ref=Xy_train)
```

In addition, since the GPU implementation relies on asynchronous memory pool, memory fragmentation can occur regardless of whether you use the CUDA async pool or RMM. You might want to start the training with a fresh pool instead of starting training right after the ETL process. If you run into out-of-memory errors and you are convinced that the pool is not full yet (pool memory usage can be profiled with `nsight-system`), consider using the `ArenaMemoryResource` memory resource with RMM, or using the CUDA asynchronous pool with the latest NVIDIA kernel driver.

During CPU benchmarking, we used an NVMe connected to a PCIe-4 slot. Other types of storage can be too slow for practical usage. However, your system will likely perform some caching to reduce the overhead of the file read. See the following sections for remarks.

Remarks

When using external memory with XGBoost, data is divided into smaller chunks so that only a fraction of it needs to be stored in memory at any given time. It's important to note that this method only applies to the predictor data (`X`), while other data, like labels and internal runtime structures are concatenated. This means that memory reduction is

most effective when dealing with wide datasets where `X` is significantly larger in size compared to other data like `y`, while it has little impact on slim datasets.

As one might expect, fetching data on demand puts significant pressure on the storage device. Today's computing devices can process way more data than storage devices can read in a single unit of time. The ratio is in the order of magnitudes. A GPU is capable of processing hundreds of Gigabytes of floating-point data in a split second. On the other hand, a four-lane NVMe storage connected to a PCIe-4 slot usually has about 6GB/s of data transfer rate. As a result, the training is likely to be severely bounded by your storage device. Before adopting the external memory solution, some back-of-envelop calculations might help you determine its viability. For instance, if your NVMe drive can transfer 4GB (a reasonably practical number) of data per second, and you have a 100GB of data in a compressed XGBoost cache (corresponding to a dense float32 numpy array with 200GB, give or take). A tree with depth 8 needs at least 16 iterations through the data when the parameter is optimal. You need about 14 minutes to train a single tree without accounting for some other overheads and assume the computation overlaps with the IO. If your dataset happens to have a TB-level size, you might need thousands of trees to get a generalized model. These calculations can help you get an estimate of the expected training time.

However, sometimes, we can ameliorate this limitation. One should also consider that the OS (mainly talking about the Linux kernel) can usually cache the data on host memory. It only evicts pages when new data comes in and there's no room left. In practice, at least some portion of the data can persist in the host memory throughout the entire training session. We are aware of this cache when optimizing the external memory fetcher. The compressed cache is usually smaller than the raw input data, especially when the input is dense without any missing value. If the host memory can fit a significant portion of this compressed cache, the performance should be decent after initialization. Our development so far focuses on following fronts of optimization for external memory:

- Avoid iterating through the data whenever appropriate.
- If the OS can cache the data, the performance should be close to in-core training.
- For GPU, the actual computation should overlap with memory copy as much as possible.

Starting with XGBoost 2.0, the CPU implementation of external memory uses `mmap`. It has not been tested against system errors like disconnected network devices (*SIGBUS*). In the face of a bus error, you will see a hard crash and need to clean up the cache files. If the training session might take a long time and you use solutions like NVMe-oF, we recommend checkpointing your model periodically. Also, it's worth noting that most tests have been conducted on Linux distributions.

Another important point to keep in mind is that creating the initial cache for XGBoost may take some time. The interface to external memory is through custom iterators, which we can not assume to be thread-safe. Therefore, initialization is performed sequentially. Using the `config_context()` with *verbosity=2* can give you some information on what XGBoost is doing during the wait if you don't mind the extra output.

Compared to the QuantileDMatrix

Passing an iterator to the `QuantileDMatrix` enables direct construction of `QuantileDMatrix` with data chunks. On the other hand, if it's passed to the `DMatrix` or the `ExtMemQuantileDMatrix`, it instead enables the external memory feature. The `QuantileDMatrix` concatenates the data in memory after compression and doesn't fetch data during training. On the other hand, the external memory `DMatrix` (`ExtMemQuantileDMatrix`) fetches data batches from external memory on demand. Use the `QuantileDMatrix` (with iterator if necessary) when you can fit most of your data in memory. For many platforms, the training speed can be an order of magnitude faster than external memory.

Brief History

For a long time, external memory support has been an experimental feature and has undergone multiple development iterations. Here's a brief summary of major changes:

- Gradient-based sampling was introduced to the GPU hist in 1.1.
- The iterator interface was introduced in 1.5, along with a major rewrite for the internal framework.
- 2.0 introduced the use of `mmap`, along with optimization in XBGoost to enable zero-copy data fetching.

- 3.0 reworked the GPU implementation to support caching data on the host and disk, introduced the `ExtMemQuantileDMatrix` class, added quantile-based objectives support.
- In addition, we begin support for distributed training in 3.0
- 3.1 added support for having divided cache pages. One can have part of a cache page in the GPU and the rest of the cache in the host memory. In addition, XGBoost works with the Grace Blackwell hardware decompression engine when data is sparse.
- The text file cache format has been removed in 3.1.0.
- The page concatenation option has been removed in 3.2.0.

1.4.17 C API Tutorial

In this tutorial, we are going to install XGBoost library & configure the CMakeLists.txt file of our C/C++ application to link XGBoost library with our application. Later on, we will see some useful tips for using C API and code snippets as examples to use various functions available in C API to perform basic task like loading, training model & predicting on test dataset. For API reference, please visit *XGBoost C Package*

- *Requirements*
- *Install XGBoost on conda environment*
- *Configure CMakeList.txt file of your application to link with XGBoost*
- *Useful Tips To Remember*
- *Sample examples along with Code snippet to use C API functions*

Requirements

Install CMake - Follow the cmake installation documentation for instructions. Install Conda - Follow the conda installation documentation for instructions

Install XGBoost on conda environment

Run the following commands on your terminal. The below commands will install the XGBoost in your XGBoost folder of the repository cloned

```
# clone the XGBoost repository & its submodules
git clone --recursive https://github.com/dmlc/xgboost
cd xgboost
# Activate the Conda environment, into which we'll install XGBoost
conda activate [env_name]
# Build the compiled version of XGBoost inside the build folder
cmake -B build -S . -DCMAKE_INSTALL_PREFIX=$CONDA_PREFIX
# install XGBoost in your conda environment (usually under [your home directory]/
↪miniconda3)
cmake --build build --target install
```

Configure CMakeList.txt file of your application to link with XGBoost

Here, we assume that your C++ application is using CMake for builds.

Use `find_package()` and `target_link_libraries()` in your application's CMakeList.txt to link with the XGBoost library:

```
cmake_minimum_required(VERSION 3.18)
project(your_project_name LANGUAGES C CXX VERSION your_project_version)
find_package(xgboost REQUIRED)
add_executable(your_project_name /path/to/project_file.c)
target_link_libraries(your_project_name xgboost::xgboost)
```

To ensure that CMake can locate the XGBoost library, supply `-DCMAKE_PREFIX_PATH=$CONDA_PREFIX` argument when invoking CMake. This option instructs CMake to locate the XGBoost library in `$CONDA_PREFIX`, which is where your Conda environment is located.

```
# Activate the Conda environment where we previously installed XGBoost
conda activate [env_name]
# Invoke CMake with CMAKE_PREFIX_PATH
cmake -B build -S . -DCMAKE_PREFIX_PATH=$CONDA_PREFIX
# Build your application
cmake --build build
```

Useful Tips To Remember

Below are some useful tips while using C API:

1. Error handling: Always check the return value of the C API functions.

a. In a C application: Use the following macro to guard all calls to XGBoost's C API functions. The macro prints all the error/ exception occurred:

```
#define safe_xgboost(call) {  \
  int err = (call); \
  if (err != 0) { \
    fprintf(stderr, "%s:%d: error in %s: %s\n", __FILE__, __LINE__, #call,
→XGBGetLastError());  \
    exit(1); \
  } \
}
```

In your application, wrap all C API function calls with the macro as follows:

```
DMatrixHandle train;
safe_xgboost(XGDMatrixCreateFromFile("/path/to/training/dataset/", silent, &train));
```

b. In a C++ application: modify the macro `safe_xgboost` to throw an exception upon an error.

```
#define safe_xgboost(call) {  \
  int err = (call); \
  if (err != 0) { \
    throw std::runtime_error(std::string(__FILE__) + ":" + std::to_string(__LINE__) + \
                        ": error in " + #call + ":" + XGBGetLastError());  \
  } \
}
```

c. Assertion technique: It works both in C/ C++. If expression evaluates to 0 (false), then the expression, source code filename, and line number are sent to the standard error, and then abort() function is called. It can be used to test assumptions made by you in the code.

```
DMatrixHandle dmat;
assert( XGDMatrixCreateFromFile("training_data.libsvm", 0, &dmat) == 0);
```

2. Always remember to free the allocated space by BoosterHandle & DMatrixHandle appropriately:

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <xgboost/c_api.h>

int main(int argc, char** argv) {
  int silent = 0;

  BoosterHandle booster;

  // do something with booster

  //free the memory
  XGBoosterFree(booster);

  DMatrixHandle DMatrixHandle_param;

  // do something with DMatrixHandle_param

  // free the memory
  XGDMatrixFree(DMatrixHandle_param);

  return 0;
}
```

3. For tree models, it is important to use consistent data formats during training and scoring/ predicting otherwise it will result in wrong outputs. Example if we our training data is in `dense matrix` format then your prediction dataset should also be a `dense matrix` or if training in `libsvm` format then dataset for prediction should also be in `libsvm` format.

4. Always use strings for setting values to the parameters in booster handle object. The parameter value can be of any data type (e.g. int, char, float, double, etc), but they should always be encoded as strings.

```
BoosterHandle booster;
XGBoosterSetParam(booster, "parameter_name", "0.1");
```

Sample examples along with Code snippet to use C API functions

1. If the dataset is available in a file, it can be loaded into a `DMatrix` object using the *XGDMatrixCreateFromFile()*

```
DMatrixHandle data; // handle to DMatrix
// Load the data from file & store it in data variable of DMatrixHandle datatype
safe_xgboost(XGDMatrixCreateFromFile("/path/to/file/filename", silent, &data));
```

2. You can also create a `DMatrix` object from a 2D Matrix using the *XGDMatrixCreateFromMat()*

```
// 1D matrix
const int data1[] = { 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0 };

// 2D matrix
const int ROWS = 6, COLS = 3;
const int data2[ROWS][COLS] = { {1, 2, 3}, {2, 4, 6}, {3, -1, 9}, {4, 8, -1}, {2, 5, 1}, {0, 1, 5} };
DMatrixHandle dmatrix1, dmatrix2;
// Pass the matrix, no of rows & columns contained in the matrix variable
// here '0' represents the missing value in the matrix dataset
// dmatrix variable will contain the created DMatrix using it
safe_xgboost(XGDMatrixCreateFromMat(data1, 1, 50, 0, &dmatrix));
// here -1 represents the missing value in the matrix dataset
safe_xgboost(XGDMatrixCreateFromMat(data2, ROWS, COLS, -1, &dmatrix2));
```

3. Create a Booster object for training & testing on dataset using *XGBoosterCreate()*

```
BoosterHandle booster;
const int eval_dmats_size;
// We assume that training and test data have been loaded into 'train' and 'test'
DMatrixHandle eval_dmats[eval_dmats_size] = {train, test};
safe_xgboost(XGBoosterCreate(eval_dmats, eval_dmats_size, &booster));
```

4. For each DMatrix object, set the labels using *XGDMatrixSetFloatInfo()*. Later you can access the label using *XGDMatrixGetFloatInfo()*.

```
const int ROWS=5, COLS=3;
const int data[ROWS][COLS] = { {1, 2, 3}, {2, 4, 6}, {3, -1, 9}, {4, 8, -1}, {2, 5, 1}, {0, 1, 5} };
DMatrixHandle dmatrix;

safe_xgboost(XGDMatrixCreateFromMat(data, ROWS, COLS, -1, &dmatrix));

// variable to store labels for the dataset created from above matrix
float labels[ROWS];

for (int i = 0; i < ROWS; i++) {
  labels[i] = i;
}

// Loading the labels
safe_xgboost(XGDMatrixSetFloatInfo(dmatrix, "label", labels, ROWS));

// reading the labels and store the length of the result
bst_ulong result_len;

// labels result
const float *result;

safe_xgboost(XGDMatrixGetFloatInfo(dmatrix, "label", &result_len, &result));

for(unsigned int i = 0; i < result_len; i++) {
```

(continues on next page)

(continued from previous page)

```
  printf("label[%i] = %f\n", i, result[i]);
}
```

5. Set the parameters for the Booster object according to the requirement using *XGBoosterSetParam()* . Check out the full list of parameters available *here* .

```
BoosterHandle booster;
safe_xgboost(XGBoosterSetParam(booster, "booster", "gblinear"));
// default max_depth =6
safe_xgboost(XGBoosterSetParam(booster, "max_depth", "3"));
// default eta  = 0.3
safe_xgboost(XGBoosterSetParam(booster, "eta", "0.1"));
```

6. Train & evaluate the model using *XGBoosterUpdateOneIter()* and *XGBoosterEvalOneIter()* respectively.

```
int num_of_iterations = 20;
const char* eval_names[eval_dmats_size] = {"train", "test"};
const char* eval_result = NULL;

for (int i = 0; i < num_of_iterations; ++i) {
  // Update the model performance for each iteration
  safe_xgboost(XGBoosterUpdateOneIter(booster, i, train));

  // Give the statistics for the learner for training & testing dataset in terms of error after each iteration
  safe_xgboost(XGBoosterEvalOneIter(booster, i, eval_dmats, eval_names, eval_dmats_size, &eval_result));
  printf("%s\n", eval_result);
}
```

Note

For customized loss function, use *XGBoosterBoostOneIter()* instead and manually specify the gradient and 2nd order gradient.

7. Predict the result on a test set using *XGBoosterPredictFromDMatrix()*

```
char const config[] =
    "{\"training\": false, \"type\": 0, "
    "\"iteration_begin\": 0, \"iteration_end\": 0, \"strict_shape\": false}";
/* Shape of output prediction */
uint64_t const* out_shape;
/* Dimension of output prediction */
uint64_t out_dim;
/* Pointer to a thread local contiguous array, assigned in prediction function. */
float const* out_result = NULL;
safe_xgboost(
    XGBoosterPredictFromDMatrix(booster, dmatrix, config, &out_shape, &out_dim, &out_result));

for (unsigned int i = 0; i < output_length; i++){
```

(continues on next page)

(continued from previous page)

```
  printf("prediction[%i] = %f \n", i, output_result[i]);
}
```

8. Get the number of features in your dataset using *XGBoosterGetNumFeature()*.

```
bst_ulong num_of_features = 0;

// Assuming booster variable of type BoosterHandle is already declared
// and dataset is loaded and trained on booster
// storing the results in num_of_features variable
safe_xgboost(XGBoosterGetNumFeature(booster, &num_of_features));

// Printing number of features by type conversion of num_of_features variable from bst_
↪ulong to unsigned long
printf("num_feature: %lu\n", (unsigned long)(num_of_features));
```

9. Save the model using *XGBoosterSaveModel()*

```
BoosterHandle booster;
const char *model_path = "/path/of/model.json";
safe_xgboost(XGBoosterSaveModel(booster, model_path));
```

10. Load the model using *XGBoosterLoadModel()*

```
BoosterHandle booster;
const char *model_path = "/path/of/model.json";

// create booster handle first
safe_xgboost(XGBoosterCreate(NULL, 0, &booster));

// set the model parameters here

// load model
safe_xgboost(XGBoosterLoadModel(booster, model_path));

// predict the model here
```

11. Free all the internal structure used in your code using *XGDMatrixFree()* and *XGBoosterFree()*. This step is important to prevent memory leak.

```
safe_xgboost(XGDMatrixFree(dmatrix));
safe_xgboost(XGBoosterFree(booster));
```

1.4.18 Text Input Format of DMatrix

Here we will briefly describe the text input formats for XGBoost. However, for users with access to a supported language environment like Python or R, it's recommended to use data parsers from that ecosystem instead. For instance, `sklearn.datasets.load_svmlight_file()`.

Warning

As stated above, users are encouraged to use third-party data parsers. The text parsers in XGBoost have been deprecated.

Basic Input Format

XGBoost currently supports two text formats for ingesting data: LIBSVM and CSV. The rest of this document will describe the LIBSVM format. (See this Wikipedia article for a description of the CSV format.). Please be careful that, XGBoost does **not** understand file extensions, nor try to guess the file format, as there is no universal agreement upon file extension of LIBSVM or CSV. Instead it employs URI format for specifying the precise input file type. For example if you provide a *csv* file `./data.train.csv` as input, XGBoost will blindly use the default LIBSVM parser to digest it and generate a parser error. Instead, users need to provide an URI in the form of `train.csv?format=csv` or `train.csv?format=libsvm`. For external memory input, the URI should of a form similar to `train.csv?format=csv#dtrain.cache`. See *Data Interface* and *Using XGBoost External Memory Version* also.

For training or predicting, XGBoost takes an instance file with the format as below:

Listing 18: `train.txt`

```
1 101:1.2 102:0.03
0 1:2.1 10001:300 10002:400
0 0:1.3 1:0.3
1 0:0.01 1:0.3
0 0:0.2 1:0.3
```

Each line represent a single instance, and in the first line '1' is the instance label, '101' and '102' are feature indices, '1.2' and '0.03' are feature values. In the binary classification case, '1' is used to indicate positive samples, and '0' is used to indicate negative samples. We also support probability values in [0,1] as label, to indicate the probability of the instance being positive.

Auxiliary Files for Additional Information

Note: all information below is applicable only to single-node version of the package. If you'd like to perform distributed training with multiple nodes, skip to the section *Embedding additional information inside LIBSVM file*.

Group Input Format

For ranking task, XGBoost supports the group input format. In ranking task, instances are categorized into *query groups* in real world scenarios. For example, in the learning to rank web pages scenario, the web page instances are grouped by their queries. XGBoost requires an file that indicates the group information. For example, if the instance file is the `train.txt` shown above, the group file should be named `train.txt.group` and be of the following format:

Listing 19: `train.txt.group`

```
2
3
```

This means that, the data set contains 5 instances, and the first two instances are in a group and the other three are in another group. The numbers in the group file are actually indicating the number of instances in each group in the instance file in order. At the time of configuration, you do not have to indicate the path of the group file. If the instance file name is `xxx`, XGBoost will check whether there is a file named `xxx.group` in the same directory.

Instance Weight File

Instances in the training data may be assigned weights to differentiate relative importance among them. For example, if we provide an instance weight file for the `train.txt` file in the example as below:

Listing 20: `train.txt.weight`

```
1
0.5
0.5
1
0.5
```

It means that XGBoost will emphasize more on the first and fourth instance (i.e. the positive instances) while training. The configuration is similar to configuring the group information. If the instance file name is `xxx`, XGBoost will look for a file named `xxx.weight` in the same directory. If the file exists, the instance weights will be extracted and used at the time of training.

> **Note**
>
> Binary buffer format and instance weights
>
> If you choose to save the training data as a binary buffer (using `save_binary()`), keep in mind that the resulting binary buffer file will include the instance weights. To update the weights, use the `set_weight()` function.

Initial Margin File

XGBoost supports providing each instance an initial margin prediction. For example, if we have a initial prediction using logistic regression for `train.txt` file, we can create the following file:

Listing 21: train.txt.base_margin

```
-0.4
1.0
3.4
```

XGBoost will take these values as initial margin prediction and boost from that. An important note about base_margin is that it should be margin prediction before transformation, so if you are doing logistic loss, you will need to put in value before logistic transformation. If you are using XGBoost predictor, use `pred_margin=1` to output margin values.

Embedding additional information inside LIBSVM file

This section is applicable to both single- and multiple-node settings.

Query ID Columns

This is most useful for ranking task, where the instances are grouped into query groups. You may embed query group ID for each instance in the LIBSVM file by adding a token of form `qid:xx` in each row:

Listing 22: train.txt

```
1 qid:1 101:1.2 102:0.03
0 qid:1 1:2.1 10001:300 10002:400
0 qid:2 0:1.3 1:0.3
1 qid:2 0:0.01 1:0.3
0 qid:3 0:0.2 1:0.3
1 qid:3 3:-0.1 10:-0.3
0 qid:3 6:0.2 10:0.15
```

Keep in mind the following restrictions:

- You are not allowed to specify query ID's for some instances but not for others. Either every row is assigned query ID's or none at all.
- The rows have to be sorted in ascending order by the query IDs. So, for instance, you may not have one row having large query ID than any of the following rows.

Instance weights

You may specify instance weights in the LIBSVM file by appending each instance label with the corresponding weight in the form of `[label]:[weight]`, as shown by the following example:

Listing 23: train.txt

```
1:1.0 101:1.2 102:0.03
0:0.5 1:2.1 10001:300 10002:400
0:0.5 0:1.3 1:0.3
1:1.0 0:0.01 1:0.3
0:0.5 0:0.2 1:0.3
```

where the negative instances are assigned half weights compared to the positive instances.

1.4.19 Notes on Parameter Tuning

Parameter tuning is a dark art in machine learning, the optimal parameters of a model can depend on many scenarios. So it is impossible to create a comprehensive guide for doing so.

This document tries to provide some guideline for parameters in XGBoost.

Understanding Bias-Variance Tradeoff

If you take a machine learning or statistics course, this is likely to be one of the most important concepts. When we allow the model to get more complicated (e.g. more depth), the model has better ability to fit the training data, resulting in a less biased model. However, such complicated model requires more data to fit.

Most of parameters in XGBoost are about bias variance tradeoff. The best model should trade the model complexity with its predictive power carefully. *Parameters Documentation* will tell you whether each parameter will make the model more conservative or not. This can be used to help you turn the knob between complicated model and simple model.

Control Overfitting

When you observe high training accuracy, but low test accuracy, it is likely that you encountered overfitting problem.

There are in general two ways that you can control overfitting in XGBoost:

- The first way is to directly control model complexity.
  - This includes `max_depth`, `min_child_weight`, `gamma`, `max_cat_threshold` and other similar regularization parameters. See *XGBoost Parameters* for a comprehensive set of parameters.
  - Set a constant `base_score` based on your own criteria. See *Intercept* for more info.
- The second way is to add randomness to make training robust to noise.
  - This includes `subsample` and `colsample_bytree`, which may be used with boosting RF `num_parallel_tree`.
  - You can also reduce stepsize `eta`, possibly with a training callback. Remember to increase `num_round` when you do so.

Handle Imbalanced Dataset

For common cases such as ads clickthrough log, the dataset is extremely imbalanced. This can affect the training of XGBoost model, and there are two ways to improve it.

- If you care only about the overall performance metric (AUC) of your prediction
  - Balance the positive and negative weights via `scale_pos_weight`
  - Use AUC for evaluation
- If you care about predicting the right probability

- In such a case, you cannot re-balance the dataset
- Set parameter `max_delta_step` to a finite number (say 1) to help convergence

Use Hyper Parameter Optimization (HPO) Frameworks

Tuning models is a sophisticated task and there are advanced frameworks to help you. For examples, some meta estimators in scikit-learn like `sklearn.model_selection.HalvingGridSearchCV` can help guide the search process. Optuna is another great option and there are many more based on different branches of statistics.

Know Your Data

It cannot be stressed enough the importance of understanding the data, sometimes that's all it takes to get a good model. Many solutions use a simple XGBoost tree model without much tuning and emphasize the data pre-processing step. XGBoost can help feature selection by providing both a global feature importance score and sample feature importance with SHAP value. Also, there are parameters specifically targeting categorical features, and tasks like survival and ranking. Feel free to explore them.

Reducing Memory Usage

If you are using a HPO library like `sklearn.model_selection.GridSearchCV`, please control the number of threads it can use. It's best to let XGBoost to run in parallel instead of asking *GridSearchCV* to run multiple experiments at the same time. For instance, creating a fold of data for cross validation can consume a significant amount of memory:

```
# This creates a copy of dataset. X and X_train are both in memory at the same time.

# This happens for every thread at the same time if you run `GridSearchCV` with
# `n_jobs` larger than 1

X_train, X_test, y_train, y_test = train_test_split(X, y)
```

```
df = pd.DataFrame()
# This creates a new copy of the dataframe, even if you specify the inplace parameter
new_df = df.drop(...)
```

```
array = np.array(...)
# This may or may not make a copy of the data, depending on the type of the data
array.astype(np.float32)
```

```
# np by default uses double, do you actually need it?
array = np.array(...)
```

You can find some more specific memory reduction practices scattered through the documents For instances: *Distributed XGBoost with Dask*, *XGBoost GPU Support*. However, before going into these, being conscious about making data copies is a good starting point. It usually consumes a lot more memory than people expect.

1.4.20 Custom Objective and Evaluation Metric

Contents

- *Overview*
- *Customized Objective Function*

- *Customized Metric Function*
- *Reverse Link Function*
- *Scikit-Learn Interface*

Overview

XGBoost is designed to be an extensible library. One way to extend it is by providing our own objective function for training and corresponding metric for performance monitoring. This document introduces implementing a customized elementwise evaluation metric and objective for XGBoost. Although the introduction uses Python for demonstration, the concepts should be readily applicable to other language bindings.

Note

- The ranking task does not support customized functions.
- Breaking change was made in XGBoost 1.6.

See also the advanced usage example for more information about limitations and workarounds for more complex objetives: *Advanced Usage of Custom Objectives*

In the following two sections, we will provide a step by step walk through of implementing the `Squared Log Error (SLE)` objective function:

$$\frac{1}{2}[\log(pred + 1) - \log(label + 1)]^2$$

and its default metric `Root Mean Squared Log Error(RMSLE)`:

$$\sqrt{\frac{1}{N}[\log(pred + 1) - \log(label + 1)]^2}$$

Although XGBoost has native support for said functions, using it for demonstration provides us the opportunity of comparing the result from our own implementation and the one from XGBoost internal for learning purposes. After finishing this tutorial, we should be able to provide our own functions for rapid experiments. And at the end, we will provide some notes on non-identity link function along with examples of using custom metric and objective with the *scikit-learn* interface.

If we compute the gradient of said objective function:

$$g = \frac{\partial objective}{\partial pred} = \frac{\log(pred + 1) - \log(label + 1)}{pred + 1}$$

As well as the hessian (the second derivative of the objective):

$$h = \frac{\partial^2 objective}{\partial pred^2} = \frac{-\log(pred + 1) + \log(label + 1) + 1}{(pred + 1)^2}$$

Customized Objective Function

During model training, the objective function plays an important role: provide gradient information, both first and second order gradient, based on model predictions and observed data labels (or targets). Therefore, a valid objective function should accept two inputs, namely prediction and labels. For implementing `SLE`, we define:

```
import numpy as np
import xgboost as xgb
from typing import Tuple

def gradient(predt: np.ndarray, dtrain: xgb.DMatrix) -> np.ndarray:
    '''Compute the gradient squared log error.'''
    y = dtrain.get_label()
    return (np.log1p(predt) - np.log1p(y)) / (predt + 1)

def hessian(predt: np.ndarray, dtrain: xgb.DMatrix) -> np.ndarray:
    '''Compute the hessian for squared log error.'''
    y = dtrain.get_label()
    return ((-np.log1p(predt) + np.log1p(y) + 1) /
            np.power(predt + 1, 2))

def squared_log(predt: np.ndarray,
                dtrain: xgb.DMatrix) -> Tuple[np.ndarray, np.ndarray]:
    '''Squared Log Error objective. A simplified version for RMSLE used as
    objective function.
    '''
    predt[predt < -1] = -1 + 1e-6
    grad = gradient(predt, dtrain)
    hess = hessian(predt, dtrain)
    return grad, hess
```

In the above code snippet, `squared_log` is the objective function we want. It accepts a numpy array `predt` as model prediction, and the training DMatrix for obtaining required information, including labels and weights (not used here). This objective is then used as a callback function for XGBoost during training by passing it as an argument to `xgb.train`:

```
xgb.train({'tree_method': 'hist', 'seed': 1994},  # any other tree method is fine.
          dtrain=dtrain,
          num_boost_round=10,
          obj=squared_log)
```

Notice that in our definition of the objective, whether we subtract the labels from the prediction or the other way around is important. If you find the training error goes up instead of down, this might be the reason.

Customized Metric Function

So after having a customized objective, we might also need a corresponding metric to monitor our model's performance. As mentioned above, the default metric for SLE is RMSLE. Similarly we define another callback like function as the new metric:

```
def rmsle(predt: np.ndarray, dtrain: xgb.DMatrix) -> Tuple[str, float]:
    ''' Root mean squared log error metric.'''
    y = dtrain.get_label()
    predt[predt < -1] = -1 + 1e-6
    elements = np.power(np.log1p(y) - np.log1p(predt), 2)
    return 'PyRMSLE', float(np.sqrt(np.sum(elements) / len(y)))
```

Since we are demonstrating in Python, the metric or objective need not be a function, any callable object should suffice. Similar to the objective function, our metric also accepts `predt` and `dtrain` as inputs, but returns the name of the metric itself and a floating point value as the result. After passing it into XGBoost as argument of `custom_metric` parameter:

```
xgb.train({'tree_method': 'hist', 'seed': 1994,
           'disable_default_eval_metric': 1},
          dtrain=dtrain,
          num_boost_round=10,
          obj=squared_log,
          custom_metric=rmsle,
          evals=[(dtrain, 'dtrain'), (dtest, 'dtest')],
          evals_result=results)
```

We will be able to see XGBoost printing something like:

```
[0] dtrain-PyRMSLE:1.37153  dtest-PyRMSLE:1.31487
[1] dtrain-PyRMSLE:1.26619  dtest-PyRMSLE:1.20899
[2] dtrain-PyRMSLE:1.17508  dtest-PyRMSLE:1.11629
[3] dtrain-PyRMSLE:1.09836  dtest-PyRMSLE:1.03871
[4] dtrain-PyRMSLE:1.03557  dtest-PyRMSLE:0.977186
[5] dtrain-PyRMSLE:0.985783 dtest-PyRMSLE:0.93057
...
```

Notice that the parameter `disable_default_eval_metric` is used to suppress the default metric in XGBoost.

For fully reproducible source code and comparison plots, see *Demo for defining a custom regression objective and metric*.

Reverse Link Function

When using builtin objective, the raw prediction is transformed according to the objective function. When a custom objective is provided XGBoost doesn't know its link function so the user is responsible for making the transformation for both objective and custom evaluation metric. For objective with identity link like `squared error` this is trivial, but for other link functions like log link or inverse link the difference is significant.

For the Python package, the behaviour of prediction can be controlled by the `output_margin` parameter in `predict` function. When using the `custom_metric` parameter without a custom objective, the metric function will receive transformed prediction since the objective is defined by XGBoost. However, when the custom objective is also provided along with that metric, then both the objective and custom metric will receive raw prediction. The following example provides a comparison between two different behavior with a multi-class classification model. Firstly we define 2 different Python metric functions implementing the same underlying metric for comparison, *merror_with_transform* is used when custom objective is also used, otherwise the simpler *merror* is preferred since XGBoost can perform the transformation itself.

```
import xgboost as xgb
import numpy as np

def merror_with_transform(predt: np.ndarray, dtrain: xgb.DMatrix):
    """Used when custom objective is supplied."""
    y = dtrain.get_label()
    n_classes = predt.size // y.shape[0]
    # Like custom objective, the predt is untransformed leaf weight when custom objective
    # is provided.

    # With the use of `custom_metric` parameter in train function, custom metric receives
    # raw input only when custom objective is also being used.  Otherwise custom metric
    # will receive transformed prediction.
    assert predt.shape == (d_train.num_row(), n_classes)
```

(continues on next page)

(continued from previous page)

```
    out = np.zeros(dtrain.num_row())
    for r in range(predt.shape[0]):
        i = np.argmax(predt[r])
        out[r] = i

    assert y.shape == out.shape

    errors = np.zeros(dtrain.num_row())
    errors[y != out] = 1.0
    return 'PyMError', np.sum(errors) / dtrain.num_row()
```

The above function is only needed when we want to use custom objective and XGBoost doesn't know how to transform the prediction. The normal implementation for multi-class error function is:

```
def merror(predt: np.ndarray, dtrain: xgb.DMatrix):
    """Used when there's no custom objective."""
    # No need to do transform, XGBoost handles it internally.
    errors = np.zeros(dtrain.num_row())
    errors[y != out] = 1.0
    return 'PyMError', np.sum(errors) / dtrain.num_row()
```

Next we need the custom softprob objective:

```
def softprob_obj(predt: np.ndarray, data: xgb.DMatrix):
    """Loss function.  Computing the gradient and approximated hessian (diagonal).
    Reimplements the `multi:softprob` inside XGBoost.
    """

    # Full implementation is available in the Python demo script linked below
    ...

    return grad, hess
```

Lastly we can train the model using `obj` and `custom_metric` parameters:

```
Xy = xgb.DMatrix(X, y)
booster = xgb.train(
    {"num_class": kClasses, "disable_default_eval_metric": True},
    m,
    num_boost_round=kRounds,
    obj=softprob_obj,
    custom_metric=merror_with_transform,
    evals_result=custom_results,
    evals=[(m, "train")],
)
```

Or if you don't need the custom objective and just want to supply a metric that's not available in XGBoost:

```
booster = xgb.train(
    {
        "num_class": kClasses,
        "disable_default_eval_metric": True,
        "objective": "multi:softmax",
```

(continues on next page)

(continued from previous page)

```
    },
    m,
    num_boost_round=kRounds,
    # Use a simpler metric implementation.
    custom_metric=merror,
    evals_result=custom_results,
    evals=[(m, "train")],
)
```

We use `multi:softmax` to illustrate the differences of transformed prediction. With `softprob` the output prediction array has shape `(n_samples, n_classes)` while for `softmax` it's `(n_samples, )`. A demo for multi-class objective function is also available at *Demo for creating customized multi-class objective function*. Also, see *Intercept* for some more explanation.

Scikit-Learn Interface

The scikit-learn interface of XGBoost has some utilities to improve the integration with standard scikit-learn functions. For instance, after XGBoost 1.6.0 users can use the cost function (not scoring functions) from scikit-learn out of the box:

```
from sklearn.datasets import load_diabetes
from sklearn.metrics import mean_absolute_error
X, y = load_diabetes(return_X_y=True)
reg = xgb.XGBRegressor(
    tree_method="hist",
    eval_metric=mean_absolute_error,
)
reg.fit(X, y, eval_set=[(X, y)])
```

Also, for custom objective function, users can define the objective without having to access `DMatrix`:

```
def softprob_obj(labels: np.ndarray, predt: np.ndarray) -> Tuple[np.ndarray, np.ndarray]:
    rows = labels.shape[0]
    classes = predt.shape[1]
    grad = np.zeros((rows, classes), dtype=float)
    hess = np.zeros((rows, classes), dtype=float)
    eps = 1e-6
    for r in range(predt.shape[0]):
        target = labels[r]
        p = softmax(predt[r, :])
        for c in range(predt.shape[1]):
            g = p[c] - 1.0 if c == target else p[c]
            h = max((2.0 * p[c] * (1.0 - p[c])).item(), eps)
            grad[r, c] = g
            hess[r, c] = h

    grad = grad.reshape((rows * classes, 1))
    hess = hess.reshape((rows * classes, 1))
    return grad, hess

clf = xgb.XGBClassifier(tree_method="hist", objective=softprob_obj)
```

1.4.21 Advanced Usage of Custom Objectives

Contents

- *Overview*
- *Dirichlet Regression Formulae*
- *Dirichlet Regression as Objective Function*
- *Practical Example*

Overview

XGBoost allows optimizing custom user-defined functions based on gradients and Hessians provided by the user for the desired objective function.

In order for a custom objective to work as intended:

- The function to optimize must be smooth and twice differentiable.
- The function must be additive with respect to rows / observations, such as a likelihood function with i.i.d. assumptions.
- The range of the scores for the function must be unbounded (i.e. it should not work exclusively with positive numbers, for example).
- The function must be convex. Note that, if the Hessian has negative values, they will be clipped, which will likely result in a model that does not fit the function well.
- For multi-output objectives, there should not be dependencies between different targets (i.e. Hessian should be diagonal for each row).

Some of these limitations can nevertheless be worked around by foregoing the true Hessian of the function, using something else instead such as an approximation with better properties - convergence might be slower when not using the true Hessian of a function, but many theoretical guarantees should still hold and result in usable models. For example, XGBoost's internal implementation of multionomial logistic regression uses an upper bound on the Hessian with diagonal structure instead of the true Hessian which is a full square matrix for each row in the data.

This tutorial provides some suggestions for use-cases that do not perfectly fit the criteria outlined above, by showing how to solve a Dirichlet regression parameterized by concentrations.

A Dirichlet regression model poses certain challenges for XGBoost:

- Concentration parameters must be positive. An easy way to achieve this is by applying an 'exp' transform on raw unbounded values, but in such case the objective becomes non-convex. Furthermore, note that this function is not in the exponential family, unlike typical distributions used for GLM models.
- The Hessian has dependencies between targets - that is, for a Dirichlet distribution with 'k' parameters, each row will have a full Hessian matrix of dimensions `[k, k]`.
- An optimal intercept for this type of model would involve a vector of values rather than the same value for every target.

In order to use this type of model as a custom objetive:

- It's possible to use the expected Hessian (a.k.a. the Fisher information matrix or expected information) instead of the true Hessian. The expected Hessian is always positive semi-definite for an additive likelihood, even if the true Hessian isn't.

- It's possible to use an upper bound on the expected Hessian with a diagonal structure, such that a second-order approximation under this diagonal bound would always yield greater or equal function values than under the non-diagonal expected Hessian.
- Since the `base_score` parameter that XGBoost uses for an intercept is limited to a scalar, one can use the `base_margin` functionality instead, but note that using it requires a bit more effort.

Dirichlet Regression Formulae

The Dirichlet distribution is a generalization of the Beta distribution to multiple dimensions. It models proportions data in which the values sum to 1, and is typically used as part of composite models (e.g. Dirichlet-multinomial) or as a prior in Bayesian models, but it also can be used on its own for proportions data for example.

Its likelihood for a given observation with values `y` and a given prediction `x` is given as follows:

$$L(\mathbf{y}|\mathbf{x}) = \frac{1}{\beta(\mathbf{x})} \prod_{i=1}^{k} y_i^{x_i - 1}$$

Where:

$$\beta(\mathbf{x}) = \frac{\prod_{i=1}^{k} \Gamma(x_i)}{\Gamma(\sum_{i=1}^{k} x_i)}$$

In this case, we want to optimize the negative of the log-likelihood summed across rows. The resulting function, gradient and Hessian could be implemented as follows:

Python

R

```
import numpy as np
from scipy.special import loggamma, psi as digamma, polygamma
trigamma = lambda x: polygamma(1, x)

def dirichlet_fun(pred: np.ndarray, Y: np.ndarray) -> float:
    epred = np.exp(pred)
    sum_epred = np.sum(epred, axis=1, keepdims=True)
    return (
        loggamma(epred).sum()
        - loggamma(sum_epred).sum()
        - np.sum(np.log(Y) * (epred - 1))
    )
def dirichlet_grad(pred: np.ndarray, Y: np.ndarray) -> np.ndarray:
    epred = np.exp(pred)
    return epred * (
        digamma(epred)
        - digamma(np.sum(epred, axis=1, keepdims=True))
        - np.log(Y)
    )
def dirichlet_hess(pred: np.ndarray, Y: np.ndarray) -> np.ndarray:
    epred = np.exp(pred)
    grad = dirichlet_grad(pred, Y)
    k = Y.shape[1]
    H = np.empty((pred.shape[0], k, k))
    for row in range(pred.shape[0]):
        H[row, :, :] = (
```

(continues on next page)

(continued from previous page)

```
            - trigamma(epred[row].sum()) * np.outer(epred[row], epred[row])
            + np.diag(grad[row] + trigamma(epred[row]) * epred[row] ** 2)
        )
    return H
```

```
softmax <- function(x) {
    max.x <- max(x)
    e <- exp(x - max.x)
    return(e / sum(e))
}

dirichlet.fun <- function(pred, y) {
    epred <- exp(pred)
    sum_epred <- rowSums(epred)
    return(
        sum(lgamma(epred))
        - sum(lgamma(sum_epred))
        - sum(log(y) * (epred - 1))
    )
}

dirichlet.grad <- function(pred, y) {
    epred <- exp(pred)
    return(
        epred * (
            digamma(epred)
            - digamma(rowSums(epred))
            - log(y)
        )
    )
}

dirichlet.hess <- function(pred, y) {
    epred <- exp(pred)
    grad <- dirichlet.grad(pred, y)
    k <- ncol(y)
    H <- array(dim = c(nrow(y), k, k))
    for (row in seq_len(nrow(y))) {
        H[row, , ] <- (
            - trigamma(sum(epred[row,])) * tcrossprod(epred[row,])
            + diag(grad[row,] + trigamma(epred[row,]) * epred[row,]^2)
        )
    }
    return(H)
}
```

Convince yourself that the implementation is correct:

Python

R

```
from math import isclose
```

(continues on next page)

(continued from previous page)

```
from scipy import stats
from scipy.optimize import check_grad
from scipy.special import softmax

def gen_random_dirichlet(rng: np.random.Generator, m: int, k: int):
    alpha = np.exp(rng.standard_normal(size=k))
    return rng.dirichlet(alpha, size=m)

def test_dirichlet_fun_grad_hess():
    k = 3
    m = 10
    rng = np.random.default_rng(seed=123)
    Y = gen_random_dirichlet(rng, m, k)
    x0 = rng.standard_normal(size=k)
    for row in range(Y.shape[0]):
        fun_row = dirichlet_fun(x0.reshape((1,-1)), Y[[row]])
        ref_logpdf = stats.dirichlet.logpdf(
            Y[row] / Y[row].sum(), # <- avoid roundoff error
            np.exp(x0),
        )
        assert isclose(fun_row, -ref_logpdf)

        gdiff = check_grad(
            lambda pred: dirichlet_fun(pred.reshape((1,-1)), Y[[row]]),
            lambda pred: dirichlet_grad(pred.reshape((1,-1)), Y[[row]]),
            x0
        )
        assert gdiff <= 1e-6

        H_numeric = np.empty((k,k))
        eps = 1e-7
        for ii in range(k):
            x0_plus_eps = x0.reshape((1,-1)).copy()
            x0_plus_eps[0,ii] += eps
            for jj in range(k):
                H_numeric[ii, jj] = (
                    dirichlet_grad(x0_plus_eps, Y[[row]])[0][jj]
                    - dirichlet_grad(x0.reshape((1,-1)), Y[[row]])[0][jj]
                ) / eps
        H = dirichlet_hess(x0.reshape((1,-1)), Y[[row]])[0]
        np.testing.assert_almost_equal(H, H_numeric, decimal=6)
test_dirichlet_fun_grad_hess()
```

```
library(DirichletReg)
library(testthat)

test_that("dirichlet formulae", {
    k <- 3L
    m <- 10L
    set.seed(123)
    alpha <- exp(rnorm(k))
    y <- rdirichlet(m, alpha)
```

(continues on next page)

(continued from previous page)

```
    x0 <- rnorm(k)

    for (row in seq_len(m)) {
        logpdf <- dirichlet.fun(matrix(x0, nrow=1), y[row,,drop=F])
        ref_logpdf <- ddirichlet(y[row,,drop=F], exp(x0), log = T)
        expect_equal(logpdf, -ref_logpdf)

        eps <- 1e-7
        grad_num <- numeric(k)
        for (col in seq_len(k)) {
            xplus <- x0
            xplus[col] <- x0[col] + eps
            grad_num[col] <- (
                dirichlet.fun(matrix(xplus, nrow=1), y[row,,drop=F])
                - dirichlet.fun(matrix(x0, nrow=1), y[row,,drop=F])
            ) / eps
        }

        grad <- dirichlet.grad(matrix(x0, nrow=1), y[row,,drop=F])
        expect_equal(grad |> as.vector(), grad_num, tolerance=1e-6)

        H_numeric <- array(dim=c(k, k))
        for (ii in seq_len(k)) {
            xplus <- x0
            xplus[ii] <- x0[ii] + eps
            for (jj in seq_len(k)) {
                H_numeric[ii, jj] <- (
                    dirichlet.grad(matrix(xplus, nrow=1), y[row,,drop=F])[1, jj]
                    - grad[1L, jj]
                ) / eps
            }
        }

        H <- dirichlet.hess(matrix(xplus, nrow=1), y[row,,drop=F])
        expect_equal(H[1,,], H_numeric, tolerance=1e-6)
    }
})
```

Dirichlet Regression as Objective Function

As mentioned earlier, the Hessian of this function is problematic for XGBoost: it can have a negative determinant, and might even have negative values in the diagonal, which is problematic for optimization methods - in XGBoost, those values would be clipped and the resulting model might not end up producing sensible predictions.

A potential workaround is to use the expected Hessian instead - that is, the expected outer product of the gradient if the response variable were distributed according to what is predicted. See the Wikipedia article for more information:

https://en.wikipedia.org/wiki/Fisher_information

In general, for objective functions in the exponential family, this is easy to obtain from the gradient of the link function and the variance of the probability distribution, but for other functions in general, it might involve other types of calculations (e.g. covariances and covariances of logarithms for Dirichlet).

It nevertheless results in a form very similar to the Hessian. One can also see from the differences here that, at an optimal point (gradient being zero), the expected and true Hessian for Dirichlet will match, which is a nice property

for optimization (i.e. the Hessian will be positive at a stationary point, which means it will be a minimum rather than a maximum or saddle point).

Python

R

```
def dirichlet_expected_hess(pred: np.ndarray) -> np.ndarray:
    epred = np.exp(pred)
    k = pred.shape[1]
    Ehess = np.empty((pred.shape[0], k, k))
    for row in range(pred.shape[0]):
        Ehess[row, :, :] = (
            - trigamma(epred[row].sum()) * np.outer(epred[row], epred[row])
            + np.diag(trigamma(epred[row]) * epred[row] ** 2)
        )
    return Ehess
def test_dirichlet_expected_hess():
    k = 3
    rng = np.random.default_rng(seed=123)
    x0 = rng.standard_normal(size=k)
    y_sample = rng.dirichlet(np.exp(x0), size=200_000)
    x_broadcast = np.broadcast_to(x0, (y_sample.shape[0], k))
    g_sample = dirichlet_grad(x_broadcast, y_sample)
    ref = (g_sample.T @ g_sample) / y_sample.shape[0]
    Ehess = dirichlet_expected_hess(x0.reshape((1,-1)))[0]
    np.testing.assert_allclose(Ehess, ref, rtol=5e-2, atol=5e-2)
test_dirichlet_expected_hess()
```

```
dirichlet.expected.hess <- function(pred) {
    epred <- exp(pred)
    k <- ncol(pred)
    H <- array(dim = c(nrow(pred), k, k))
    for (row in seq_len(nrow(pred))) {
        H[row, , ] <- (
            - trigamma(sum(epred[row,])) * tcrossprod(epred[row,])
            + diag(trigamma(epred[row,]) * epred[row,]^2)
        )
    }
    return(H)
}

test_that("expected hess", {
    k <- 3L
    set.seed(123)
    x0 <- rnorm(k)
    alpha <- exp(x0)
    n.samples <- 2e5
    y.samples <- rdirichlet(n.samples, alpha)

    x.broadcast <- rep(x0, n.samples) |> matrix(ncol=k, byrow=T)
    grad.samples <- dirichlet.grad(x.broadcast, y.samples)
    ref <- crossprod(grad.samples) / n.samples
    Ehess <- dirichlet.expected.hess(matrix(x0, nrow=1))
```

(continues on next page)

(continued from previous page)

```
    expect_equal(Ehess[1,,], ref, tolerance=5e-2)
})
```

But note that this is still not usable for XGBoost, since the expected Hessian, just like the true Hessian, has shape `[nrows, k, k]`, while XGBoost requires something with shape `[nrows, k]`.

One may use the diagonal of the expected Hessian for each row, but it's possible to do better: one can use instead an upper bound with diagonal structure, since it should lead to better convergence properties, just like for other Hessian-based optimization methods.

In the absence of any obvious way of obtaining an upper bound, a possibility here is to construct such a bound numerically based directly on the definition of a diagonally dominant matrix:

https://en.wikipedia.org/wiki/Diagonally_dominant_matrix

That is: take the absolute value of the expected Hessian for each row of the data, and sum by rows of the `[k, k]`-shaped Hessian for that row in the data:

Python

R

```
def dirichlet_diag_upper_bound_expected_hess(
    pred: np.ndarray, Y: np.ndarray
) -> np.ndarray:
    Ehess = dirichlet_expected_hess(pred)
    diag_bound_Ehess = np.empty((pred.shape[0], Y.shape[1]))
    for row in range(pred.shape[0]):
        diag_bound_Ehess[row, :] = np.abs(Ehess[row, :, :]).sum(axis=1)
    return diag_bound_Ehess
```

```
dirichlet.diag.upper.bound.expected.hess <- function(pred, y) {
    Ehess <- dirichlet.expected.hess(pred)
    diag.bound.Ehess <- array(dim=dim(pred))
    for (row in seq_len(nrow(pred))) {
        diag.bound.Ehess[row,] <- abs(Ehess[row,,]) |> rowSums()
    }
    return(diag.bound.Ehess)
}
```

(*note: the calculation can be made more efficiently than what is shown here by not calculating the full matrix, and in R, by making the rows be the last dimension and transposing after the fact*)

With all these pieces in place, one can now frame this model into the format required for XGBoost's custom objectives:

Python

R

```
import xgboost as xgb
from typing import Tuple

def dirichlet_xgb_objective(
    pred: np.ndarray, dtrain: xgb.DMatrix
) -> Tuple[np.ndarray, np.ndarray]:
    Y = dtrain.get_label().reshape(pred.shape)
    return (
```

(continues on next page)

(continued from previous page)

```
        dirichlet_grad(pred, Y),
        dirichlet_diag_upper_bound_expected_hess(pred, Y),
    )
```

```
library(xgboost)

dirichlet.xgb.objective <- function(pred, dtrain) {
    y <- getinfo(dtrain, "label")
    return(
        list(
            grad = dirichlet.grad(pred, y),
            hess = dirichlet.diag.upper.bound.expected.hess(pred, y)
        )
    )
}
```

And for an evaluation metric monitoring based on the Dirichlet log-likelihood:

Python

R

```
def dirichlet_eval_metric(
    pred: np.ndarray, dtrain: xgb.DMatrix
) -> Tuple[str, float]:
    Y = dtrain.get_label().reshape(pred.shape)
    return "dirichlet_ll", dirichlet_fun(pred, Y)
```

```
dirichlet.eval.metric <- function(pred, dtrain) {
    y <- getinfo(dtrain, "label")
    ll <- dirichlet.fun(pred, y)
    return(
        list(
            metric = "dirichlet_ll",
            value = ll
        )
    )
}
```

Practical Example

A good source for test datasets for proportions data is the R package `DirichletReg`:

https://cran.r-project.org/package=DirichletReg

For this example, we'll now use the Arctic Lake dataset (Aitchison, J. (2003). The Statistical Analysis of Compositional Data. The Blackburn Press, Caldwell, NJ.), taken from the `DirichletReg` R package, which consists of 39 rows with one predictor variable 'depth' and a three-valued response variable denoting the sediment composition of the measurements in this arctic lake (sand, silt, clay).

The data:

Python

R

```
# depth
X = np.array([
    10.4,11.7,12.8,13,15.7,16.3,18,18.7,20.7,22.1,
    22.4,24.4,25.8,32.5,33.6,36.8,37.8,36.9,42.2,47,
    47.1,48.4,49.4,49.5,59.2,60.1,61.7,62.4,69.3,73.6,
    74.4,78.5,82.9,87.7,88.1,90.4,90.6,97.7,103.7,
]).reshape((-1,1))
# sand, silt, clay
Y = np.array([
    [0.775,0.195,0.03], [0.719,0.249,0.032], [0.507,0.361,0.132],
    [0.522,0.409,0.066], [0.7,0.265,0.035], [0.665,0.322,0.013],
    [0.431,0.553,0.016], [0.534,0.368,0.098], [0.155,0.544,0.301],
    [0.317,0.415,0.268], [0.657,0.278,0.065], [0.704,0.29,0.006],
    [0.174,0.536,0.29], [0.106,0.698,0.196], [0.382,0.431,0.187],
    [0.108,0.527,0.365], [0.184,0.507,0.309], [0.046,0.474,0.48],
    [0.156,0.504,0.34], [0.319,0.451,0.23], [0.095,0.535,0.37],
    [0.171,0.48,0.349], [0.105,0.554,0.341], [0.048,0.547,0.41],
    [0.026,0.452,0.522], [0.114,0.527,0.359], [0.067,0.469,0.464],
    [0.069,0.497,0.434], [0.04,0.449,0.511], [0.074,0.516,0.409],
    [0.048,0.495,0.457], [0.045,0.485,0.47], [0.066,0.521,0.413],
    [0.067,0.473,0.459], [0.074,0.456,0.469], [0.06,0.489,0.451],
    [0.063,0.538,0.399], [0.025,0.48,0.495], [0.02,0.478,0.502],
])
```

```
data("ArcticLake", package="DirichletReg")
x <- ArcticLake[, c("depth"), drop=F]
y <- ArcticLake[, c("sand", "silt", "clay")] |> as.matrix()
```

Fitting an XGBoost model and making predictions:

Python

R

```
from typing import Dict, List

dtrain = xgb.DMatrix(X, label=Y)
results: Dict[str, Dict[str, List[float]]] = {}
booster = xgb.train(
    params={
        "tree_method": "hist",
        "num_target": Y.shape[1],
        "base_score": 0,
        "disable_default_eval_metric": True,
        "max_depth": 3,
        "seed": 123,
    },
    dtrain=dtrain,
    num_boost_round=10,
    obj=dirichlet_xgb_objective,
    evals=[(dtrain, "Train")],
    evals_result=results,
    custom_metric=dirichlet_eval_metric,
    verbose_eval=False,
```

(continues on next page)

(continued from previous page)

```
)
yhat = softmax(booster.inplace_predict(X), axis=1)
```

```
dtrain <- xgb.DMatrix(x, y)
booster <- xgb.train(
    params = list(
        tree_method="hist",
        num_target=ncol(y),
        base_score=0,
        disable_default_eval_metric=TRUE,
        max_depth=3,
        seed=123
    ),
    data = dtrain,
    nrounds = 10,
    obj = dirichlet.xgb.objective,
    evals = list(Train=dtrain),
    eval_metric = dirichlet.eval.metric
)
raw.pred <- predict(booster, x, reshape=TRUE)
yhat <- apply(raw.pred, 1, softmax) |> t()
```

Should produce an evaluation log as follows (note: the function is decreasing as expected - but unlike other objectives, the minimum value here can reach below zero):

```
[0] Train-dirichlet_ll:-40.25009
[1] Train-dirichlet_ll:-47.69122
[2] Train-dirichlet_ll:-52.64620
[3] Train-dirichlet_ll:-56.36977
[4] Train-dirichlet_ll:-59.33048
[5] Train-dirichlet_ll:-61.93359
[6] Train-dirichlet_ll:-64.17280
[7] Train-dirichlet_ll:-66.29709
[8] Train-dirichlet_ll:-68.21001
[9] Train-dirichlet_ll:-70.03442
```

One can confirm that the obtained `yhat` resembles the actual concentrations to a large degree, beyond what would be expected from random predictions by a simple look at both `yhat` and `Y`.

For better results, one might want to add an intercept. XGBoost only allows using scalars for intercepts, but for a vector-valued model, the optimal intercept should also have vector form.

This can be done by supplying `base_margin` instead - unlike the intercept, one must specifically supply values for every row here, and said `base_margin` must be supplied again at the moment of making predictions (i.e. does not get added automatically like `base_score` does).

For the case of a Dirichlet model, the optimal intercept can be obtained efficiently using a general solver (e.g. SciPy's Newton solver) with dedicated likelihood, gradient and Hessian functions for just the intercept part. Further, note that if one frames it instead as bounded optimization without applying 'exp' transform to the concentrations, it becomes instead a convex problem, for which the true Hessian can be used without issues in other classes of solvers.

For simplicity, this example will nevertheless reuse the same likelihood and gradient functions that were defined earlier alongside with SciPy's / R's L-BFGS solver to obtain the optimal vector-valued intercept:

Python

R

```
from scipy.optimize import minimize

def get_optimal_intercepts(Y: np.ndarray) -> np.ndarray:
    k = Y.shape[1]
    res = minimize(
        fun=lambda pred: dirichlet_fun(
            np.broadcast_to(pred, (Y.shape[0], k)),
            Y
        ),
        x0=np.zeros(k),
        jac=lambda pred: dirichlet_grad(
            np.broadcast_to(pred, (Y.shape[0], k)),
            Y
        ).sum(axis=0)
    )
    return res["x"]
intercepts = get_optimal_intercepts(Y)
```

```
get.optimal.intercepts <- function(y) {
    k <- ncol(y)
    broadcast.vec <- function(x) rep(x, nrow(y)) |> matrix(ncol=k, byrow=T)
    res <- optim(
        par = numeric(k),
        fn = function(x) dirichlet.fun(broadcast.vec(x), y),
        gr = function(x) dirichlet.grad(broadcast.vec(x), y) |> colSums(),
        method = "L-BFGS-B"
    )
    return(res$par)
}
intercepts <- get.optimal.intercepts(y)
```

Now fitting a model again, this time with the intercept:

Python

R

```
base_margin = np.broadcast_to(intercepts, Y.shape)
dtrain_w_intercept = xgb.DMatrix(X, label=Y, base_margin=base_margin)
results: Dict[str, Dict[str, List[float]]] = {}
booster = xgb.train(
    params={
        "tree_method": "hist",
        "num_target": Y.shape[1],
        "base_score": 0,
        "disable_default_eval_metric": True,
        "max_depth": 3,
        "seed": 123,
    },
    dtrain=dtrain_w_intercept,
    num_boost_round=10,
    obj=dirichlet_xgb_objective,
```

(continues on next page)

(continued from previous page)

```
    evals=[(dtrain, "Train")],
    evals_result=results,
    custom_metric=dirichlet_eval_metric,
    verbose_eval=False,
)
yhat = softmax(
    booster.predict(
        xgb.DMatrix(X, base_margin=base_margin)
    ),
    axis=1
)
```

```
base.margin <- rep(intercepts, nrow(y)) |> matrix(nrow=nrow(y), byrow=T)
dtrain <- xgb.DMatrix(x, y, base_margin=base.margin)
booster <- xgb.train(
    params = list(
        tree_method="hist",
        num_target=ncol(y),
        base_score=0,
        disable_default_eval_metric=TRUE,
        max_depth=3,
        seed=123
    ),
    data = dtrain,
    nrounds = 10,
    obj = dirichlet.xgb.objective,
    evals = list(Train=dtrain),
    eval_metric = dirichlet.eval.metric
)
raw.pred <- predict(
    booster,
    x,
    base_margin=base.margin,
    reshape=TRUE
)
yhat <- apply(raw.pred, 1, softmax) |> t()
```

```
[0] Train-dirichlet_ll:-37.01861
[1] Train-dirichlet_ll:-42.86120
[2] Train-dirichlet_ll:-46.55133
[3] Train-dirichlet_ll:-49.15111
[4] Train-dirichlet_ll:-51.02638
[5] Train-dirichlet_ll:-52.53880
[6] Train-dirichlet_ll:-53.77409
[7] Train-dirichlet_ll:-54.88851
[8] Train-dirichlet_ll:-55.95961
[9] Train-dirichlet_ll:-56.95497
```

For this small example problem, predictions should be very similar between the two and the version without intercepts achieved a lower objective function in the training data (for the Python version at least), but for more serious usage with real-world data, one is likely to observe better results when adding the intercepts.

1.4.22 Intercept

Added in version 2.0.0.

Since 2.0.0, XGBoost supports estimating the model intercept (named `base_score`) automatically based on targets upon training. The behavior can be controlled by setting `base_score` to a constant value. The following snippet disables the automatic estimation:

Python

R

```python
import xgboost as xgb

clf = xgb.XGBClassifier(n_estimators=10)
clf.set_params(base_score=0.5)
```

```r
library(xgboost)

# Load built-in dataset
data(agaricus.train, package = "xgboost")

# Set base_score parameter directly
model <- xgboost(
  x = agaricus.train$data,
  y = factor(agaricus.train$label),
  base_score = 0.5,
  nrounds = 10
)
```

In addition, here 0.5 represents the value after applying the inverse link function. See the end of the document for a description.

Other than the `base_score`, users can also provide global bias via the data field `base_margin`, which is a vector or a matrix depending on the task. With multi-output and multi-class, the `base_margin` is a matrix with size `(n_samples, n_targets)` or `(n_samples, n_classes)`.

Python

R

```python
import xgboost as xgb
from sklearn.datasets import make_classification

X, y = make_classification()

clf = xgb.XGBClassifier()
clf.fit(X, y)
# Request for raw prediction
m = clf.predict(X, output_margin=True)

clf_1 = xgb.XGBClassifier()
# Feed the prediction into the next model
# Using base margin overrides the base score, see below sections.
clf_1.fit(X, y, base_margin=m)
clf_1.predict(X, base_margin=m)
```

```r
library(xgboost)

# Load built-in dataset
data(agaricus.train, package = "xgboost")

# Train first model
model_1 <- xgboost(
  x = agaricus.train$data,
  y = factor(agaricus.train$label),
  nrounds = 10
)

# Request for raw prediction
m <- predict(model_1, agaricus.train$data, type = "raw")

# Feed the prediction into the next model using base_margin
# Using base margin overrides the base score, see below sections.
model_2 <- xgboost(
  x = agaricus.train$data,
  y = factor(agaricus.train$label),
  base_margin = m,
  nrounds = 10
)

# Make predictions with base_margin
pred <- predict(model_2, agaricus.train$data, base_margin = m)
```

It specifies the bias for each sample and can be used for stacking an XGBoost model on top of other models, see *Demo for boosting from prediction* for a worked example. When `base_margin` is specified, it automatically overrides the `base_score` parameter. If you are stacking XGBoost models, then the usage should be relatively straightforward, with the previous model providing raw prediction and a new model using the prediction as bias. For more customized inputs, users need to take extra care of the link function. Let F be the model and g be the link function, since `base_score` is overridden when sample-specific `base_margin` is available, we will omit it here:

$$g(E[y_i]) = F(x_i)$$

When base margin b is provided, it's added to the raw model output F:

$$g(E[y_i]) = F(x_i) + b_i$$

and the output of the final model is:

$$g^{-1}(F(x_i) + b_i)$$

Using the gamma deviance objective `reg:gamma` as an example, which has a log link function, hence:

$$\ln(E[y_i]) = F(x_i) + b_i$$
$$E[y_i] = \exp(F(x_i) + b_i)$$

As a result, if you are feeding outputs from models like GLM with a corresponding objective function, make sure the outputs are not yet transformed by the inverse link (activation).

In the case of `base_score` (intercept), it can be accessed through `save_config()` after estimation. Unlike the `base_margin`, the returned value represents a value after applying inverse link. With logistic regression and the logit

link function as an example, given the `base_score` as 0.5, $g(intercept) = logit(0.5) = 0$ is added to the raw model output:

$$E[y_i] = g^{-1}(F(x_i) + g(intercept))$$

and 0.5 is the same as $base_score = g^{-1}(0) = 0.5$. This is more intuitive if you remove the model and consider only the intercept, which is estimated before the model is fitted:

$$E[y] = g^{-1}(g(intercept))$$
$$E[y] = intercept$$

For some objectives like MAE, there are close solutions, while for others it's estimated with one step Newton method.

Offset

The `base_margin` is a form of `offset` in GLM. Using the Poisson objective as an example, we might want to model the rate instead of the count:

$$rate = \frac{count}{exposure}$$

And the offset is defined as log link applied to the exposure variable: $\ln exposure$. Let c be the count and γ be the exposure, substituting the response y in our previous formulation of base margin:

$$g(\frac{E[c_i]}{\gamma_i}) = F(x_i)$$

Substitute g with $\ln$ for Poisson regression:

$$\ln \frac{E[c_i]}{\gamma_i} = F(x_i)$$

We have:

$$E[c_i] = \exp{(F(x_i) + \ln{\gamma_i})}$$
$$E[c_i] = g^{-1}(F(x_i) + g(\gamma_i))$$

As you can see, we can use the `base_margin` for modeling with offset similar to GLMs

Example

The following example shows the relationship between `base_score` and `base_margin` using binary logistic with a *logit* link function:

Python

R

```
import numpy as np
from scipy.special import logit
from sklearn.datasets import make_classification

import xgboost as xgb

X, y = make_classification(random_state=2025)
```

```
library(xgboost)

# Load built-in dataset
data(agaricus.train, package = "xgboost")
X <- agaricus.train$data
y <- agaricus.train$label
```

The intercept is a valid probability (0.5). It's used as the initial estimation of the probability of obtaining a positive sample.

Python

R

```
intercept = 0.5
```

```
intercept <- 0.5
```

First we use the intercept to train a model:

Python

R

```
booster = xgb.train(
    {"base_score": intercept, "objective": "binary:logistic"},
    dtrain=xgb.DMatrix(X, y),
    num_boost_round=1,
)
predt_0 = booster.predict(xgb.DMatrix(X, y))
```

```
# First model with base_score
model_0 <- xgboost(
  x = X, y = factor(y),
  base_score = intercept,
  objective = "binary:logistic",
  nrounds = 1
)
predt_0 <- predict(model_0, X)
```

Apply `logit()` to obtain the "margin":

Python

R

```
# Apply logit function to obtain the "margin"
margin = np.full(y.shape, fill_value=logit(intercept), dtype=np.float32)
Xy = xgb.DMatrix(X, y, base_margin=margin)
# Second model with base_margin
# 0.2 is a dummy value to show that `base_margin` overrides `base_score`.
booster = xgb.train(
    {"base_score": 0.2, "objective": "binary:logistic"},
    dtrain=Xy,
    num_boost_round=1,
```

(continues on next page)

(continued from previous page)

```
)
predt_1 = booster.predict(Xy)
```

```
# Apply logit function to obtain the "margin"
logit_intercept <- log(intercept / (1 - intercept))
margin <- rep(logit_intercept, length(y))
# Second model with base_margin
# 0.2 is a dummy value to show that `base_margin` overrides `base_score`
model_1 <- xgboost(
  x = X, y = factor(y),
  base_margin = margin,
  base_score = 0.2,
  objective = "binary:logistic",
  nrounds = 1
)
predt_1 <- predict(model_1, X, base_margin = margin)
```

Compare the results:

Python

R

```
np.testing.assert_allclose(predt_0, predt_1)
```

```
all.equal(predt_0, predt_1, tolerance = 1e-6)
```

1.4.23 Privacy Preserving Inference with Concrete ML

Concrete ML is a specialized library developed by Zama that allows the execution of machine learning models on encrypted data through Fully Homomorphic Encryption (FHE), thereby preserving data privacy.

To use models such as XGBClassifier, use the following import:

```
from concrete.ml.sklearn import XGBClassifier
```

Performing Privacy Preserving Inference

Initialization of a XGBClassifier can be done as follows:

```
classifier = XGBClassifier(n_bits=6, [other_hyperparameters])
```

where `n_bits` determines the precision of the input features. Note that a higher value of `n_bits` increases the precision of the input features and possibly the final model accuracy but also ends up with longer FHE execution time.

Other hyper-parameters that exist in xgboost library can be used.

Model Training and Compilation

As commonly used in scikit-learn like models, it can be trained with the .fit() method.

```
classifier.fit(X_train, y_train)
```

After training, the model can be compiled with a calibration dataset, potentially a subset of the training data:

```
classifier.compile(X_calibrate)
```

This calibration dataset, `X_calibrate`, is used in Concrete ML compute the precision (bit-width) of each intermediate value in the model. This is a necessary step to optimize the equivalent FHE circuit.

FHE Simulation and Execution

To verify model accuracy in encrypted computations, you can run an FHE simulation:

```
predictions = classifier.predict(X_test, fhe="simulate")
```

This simulation can be used to evaluate the model. The resulting accuracy of this simulation step is representative of the actual FHE execution without having to pay the cost of an actual FHE execution.

When the model is ready, actual Fully Homomorphic Encryption execution can be performed:

```
predictions = classifier.predict(X_test, fhe="execute")
```

Note that using FHE="execute" is a convenient way to assess the model in FHE, but for real deployment, functions to encrypt (on the client), run in FHE (on the server), and finally decrypt (on the client) have to be used for end-to-end privacy-preserving inferences.

Concrete ML provides a deployment API to facilitate this process, ensuring end-to-end privacy.

To go further in the deployment API you can read:

- the deployment documentation
- the deployment notebook

Parameter Tuning in Concrete ML

Concrete ML is compatible with standard scikit-learn pipelines such as GridSearchCV or any other hyper-parameter tuning techniques.

Examples and Demos

- Sentiment analysis (based on transformers + xgboost)
- XGBoost Classifier
- XGBoost Regressor

Conclusion

Concrete ML provides a framework for executing privacy-preserving inferences by leveraging Fully Homomorphic Encryption, allowing secure and private computations on encrypted data.

More information and examples are given in the Concrete ML documentation.

1.5 Frequently Asked Questions

This document contains frequently asked questions about XGBoost.

1.5.1 How to tune parameters

See *Parameter Tuning Guide*.

1.5.2 Description of the model

See *Introduction to Boosted Trees*.

1.5.3 I have a big dataset

XGBoost is designed to be memory efficient. Usually it can handle problems as long as the data fits into your memory. This usually means millions of instances.

If you are running out of memory, checkout the tutorial page for using *distributed training* with one of the many frameworks, or the *external memory version* for using external memory.

1.5.4 How to handle categorical feature?

Visit *this tutorial* for a walkthrough of categorical data handling and some worked examples.

1.5.5 Why not implement distributed XGBoost on top of X (Spark, Hadoop)?

The first fact we need to know is going distributed does not necessarily solve all the problems. Instead, it creates more problems such as more communication overhead and fault tolerance. The ultimate question will still come back to how to push the limit of each computation node and use less resources to complete the task (thus with less communication and chance of failure).

To achieve these, we decide to reuse the optimizations in the single node XGBoost and build the distributed version on top of it. The demand for communication in machine learning is rather simple, in the sense that we can depend on a limited set of APIs. Such design allows us to reuse most of the code, while being portable to major platforms such as Hadoop/Yarn, MPI, SGE. Most importantly, it pushes the limit of the computation resources we can use.

1.5.6 How can I port a model to my own system?

The model and data format of XGBoost are exchangeable, which means the model trained by one language can be loaded in another. This means you can train the model using R, while running prediction using Java or C++, which are more common in production systems. You can also train the model using distributed versions, and load them in from Python to do some interactive analysis. See *Model IO* for more information.

1.5.7 Do you support LambdaMART?

Yes, XGBoost implements LambdaMART. Checkout the objective section in *parameters*.

1.5.8 How to deal with missing values

XGBoost supports missing values by default. In tree algorithms, branch directions for missing values are learned during training. Note that the gblinear booster treats missing values as zeros.

When the `missing` parameter is specified, values in the input predictor that is equal to `missing` will be treated as missing and removed. By default it's set to `NaN`.

1.5.9 Slightly different result between runs

This could happen, due to non-determinism in floating point summation order and multi-threading. Also, data partitioning changes by distributed framework can be an issue as well. Though the general accuracy will usually remain the same.

1.5.10 Why do I see different results with sparse and dense data?

"Sparse" elements are treated as if they were "missing" by the tree booster, and as zeros by the linear booster. However, if we convert the sparse matrix back to dense matrix, the sparse matrix might fill the missing entries with 0, which is a valid value for xgboost. In short, sparse matrix implementations like scipy treats 0 as missing, while 0 is a valid split value for XGBoost decision trees.

1.6 XGBoost GPU Support

This page contains information about GPU algorithms supported in XGBoost.

Note

CUDA 12.0, Compute Capability 5.0 required (See this list to look up compute capability of your GPU card.)

1.6.1 CUDA Accelerated Tree Construction Algorithms

Most of the algorithms in XGBoost including training, prediction and evaluation can be accelerated with CUDA-capable GPUs.

Usage

To enable GPU acceleration, specify the `device` parameter as `cuda`. In addition, the device ordinal (which GPU to use if you have multiple devices in the same node) can be specified using the `cuda:<ordinal>` syntax, where `<ordinal>` is an integer that represents the device ordinal. XGBoost defaults to 0 (the first device reported by CUDA runtime).

The GPU algorithms currently work with CLI, Python, R, and JVM packages. See *Installation Guide* for details.

Listing 24: Python example

```
params = dict()
params["device"] = "cuda"
params["tree_method"] = "hist"
Xy = xgboost.QuantileDMatrix(X, y)
xgboost.train(params, Xy)
```

Listing 25: With the Scikit-Learn interface

```
XGBRegressor(tree_method="hist", device="cuda")
```

GPU-Accelerated SHAP values

XGBoost uses its in-tree `QuadratureTreeSHAP` implementation for computing SHAP values on both CPU and GPU. The GPU path uses the same Quadrature-TreeSHAP formulation described by Wettenstein et al. (2026) for exact TreeSHAP feature attributions when the GPU is selected.

```
booster.set_param({"device": "cuda:0"})
shap_values = booster.predict(dtrain, pred_contribs=True)
shap_interaction_values = model.predict(dtrain, pred_interactions=True)
```

See *Use GPU to speedup SHAP value computation* for a worked example.

Multi-node Multi-GPU Training

XGBoost supports fully distributed GPU training using Dask, `Spark` and `PySpark`. For getting started with Dask see our tutorial *Distributed XGBoost with Dask* and worked examples *XGBoost Dask Feature Walkthrough*, also Python documentation *Dask API* for complete reference. For usage with `Spark` using Scala see *XGBoost4J-Spark-GPU Tutorial*. Lastly for distributed GPU training with `PySpark`, see *Distributed XGBoost with PySpark*.

RMM integration

XGBoost provides optional support for RMM integration. See *Using XGBoost with RAPIDS Memory Manager (RMM) plugin* for more info.

Memory usage

The following are some guidelines on the device memory usage of the `hist` tree method on GPU.

Memory inside xgboost training is generally allocated for two reasons - storing the dataset and working memory.

The dataset itself is stored on device in a compressed ELLPACK format. The ELLPACK format is a type of sparse matrix that stores elements with a constant row stride. This format is convenient for parallel computation when compared to CSR because the row index of each element is known directly from its address in memory. The disadvantage of the ELLPACK format is that it becomes less memory efficient if the maximum row length is significantly more than the average row length. Elements are quantised and stored as integers. These integers are compressed to a minimum bit length. Depending on the number of features, we usually don't need the full range of a 32 bit integer to store elements and so compress this down. The compressed, quantised ELLPACK format will commonly use 1/4 the space of a CSR matrix stored in floating point.

Working memory is allocated inside the algorithm proportional to the number of rows to keep track of gradients, tree positions and other per row statistics. Memory is allocated for histogram bins proportional to the number of bins, number of features and nodes in the tree. For performance reasons we keep histograms in memory from previous nodes in the tree, when a certain threshold of memory usage is passed we stop doing this to conserve memory at some performance loss.

If you are getting out-of-memory errors on a big dataset, try the `xgboost.QuantileDMatrix` first. If you have access to NVLink-C2C devices, see *external memory version*. In addition, `inplace_predict()` should be preferred over `predict` when data is already on GPU. Both `xgboost.QuantileDMatrix` and `inplace_predict()` are automatically enabled if you are using the scikit-learn interface. Last but not least, using `QuantileDMatrix` with a data iterator as input is a great way to increase memory capacity, see *Demo for using data iterator with Quantile DMatrix*.

CPU-GPU Interoperability

The model can be used on any device regardless of the one used to train it. For instance, a model trained using GPU can still work on a CPU-only machine and vice versa. For more information about model serialization, see *Introduction to Model IO*.

Developer notes

The application may be profiled with annotations by specifying `USE_NTVX` to cmake. Regions covered by the 'Monitor' class in CUDA code will automatically appear in the nsight profiler when *verbosity* is set to 3.

1.6.2 References

Mitchell R, Frank E. (2017) Accelerating the XGBoost algorithm using GPU computing. PeerJ Computer Science 3:e127 https://doi.org/10.7717/peerj-cs.127

Lundberg SM, Erion GG, Lee S-I. (2018) Consistent Individualized Feature Attribution for Tree Ensembles. arXiv:1802.03888

Wettenstein R, Mitchell R, Yu P. (2026) Quadrature-TreeSHAP: Depth-Independent TreeSHAP and Shapley Interactions. arXiv:2605.04497

NVIDIA Parallel Forall: Gradient Boosting, Decision Trees and XGBoost with CUDA

Out-of-Core GPU Gradient Boosting

Contributors

Many thanks to the following contributors (alphabetical order):

- Andrey Adinets
- Jiaming Yuan
- Jonathan C. McKinney
- Matthew Jones
- Philip Cho
- Rong Ou
- Rory Mitchell
- Shankara Rao Thejaswi Nanditale
- Sriram Chandramouli
- Vinay Deshpande

Please report bugs to the XGBoost issues list.

1.7 XGBoost Parameters

Before running XGBoost, we must set three types of parameters: general parameters, booster parameters and task parameters.

- **General parameters** relate to which booster we are using to do boosting, commonly tree or linear model
- **Booster parameters** depend on which booster you have chosen
- **Learning task parameters** decide on the learning scenario. For example, regression tasks may use different parameters with ranking tasks.

Note

Parameters in R package

In R-package, you can use `.` (dot) to replace underscore in the parameters, for example, you can use `max.depth` to indicate `max_depth`. The underscore parameters are also valid in R.

- *Global Configuration*
- *General Parameters*
  - *Parameters for Tree Booster*
  - *Parameters for Non-Exact Tree Methods*
  - *Parameters for Categorical Feature*

- *Additional dropout parameters for tree boosters*
- *Parameters for Linear Booster (`booster=gblinear`)*

- *Learning Task Parameters*
  - *Parameters for Tweedie Regression (`objective=reg:tweedie`)*
  - *Parameter for using Pseudo-Huber (`reg:pseudohubererror`)*
  - *Parameter for using Quantile Loss (`reg:quantileerror`)*
  - *Parameter for using Expectile Loss (`reg:expectileerror`)*
  - *Parameter for using AFT Survival Loss (`survival:aft`) and Negative Log Likelihood of AFT metric (`aft-nloglik`)*
  - *Parameters for learning to rank (`rank:ndcg`, `rank:map`, `rank:pairwise`)*

1.7.1 Global Configuration

The following parameters can be set in the global scope, using *`xgboost.config_context()`* (Python) or `xgb.set.config()` (R).

- `verbosity`: Verbosity of printing messages. Valid values of 0 (silent), 1 (warning), 2 (info), and 3 (debug).
- `use_rmm`: Whether to use RAPIDS Memory Manager (RMM) to allocate cache GPU memory. The primary memory is always allocated on the RMM pool when XGBoost is built (compiled) with the RMM plugin enabled. Valid values are `true` and `false`. See *Using XGBoost with RAPIDS Memory Manager (RMM) plugin* for details.
- `use_cuda_async_pool` [default=false]

  Whether to use the device memory pool in the CUDA driver. This option is not available if XGBoost is built with RMM support, as it is the same as using the RMM *CudaAsyncMemoryResource* pool.

  Added in version 3.2.0.

  Warning

  This is an experimental feature and is subject to change without notice. Windows is not supported yet.

- `nthread`: Set the global number of threads for OpenMP. Use this only when you need to override some OpenMP-related environment variables like `OMP_NUM_THREADS`. Otherwise, the `nthread` parameter from the Booster and the DMatrix should be preferred as the former sets the global variable and might cause conflicts with other libraries.

1.7.2 General Parameters

- `booster` [default= `gbtree`]
  - Which booster to use. Can be `gbtree`, `gblinear` or `dart`; `gbtree` and `dart` use tree based models while `gblinear` uses linear functions.
  - Dropout parameters like `rate_drop` can be used directly with tree models. `booster=dart` remains supported for compatibility.

  Deprecated since version 3.3.0: `booster=gblinear` is deprecated and support will be removed in a future release.

- `device` [default= `cpu`]

  Added in version 2.0.0.

  - Device for XGBoost to run. User can set it to one of the following values:
    - `cpu`: Use CPU.
    - `cuda`: Use a GPU (CUDA device).
    - `cuda:<ordinal>`: `<ordinal>` is an integer that specifies the ordinal of the GPU (which GPU do you want to use if you have more than one devices).
    - `gpu`: Default GPU device selection from the list of available and supported devices. Only `cuda` devices are supported currently.
    - `gpu:<ordinal>`: Default GPU device selection from the list of available and supported devices. Only `cuda` devices are supported currently.

    For more information about GPU acceleration, see *XGBoost GPU Support*. In distributed environments, ordinal selection is handled by distributed frameworks instead of XGBoost. As a result, using `cuda:<ordinal>` will result in an error. Use `cuda` instead.

- `verbosity` [default=1]
  - Verbosity of printing messages. Valid values are 0 (silent), 1 (warning), 2 (info), 3 (debug). Sometimes XGBoost tries to change configurations based on heuristics, which is displayed as warning message. If there's unexpected behaviour, please try to increase value of verbosity.
- `validate_parameters` [default to `false`, except for Python, R and CLI interface]
  - When set to True, XGBoost will perform validation of input parameters to check whether a parameter is used or not. A warning is emitted when there's unknown parameter.
- `nthread` [default to maximum number of threads available if not set]
  - Number of parallel threads used to run XGBoost. When choosing it, please keep thread contention and hyperthreading in mind.
- `disable_default_eval_metric` [default= `false`]
  - Flag to disable default metric. Set to 1 or `true` to disable.

Parameters for Tree Booster

- `eta` [default=0.3, alias: `learning_rate`]
  - Step size shrinkage used in update to prevent overfitting. After each boosting step, we can directly get the weights of new features, and `eta` shrinks the feature weights to make the boosting process more conservative.
  - range: [0,1]
- `gamma` [default=0, alias: `min_split_loss`]
  - Minimum loss reduction required to make a further partition on a leaf node of the tree. The larger `gamma` is, the more conservative the algorithm will be. Note that a tree where no splits were made might still contain a single terminal node with a non-zero score. This is the same γ described in the *Introduction to Boosted Trees*.
  - range: [0,∞]
- `max_depth` [default=6, type=int32]

- Maximum depth of a tree. Increasing this value will make the model more complex and more likely to overfit. 0 indicates no limit on depth. Beware that XGBoost aggressively consumes memory when training a deep tree. `exact` tree method requires non-zero value.
- range: $[0,\infty]$

• `min_child_weight` [default=1]

- Minimum sum of instance weight (hessian) needed in a child. If the tree partition step results in a leaf node with the sum of instance weight less than `min_child_weight`, then the building process will give up further partitioning. In linear regression task, this simply corresponds to minimum number of instances needed to be in each node. The larger `min_child_weight` is, the more conservative the algorithm will be.
- range: $[0,\infty]$

• `max_delta_step` [default=0]

- Maximum delta step we allow each leaf output to be. If the value is set to 0, it means there is no constraint. If it is set to a positive value, it can help making the update step more conservative. Usually this parameter is not needed, but it might help in logistic regression when class is extremely imbalanced. Set it to value of 1-10 might help control the update.
- range: $[0,\infty]$

• `subsample` [default=1]

- Subsample ratio of the training instances. Setting it to 0.5 means that XGBoost would randomly sample half of the training data prior to growing trees. and this will prevent overfitting. Subsampling will occur once in every boosting iteration.
- range: (0,1]

• `sampling_method` [default= `uniform`]

Changed in version 3.2.0:

XGBoost supports both CPU and GPU for gradient-based sampling.

• The method to use to sample the training instances.

• `uniform`: each training instance has an equal probability of being selected. Typically set `subsample` >= 0.5 for good results.

• `gradient_based`: the selection probability for each training instance is proportional to the *regularized absolute value* of gradients (more specifically, $\sqrt{g^2 + \lambda h^2}$). `subsample` may be set to as low as 0.1 without loss of model accuracy. Note that this sampling method is only supported when `tree_method` is set to `hist`; other tree methods only support `uniform` sampling.

Note

When working with reduced gradient for multi-target models, the accuracy of gradient-based sampling might be sub-optimal. The sampling is performed using the split gradient, which may not be optimal with the full gradient. Use uniform sampling as an alternative.

• `colsample_bytree`, `colsample_bylevel`, `colsample_bynode` [default=1]

- This is a family of parameters for subsampling of columns.
- All `colsample_by*` parameters have a range of (0, 1], the default value of 1, and specify the fraction of columns to be subsampled.
- `colsample_bytree` is the subsample ratio of columns when constructing each tree. Subsampling occurs once for every tree constructed.

- colsample_bylevel is the subsample ratio of columns for each level. Subsampling occurs once for every new depth level reached in a tree. Columns are subsampled from the set of columns chosen for the current tree.
- colsample_bynode is the subsample ratio of columns for each node (split). Subsampling occurs once every time a new split is evaluated. Columns are subsampled from the set of columns chosen for the current level. This is not supported by the exact tree method.
- colsample_by* parameters work cumulatively. For instance, the combination {'colsample_bytree':0.5, 'colsample_bylevel':0.5, 'colsample_bynode':0.5} with 64 features will leave 8 features to choose from at each split.

  Using the Python or the R package, one can set the feature_weights for DMatrix to define the probability of each feature being selected when using column sampling. There's a similar parameter for fit method in sklearn interface.

- lambda [default=1, alias: reg_lambda]
  - L2 regularization term on weights. Increasing this value will make model more conservative. This is the λ described in the *Introduction to Boosted Trees*.
  - range: [0, ∞]
- alpha [default=0, alias: reg_alpha]
  - L1 regularization term on weights. Increasing this value will make model more conservative.
  - range: [0, ∞]
- tree_method string [default= auto]
  - The tree construction algorithm used in XGBoost. See description in the reference paper and *Tree Methods*.
  - Choices: auto, exact, approx, hist, this is a combination of commonly used updaters. For other updaters like refresh, set the parameter updater directly.
    - auto: Same as the hist tree method.
    - exact: Exact greedy algorithm. Enumerates all split candidates.
    - approx: Approximate greedy algorithm using quantile sketch and gradient histogram.
    - hist: Faster histogram optimized approximate greedy algorithm.
- scale_pos_weight [default=1]
  - Control the balance of positive and negative weights, useful for unbalanced classes. A typical value to consider: sum(negative instances) / sum(positive instances). See *Parameters Tuning* for more discussion. Also, see Higgs Kaggle competition demo for examples: R, py1, py2, py3.
- updater
  - A comma separated string defining the sequence of tree updaters to run, providing a modular way to construct and to modify the trees. This is an advanced parameter that is usually set automatically, depending on some other parameters. However, it could be also set explicitly by a user. The following updaters exist:
    - grow_colmaker: non-distributed column-based construction of trees.
    - grow_histmaker: distributed tree construction with row-based data splitting based on global proposal of histogram counting.
    - grow_quantile_histmaker: Grow tree using quantized histogram.
    - grow_gpu_hist: Enabled when tree_method is set to hist along with device=cuda.
    - grow_gpu_approx: Enabled when tree_method is set to approx along with device=cuda.

- `sync`: synchronizes trees in all distributed nodes.
- `refresh`: refreshes tree's statistics and/or leaf values based on the current data. Note that no random subsampling of data rows is performed.
- `prune`: prunes the splits where loss < min_split_loss (or gamma) and nodes that have depth greater than `max_depth`.

- `refresh_leaf` [default=1]
  - This is a parameter of the `refresh` updater. When this flag is 1, tree leafs as well as tree nodes' stats are updated. When it is 0, only node stats are updated.
- `process_type` [default= `default`]
  - A type of boosting process to run.
  - Choices: `default`, `update`
    - `default`: The normal boosting process which creates new trees.
    - `update`: Starts from an existing model and only updates its trees. In each boosting iteration, a tree from the initial model is taken, a specified sequence of updaters is run for that tree, and a modified tree is added to the new model. The new model would have either the same or smaller number of trees, depending on the number of boosting iterations performed. Currently, the following built-in updaters could be meaningfully used with this process type: `refresh`, `prune`. With `process_type=update`, one cannot use updaters that create new trees.
- `grow_policy` [default= `depthwise`]
  - Controls a way new nodes are added to the tree.
  - Currently supported only if `tree_method` is set to `hist` or `approx`.
  - Choices: `depthwise`, `lossguide`
    - `depthwise`: split at nodes closest to the root.
    - `lossguide`: split at nodes with highest loss change.
- `max_leaves` [default=0, type=int32]
  - Maximum number of nodes to be added. Not used by `exact` tree method.
- `max_bin`, [default=256, type=int32]
  - Only used if `tree_method` is set to `hist` or `approx`.
  - Maximum number of discrete bins to bucket continuous features.
  - Increasing this number improves the optimality of splits at the cost of higher computation time.
- `num_parallel_tree`, [default=1]
  - Number of parallel trees constructed during each iteration. This option is used to support boosted random forest.
- `monotone_constraints`
  - Constraint of variable monotonicity. See *Monotonic Constraints* for more information.
- `interaction_constraints`
  - Constraints for interaction representing permitted interactions. The constraints must be specified in the form of a nest list, e.g. `[[0, 1], [2, 3, 4]]`, where each inner list is a group of indices of features that are allowed to interact with each other. See *Feature Interaction Constraints* for more information.

- `multi_strategy`, [default = `one_output_per_tree`]

  Added in version 2.0.0.

  > **Note**
  >
  > This parameter is working-in-progress.

  - The strategy used for training multi-target models, including multi-target regression and multi-class classification. See *Multiple Outputs* for more information.
    - `one_output_per_tree`: One model for each target.
    - `multi_output_tree`: Use multi-target trees.

Parameters for Non-Exact Tree Methods

- `max_cached_hist_node`, [default = 65536]

  Maximum number of cached nodes for histogram. This can be used with the `hist` and the `approx` tree methods.

  Added in version 2.0.0.

  - For most of the cases this parameter should not be set except for growing deep trees. After 3.0, this parameter affects GPU algorithms as well.

Parameters for Categorical Feature

These parameters are only used for training with categorical data. See *Categorical Data* for more information.

> **Note**
>
> The `exact` tree method is not supported for categorical features.

- `max_cat_to_onehot`

  Added in version 1.6.0.

  - A threshold for deciding whether XGBoost should use one-hot encoding based split for categorical data. When number of categories is lesser than the threshold then one-hot encoding is chosen, otherwise the categories will be partitioned into children nodes.

- `max_cat_threshold`

  Added in version 1.7.0.

  - Maximum number of categories considered for each split. Used only by partition-based splits for preventing over-fitting.

Additional dropout parameters for tree boosters

- `sample_type` [default= `uniform`]
  - Type of sampling algorithm.
    - `uniform`: dropped trees are selected uniformly.
    - `weighted`: dropped trees are selected in proportion to weight.
- `normalize_type` [default= `tree`]
  - Type of normalization algorithm.

    * `tree`: new trees have the same weight of each of dropped trees.
        · Weight of new trees are `1 / (k + learning_rate)`.
        · Dropped trees are scaled by a factor of `k / (k + learning_rate)`.
    * `forest`: new trees have the same weight of sum of dropped trees (forest).
        · Weight of new trees are `1 / (1 + learning_rate)`.
        · Dropped trees are scaled by a factor of `1 / (1 + learning_rate)`.

- `rate_drop` [default=0.0]
  - Dropout rate (a fraction of previous trees to drop during the dropout).
  - range: [0.0, 1.0]
- `one_drop` [default=0]
  - When this flag is enabled, at least one tree is always dropped during the dropout (allows Binomial-plus-one or epsilon-dropout from the original DART paper).
- `skip_drop` [default=0.0]
  - Probability of skipping the dropout procedure during a boosting iteration.
    * If a dropout is skipped, new trees are added in the same manner as `gbtree`.
    * Note that non-zero `skip_drop` has higher priority than `rate_drop` or `one_drop`.
  - range: [0.0, 1.0]

Parameters for Linear Booster (`booster=gblinear`)

Deprecated since version 3.3.0: `booster=gblinear` is deprecated and support will be removed in a future release.

- `lambda` [default=0, alias: `reg_lambda`]
  - L2 regularization term on weights. Increasing this value will make model more conservative. Normalised to number of training examples.
- `alpha` [default=0, alias: `reg_alpha`]
  - L1 regularization term on weights. Increasing this value will make model more conservative. Normalised to number of training examples.
- `eta` [default=0.5, alias: `learning_rate`]
  - Step size shrinkage used in update to prevent overfitting. After each boosting step, we can directly get the weights of new features, and `eta` shrinks the feature weights to make the boosting process more conservative.
  - range: [0,1]
- `updater` [default= `shotgun`]
  - Choice of algorithm to fit linear model
    * `shotgun`: Parallel coordinate descent algorithm based on shotgun algorithm. Uses 'hogwild' parallelism and therefore produces a nondeterministic solution on each run.
    * `coord_descent`: Ordinary coordinate descent algorithm. Also multithreaded but still produces a deterministic solution. When the `device` parameter is set to `cuda` or `gpu`, a GPU variant would be used.
- `feature_selector` [default= `cyclic`]

- Feature selection and ordering method
  * `cyclic`: Deterministic selection by cycling through features one at a time.
  * `shuffle`: Similar to `cyclic` but with random feature shuffling prior to each update.
  * `random`: A random (with replacement) coordinate selector.
  * `greedy`: Select coordinate with the greatest gradient magnitude. It has `O(num_feature^2)` complexity. It is fully deterministic. It allows restricting the selection to `top_k` features per group with the largest magnitude of univariate weight change, by setting the `top_k` parameter. Doing so would reduce the complexity to `O(num_feature*top_k)`.
  * `thrifty`: Thrifty, approximately-greedy feature selector. Prior to cyclic updates, reorders features in descending magnitude of their univariate weight changes. This operation is multithreaded and is a linear complexity approximation of the quadratic greedy selection. It allows restricting the selection to `top_k` features per group with the largest magnitude of univariate weight change, by setting the `top_k` parameter.

- `top_k` [default=0]
  - The number of top features to select in `greedy` and `thrifty` feature selector. The value of 0 means using all the features.

1.7.3 Learning Task Parameters

Specify the learning task and the corresponding learning objective. The objective options are below:

- `objective` [default=reg:squarederror]
  - `reg:squarederror`: regression with squared loss.
  - `reg:squaredlogerror`: regression with squared log loss $\frac{1}{2}[log(pred+1) - log(label+1)]^2$. All input labels are required to be greater than -1. Also, see metric `rmsle` for possible issue with this objective.
  - `reg:logistic`: logistic regression, output probability
  - `reg:pseudohubererror`: regression with Pseudo Huber loss, a twice differentiable alternative to absolute loss.
  - `reg:absoluteerror`: Regression with L1 error. When tree model is used, leaf value is refreshed after tree construction. If used in distributed training, the leaf value is calculated as the mean value from all workers, which is not guaranteed to be optimal.

    Added in version 1.7.0.
  - `reg:quantileerror`: Quantile loss, also known as `pinball loss`. See later sections for its parameter and *Prediction Intervals with Quantile and Expectile Regression* for a worked example.

    Added in version 2.0.0.
  - `reg:expectileerror`: Expectile loss (asymmetric squared error). See later sections for its parameter and properties.
  - `binary:logistic`: logistic regression for binary classification, output probability
  - `binary:logitraw`: logistic regression for binary classification, output score before logistic transformation
  - `binary:hinge`: hinge loss for binary classification. This makes predictions of 0 or 1, rather than producing probabilities.
  - `count:poisson`: Poisson regression for count data, output mean of Poisson distribution.
    * `max_delta_step` is set to 0.7 by default in Poisson regression (used to safeguard optimization)

  - `survival:cox`: Cox regression for right censored survival time data (negative values are considered right censored). Note that predictions are returned on the hazard ratio scale (i.e., as HR = exp(marginal_prediction) in the proportional hazard function `h(t) = h0(t) * HR`).
  - `survival:aft`: Accelerated failure time model for censored survival time data. See *Survival Analysis with Accelerated Failure Time* for details.
  - `multi:softmax`: set XGBoost to do multiclass classification using the softmax objective, you also need to set num_class(number of classes)
  - `multi:softprob`: same as softmax, but output a vector of `ndata * nclass`, which can be further reshaped to `ndata * nclass` matrix. The result contains predicted probability of each data point belonging to each class.
  - `rank:ndcg`: Use LambdaMART to perform pair-wise ranking where Normalized Discounted Cumulative Gain (NDCG) is maximized. This objective supports position debiasing for click data.
  - `rank:map`: Use LambdaMART to perform pair-wise ranking where Mean Average Precision (MAP) is maximized
  - `rank:pairwise`: Use LambdaRank to perform pair-wise ranking using the *ranknet* objective.
  - `reg:gamma`: gamma regression with log-link. Output is a mean of gamma distribution. It might be useful, e.g., for modeling insurance claims severity, or for any outcome that might be gamma-distributed.
  - `reg:tweedie`: Tweedie regression with log-link. It might be useful, e.g., for modeling total loss in insurance, or for any outcome that might be Tweedie-distributed.

- `base_score`

  The initial prediction score of all instances, also known as the global bias, or the intercept.

  Changed in version 3.1.0: XGBoost is updated to use vector-valued intercept by default.

  - The parameter is automatically estimated for selected objectives before training. To disable the estimation, specify a real number argument, e.g. `base_score = 0.5`.
  - If `base_margin` is supplied, `base_score` will not be used.
  - If we train the model with a sufficient number of iterations, changing this value does not offer significant benefit.

  See *Intercept* for more information, including different use cases.

- `eval_metric` [default according to objective]

  - Evaluation metrics for validation data, a default metric will be assigned according to objective (rmse for regression, and logloss for classification, *mean average precision* for `rank:map`, etc.)
  - User can add multiple evaluation metrics. Python users: remember to pass the metrics in as list of parameters pairs instead of map, so that latter `eval_metric` won't override previous ones
  - The choices are listed below:
    * `rmse`: root mean square error
    * `rmsle`: root mean square log error: $\sqrt{\frac{1}{N}[log(pred+1)-log(label+1)]^2}$. Default metric of `reg:squaredlogerror` objective. This metric reduces errors generated by outliers in dataset. But because `log` function is employed, `rmsle` might output `nan` when prediction value is less than -1. See `reg:squaredlogerror` for other requirements.
    * `mae`: mean absolute error
    * `mape`: mean absolute percentage error

* `mphe`: mean Pseudo Huber error. Default metric of `reg:pseudohubererror` objective.
* `expectile`: Expectile regression error (asymmetric squared error). Default metric of `reg:expectileerror` objective.
* `logloss`: negative log-likelihood
* `error`: Binary classification error rate. It is calculated as `#(wrong cases)/#(all cases)`. For the predictions, the evaluation will regard the instances with prediction value larger than 0.5 as positive instances, and the others as negative instances.
* `error@t`: a different than 0.5 binary classification threshold value could be specified by providing a numerical value through 't'.
* `merror`: Multiclass classification error rate. It is calculated as `#(wrong cases)/#(all cases)`.
* `mlogloss`: Multiclass logloss.
* `auc`: Receiver Operating Characteristic Area under the Curve. Available for classification and learning-to-rank tasks.
  - When used with binary classification, the objective should be `binary:logistic` or similar functions that work on probability.
  - When used with multi-class classification, objective should be `multi:softprob` instead of `multi:softmax`, as the latter doesn't output probability. Also the AUC is calculated by 1-vs-rest with reference class weighted by class prevalence.
  - When used with LTR task, the AUC is computed by comparing pairs of documents to count correctly sorted pairs. This corresponds to pairwise learning to rank. The implementation has some issues with average AUC around groups and distributed workers not being well-defined.
  - On a single machine the AUC calculation is exact. In a distributed environment the AUC is a weighted average over the AUC of training rows on each node - therefore, distributed AUC is an approximation sensitive to the distribution of data across workers. Use another metric in distributed environments if precision and reproducibility are important.
  - When input dataset contains only negative or positive samples, the output is *NaN*. The behavior is implementation defined, for instance, `scikit-learn` returns 0.5 instead.
* `aucpr`: Area under the PR curve. Available for classification and learning-to-rank tasks.

  After XGBoost 1.6, both of the requirements and restrictions for using `aucpr` in classification problem are similar to `auc`. For ranking task, only binary relevance label $y \in [0, 1]$ is supported. Different from `map (mean average precision)`, `aucpr` calculates the *interpolated* area under precision recall curve using continuous interpolation.
* `pre`: Precision at k. Supports only learning to rank task.
* `ndcg`: Normalized Discounted Cumulative Gain
* `map`: Mean Average Precision

  The *average precision* is defined as:

  $$AP@l = \frac{1}{min(l, N)} \sum_{k=1}^{l} P@k \cdot I_{(k)}$$

  where $I_{(k)}$ is an indicator function that equals to 1 when the document at k is relevant and 0 otherwise. The $P@k$ is the precision at k, and N is the total number of relevant documents. Lastly, the *mean average precision* is defined as the weighted average across all queries.
* `ndcg@n`, `map@n`, `pre@n`: n can be assigned as an integer to cut off the top positions in the lists for evaluation.

    * `ndcg-`, `map-`, `ndcg@n-`, `map@n-`: In XGBoost, the NDCG and MAP evaluate the score of a list without any positive samples as 1. By appending "-" to the evaluation metric name, we can ask XGBoost to evaluate these scores as 0 to be consistent under some conditions.
    * `poisson-nloglik`: negative log-likelihood for Poisson regression
    * `gamma-nloglik`: negative log-likelihood for gamma regression
    * `cox-nloglik`: negative partial log-likelihood for Cox proportional hazards regression
    * `gamma-deviance`: residual deviance for gamma regression
    * `tweedie-nloglik`: negative log-likelihood for Tweedie regression (at a specified value of the `tweedie_variance_power` parameter)
    * `aft-nloglik`: Negative log likelihood of Accelerated Failure Time model. See *Survival Analysis with Accelerated Failure Time* for details.
    * `interval-regression-accuracy`: Fraction of data points whose predicted labels fall in the interval-censored labels. Only applicable for interval-censored data. See *Survival Analysis with Accelerated Failure Time* for details.

- `seed` [default=0]
  - Random number seed. In the R package, if not specified, instead of defaulting to seed 'zero', will take a random seed through R's own RNG engine.
- `seed_per_iteration` [default= `false`]
  - Seed PRNG determnisticly via iterator number.

Parameters for Tweedie Regression (`objective=reg:tweedie`)

- `tweedie_variance_power` [default=1.5]
  - Parameter that controls the variance of the Tweedie distribution `var(y) ~ E(y)^tweedie_variance_power`
  - range: (1,2)
  - Set closer to 2 to shift towards a gamma distribution
  - Set closer to 1 to shift towards a Poisson distribution.

Parameter for using Pseudo-Huber (`reg:pseudohubererror`)

- `huber_slope` : A parameter used for Pseudo-Huber loss to define the δ term. [default = 1.0]

Parameter for using Quantile Loss (`reg:quantileerror`)

- `quantile_alpha`: A scalar or a list of targeted quantiles.

  Added in version 2.0.0.

Parameter for using Expectile Loss (`reg:expectileerror`)

- `expectile_alpha`: A scalar or a list of targeted expectiles. Range: [0, 1]. Required for `reg:expectileerror`.

  Added in version 3.3.0.

Note

Multiple alphas must be sorted in ascending order. Unlike the quantile objective, expectile does not suffer from curve crossing. When predicting with `output_margin=True` and multiple alphas, the first margin corresponds to the smallest alpha; subsequent margins are reparameterized gaps between consecutive expectile predictions, use normal prediction to obtain the actual expectile values.

Parameter for using AFT Survival Loss (`survival:aft`) and Negative Log Likelihood of AFT metric (`aft-nloglik`)

- `aft_loss_distribution`: Probability Density Function for the AFT distribution; `normal`, `logistic`, or `extreme`.
- `aft_loss_distribution_scale`: Scaling factor for the AFT distribution. Range: (0,∞)

Parameters for learning to rank (`rank:ndcg`, `rank:map`, `rank:pairwise`)

These are parameters specific to learning to rank task. See *Learning to Rank* for an in-depth explanation.

- `lambdarank_pair_method` [default = `topk`]

  How to construct pairs for pair-wise learning.

  - `mean`: Sample `lambdarank_num_pair_per_sample` pairs for each document in the query list.
  - `topk`: Focus on top-`lambdarank_num_pair_per_sample` documents. Construct $|query|$ pairs for each document at the top-`lambdarank_num_pair_per_sample` ranked by the model.

- `lambdarank_num_pair_per_sample` [range = $[1, \infty]$]

  It specifies the number of pairs sampled for each document when pair method is `mean`, or the truncation level for queries when the pair method is `topk`. For example, to train with `ndcg@6`, set `lambdarank_num_pair_per_sample` to 6 and `lambdarank_pair_method` to `topk`.

- `lambdarank_normalization` [default = `true`]

  Added in version 2.1.0.

  Whether to normalize the leaf value by lambda gradient. This can sometimes stagnate the training progress.

  Changed in version 3.0.0.

  When the `mean` method is used, it's normalized by the `lambdarank_num_pair_per_sample` instead of gradient.

- `lambdarank_score_normalization` [default = `true`]

  Added in version 3.0.0.

  Whether to normalize the delta metric by the difference of prediction scores. This can sometimes stagnate the training progress. With pairwise ranking, we can normalize the gradient using the difference between two samples in each pair to reduce influence from the pairs that have large difference in ranking scores. This can help us regularize the model to reduce bias and prevent overfitting. Similar to other regularization techniques, this might prevent training from converging.

  There was no normalization before 2.0. In 2.0 and later versions this is used by default. In 3.0, we made this an option that users can disable.

- `lambdarank_unbiased` [default = `false`]

  Specify whether do we need to debias input click data.

- lambdarank_bias_norm [default = 2.0]

  L_p normalization for position debiasing, default is L_2. Only relevant when lambdarank_unbiased is set to true.

- ndcg_exp_gain [default = true]

  Whether we should use exponential gain function for NDCG. There are two forms of gain function for NDCG, one is using relevance value directly while the other is using $2^{rel} - 1$ to emphasize on retrieving relevant documents. When ndcg_exp_gain is true (the default), relevance degree cannot be greater than 31.

1.8 Prediction

There are a number of prediction functions in XGBoost with various parameters. This document attempts to clarify some of confusions around prediction with a focus on the Python binding, R package is similar when strict_shape is specified (see below).

1.8.1 Prediction Options

There are a number of different prediction options for the *xgboost.Booster.predict()* method, ranging from pred_contribs to pred_leaf. The output shape depends on types of prediction. Also for multi-class classification problem, XGBoost builds one tree for each class and the trees for each class are called a "group" of trees, so output dimension may change due to used model. After 1.4 release, we added a new parameter called strict_shape, one can set it to True to indicate a more restricted output is desired. Assuming you are using *xgboost.Booster*, here is a list of possible returns:

- When using normal prediction with strict_shape set to True:

  Output is a 2-dim array with first dimension as rows and second as groups. For regression/survival/ranking/binary classification this is equivalent to a column vector with shape[1] == 1. But for multi-class with multi:softprob the number of columns equals to number of classes. If strict_shape is set to False then XGBoost might output 1 or 2 dim array.

- When using output_margin to avoid transformation and strict_shape is set to True:

  Similar to the previous case, output is a 2-dim array, except for that multi:softmax has equivalent output shape of multi:softprob due to dropped transformation. If strict shape is set to False then output can have 1 or 2 dim depending on used model.

- When using pred_contribs with strict_shape set to True:

  Output is a 3-dim array, with (rows, groups, columns + 1) as shape. Whether approx_contribs is used does not change the output shape. If the strict shape parameter is not set, it can be a 2 or 3 dimension array depending on whether multi-class model is being used. See Lundberg et al. (2018) for the original TreeSHAP feature attribution method. When approx_contribs is False, XGBoost uses the QuadratureTreeSHAP implementation described by Wettenstein et al. (2026) for exact TreeSHAP feature attributions on both CPU and GPU. When approx_contribs is True, XGBoost uses an approximate contribution method on CPU; the GPU predictor does not implement approximated contributions.

- When using pred_interactions with strict_shape set to True:

  Output is a 4-dim array, with (rows, groups, columns + 1, columns + 1) as shape. Like the predict contribution case, whether approx_contribs is used does not change the output shape. If strict shape is set to False, it can have 3 or 4 dims depending on the underlying model.

- When using pred_leaf with strict_shape set to True:

  Output is a 4-dim array with (n_samples, n_iterations, n_classes, n_trees_in_forest) as shape. n_trees_in_forest is specified by the numb_parallel_tree during training. When strict shape is set to

False, output is a 2-dim array with last 3 dims concatenated into 1. Also the last dimension is dropped if it equals to 1. When using `apply` method in scikit learn interface, this is set to False by default.

For R package, when `strict_shape` is specified, an `array` is returned, with the same value as Python except R array is column-major while Python numpy array is row-major, so all the dimensions are reversed. For example, for a Python `predict_leaf` output obtained by having `strict_shape=True` has 4 dimensions: `(n_samples, n_iterations, n_classes, n_trees_in_forest)`, while R with `strict_shape=TRUE` outputs `(n_trees_in_forest, n_classes, n_iterations, n_samples)`.

Other than these prediction types, there's also a parameter called `iteration_range`, which is similar to model slicing. But instead of actually splitting up the model into multiple stacks, it simply returns the prediction formed by the trees within range. Number of trees created in each iteration equals to $trees_i = num_class \times num_parallel_tree$. So if you are training a boosted random forest with size of 4, on the 3-class classification dataset, and want to use the first 2 iterations of trees for prediction, you need to provide `iteration_range=(0, 2)`. Then the first $2 \times 3 \times 4$ trees will be used in this prediction.

1.8.2 Early Stopping

When a model is trained with early stopping, there is an inconsistent behavior between native Python interface and sklearn/R interfaces. By default on R and sklearn interfaces, the `best_iteration` is automatically used so prediction comes from the best model. But with the native Python interface `xgboost.Booster.predict()` and `xgboost.Booster.inplace_predict()` uses the full model. Users can use `best_iteration` attribute with `iteration_range` parameter to achieve the same behavior. Also the `save_best` parameter from `xgboost.callback.EarlyStopping` might be useful.

1.8.3 Base Margin

There's a training parameter in XGBoost called `base_score`, and a meta data for `DMatrix` called `base_margin` (which can be set in `fit` method if you are using scikit-learn interface). They specifies the global bias for boosted model. If the latter is supplied then former is ignored. `base_margin` can be used to train XGBoost model based on other models. See demos on boosting from predictions.

1.8.4 Staged Prediction

Using the native interface with `DMatrix`, prediction can be staged (or cached). For example, one can first predict on the first 4 trees then run prediction on 8 trees. After running the first prediction, result from first 4 trees are cached so when you run the prediction with 8 trees XGBoost can reuse the result from previous prediction. The cache expires automatically upon next prediction, train or evaluation if the cached `DMatrix` object is expired (like going out of scope and being collected by garbage collector in your language environment).

1.8.5 In-place Prediction

Traditionally XGBoost accepts only `DMatrix` for prediction, with wrappers like scikit-learn interface the construction happens internally. We added support for in-place predict to bypass the construction of `DMatrix`, which is slow and memory consuming. The new predict function has limited features but is often sufficient for simple inference tasks. It accepts some commonly found data types in Python like `numpy.ndarray`, `scipy.sparse.csr_matrix` and `cudf.DataFrame` instead of `xgboost.DMatrix`. You can call `xgboost.Booster.inplace_predict()` to use it. Be aware that the output of in-place prediction depends on input data type, when input is on GPU data output is `cupy.ndarray`, otherwise a `numpy.ndarray` is returned.

1.8.6 Thread Safety

After 1.4 release, all prediction functions including normal `predict` with various parameters like shap value computation and `inplace_predict` are thread safe when underlying booster is `gbtree` or `dart`, which means as long as tree model is used, prediction itself should thread safe. But the safety is only guaranteed with prediction. If one tries to train a model in one thread and provide prediction at the other using the same model the behaviour is undefined. This

happens easier than one might expect, for instance we might accidentally call `clf.set_params()` inside a predict function:

```
def predict_fn(clf: xgb.XGBClassifier, X):
    X = preprocess(X)
    clf.set_params(n_jobs=1)  # NOT safe!
    return clf.predict_proba(X, iteration_range=(0, 10))

with ThreadPoolExecutor(max_workers=10) as e:
    e.submit(predict_fn, ...)
```

1.8.7 Privacy-Preserving Prediction

Concrete ML is a third-party open-source library developed by Zama that proposes gradient boosting classes similar to ours, but predicting directly over encrypted data, thanks to Fully Homomorphic Encryption. A simple example would be as follows:

```
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from concrete.ml.sklearn import XGBClassifier

x, y = make_classification(n_samples=100, class_sep=2, n_features=30, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(
    x, y, test_size=10, random_state=42
)

# Train in the clear and quantize the weights
model = XGBClassifier()
model.fit(X_train, y_train)

# Simulate the predictions in the clear
y_pred_clear = model.predict(X_test)

# Compile in FHE
model.compile(X_train)

# Generate keys
model.fhe_circuit.keygen()

# Run the inference on encrypted inputs!
y_pred_fhe = model.predict(X_test, fhe="execute")

print("In clear  :", y_pred_clear)
print("In FHE    :", y_pred_fhe)
print(f"Similarity: {int((y_pred_fhe == y_pred_clear).mean()*100)}%")
```

More information and examples are given in the Concrete ML documentation.

1.9 Tree Methods

For training boosted tree models, there are 2 parameters used for choosing algorithms, namely `updater` and `tree_method`. XGBoost has 3 builtin tree methods, namely `exact`, `approx` and `hist`. Along with these tree methods, there are also some free standing updaters including `refresh`, `prune` and `sync`. The parameter `updater` is more

primitive than tree_method as the latter is just a pre-configuration of the former. The difference is mostly due to historical reasons that each updater requires some specific configurations and might have missing features. As we are moving forward, the gap between them is becoming more and more irrelevant. We will collectively document them under tree methods.

1.9.1 Exact Solution

Exact means XGBoost considers all candidates from data for tree splitting, but underlying the objective is still interpreted as a Taylor expansion.

1. exact: The vanilla gradient boosting tree algorithm described in reference paper. During split-finding, it iterates over all entries of input data. It's more accurate (among other greedy methods) but computationally slower in compared to other tree methods. Further more, its feature set is limited. Features like distributed training and external memory that require approximated quantiles are not supported. This tree method can be used with the parameter tree_method set to exact.

1.9.2 Approximated Solutions

As exact tree method is slow in computation performance and difficult to scale, we often employ approximated training algorithms. These algorithms build a gradient histogram for each node and iterate through the histogram instead of real dataset. Here we introduce the implementations in XGBoost.

1. approx tree method: An approximation tree method described in reference paper. It runs sketching before building each tree using all the rows (rows belonging to the root). Hessian is used as weights during sketch. The algorithm can be accessed by setting tree_method to approx.
2. hist tree method: An approximation tree method used in LightGBM with slight differences in implementation. It runs sketching before training using only user provided weights instead of hessian. The subsequent per-node histogram is built upon this global sketch. This is the fastest algorithm as it runs sketching only once. The algorithm can be accessed by setting tree_method to hist.

1.9.3 Implications

Some objectives like reg:squarederror have constant hessian. In this case, the hist should be preferred as weighted sketching doesn't make sense with constant weights. When using non-constant hessian objectives, sometimes approx yields better accuracy, but with slower computation performance. Most of the time using hist with higher max_bin can achieve similar or even superior accuracy while maintaining good performance. However, as xgboost is largely driven by community effort, the actual implementations have some differences than pure math description. Result might be slightly different than expectation, which we are currently trying to overcome.

1.9.4 Other Updaters

1. Prune: It prunes the existing trees. prune is usually used as part of other tree methods. To use pruner independently, one needs to set the process type to update by: {"process_type": "update", "updater": "prune"}. With this set of parameters, during training, XGBoost will prune the existing trees according to 2 parameters min_split_loss (gamma) and max_depth.
2. Refresh: Refresh the statistic of built trees on a new training dataset. Like the pruner, To use refresh independently, one needs to set the process type to update: {"process_type": "update", "updater": "refresh"}. During training, the updater will change statistics like cover and weight according to the new training dataset. When refresh_leaf is also set to true (default), XGBoost will update the leaf value according to the new leaf weight, but the tree structure (split condition) itself doesn't change.

   There are examples on both training continuation (adding new trees) and using update process on demo/guide-python. Also checkout the process_type parameter in *XGBoost Parameters*.
3. Sync: Synchronize the tree among workers when running distributed training.

1.9.5 Removed Updaters

3 Updaters were removed during development due to maintainability. We describe them here solely for the interest of documentation.

1. Distributed colmaker, which was a distributed version of exact tree method. It required specialization for column based splitting strategy and a different prediction procedure. As the exact tree method is slow by itself and scaling is even less efficient, we removed it entirely.
2. `skmaker`. Per-node weighted sketching employed by `grow_local_histmaker` is slow, the `skmaker` was unmaintained and seems to be a workaround trying to eliminate the histogram creation step and uses sketching values directly during split evaluation. It was never tested and contained some unknown bugs, we decided to remove it and focus our resources on more promising algorithms instead. For accuracy, most of the time `approx` and `hist` are enough with some parameters tuning, so removing them don't have any real practical impact.
3. `grow_local_histmaker` updater: An approximation tree method described in reference paper. This updater was rarely used in practice so it was still an updater rather than tree method. During split finding, it first runs a weighted GK sketching for data points belong to current node to find split candidates, using hessian as weights. The histogram is built upon this per-node sketch. It was faster than `exact` in some applications, but still slow in computation. It was removed because it depended on Rabit's customized reduction function that handles all the data structure that can be serialized/deserialized into fixed size buffer, which is not directly supported by NCCL or federated learning gRPC, making it hard to refactor into a common allreducer interface.

1.9.6 Feature Matrix

Following table summarizes some differences in supported features between 4 tree methods, T means supported while F means unsupported.

	Exact	Approx	Approx (GPU)	Hist
grow_policy	Depthwise	depthwise/lossguide	depthwise/lossguide	depthwise/lossguide
max_leaves	F	T	T	T
sampling method	uniform	uniform	gradient_based/uniform	gradient_based/uniform
categorical data	F	T	T	T
External memory	F	T	P	T
Distributed	F	T	T	T

Features/parameters that are not mentioned here are universally supported for all 3 tree methods (for instance, column sampling and constraints). The P in external memory means special handling. Please note that both categorical data and external memory are experimental.

1.10 XGBoost Python Package

This page contains links to all the python related documents on python package. To install the package, checkout *Installation Guide*.

1.10.1 Contents

Python Package Introduction

This document gives a basic walkthrough of the xgboost package for Python. The Python package is consisted of 3 different interfaces, including native interface, scikit-learn interface and dask interface. For introduction to dask interface please see *Distributed XGBoost with Dask*.

List of other Helpful Links

- *XGBoost Python Feature Walkthrough*

- *Python API Reference*

Contents

- *Install XGBoost*
- *Data Interface*
- *Setting Parameters*
- *Training*
- *Early Stopping*
- *Prediction*
- *Plotting*
- *Scikit-Learn interface*

Install XGBoost

To install XGBoost, follow instructions in *Installation Guide*.

To verify your installation, run the following in Python:

```
import xgboost as xgb
```

Data Interface

The XGBoost Python module is able to load data from many different types of data format including both CPU and GPU data structures. For a comprehensive list of supported data types, please reference the *Supported Python data structures*. For a detailed description of text input formats, please visit *Text Input Format of DMatrix*.

The input data is stored in a `DMatrix` object. For the sklearn estimator interface, a `DMatrix` or a `QuantileDMatrix` is created depending on the chosen algorithm and the input, see the sklearn API reference for details. We will illustrate some of the basic input types using the `DMatrix` here.

- To load a NumPy array into `DMatrix`:

```
data = np.random.rand(5, 10)  # 5 entities, each contains 10 features
label = np.random.randint(2, size=5)  # binary target
dtrain = xgb.DMatrix(data, label=label)
```

- To load a `scipy.sparse` array into `DMatrix`:

```
csr = scipy.sparse.csr_matrix((dat, (row, col)))
dtrain = xgb.DMatrix(csr)
```

- To load a Pandas data frame into `DMatrix`:

```
data = pandas.DataFrame(np.arange(12).reshape((4,3)), columns=['a', 'b', 'c'])
label = pandas.DataFrame(np.random.randint(2, size=4))
dtrain = xgb.DMatrix(data, label=label)
```

- Saving `DMatrix` into a XGBoost binary file:

```
data = np.random.rand(5, 10)  # 5 entities, each contains 10 features
label = np.random.randint(2, size=5)  # binary target
dtrain.save_binary('train.buffer')
```

- Missing values can be replaced by a default value in the *DMatrix* constructor:

```
dtrain = xgb.DMatrix(data, label=label, missing=np.NaN)
```

- Weights can be set when needed:

```
w = np.random.rand(5, 1)
dtrain = xgb.DMatrix(data, label=label, missing=np.NaN, weight=w)
```

Setting Parameters

XGBoost can use either a list of pairs or a dictionary to set *parameters*. For instance:

- Booster parameters

```
param = {'max_depth': 2, 'eta': 1, 'objective': 'binary:logistic'}
param['nthread'] = 4
param['eval_metric'] = 'auc'
```

- You can also specify multiple eval metrics:

```
param['eval_metric'] = ['auc', 'ams@0']

# alternatively:
# plst = param.items()
# plst += [('eval_metric', 'ams@0')]
```

- Specify validations set to watch performance

```
evallist = [(dtrain, 'train'), (dtest, 'eval')]
```

Training

Training a model requires a parameter list and data set.

```
num_round = 10
bst = xgb.train(param, dtrain, num_round, evallist)
```

After training, the model can be saved into `JSON` or `UBJSON`:

```
bst.save_model('model.ubj')
```

The model and its feature map can also be dumped to a text file.

```
# dump model
bst.dump_model('dump.raw.txt')
# dump model with feature map
bst.dump_model('dump.raw.txt', 'featmap.txt')
```

A saved model can be loaded as follows:

```
bst = xgb.Booster({'nthread': 4})  # init model
bst.load_model('model.ubj')  # load model data
```

Methods including *update* and *boost* from `xgboost.Booster` are designed for internal usage only. The wrapper function `xgboost.train` does some pre-configuration including setting up caches and some other parameters.

Early Stopping

If you have a validation set, you can use early stopping to find the optimal number of boosting rounds. Early stopping requires at least one set in `evals`. If there's more than one, it will use the last.

```
train(..., evals=evals, early_stopping_rounds=10)
```

The model will train until the validation score stops improving. Validation error needs to decrease at least every `early_stopping_rounds` to continue training.

If early stopping occurs, the model will have two additional fields: `bst.best_score`, `bst.best_iteration`. Note that `xgboost.train()` will return a model from the last iteration, not the best one.

This works with both metrics to minimize (RMSE, log loss, etc.) and to maximize (MAP, NDCG, AUC). Note that if you specify more than one evaluation metric the last one in `param['eval_metric']` is used for early stopping.

Prediction

A model that has been trained or loaded can perform predictions on data sets.

```
# 7 entities, each contains 10 features
data = np.random.rand(7, 10)
dtest = xgb.DMatrix(data)
ypred = bst.predict(dtest)
```

If early stopping is enabled during training, you can get predictions from the best iteration with `bst.best_iteration`:

```
ypred = bst.predict(dtest, iteration_range=(0, bst.best_iteration + 1))
```

Plotting

You can use plotting module to plot importance and output tree.

To plot importance, use `xgboost.plot_importance()`. This function requires `matplotlib` to be installed.

```
xgb.plot_importance(bst)
```

To plot the output tree via `matplotlib`, use `xgboost.plot_tree()`, specifying the ordinal number of the target tree. This function requires `graphviz` and `matplotlib`.

```
xgb.plot_tree(bst, num_trees=2)
```

When you use `IPython`, you can use the `xgboost.to_graphviz()` function, which converts the target tree to a `graphviz` instance. The `graphviz` instance is automatically rendered in `IPython`.

```
xgb.to_graphviz(bst, num_trees=2)
```

Scikit-Learn interface

XGBoost provides an easy to use scikit-learn interface for some pre-defined models including regression, classification and ranking. See *Using the Scikit-Learn Estimator Interface* for more info.

```
# Use "hist" for training the model.
reg = xgb.XGBRegressor(tree_method="hist", device="cuda")
# Fit the model using predictor X and response y.
reg.fit(X, y)
# Save model into JSON format.
reg.save_model("regressor.json")
```

User can still access the underlying booster model when needed:

```
booster: xgb.Booster = reg.get_booster()
```

Using the Scikit-Learn Estimator Interface

Contents

- *Overview*
- *Early Stopping*
- *Obtaining the native booster object*
- *Prediction*
- *Number of parallel threads*

Overview

In addition to the native interface, XGBoost features a sklearn estimator interface that conforms to sklearn estimator guideline. It supports regression, classification, and learning to rank. Survival training for the sklearn estimator interface is still working in progress.

You can find some some quick start examples at *Collection of examples for using sklearn interface*. The main advantage of using sklearn interface is that it works with most of the utilities provided by sklearn like `sklearn.model_selection.cross_validate()`. Also, many other libraries recognize the sklearn estimator interface thanks to its popularity.

With the sklearn estimator interface, we can train a classification model with only a couple lines of Python code. Here's an example for training a classification model:

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split

import xgboost as xgb

X, y = load_breast_cancer(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y, random_state=94)

# Use "hist" for constructing the trees, with early stopping enabled.
clf = xgb.XGBClassifier(tree_method="hist", early_stopping_rounds=2)
```

(continues on next page)

(continued from previous page)

```
# Fit the model, test sets are used for early stopping.
clf.fit(X_train, y_train, eval_set=[(X_test, y_test)])
# Save model into JSON format.
clf.save_model("clf.json")
```

The `tree_method` parameter specifies the method to use for constructing the trees, and the early_stopping_rounds parameter enables early stopping. Early stopping can help prevent overfitting and save time during training.

Early Stopping

As demonstrated in the previous example, early stopping can be enabled by the parameter `early_stopping_rounds`. Alternatively, there's a callback function that can be used `xgboost.callback.EarlyStopping` to specify more details about the behavior of early stopping, including whether XGBoost should return the best model instead of the full stack of trees:

```
early_stop = xgb.callback.EarlyStopping(
    rounds=2, metric_name='logloss', data_name='validation_0', save_best=True
)
clf = xgb.XGBClassifier(tree_method="hist", callbacks=[early_stop])
clf.fit(X_train, y_train, eval_set=[(X_test, y_test)])
```

At present, XGBoost doesn't implement data spliting logic within the estimator and relies on the `eval_set` parameter of the `xgboost.XGBModel.fit()` method. If you want to use early stopping to prevent overfitting, you'll need to manually split your data into training and testing sets using the `sklearn.model_selection.train_test_split()` function from the *sklearn* library. Some other machine learning algorithms, like those in *sklearn*, include early stopping as part of the estimator and may work with cross validation. However, using early stopping during cross validation may not be a perfect approach because it changes the model's number of trees for each validation fold, leading to different model. A better approach is to retrain the model after cross validation using the best hyperparameters along with early stopping. If you want to experiment with idea of using cross validation with early stopping, here is a snippet to begin with:

```
from sklearn.base import clone
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import StratifiedKFold, cross_validate

import xgboost as xgb

X, y = load_breast_cancer(return_X_y=True)


def fit_and_score(estimator, X_train, X_test, y_train, y_test):
    """Fit the estimator on the train set and score it on both sets"""
    estimator.fit(X_train, y_train, eval_set=[(X_test, y_test)])

    train_score = estimator.score(X_train, y_train)
    test_score = estimator.score(X_test, y_test)

    return estimator, train_score, test_score


cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=94)
```

(continues on next page)

(continued from previous page)

```
clf = xgb.XGBClassifier(tree_method="hist", early_stopping_rounds=3)

results = {}

for train, test in cv.split(X, y):
    X_train = X[train]
    X_test = X[test]
    y_train = y[train]
    y_test = y[test]
    est, train_score, test_score = fit_and_score(
        clone(clf), X_train, X_test, y_train, y_test
    )
    results[est] = (train_score, test_score)
```

Obtaining the native booster object

The sklearn estimator interface primarily facilitates training and doesn't implement all features available in XGBoost. For instance, in order to have cached predictions, *xgboost.DMatrix* needs to be used with *xgboost.Booster.predict()*. One can obtain the booster object from the sklearn interface using `xgboost.XGBModel.get_booster()`:

```
booster = clf.get_booster()
print(booster.num_boosted_rounds())
```

Prediction

When early stopping is enabled, prediction functions including the `xgboost.XGBModel.predict()`, `xgboost.XGBModel.score()`, and `xgboost.XGBModel.apply()` methods will use the best model automatically. Meaning the `xgboost.XGBModel.best_iteration` is used to specify the range of trees used in prediction.

To have cached results for incremental prediction, please use the *xgboost.Booster.predict()* method instead.

Number of parallel threads

When working with XGBoost and other sklearn tools, you can specify how many threads you want to use by using the `n_jobs` parameter. By default, XGBoost uses all the available threads on your computer, which can lead to some interesting consequences when combined with other sklearn functions like `sklearn.model_selection.cross_validate()`. If both XGBoost and sklearn are set to use all threads, your computer may start to slow down significantly due to something called "thread thrashing". To avoid this, you can simply set the `n_jobs` parameter for XGBoost to *None* (which uses all threads) and the `n_jobs` parameter for sklearn to *1*. This way, both programs will be able to work together smoothly without causing any unnecessary computer strain.

Python API Reference

This page gives the Python API reference of xgboost, please also refer to Python Package Introduction for more information about the Python package.

- *Global Configuration*
- *Core Data Structure*
- *Learning API*

- *Scikit-Learn API*
- *Plotting API*
- *Callback API*
- *Dask API*
- *PySpark API*
- *Collective*

Global Configuration

xgboost.**config_context**(***new_config*)

Context manager for global XGBoost configuration.

Global configuration consists of a collection of parameters that can be applied in the global scope. See *Global Configuration* for the full list of parameters supported in the global configuration.

Note

All settings, not just those presently modified, will be returned to their previous values when the context manager is exited. This is not thread-safe.

Added in version 1.4.0.

Parameters
new_config (*Dict[str, Any]*) – Keyword arguments representing the parameters and their values

Return type
Iterator[None]

Example

```
import xgboost as xgb

# Show all messages, including ones pertaining to debugging
xgb.set_config(verbosity=2)

# Get current value of global configuration
# This is a dict containing all parameters in the global configuration,
# including 'verbosity'
config = xgb.get_config()
assert config['verbosity'] == 2

# Example of using the context manager xgb.config_context().
# The context manager will restore the previous value of the global
# configuration upon exiting.
with xgb.config_context(verbosity=0):
    # Suppress warning caused by model generated with XGBoost version < 1.0.0
    bst = xgb.Booster(model_file='./old_model.bin')
assert xgb.get_config()['verbosity'] == 2  # old value restored
```

Nested configuration context is also supported:

Example

```
with xgb.config_context(verbosity=3):
    assert xgb.get_config()["verbosity"] == 3
    with xgb.config_context(verbosity=2):
        assert xgb.get_config()["verbosity"] == 2

xgb.set_config(verbosity=2)
assert xgb.get_config()["verbosity"] == 2
with xgb.config_context(verbosity=3):
    assert xgb.get_config()["verbosity"] == 3
```

See also

set_config
: Set global XGBoost configuration

get_config
: Get current values of the global configuration

xgboost.**set_config**(***new_config*)

Set global configuration.

Global configuration consists of a collection of parameters that can be applied in the global scope. See *Global Configuration* for the full list of parameters supported in the global configuration.

Added in version 1.4.0.

Parameters
: **new_config** (*Dict[str, Any]*) – Keyword arguments representing the parameters and their values

Return type
: None

Example

```
import xgboost as xgb

# Show all messages, including ones pertaining to debugging
xgb.set_config(verbosity=2)

# Get current value of global configuration
# This is a dict containing all parameters in the global configuration,
# including 'verbosity'
config = xgb.get_config()
assert config['verbosity'] == 2

# Example of using the context manager xgb.config_context().
# The context manager will restore the previous value of the global
# configuration upon exiting.
```

(continues on next page)

(continued from previous page)

```
with xgb.config_context(verbosity=0):
    # Suppress warning caused by model generated with XGBoost version < 1.0.0
    bst = xgb.Booster(model_file='./old_model.bin')
assert xgb.get_config()['verbosity'] == 2  # old value restored
```

Nested configuration context is also supported:

Example

```
with xgb.config_context(verbosity=3):
    assert xgb.get_config()["verbosity"] == 3
    with xgb.config_context(verbosity=2):
        assert xgb.get_config()["verbosity"] == 2

xgb.set_config(verbosity=2)
assert xgb.get_config()["verbosity"] == 2
with xgb.config_context(verbosity=3):
    assert xgb.get_config()["verbosity"] == 3
```

`xgboost.get_config()`

Get current values of the global configuration.

Global configuration consists of a collection of parameters that can be applied in the global scope. See *Global Configuration* for the full list of parameters supported in the global configuration.

Added in version 1.4.0.

Returns
args – The list of global parameters and their values

Return type
Dict[str, Any]

Example

```
import xgboost as xgb

# Show all messages, including ones pertaining to debugging
xgb.set_config(verbosity=2)

# Get current value of global configuration
# This is a dict containing all parameters in the global configuration,
# including 'verbosity'
config = xgb.get_config()
assert config['verbosity'] == 2

# Example of using the context manager xgb.config_context().
# The context manager will restore the previous value of the global
# configuration upon exiting.
with xgb.config_context(verbosity=0):
    # Suppress warning caused by model generated with XGBoost version < 1.0.0
    bst = xgb.Booster(model_file='./old_model.bin')
assert xgb.get_config()['verbosity'] == 2  # old value restored
```

Nested configuration context is also supported:

Example

```
with xgb.config_context(verbosity=3):
    assert xgb.get_config()["verbosity"] == 3
    with xgb.config_context(verbosity=2):
        assert xgb.get_config()["verbosity"] == 2

xgb.set_config(verbosity=2)
assert xgb.get_config()["verbosity"] == 2
with xgb.config_context(verbosity=3):
    assert xgb.get_config()["verbosity"] == 3
```

xgboost.**build_info**()

Build information of XGBoost. The returned value format is not stable. Also, please note that build time dependency is not the same as runtime dependency. For instance, it's possible to build XGBoost with older CUDA version but run it with the lastest one.

Added in version 1.6.0.

Return type
dict

Core Data Structure

Core XGBoost Library.

class xgboost.**DMatrix**(*data*, *label=None*, ** (Keyword-only parameters separator (PEP 3102))*, *weight=None*, *base_margin=None*, *missing=None*, *silent=False*, *feature_names=None*, *feature_types=None*, *nthread=None*, *group=None*, *qid=None*, *label_lower_bound=None*, *label_upper_bound=None*, *feature_weights=None*, *enable_categorical=True*, *data_split_mode=DataSplitMode.ROW*)

Bases: `object`

Data Matrix used in XGBoost.

DMatrix is an internal data structure that is used by XGBoost, which is optimized for both memory efficiency and training speed. You can construct DMatrix from multiple different sources of data.

Parameters

- **data** (*Any*) – Data source of DMatrix. See *Markers* for a list of supported input types.

  Note that, if passing an iterator, it **will cache data on disk**, and note that fields like `label` will be concatenated in-memory from multiple calls to the iterator.
- **label** (*Any | None*) – Label of the training data.
- **weight** (*Any | None*) – Weight for each instance.

> **Note**
>
> For ranking task, weights are per-group. In ranking task, one weight is assigned to each group (not each data point). This is because we only care about the relative ordering of data points within each group, so it doesn't make sense to assign weights to individual data points.

- **base_margin** (*Any | None*) – Global bias for each instance. See *Intercept* for details.

- **missing** (*float | None*) – Value in the input data which needs to be present as a missing value. If None, defaults to np.nan.
- **silent** (*bool*) – Whether print messages during construction
- **feature_names** (*Sequence[str] | None*) – Set names for features.
- **feature_types** (*Sequence[str] | Categories | None*) – Set types for features. If *data* is a DataFrame type and passing *enable_categorical=True*, the types will be deduced automatically from the column types.

  Otherwise, one can pass a list-like input with the same length as number of columns in *data*, with the following possible values:

  - "c", which represents categorical columns.
  - "q", which represents numeric columns.
  - "int", which represents integer columns.
  - "i", which represents boolean columns.

  Note that, while categorical types are treated differently from the rest for model fitting purposes, the other types do not influence the generated model, but have effects in other functionalities such as feature importances.

  For categorical features, the input is assumed to be preprocessed and encoded by the users. The encoding can be done via `sklearn.preprocessing.OrdinalEncoder` or pandas dataframe *.cat.codes* method. This is useful when users want to specify categorical features without having to construct a dataframe as input.

  Added in version 3.1.0.

  Alternatively, user can pass a `Categories` object returned from previous training as a reference for re-coding. One can obtain the reference with the `get_categories()` from the previous training DMatrix or the Booster. This feature is experimental.

- **nthread** (*int | None*) – Number of threads to use for loading data when parallelization is applicable. If -1, uses maximum threads available on the system.
- **group** (*Any | None*) – Group size for all ranking group.
- **qid** (*Any | None*) – Query ID for data samples, used for ranking.
- **label_lower_bound** (*Any | None*) – Lower bound for survival training.
- **label_upper_bound** (*Any | None*) – Upper bound for survival training.
- **feature_weights** (*Any | None*) – Set feature weights for column sampling.
- **enable_categorical** (*bool*) – Added in version 1.3.0.

  Note

  This parameter is experimental

  Experimental support of specializing for categorical features. See *Categorical Data* for more info.

  If passing *True* and *data* is a data frame (from supported libraries such as Pandas, Modin, polars, and cuDF), The DMatrix recognizes categorical columns and automatically set the *feature_types* parameter. If *data* is not a data frame, this argument is ignored.

If passing *False* and *data* is a data frame with categorical columns, it will result in an error.

See notes in the `DataIter` for consistency requirement when the input is an iterator.

Changed in version 3.1.0.

XGBoost can remember the encoding of categories when the input is a dataframe.

- **data_split_mode** (*DataSplitMode*)

data_split_mode()

Get the data split mode of the DMatrix.

Added in version 2.1.0.

Return type
DataSplitMode

property feature_names: Sequence[str] | None

Labels for features (column labels).

Setting it to `None` resets existing feature names.

property feature_types: Sequence[str] | None

Type of features (column types).

This is for displaying the results and categorical data support. See `DMatrix` for details.

Setting it to `None` resets existing feature types.

get_base_margin()

Get the base margin of the DMatrix.

Return type
ndarray | *Any*

get_categories(*export_to_arrow=False*)

Get the categories in the dataset.

Added in version 3.1.0.

Warning

This function is experimental.

Parameters
export_to_arrow (*bool*) – The returned container will contain a list of `pyarrow` arrays for the categories. See the `to_arrow()` for more info.

Return type
Categories

get_data()

Get the predictors from DMatrix as a CSR matrix. This getter is mostly for testing purposes. If this is a quantized DMatrix then quantized values are returned instead of input values.

Added in version 1.7.0.

Return type
csr_matrix

`get_float_info(field)`

Get float property from the DMatrix.

Parameters
`field` (*str*) – The field name of the information.

Return type
ndarray | *Any*

`get_group()`

Get the group of the DMatrix.

Return type
group

`get_label()`

Get the label of the DMatrix.

Return type
ndarray | *Any*

`get_quantile_cut()`

Get quantile cuts for quantization.

Added in version 2.0.0.

Return type
Tuple[*ndarray*, *ndarray*]

`get_uint_info(field)`

Get unsigned integer property from the DMatrix.

Parameters
`field` (*str*) – The field name of the information.

Return type
ndarray | *Any*

`get_weight()`

Get the weight of the DMatrix.

Return type
ndarray | *Any*

`num_col()`

Get the number of columns (features) in the DMatrix.

Return type
int

`num_nonmissing()`

Get the number of non-missing values in the DMatrix.

Added in version 1.7.0.

Return type
int

`num_row()`

Get the number of rows in the DMatrix.

Return type
int

`save_binary`(*fname*, *silent=True*)

Save DMatrix to an XGBoost buffer. Saved binary can be later loaded by providing the path to `xgboost.DMatrix()` as input.

Parameters

- **fname** (*string or os.PathLike*) – Name of the output buffer file.
- **silent** (*bool (optional; default: True)*) – If set, the output is suppressed.

Return type
None

`set_base_margin`(*margin*)

Set base margin of booster to start from.

This can be used to specify a prediction value of existing model to be base_margin However, remember margin is needed, instead of transformed prediction e.g. for logistic regression: need to put in value before logistic transformation see also example/demo.py

Parameters
margin (*array like*) – Prediction margin of each datapoint

Return type
None

`set_float_info`(*field*, *data*)

Set float type property into the DMatrix.

Parameters

- **field** (*str*) – The field name of the information
- **data** (*numpy array*) – The array of data to be set

Return type
None

`set_float_info_npy2d`(*field*, *data*)

Set float type property into the DMatrix
for numpy 2d array input

Parameters

- **field** (*str*) – The field name of the information
- **data** (*numpy array*) – The array of data to be set

Return type
None

`set_group`(*group*)

Set group size of DMatrix (used for ranking).

Parameters
group (*array like*) – Group size of each group

Return type
None

set_info(**, label=None, weight=None, base_margin=None, group=None, qid=None, label_lower_bound=None, label_upper_bound=None, feature_names=None, feature_types=None, feature_weights=None*)

Set meta info for DMatrix. See doc string for `xgboost.DMatrix`.

Parameters

- **label** (*Any | None*)
- **weight** (*Any | None*)
- **base_margin** (*Any | None*)
- **group** (*Any | None*)
- **qid** (*Any | None*)
- **label_lower_bound** (*Any | None*)
- **label_upper_bound** (*Any | None*)
- **feature_names** (*Sequence[str] | None*)
- **feature_types** (*Sequence[str] | None*)
- **feature_weights** (*Any | None*)

Return type

None

set_label(*label*)

Set label of dmatrix

Parameters

label (*array like*) – The label information to be set into DMatrix

Return type

None

set_uint_info(*field, data*)

Set uint type property into the DMatrix.

Parameters

- **field** (*str*) – The field name of the information
- **data** (*numpy array*) – The array of data to be set

Return type

None

set_weight(*weight*)

Set weight of each instance.

Parameters

weight (*array like*) – Weight for each data point

> **Note**
>
> For ranking task, weights are per-group.
>
> In ranking task, one weight is assigned to each group (not each data point). This is because we only care about the relative ordering of data points within each group, so it doesn't make sense to assign weights to individual data points.

Return type
None

slice(*rindex*, *allow_groups=False*)

Slice the DMatrix and return a new DMatrix that only contains *rindex*.

Parameters

- **rindex** (*List[int] | ndarray*) – List of indices to be selected.
- **allow_groups** (*bool*) – Allow slicing of a matrix with a groups attribute

Returns
A new DMatrix containing only selected indices.

Return type
res

class xgboost.**QuantileDMatrix**(*data*, *label=None*, ***, *weight=None*, *base_margin=None*, *missing=None*, *silent=False*, *feature_names=None*, *feature_types=None*, *nthread=None*, *max_bin=None*, *ref=None*, *group=None*, *qid=None*, *label_lower_bound=None*, *label_upper_bound=None*, *feature_weights=None*, *enable_categorical=True*, *max_quantile_batches=None*, *data_split_mode=DataSplitMode.ROW*)

Bases: *DMatrix*, _RefMixIn

A DMatrix variant that generates quantilized data directly from input for the `hist` tree method. This DMatrix is primarily designed to save memory in training by avoiding intermediate storage. Set `max_bin` to control the number of bins during quantisation, which should be consistent with the training parameter `max_bin`. When `QuantileDMatrix` is used for validation/test dataset, `ref` should be another `QuantileDMatrix` or `DMatrix`, but not recommended as it defeats the purpose of saving memory) constructed from training dataset. See *xgboost.DMatrix* for documents on meta info.

> **Note**
>
> Do not use `QuantileDMatrix` as validation/test dataset without supplying a reference (the training dataset) `QuantileDMatrix` using `ref` as some information may be lost in quantisation.

Added in version 1.7.0.

Examples

```
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split

X, y = make_regression()
X_train, X_test, y_train, y_test = train_test_split(X, y)
```

(continues on next page)

(continued from previous page)

```
Xy_train = xgb.QuantileDMatrix(X_train, y_train)
# It's necessary to have the training DMatrix as a reference for valid
# quantiles.
Xy_test = xgb.QuantileDMatrix(X_test, y_test, ref=Xy_train)
```

Parameters

- **max_bin** (*int | None*) – The number of histogram bin, should be consistent with the training parameter `max_bin`.
- **ref** (DMatrix | *None*) – The training dataset that provides quantile information, needed when creating validation/test dataset with `QuantileDMatrix`. Supplying the training DMatrix as a reference means that the same quantisation applied to the training data is applied to the validation/test data
- **max_quantile_batches** (*int | None*) – Deprecated. This parameter no longer has any effect and will be removed in a future release.

  Added in version 3.0.0.

  Deprecated since version 3.3.0.
- **data** (*Any*) – Data source of DMatrix. See *Markers* for a list of supported input types.

  Note that, if passing an iterator, it **will cache data on disk**, and note that fields like `label` will be concatenated in-memory from multiple calls to the iterator.
- **label** (*Any | None*) – Label of the training data.
- **weight** (*Any | None*) – Weight for each instance.

  > **Note**
  >
  > For ranking task, weights are per-group. In ranking task, one weight is assigned to each group (not each data point). This is because we only care about the relative ordering of data points within each group, so it doesn't make sense to assign weights to individual data points.

- **base_margin** (*Any | None*) – Global bias for each instance. See *Intercept* for details.
- **missing** (*float | None*) – Value in the input data which needs to be present as a missing value. If None, defaults to np.nan.
- **silent** (*bool*) – Whether print messages during construction
- **feature_names** (*Sequence[str] | None*) – Set names for features.
- **feature_types** (*Sequence[str] | None*) – Set types for features. If *data* is a DataFrame type and passing *enable_categorical=True*, the types will be deduced automatically from the column types.

  Otherwise, one can pass a list-like input with the same length as number of columns in *data*, with the following possible values:

  - "c", which represents categorical columns.
  - "q", which represents numeric columns.
  - "int", which represents integer columns.

– ”i”, which represents boolean columns.

Note that, while categorical types are treated differently from the rest for model fitting purposes, the other types do not influence the generated model, but have effects in other functionalities such as feature importances.

For categorical features, the input is assumed to be preprocessed and encoded by the users. The encoding can be done via `sklearn.preprocessing.OrdinalEncoder` or pandas dataframe *.cat.codes* method. This is useful when users want to specify categorical features without having to construct a dataframe as input.

Added in version 3.1.0.

Alternatively, user can pass a `Categories` object returned from previous training as a reference for re-coding. One can obtain the reference with the `get_categories()` from the previous training DMatrix or the Booster. This feature is experimental.

- **nthread** (*int | None*) – Number of threads to use for loading data when parallelization is applicable. If -1, uses maximum threads available on the system.
- **group** (*Any | None*) – Group size for all ranking group.
- **qid** (*Any | None*) – Query ID for data samples, used for ranking.
- **label_lower_bound** (*Any | None*) – Lower bound for survival training.
- **label_upper_bound** (*Any | None*) – Upper bound for survival training.
- **feature_weights** (*Any | None*) – Set feature weights for column sampling.
- **enable_categorical** (*bool*) – Added in version 1.3.0.

  Note

  This parameter is experimental

  Experimental support of specializing for categorical features. See *Categorical Data* for more info.

  If passing *True* and *data* is a data frame (from supported libraries such as Pandas, Modin, polars, and cuDF), The DMatrix recognizes categorical columns and automatically set the *feature_types* parameter. If *data* is not a data frame, this argument is ignored.

  If passing *False* and *data* is a data frame with categorical columns, it will result in an error.

  See notes in the `DataIter` for consistency requirement when the input is an iterator.

  Changed in version 3.1.0.

  XGBoost can remember the encoding of categories when the input is a dataframe.
- **data_split_mode** (*DataSplitMode*)

data_split_mode()

Get the data split mode of the DMatrix.

Added in version 2.1.0.

Return type

DataSplitMode

property feature_names: Sequence[str] | None

Labels for features (column labels).

Setting it to None resets existing feature names.

property feature_types: Sequence[str] | None

Type of features (column types).

This is for displaying the results and categorical data support. See *DMatrix* for details.

Setting it to None resets existing feature types.

get_base_margin()

Get the base margin of the DMatrix.

Return type

ndarray | *Any*

get_categories(*export_to_arrow=False*)

Get the categories in the dataset.

Added in version 3.1.0.

Warning

This function is experimental.

Parameters

export_to_arrow (*bool*) – The returned container will contain a list of pyarrow arrays for the categories. See the to_arrow() for more info.

Return type

Categories

get_data()

Get the predictors from DMatrix as a CSR matrix. This getter is mostly for testing purposes. If this is a quantized DMatrix then quantized values are returned instead of input values.

Added in version 1.7.0.

Return type

csr_matrix

get_float_info(*field*)

Get float property from the DMatrix.

Parameters

field (*str*) – The field name of the information.

Return type

ndarray | *Any*

get_group()

Get the group of the DMatrix.

Return type

group

get_label()

Get the label of the DMatrix.

Return type
ndarray | *Any*

get_quantile_cut()

Get quantile cuts for quantization.

Added in version 2.0.0.

Return type
Tuple[*ndarray*, *ndarray*]

get_uint_info(*field*)

Get unsigned integer property from the DMatrix.

Parameters
field (*str*) – The field name of the information.

Return type
ndarray | *Any*

get_weight()

Get the weight of the DMatrix.

Return type
ndarray | *Any*

num_col()

Get the number of columns (features) in the DMatrix.

Return type
int

num_nonmissing()

Get the number of non-missing values in the DMatrix.

Added in version 1.7.0.

Return type
int

num_row()

Get the number of rows in the DMatrix.

Return type
int

property ref: ReferenceType | None

Internal method for retrieving a reference to the training DMatrix.

save_binary(*fname*, *silent=True*)

Save DMatrix to an XGBoost buffer. Saved binary can be later loaded by providing the path to `xgboost.DMatrix()` as input.

Parameters

- **fname** (*string or os.PathLike*) – Name of the output buffer file.
- **silent** (*bool (optional; default: True)*) – If set, the output is suppressed.

Return type
None

`set_base_margin`(*margin*)

Set base margin of booster to start from.

This can be used to specify a prediction value of existing model to be base_margin However, remember margin is needed, instead of transformed prediction e.g. for logistic regression: need to put in value before logistic transformation see also example/demo.py

Parameters
margin (*array like*) – Prediction margin of each datapoint

Return type
None

`set_float_info`(*field*, *data*)

Set float type property into the DMatrix.

Parameters

- **field** (*str*) – The field name of the information
- **data** (*numpy array*) – The array of data to be set

Return type
None

`set_float_info_npy2d`(*field*, *data*)

Set float type property into the DMatrix
for numpy 2d array input

Parameters

- **field** (*str*) – The field name of the information
- **data** (*numpy array*) – The array of data to be set

Return type
None

`set_group`(*group*)

Set group size of DMatrix (used for ranking).

Parameters
group (*array like*) – Group size of each group

Return type
None

`set_info`(*, *label=None*, *weight=None*, *base_margin=None*, *group=None*, *qid=None*, *label_lower_bound=None*, *label_upper_bound=None*, *feature_names=None*, *feature_types=None*, *feature_weights=None*)

Set meta info for DMatrix. See doc string for `xgboost.DMatrix`.

Parameters

- **label** (*Any | None*)
- **weight** (*Any | None*)
- **base_margin** (*Any | None*)

- **group** (*Any | None*)
- **qid** (*Any | None*)
- **label_lower_bound** (*Any | None*)
- **label_upper_bound** (*Any | None*)
- **feature_names** (*Sequence[str] | None*)
- **feature_types** (*Sequence[str] | None*)
- **feature_weights** (*Any | None*)

Return type
None

set_label(*label*)

Set label of dmatrix

Parameters
label (*array like*) – The label information to be set into DMatrix

Return type
None

set_uint_info(*field*, *data*)

Set uint type property into the DMatrix.

Parameters

- **field** (*str*) – The field name of the information
- **data** (*numpy array*) – The array of data to be set

Return type
None

set_weight(*weight*)

Set weight of each instance.

Parameters
weight (*array like*) – Weight for each data point

Note

For ranking task, weights are per-group.

In ranking task, one weight is assigned to each group (not each data point). This is because we only care about the relative ordering of data points within each group, so it doesn't make sense to assign weights to individual data points.

Return type
None

slice(*rindex*, *allow_groups=False*)

Slice the DMatrix and return a new DMatrix that only contains *rindex*.

Parameters

- **rindex** (*List[int] | ndarray*) – List of indices to be selected.
- **allow_groups** (*bool*) – Allow slicing of a matrix with a groups attribute

Returns
A new DMatrix containing only selected indices.

Return type
res

class xgboost.ExtMemQuantileDMatrix(*data*, **, *missing=None*, *nthread=None*, *max_bin=None*, *ref=None*, *enable_categorical=True*, *max_quantile_batches=None*, *cache_host_ratio=None*)

Bases: `DMatrix`, `_RefMixIn`

The external memory version of the `QuantileDMatrix`.

See *Using XGBoost External Memory Version* for explanation and usage examples, and `QuantileDMatrix` for parameter document.

> **Warning**
>
> This is an experimental feature and subject to change.

Added in version 3.0.0.

Parameters

- **data** (`DataIter`) – A user-defined `DataIter` for loading data.
- **max_quantile_batches** (*int | None*) – Deprecated. See `QuantileDMatrix`.
- **cache_host_ratio** (*float | None*) – Added in version 3.1.0.

  Used by the GPU implementation. For GPU-based inputs, XGBoost can split the cache into host and device caches to reduce the data transfer overhead. This parameter specifies the size of host cache compared to the size of the entire cache: $host/(host + device)$.

  See *Adaptive Cache* for more info.
- **missing** (*float | None*)
- **nthread** (*int | None*)
- **max_bin** (*int | None*)
- **ref** (`DMatrix` *| None*)
- **enable_categorical** (*bool*)

data_split_mode()

Get the data split mode of the DMatrix.

Added in version 2.1.0.

Return type
DataSplitMode

property feature_names: Sequence[str] | None

Labels for features (column labels).

Setting it to `None` resets existing feature names.

property feature_types: Sequence[str] | None

Type of features (column types).

This is for displaying the results and categorical data support. See *DMatrix* for details.

Setting it to `None` resets existing feature types.

get_base_margin()

Get the base margin of the DMatrix.

Return type
ndarray | *Any*

get_categories(*export_to_arrow=False*)

Get the categories in the dataset.

Added in version 3.1.0.

> **Warning**
>
> This function is experimental.

Parameters
export_to_arrow (*bool*) – The returned container will contain a list of `pyarrow` arrays for the categories. See the `to_arrow()` for more info.

Return type
Categories

get_data()

Get the predictors from DMatrix as a CSR matrix. This getter is mostly for testing purposes. If this is a quantized DMatrix then quantized values are returned instead of input values.

Added in version 1.7.0.

Return type
csr_matrix

get_float_info(*field*)

Get float property from the DMatrix.

Parameters
field (*str*) – The field name of the information.

Return type
ndarray | *Any*

get_group()

Get the group of the DMatrix.

Return type
group

get_label()

Get the label of the DMatrix.

Return type
ndarray | *Any*

get_quantile_cut()

Get quantile cuts for quantization.

Added in version 2.0.0.

Return type
Tuple[*ndarray*, *ndarray*]

get_uint_info(*field*)

Get unsigned integer property from the DMatrix.

Parameters
field (*str*) – The field name of the information.

Return type
ndarray | *Any*

get_weight()

Get the weight of the DMatrix.

Return type
ndarray | *Any*

num_col()

Get the number of columns (features) in the DMatrix.

Return type
int

num_nonmissing()

Get the number of non-missing values in the DMatrix.

Added in version 1.7.0.

Return type
int

num_row()

Get the number of rows in the DMatrix.

Return type
int

property ref: ReferenceType | None

Internal method for retrieving a reference to the training DMatrix.

save_binary(*fname*, *silent=True*)

Save DMatrix to an XGBoost buffer. Saved binary can be later loaded by providing the path to `xgboost.DMatrix()` as input.

Parameters

- **fname** (*string or os.PathLike*) – Name of the output buffer file.
- **silent** (*bool (optional; default: True)*) – If set, the output is suppressed.

Return type
None

`set_base_margin`(*margin*)

Set base margin of booster to start from.

This can be used to specify a prediction value of existing model to be base_margin However, remember margin is needed, instead of transformed prediction e.g. for logistic regression: need to put in value before logistic transformation see also example/demo.py

Parameters
margin (*array like*) – Prediction margin of each datapoint

Return type
None

`set_float_info`(*field*, *data*)

Set float type property into the DMatrix.

Parameters

- **field** (*str*) – The field name of the information
- **data** (*numpy array*) – The array of data to be set

Return type
None

`set_float_info_npy2d`(*field*, *data*)

Set float type property into the DMatrix
for numpy 2d array input

Parameters

- **field** (*str*) – The field name of the information
- **data** (*numpy array*) – The array of data to be set

Return type
None

`set_group`(*group*)

Set group size of DMatrix (used for ranking).

Parameters
group (*array like*) – Group size of each group

Return type
None

`set_info`(*, *label=None*, *weight=None*, *base_margin=None*, *group=None*, *qid=None*, *label_lower_bound=None*, *label_upper_bound=None*, *feature_names=None*, *feature_types=None*, *feature_weights=None*)

Set meta info for DMatrix. See doc string for `xgboost.DMatrix`.

Parameters

- **label** (*Any | None*)
- **weight** (*Any | None*)
- **base_margin** (*Any | None*)
- **group** (*Any | None*)
- **qid** (*Any | None*)

- **label_lower_bound** (*Any | None*)
- **label_upper_bound** (*Any | None*)
- **feature_names** (*Sequence[str] | None*)
- **feature_types** (*Sequence[str] | None*)
- **feature_weights** (*Any | None*)

Return type
None

`set_label`(*label*)

Set label of dmatrix

Parameters
label (*array like*) – The label information to be set into DMatrix

Return type
None

`set_uint_info`(*field*, *data*)

Set uint type property into the DMatrix.

Parameters

- **field** (*str*) – The field name of the information
- **data** (*numpy array*) – The array of data to be set

Return type
None

`set_weight`(*weight*)

Set weight of each instance.

Parameters
weight (*array like*) – Weight for each data point

Note

For ranking task, weights are per-group.

In ranking task, one weight is assigned to each group (not each data point). This is because we only care about the relative ordering of data points within each group, so it doesn't make sense to assign weights to individual data points.

Return type
None

`slice`(*rindex*, *allow_groups=False*)

Slice the DMatrix and return a new DMatrix that only contains *rindex*.

Parameters

- **rindex** (*List[int] | ndarray*) – List of indices to be selected.
- **allow_groups** (*bool*) – Allow slicing of a matrix with a groups attribute

Returns
A new DMatrix containing only selected indices.

Return type
res

class xgboost.**Booster**(*params=None*, *cache=None*, *model_file=None*)

Bases: object

A Booster of XGBoost.

Booster is the model of xgboost, that contains low level routines for training, prediction and evaluation.

Parameters

- **params** (*List | Dict[str, Any] | None*) – Parameters for boosters.
- **cache** (*Sequence[DMatrix] | None*) – List of cache items.
- **model_file** (Booster *| bytearray | PathLike | str | None*) – Path to the model file if it's string or PathLike.

__getitem__(*val*)

Get a slice of the tree-based model. Attributes like *best_iteration* and *best_score* are removed in the resulting booster.

Added in version 1.3.0.

Parameters
val (*int | integer | tuple | slice | EllipsisType*)

Return type
Booster

attr(*key*)

Get attribute string from the Booster.

Parameters
key (*str*) – The key to get attribute from.

Returns
The attribute value of the key, returns None if attribute do not exist.

Return type
value

attributes()

Get attributes stored in the Booster as a dictionary.

Returns
result – Returns an empty dict if there's no attributes.

Return type
dictionary of attribute_name: attribute_value pairs of strings.

property best_iteration: int

The best iteration during training.

property best_score: float

The best evaluation score during training.

boost(*dtrain*, *iteration*, ***, *grad=None*, *hess=None*, *fobj=None*)

Boost the booster for one iteration with customized gradient statistics.

> **Warning**
>
> Like `update()`, this function should not be called directly by users.

Parameters

- **dtrain** (DMatrix) – The training DMatrix.
- **iteration** (*int*) – The current training iteration.
- **grad** (*ndarray | Any | None*) – The first order of gradient.
- **hess** (*ndarray | Any | None*) – The second order of gradient.
- **fobj** (*Callable[[ndarray, DMatrix], Tuple[ndarray, ndarray]] | None*) – A custom objective function. If gradient is None, then an objective function is required.

Return type
None

copy()

Copy the booster object.

Returns
A copied booster model

Return type
booster

dump_model(*fout*, *fmap=''*, *with_stats=False*, *dump_format='text'*)

Dump model into a text or JSON file. Unlike `save_model()`, the output format is primarily used for visualization or interpretation, hence it's more human readable but cannot be loaded back to XGBoost.

Parameters

- **fout** (*str | PathLike*) – Output file name.
- **fmap** (*str | PathLike*) – Name of the file containing feature map names.
- **with_stats** (*bool*) – Controls whether the split statistics are output.
- **dump_format** (*str*) – Format of model dump file. Can be 'text' or 'json'.

Return type
None

eval(*data*, *name='eval'*, *iteration=0*)

Evaluate the model on mat.

Parameters

- **data** (DMatrix) – The dmatrix storing the input.
- **name** (*str*) – The name of the dataset.
- **iteration** (*int*) – The current iteration number.

Returns
result – Evaluation result string.

Return type
str

`eval_set`(*evals*, *iteration=0*, *feval=None*, *output_margin=True*)

Evaluate a set of data.

Parameters

- **evals** (*Sequence[Tuple[DMatrix, str]]*) – List of items to be evaluated.
- **iteration** (*int*) – Current iteration.
- **feval** (*Callable[[ndarray, DMatrix], Tuple[str, float]] | None*) – Custom evaluation function.
- **output_margin** (*bool*)

Returns

result – Evaluation result string.

Return type

str

property feature_names: Sequence[str] | None

Feature names for this booster. Can be directly set by input data or by assignment.

property feature_types: Sequence[str] | None

Feature types for this booster. Can be directly set by input data or by assignment. See *DMatrix* for details.

`get_categories`(*export_to_arrow=False*)

Same method as *DMatrix.get_categories()*.

Parameters

export_to_arrow (*bool*)

Return type

Categories

`get_dump`(*fmap=''*, *with_stats=False*, *dump_format='text'*)

Returns the model dump as a list of strings. Unlike *save_model()*, the output format is primarily used for visualization or interpretation, hence it's more human readable but cannot be loaded back to XGBoost.

Parameters

- **fmap** (*str | PathLike*) – Name of the file containing feature map names.
- **with_stats** (*bool*) – Controls whether the split statistics should be included.
- **dump_format** (*str*) – Format of model dump. Can be 'text', 'json' or 'dot'.

Return type

List[str]

`get_fscore`(*fmap=''*)

Get feature importance of each feature.

> **Note**
>
> Zero-importance features will not be included
>
> Keep in mind that this function does not include zero-importance feature, i.e. those features that have not been used in any split conditions.

Parameters

fmap (*str | PathLike*) – The name of feature map file

Return type
Dict[str, float | *List*[float]]

`get_score`(*fmap=''*, *importance_type='weight'*)
Get feature importance of each feature. For tree model Importance type can be defined as:

- **'weight': the number of times a feature is used to split the data across all**
  trees.
- 'gain': the average gain across all splits the feature is used in.
- 'cover': the average coverage across all splits the feature is used in.
- 'total_gain': the total gain across all splits the feature is used in.
- 'total_cover': the total coverage across all splits the feature is used in.

> **Note**
>
> For linear model, only "weight" is defined and it's the normalized coefficients without bias.

> **Note**
>
> Zero-importance features will not be included
>
> Keep in mind that this function does not include zero-importance feature, i.e. those features that have not been used in any split conditions.

Parameters

- **fmap** (*str | PathLike*) – The name of feature map file.
- **importance_type** (*str*) – One of the importance types defined above.

Returns

- A map between feature names and their scores. When *gblinear* is used for
- *multi-class classification the scores for each feature is a list with length*
- *n_classes*, otherwise they're scalars.

Return type
Dict[str, float | *List*[float]]

`get_split_value_histogram`(*feature*, *fmap=''*, *bins=None*, *as_pandas=True*)
Get split value histogram of a feature

Parameters

- **feature** (*str*) – The name of the feature.
- **fmap** (*str | PathLike*) – The name of feature map file.
- **bin** – The maximum number of bins. Number of bins equals number of unique split values n_unique, if bins == None or bins > n_unique.
- **as_pandas** (*bool*) – Return pd.DataFrame when pandas is installed. If False or pandas is not installed, return numpy ndarray.

- **bins** (*int | None*)

Returns

- *a histogram of used splitting values for the specified feature*
- *either as numpy array or pandas DataFrame.*

Return type

ndarray | PdDataFrame

inplace_predict(*data*, *, *iteration_range=(0, 0)*, *predict_type='value'*, *missing=nan*, *validate_features=True*, *base_margin=None*, *strict_shape=False*)

Run prediction in-place when possible, Unlike `predict()` method, inplace prediction does not cache the prediction result.

Calling only `inplace_predict` in multiple threads is safe and lock free. But the safety does not hold when used in conjunction with other methods. E.g. you can't train the booster in one thread and perform prediction in the other.

Note

If the device ordinal of the input data doesn't match the one configured for the booster, data will be copied to the booster device.

```
booster.set_param({"device": "cuda:0"})
booster.inplace_predict(cupy_array)

booster.set_param({"device": "cpu"})
booster.inplace_predict(numpy_array)
```

Added in version 1.1.0.

Parameters

- **data** (*Any*) – The input data.
- **iteration_range** (*Tuple[int | integer, int | integer]*) – See `predict()` for details.
- **predict_type** (*str*) –
  - *value* Output model prediction values.
  - *margin* Output the raw untransformed margin value.
- **missing** (*float*) – See `xgboost.DMatrix` for details.
- **validate_features** (*bool*) – See `xgboost.Booster.predict()` for details.
- **base_margin** (*Any*) – See `xgboost.DMatrix` for details.

  Added in version 1.4.0.
- **strict_shape** (*bool*) – See `xgboost.Booster.predict()` for details.

  Added in version 1.4.0.

Returns

prediction – The prediction result. When input data is on GPU, prediction result is stored in a cupy array.

Return type
numpy.ndarray/cupy.ndarray

load_config(*config*)

Load configuration returned by *save_config*.

Added in version 1.0.0.

Parameters
config (*str*)

Return type
None

load_model(*fname*)

Load the model from a file or a bytearray.

The model is saved in an XGBoost internal format which is universal among the various XGBoost interfaces. Auxiliary attributes of the Python Booster object (such as feature_names) are only saved when using JSON or UBJSON (default) format. Also, parameters that are not part of the model (like metrics, *max_depth*, etc) are not saved, see *Model IO* for more info.

```
model.save_model("model.json")
model.load_model("model.json")

# or
model.save_model("model.ubj")
model.load_model("model.ubj")

# or
buf = model.save_raw()
model.load_model(buf)
```

Parameters
fname (*PathLike | bytearray | str*) – Input file name or memory buffer(see also save_raw)

Return type
None

num_boosted_rounds()

Get number of boosted rounds. For gblinear this is reset to 0 after serializing the model.

Return type
int

num_features()

Number of features in booster.

Return type
int

predict(*data*, ***, *output_margin=False*, *pred_leaf=False*, *pred_contribs=False*, *approx_contribs=False*, *pred_interactions=False*, *validate_features=True*, *training=False*, *iteration_range=(0, 0)*, *strict_shape=False*)

Predict with data. The full model will be used unless *iteration_range* is specified, meaning users have to either slice the model or use the `best_iteration` attribute to get prediction from best model returned from early stopping.

Note

See *Prediction* for issues like thread safety and a summary of outputs from this function.

Parameters

- **data** (DMatrix) – The dmatrix storing the input.
- **output_margin** (*bool*) – Whether to output the raw untransformed margin value.
- **pred_leaf** (*bool*) – When this option is on, the output will be a matrix of (nsample, ntrees) with each record indicating the predicted leaf index of each sample in each tree. Note that the leaf index of a tree is unique per tree, so you may find leaf 1 in both tree 1 and tree 0.
- **pred_contribs** (*bool*) – When this is True the output will be a matrix of size (nsample, nfeats + 1) with each record indicating the feature contributions (SHAP values) for that prediction. The sum of all feature contributions is equal to the raw untransformed margin value of the prediction. Note the final column is the bias term.
- **approx_contribs** (*bool*) – Approximate the contributions of each feature. Used when `pred_contribs` or `pred_interactions` is set to True. Changing the default of this parameter (False) is not recommended.
- **pred_interactions** (*bool*) – When this is True the output will be a matrix of size (nsample, nfeats + 1, nfeats + 1) indicating the SHAP interaction values for each pair of features. The sum of each row (or column) of the interaction values equals the corresponding SHAP value (from pred_contribs), and the sum of the entire matrix equals the raw untransformed margin value of the prediction. Note the last row and column correspond to the bias term.
- **validate_features** (*bool*) – When this is True, validate that the Booster's and data's feature_names are identical. Otherwise, it is assumed that the feature_names are the same.
- **training** (*bool*) – Whether the prediction value is used for training. This can effect *dart* booster, which performs dropouts during training iterations but use all trees for inference. If you want to obtain result with dropouts, set this parameter to *True*. Also, the parameter is set to true when obtaining prediction for custom objective function.

  Added in version 1.0.0.
- **iteration_range** (*Tuple[int | integer, int | integer]*) – Specifies which layer of trees are used in prediction. For example, if a random forest is trained with 100 rounds. Specifying *iteration_range=(10, 20)*, then only the forests built during [10, 20) (half open set) rounds are used in this prediction.

  Added in version 1.4.0.
- **strict_shape** (*bool*) – When set to True, output shape is invariant to whether classification is used. For both value and margin prediction, the output shape is (n_samples, n_groups), n_groups == 1 when multi-class is not used. Default to False, in which case the output shape can be (n_samples,) if multi-class is not used.

  Added in version 1.4.0.

Returns

prediction

Return type

numpy array

reset()

Reset the booster object to release data caches used for training.

Added in version 3.0.0.

Return type
Booster

save_config()

Output internal parameter configuration of Booster as a JSON string.

Added in version 1.0.0.

Return type
str

save_model(*fname*)

Save the model to a file.

The model is saved in an XGBoost internal format which is universal among the various XGBoost interfaces. Auxiliary attributes of the Python Booster object (such as feature_names) are only saved when using JSON or UBJSON (default) format. Also, parameters that are not part of the model (like metrics, *max_depth*, etc) are not saved, see *Model IO* for more info.

```
model.save_model("model.json")
# or
model.save_model("model.ubj")
```

Parameters
fname (*str* | *PathLike*) – Output file name

Return type
None

save_raw(*raw_format='ubj'*)

Save the model to a in memory buffer representation instead of file.

The model is saved in an XGBoost internal format which is universal among the various XGBoost interfaces. Auxiliary attributes of the Python Booster object (such as feature_names) are only saved when using JSON or UBJSON (default) format. Also, parameters that are not part of the model (like metrics, *max_depth*, etc) are not saved, see *Model IO* for more info.

Parameters
raw_format (*str*) – Format of output buffer. Can be *json* or *ubj*.

Return type
An in memory buffer representation of the model

set_attr(***kwargs*)

Set the attribute of the Booster.

Parameters
****kwargs** (*Any* | *None*) – The attributes to set. Setting a value to None deletes an attribute.

Return type
None

set_param(*params*, *value=None*)

Set parameters into the Booster.

Parameters

- **params** (*Dict | Iterable[Tuple[str, Any]] | str*) – list of key,value pairs, dict of key to value or simply str key
- **value** (*str | None*) – value of the specified parameter, when params is str key

Return type
None

trees_to_dataframe(*fmap=''*)

Parse a boosted tree model text dump into a pandas DataFrame structure.

This feature is only defined when the decision tree model is chosen as base learner (*booster in {gbtree, dart}*). It is not defined for other base learner types, such as linear learners (*booster=gblinear*).

Parameters
fmap (*str | PathLike*) – The name of feature map file.

Return type
PdDataFrame

update(*dtrain*, *iteration*, *fobj=None*)

Update for one iteration, with objective function calculated internally.

Warning

This function should not be called directly by users.

Parameters

- **dtrain** (DMatrix) – Training data.
- **iteration** (*int*) – The current training iteration.
- **fobj** (*Callable[[ndarray, DMatrix], Tuple[ndarray, ndarray]] | None*) – Custom objective function.

Return type
None

class xgboost.**DataIter**(*cache_prefix=None*, *release_data=True*, ***, *on_host=True*, *min_cache_page_bytes=None*)

Bases: ABC

The interface for user defined data iterator. The iterator facilitates distributed training, *QuantileDMatrix*, and external memory support using *DMatrix* or *ExtMemQuantileDMatrix*. Most of time, users don't need to interact with this class directly.

Note

The class caches some intermediate results using the *data* input (predictor *X*) as key. Don't repeat the *X* for multiple batches with different meta data (like *label*), make a copy if necessary.

Note

When the input for each batch is a DataFrame, we assume categories are consistently encoded for all batches. For example, given two dataframes for two batches, this is invalid:

```
import pandas as pd

x0 = pd.DataFrame({"a": [0, 1]}, dtype="category")
x1 = pd.DataFrame({"a": [1, 2]}, dtype="category")
```

This is invalid because the *x0* has *[0, 1]* as categories while *x2* has *[1, 2]*. They should share the same set of categories and encoding:

```
import numpy as np

categories = np.array([0, 1, 2])
x0["a"] = pd.Categorical.from_codes(
    codes=np.array([0, 1]), categories=categories
)
x1["a"] = pd.Categorical.from_codes(
    codes=np.array([1, 2]), categories=categories
)
```

You can make sure the consistent encoding in your preprocessing step be careful that the data is stored in formats that preserve the encoding when chunking the data.

Parameters

- **cache_prefix** (*str | None*) – Prefix to the cache files, only used in external memory.

  Note that using this class for external memory **will cache data on disk** under the path passed here.

- **release_data** (*bool*) – Whether the iterator should release the data during iteration. Set it to True if the data transformation (converting data to np.float32 type) is memory intensive. Otherwise, if the transformation is computation intensive then we can keep the cache.

- **on_host** (*bool*) – Whether the data should be cached on the host memory instead of the file system when using GPU with external memory. When set to true (the default), the "external memory" is the CPU (host) memory. See *Using XGBoost External Memory Version* for more info.

  Added in version 3.0.0.

  Warning

  This is an experimental parameter and subject to change.

- **min_cache_page_bytes** (*int | None*) – The minimum number of bytes of each cached pages. Only used for on-host cache with GPU-based `ExtMemQuantileDMatrix`. When using GPU-based external memory with the data cached in the host memory, XGBoost can concatenate the pages internally to increase the batch size for the GPU. The default page size is about 1/16 of the total device memory. Users can manually set the value based on the actual hardware and datasets. Set this to 0 to disable page concatenation.

  Added in version 3.0.0.

> **Warning**
>
> This is an experimental parameter and subject to change.

get_callbacks(*enable_categorical*)

Get callback functions for iterating in C. This is an internal function.

Parameters
enable_categorical (*bool*)

Return type
Tuple[*Callable*, *Callable*]

abstractmethod next(*input_data*)

Set the next batch of data.

Parameters
input_data (*Callable*) – A function with same data fields like *data*, *label* with *xgboost.DMatrix*.

Return type
False if there's no more batch, otherwise True.

property proxy: _ProxyDMatrix

Handle of DMatrix proxy.

reraise()

Reraise the exception thrown during iteration.

Return type
None

abstractmethod reset()

Reset the data iterator. Prototype for user defined function.

Return type
None

class xgboost.core.**Categories**(*handle*, *arrow_arrays*)

An internal storage class for categories returned by the DMatrix and the Booster. This class is designed to be opaque. It is intended to be used exclusively by XGBoost as an intermediate storage for re-coding categorical data.

The categories are saved along with the booster object. As a result, users don't need to preserve this class for re-coding. Use the booster model IO instead if you want to preserve the categories in a stable format.

Added in version 3.1.0.

> **Warning**
>
> This class is internal.

```
Xy = xgboost.QuantileDMatrix(X, y, enable_categorical=True)
booster = xgboost.train({}, Xy)

categories = booster.get_categories() # Get categories
```

(continues on next page)

(continued from previous page)

```
# Use categories as a reference for re-coding
Xy_new = xgboost.QuantileDMatrix(
    X_new, y_new, feature_types=categories, enable_categorical=True, ref=Xy
)

# Categories will be part of the `model.json`.
booster.save_model("model.json")
```

Parameters

- **handle** (*Tuple[c_void_p, Callable[[], None]]*)
- **arrow_arrays** (*List[Tuple[str, pa.StringArray | pa.NumericArray | None]] | None*)

Learning API

Training Library containing training routines.

`xgboost.train`(*params*, *dtrain*, *num_boost_round=10*, ***, *evals=None*, *obj=None*, *maximize=None*, *early_stopping_rounds=None*, *evals_result=None*, *verbose_eval=True*, *xgb_model=None*, *callbacks=None*, *custom_metric=None*)

Train a booster with given parameters.

Parameters

- **params** (*Dict[str, Any]*) – Booster params.
- **dtrain** (DMatrix) – Data to be trained.
- **num_boost_round** (*int*) – Number of boosting iterations.
- **evals** (*Sequence[Tuple[DMatrix, str]] | None*) – List of validation sets for which metrics will evaluated during training. Validation metrics will help us track the performance of the model.
- **obj** (*Callable[[ndarray, DMatrix], Tuple[ndarray, ndarray]] | None*) – Custom objective function. See *Custom Objective* for details.
- **maximize** (*bool | None*) – Whether to maximize custom_metric.
- **early_stopping_rounds** (*int | None*) – Activates early stopping. Validation metric needs to improve at least once in every **early_stopping_rounds** round(s) to continue training.

  Requires at least one item in **evals**.

  The method returns the model from the last iteration (not the best one). Use custom callback *EarlyStopping* or *model slicing* if the best model is desired. If there's more than one item in **evals**, the last entry will be used for early stopping.

  If there's more than one metric in the **eval_metric** parameter given in **params**, the last metric will be used for early stopping.

  If early stopping occurs, the model will have two additional fields: `bst.best_score`, `bst.best_iteration`.
- **evals_result** (*Dict[str, Dict[str, List[float] | List[Tuple[float, float]]]] | None*) – This dictionary stores the evaluation results of all the items in watchlist.

Example: with a watchlist containing `[(dtest,'eval'), (dtrain,'train')]` and a parameter containing `('eval_metric': 'logloss')`, the **evals_result** returns

```
{'train': {'logloss': ['0.48253', '0.35953']},
 'eval': {'logloss': ['0.480385', '0.357756']}}
```

- **verbose_eval** (*bool | int | None*) – Requires at least one item in **evals**.

  If **verbose_eval** is True then the evaluation metric on the validation set is printed at each boosting stage.

  If **verbose_eval** is an integer then the evaluation metric on the validation set is printed at every given **verbose_eval** boosting stage. The last boosting stage / the boosting stage found by using **early_stopping_rounds** is also printed.

  Example: with `verbose_eval=4` and at least one item in **evals**, an evaluation metric is printed every 4 boosting stages, instead of every boosting stage.

- **xgb_model** (*str | PathLike | Booster | bytearray | None*) – Xgb model to be loaded before training (allows training continuation).

- **callbacks** (*Sequence[TrainingCallback] | None*) – List of callback functions that are applied at end of each iteration. It is possible to use predefined callbacks by using *Callback API*.

  Note

  States in callback are not preserved during training, which means callback objects can not be reused for multiple training sessions without reinitialization or deepcopy.

```
for params in parameters_grid:
    # be sure to (re)initialize the callbacks before each run
    callbacks = [xgb.callback.LearningRateScheduler(custom_rates)]
    xgboost.train(params, Xy, callbacks=callbacks)
```

- **custom_metric** (*Callable[[ndarray, DMatrix], Tuple[str, float]] | None*) – Custom metric function. See *Custom Metric* for details. The metric receives transformed prediction (after applying the reverse link function) when using a builtin objective, and raw output when using a custom objective.

Returns
  Booster

Return type
  a trained booster model

`xgboost.cv`(*params*, *dtrain*, *num_boost_round=10*, ***, *nfold=3*, *stratified=False*, *folds=None*, *metrics=()*, *obj=None*, *maximize=None*, *early_stopping_rounds=None*, *fpreproc=None*, *as_pandas=True*, *verbose_eval=None*, *show_stdv=True*, *seed=0*, *callbacks=None*, *shuffle=True*, *custom_metric=None*)

Cross-validation with given parameters.

Parameters

- **params** (*dict*) – Booster params.
- **dtrain** (*DMatrix*) – Data to be trained. Only the `DMatrix` without external memory is supported.
- **num_boost_round** (*int*) – Number of boosting iterations.

- **nfold** (*int*) – Number of folds in CV.
- **stratified** (*bool*) – Perform stratified sampling.
- **folds** (*a KFold or StratifiedKFold instance or list of fold indices*) – Sklearn KFolds or StratifiedKFolds object. Alternatively may explicitly pass sample indices for each fold. For `n` folds, **folds** should be a length `n` list of tuples. Each tuple is `(in,out)` where `in` is a list of indices to be used as the training samples for the `n` th fold and `out` is a list of indices to be used as the testing samples for the `n` th fold.
- **metrics** (*string or list of strings*) – Evaluation metrics to be watched in CV.
- **obj** (*Callable[[ndarray, DMatrix], Tuple[ndarray, ndarray]] | None*) – Custom objective function. See *Custom Objective* for details.
- **maximize** (*bool*) – Whether to maximize the evaluataion metric (score or error).
- **early_stopping_rounds** (*int*) – Activates early stopping. Cross-Validation metric (average of validation metric computed over CV folds) needs to improve at least once in every **early_stopping_rounds** round(s) to continue training. The last entry in the evaluation history will represent the best iteration. If there's more than one metric in the **eval_metric** parameter given in **params**, the last metric will be used for early stopping.
- **fpreproc** (*function*) – Preprocessing function that takes (dtrain, dtest, param) and returns transformed versions of those.
- **as_pandas** (*bool, default True*) – Return pd.DataFrame when pandas is installed. If False or pandas is not installed, return np.ndarray
- **verbose_eval** (*bool, int, or None, default None*) – Whether to display the progress. If None, progress will be displayed when np.ndarray is returned. If True, progress will be displayed at boosting stage. If an integer is given, progress will be displayed at every given *verbose_eval* boosting stage.
- **show_stdv** (*bool, default True*) – Whether to display the standard deviation in progress. Results are not affected, and always contains std.
- **seed** (*int*) – Seed used to generate the folds (passed to numpy.random.seed).
- **callbacks** (*Sequence[TrainingCallback] | None*) – List of callback functions that are applied at end of each iteration. It is possible to use predefined callbacks by using *Callback API*.

  Note

  States in callback are not preserved during training, which means callback objects can not be reused for multiple training sessions without reinitialization or deepcopy.

  ```
  for params in parameters_grid:
      # be sure to (re)initialize the callbacks before each run
      callbacks = [xgb.callback.LearningRateScheduler(custom_rates)]
      xgboost.train(params, Xy, callbacks=callbacks)
  ```

- **shuffle** (*bool*) – Shuffle data before creating folds.
- **custom_metric** (*Callable[[ndarray, DMatrix], Tuple[str, float]] | None*) – Custom metric function. See *Custom Metric* for details.

Returns
: **evaluation history**

Return type
list(string)

Scikit-Learn API

Scikit-Learn Wrapper interface for XGBoost.

class xgboost.**XGBRegressor**(*, *objective='reg:squarederror'*, ***kwargs*)

Bases: `RegressorMixin`, `XGBModel`

Implementation of the scikit-learn API for XGBoost regression. See *Using the Scikit-Learn Estimator Interface* for more information.

Parameters

- **n_estimators** (*Optional[int]*) – Number of gradient boosted trees. Equivalent to number of boosting rounds.
- **max_depth** (*Optional[int]*) – Maximum tree depth for base learners.
- **max_leaves** (*Optional[int]*) – Maximum number of leaves; 0 indicates no limit.
- **max_bin** (*Optional[int]*) – If using histogram-based algorithm, maximum number of bins per feature
- **grow_policy** (*Optional[str]*) – Tree growing policy.
  - depthwise: Favors splitting at nodes closest to the node,
  - lossguide: Favors splitting at nodes with highest loss change.
- **learning_rate** (*Optional[float]*) – Boosting learning rate (xgb's "eta")
- **verbosity** (*Optional[int]*) – The degree of verbosity. Valid values are 0 (silent) - 3 (debug).
- **objective** (*Union[str, xgboost.sklearn._SklObjWProto, Callable[[Any, Any], Tuple[numpy.ndarray, numpy.ndarray]], NoneType]*) – Specify the learning task and the corresponding learning objective or a custom objective function to be used.

  For custom objective, see *Custom Objective and Evaluation Metric* and *Custom objective and metric* for more information, along with the end note for function signatures.
- **booster** (*Optional[str]*) – Specify which booster to use: `gbtree`, `gblinear` or `dart`.

  Deprecated since version 3.3.0: `gblinear` is deprecated and support will be removed in a future release.
- **tree_method** (*Optional[str]*) – Specify which tree method to use. Default to auto. If this parameter is set to default, XGBoost will choose the most conservative option available. It's recommended to study this option from the parameters document *tree method*
- **n_jobs** (*Optional[int]*) – Number of parallel threads used to run xgboost. When used with other Scikit-Learn algorithms like grid search, you may choose which algorithm to parallelize and balance the threads. Creating thread contention will significantly slow down both algorithms.
- **gamma** (*Optional[float]*) – (min_split_loss) Minimum loss reduction required to make a further partition on a leaf node of the tree.
- **min_child_weight** (*Optional[float]*) – Minimum sum of instance weight(hessian) needed in a child.

- **max_delta_step** (*Optional[float]*) – Maximum delta step we allow each tree's weight estimation to be.
- **subsample** (*Optional[float]*) – Subsample ratio of the training instance.
- **sampling_method** (*Optional[str]*) – Sampling method. Used only by the GPU version of `hist` tree method.
  - `uniform`: Select random training instances uniformly.
  - **gradient_based: Select random training instances with higher probability**
    when the gradient and hessian are larger. (cf. CatBoost)
- **colsample_bytree** (*Optional[float]*) – Subsample ratio of columns when constructing each tree.
- **colsample_bylevel** (*Optional[float]*) – Subsample ratio of columns for each level.
- **colsample_bynode** (*Optional[float]*) – Subsample ratio of columns for each split.
- **reg_alpha** (*Optional[float]*) – L1 regularization term on weights (xgb's alpha).
- **reg_lambda** (*Optional[float]*) – L2 regularization term on weights (xgb's lambda).
- **scale_pos_weight** (*Optional[float]*) – Balancing of positive and negative weights.
- **base_score** (*Union[float, List[float], NoneType]*) – The initial prediction score of all instances, global bias.
- **random_state** (*Union[numpy.random.mtrand.RandomState, numpy.random._generator.Generator, int, NoneType]*) – Random number seed.

  > **Note**
  >
  > Using gblinear booster with shotgun updater is nondeterministic as it uses Hogwild algorithm.

- **missing** (*float*) – Value in the data which needs to be present as a missing value. Default to `numpy.nan`.
- **num_parallel_tree** (*Optional[int]*) – Used for boosting random forest.
- **monotone_constraints** (*Union[Dict[str, int], str, NoneType]*) – Constraint of variable monotonicity. See *tutorial* for more information.
- **interaction_constraints** (*Union[str, List[Tuple[str]], NoneType]*) – Constraints for interaction representing permitted interactions. The constraints must be specified in the form of a nested list, e.g. `[[0, 1], [2, 3, 4]]`, where each inner list is a group of indices of features that are allowed to interact with each other. See *tutorial* for more information
- **importance_type** (*Optional[str]*) – The feature importance type for the feature_importances_ property:
  - For tree model, it's either "gain", "weight", "cover", "total_gain" or "total_cover".
  - For linear model, only "weight" is defined and it's the normalized coefficients without bias.
- **device** (*Optional[str]*) – Added in version 2.0.0.

  Device ordinal, available options are *cpu*, *cuda*, and *gpu*.

- **validate_parameters** (*Optional[bool]*) – Give warnings for unknown parameter.
- **enable_categorical** (*bool*) – See the same parameter of *DMatrix* for details.
- **feature_types** (*Optional[Sequence[str]]*) – Added in version 1.7.0.

  Used for specifying feature types without constructing a dataframe. See the *DMatrix* for details.
- **feature_weights** (*Optional[ArrayLike]*) – Weight for each feature, defines the probability of each feature being selected when colsample is being used. All values must be greater than 0, otherwise a *ValueError* is thrown.
- **max_cat_to_onehot** (*Optional[int]*) – Added in version 1.6.0.

  Note

  This parameter is experimental

  A threshold for deciding whether XGBoost should use one-hot encoding based split for categorical data. When number of categories is lesser than the threshold then one-hot encoding is chosen, otherwise the categories will be partitioned into children nodes. Also, *enable_categorical* needs to be set to have categorical feature support. See *Categorical Data* and *Parameters for Categorical Feature* for details.
- **max_cat_threshold** (*Optional[int]*) – Added in version 1.7.0.

  Note

  This parameter is experimental

  Maximum number of categories considered for each split. Used only by partition-based splits for preventing over-fitting. Also, *enable_categorical* needs to be set to have categorical feature support. See *Categorical Data* and *Parameters for Categorical Feature* for details.
- **multi_strategy** (*Optional[str]*) – Added in version 2.0.0.

  Note

  This parameter is working-in-progress.

  The strategy used for training multi-target models, including multi-target regression and multi-class classification. See *Multiple Outputs* for more information.

  - `one_output_per_tree`: One model for each target.
  - `multi_output_tree`: Use multi-target trees.
- **eval_metric** (*Union[str, List[Union[str, Callable]], Callable, NoneType]*) – Added in version 1.6.0.

  Metric used for monitoring the training result and early stopping. It can be a string or list of strings as names of predefined metric in XGBoost (See *XGBoost Parameters*), one of the metrics in `sklearn.metrics`, or any other user defined metric that looks like *sklearn.metrics*.

  If custom objective is also provided, then custom metric should implement the corresponding reverse link function.

Unlike the *scoring* parameter commonly used in scikit-learn, when a callable object is provided, it's assumed to be a cost function and by default XGBoost will minimize the result during early stopping.

For advanced usage on Early stopping like directly choosing to maximize instead of minimize, see `xgboost.callback.EarlyStopping`.

See *Custom Objective and Evaluation Metric* and *Custom objective and metric* for more information.

```
from sklearn.datasets import load_diabetes
from sklearn.metrics import mean_absolute_error
X, y = load_diabetes(return_X_y=True)
reg = xgb.XGBRegressor(
    tree_method="hist",
    eval_metric=mean_absolute_error,
)
reg.fit(X, y, eval_set=[(X, y)])
```

- **early_stopping_rounds** (*Optional[int]*) – Added in version 1.6.0.
  - Activates early stopping. Validation metric needs to improve at least once in every **early_stopping_rounds** round(s) to continue training. Requires at least one item in **eval_set** in `fit()`.
  - If early stopping occurs, the model will have two additional attributes: `best_score` and `best_iteration`. These are used by the `predict()` and `apply()` methods to determine the optimal number of trees during inference. If users want to access the full model (including trees built after early stopping), they can specify the *iteration_range* in these inference methods. In addition, other utilities like model plotting can also use the entire model.
  - If you prefer to discard the trees after *best_iteration*, consider using the callback function `xgboost.callback.EarlyStopping`.
  - If there's more than one item in **eval_set**, the last entry will be used for early stopping. If there's more than one metric in **eval_metric**, the last metric will be used for early stopping.
- **callbacks** (*Optional[List[xgboost.callback.TrainingCallback]]*) – List of callback functions that are applied at end of each iteration. It is possible to use predefined callbacks by using *Callback API*.

  Note

  States in callback are not preserved during training, which means callback objects can not be reused for multiple training sessions without reinitialization or deepcopy.

```
for params in parameters_grid:
    # be sure to (re)initialize the callbacks before each run
    callbacks = [xgb.callback.LearningRateScheduler(custom_rates)]
    reg = xgboost.XGBRegressor(**params, callbacks=callbacks)
    reg.fit(X, y)
```

- **kwargs** (*Optional[Any]*) – Keyword arguments for XGBoost Booster object. Full documentation of parameters can be found *here*. Attempting to set a parameter via the constructor args and **kwargs dict simultaneously will result in a TypeError.

> **Note**
>
> **kwargs unsupported by scikit-learn
>
> **kwargs is unsupported by scikit-learn. We do not guarantee that parameters passed via this argument will interact properly with scikit-learn.

> **Note**
>
> Custom objective function
>
> A custom objective function can be provided for the `objective` parameter. In this case, it should have the signature `objective(y_true, y_pred) -> [grad, hess]` or `objective(y_true, y_pred, *, sample_weight) -> [grad, hess]`:
>
> **y_true: array_like of shape [n_samples]**
> The target values
>
> **y_pred: array_like of shape [n_samples]**
> The predicted values
>
> **sample_weight :**
> Optional sample weights.
>
> **grad: array_like of shape [n_samples]**
> The value of the gradient for each sample point.
>
> **hess: array_like of shape [n_samples]**
> The value of the second derivative for each sample point
>
> Note that, if the custom objective produces negative values for the Hessian, these will be clipped. If the objective is non-convex, one might also consider using the expected Hessian (Fisher information) instead.

apply(*X*, *iteration_range=None*)

Return the predicted leaf every tree for each sample. If the model is trained with early stopping, then *best_iteration* is used automatically.

Parameters

- **X** (*Any*) – Input features matrix. See *Markers* for a list of supported types.
- **iteration_range** (*Tuple[int | integer, int | integer] | None*) – See *predict()*.

Returns

X_leaves – For each datapoint x in X and for each tree, return the index of the leaf x ends up in. Leaves are numbered within `[0; 2**(self.max_depth+1))`, possibly with gaps in the numbering.

Return type

array_like, shape=[n_samples, n_trees]

property best_iteration: int

The best iteration obtained by early stopping. This attribute is 0-based, for instance if the best iteration is the first round, then best_iteration is 0.

property best_score: float

The best score obtained by early stopping.

property coef_: ndarray

Coefficients property

> **Note**
>
> Coefficients are defined only for linear learners
>
> Coefficients are only defined when the linear model is chosen as base learner (*booster=gblinear*). It is not defined for other base learner types, such as tree learners (*booster=gbtree*).

Returns
coef_

Return type
array of shape `[n_features]` or `[n_classes, n_features]`

evals_result()

Return the evaluation results.

If **eval_set** is passed to the `fit()` function, you can call `evals_result()` to get evaluation results for all passed **eval_sets**. When **eval_metric** is also passed to the `fit()` function, the **evals_result** will contain the **eval_metrics** passed to the `fit()` function.

The returned evaluation result is a dictionary:

```
{'validation_0': {'logloss': ['0.604835', '0.531479']},
 'validation_1': {'logloss': ['0.41965', '0.17686']}}
```

Return type
evals_result

property feature_importances_: ndarray

Feature importances property, return depends on *importance_type* parameter. When model trained with multi-class/multi-label/multi-target dataset, the feature importance is “averaged” over all targets. The “average” is defined based on the importance type. For instance, if the importance type is “total_gain”, then the score is sum of loss change for each split from all trees.

Returns

- **feature_importances_** (array of shape `[n_features]` except for multi-class)
- linear model, which returns an array with shape *(n_features, n_classes)*

property feature_names_in_: ndarray

Names of features seen during `fit()`. Defined only when *X* has feature names that are all strings.

fit(*X*, *y*, ***, *sample_weight=None*, *base_margin=None*, *eval_set=None*, *verbose=True*, *xgb_model=None*, *sample_weight_eval_set=None*, *base_margin_eval_set=None*, *feature_weights=None*)

Fit gradient boosting model.

Note that calling `fit()` multiple times will cause the model object to be re-fit from scratch. To resume training from a previous checkpoint, explicitly pass `xgb_model` argument.

Parameters

- **X** (*Any*) – Input feature matrix. See *Markers* for a list of supported types.

  When the `tree_method` is set to `hist`, internally, the `QuantileDMatrix` will be used instead of the `DMatrix` for conserving memory. However, this has performance implications when the device of input data is not matched with algorithm. For instance, if the input is a numpy array on CPU but `cuda` is used for training, then the data is first processed on CPU then transferred to GPU.

- **y** (*Any*) – Labels
- **sample_weight** (*Any | None*) – instance weights
- **base_margin** (*Any | None*) – Global bias for each instance. See *Intercept* for details.
- **eval_set** (*Sequence[Tuple[Any, Any]] | None*) – A list of (X, y) tuple pairs to use as validation sets, for which metrics will be computed. Validation metrics will help us track the performance of the model.
- **verbose** (*bool | int | None*) – If *verbose* is True and an evaluation set is used, the evaluation metric measured on the validation set is printed to stdout at each boosting stage. If *verbose* is an integer, the evaluation metric is printed at each *verbose* boosting stage. The last boosting stage / the boosting stage found by using *early_stopping_rounds* is also printed.
- **xgb_model** (Booster *| XGBModel | str | None*) – file name of stored XGBoost model or 'Booster' instance XGBoost model to be loaded before training (allows training continuation).
- **sample_weight_eval_set** (*Sequence[Any] | None*) – A list of the form [L_1, L_2, …, L_n], where each L_i is an array like object storing instance weights for the i-th validation set.
- **base_margin_eval_set** (*Sequence[Any] | None*) – A list of the form [M_1, M_2, …, M_n], where each M_i is an array like object storing base margin for the i-th validation set.
- **feature_weights** (*Any | None*) – Deprecated since version 3.0.0.

  Use *feature_weights* in `__init__()` or `set_params()` instead.

Return type
XGBModel

get_booster()

Get the underlying xgboost Booster of this model.

This will raise an exception when fit was not called

Returns
booster

Return type
a xgboost booster of underlying model

get_metadata_routing()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns
routing – A `MetadataRequest` encapsulating routing information.

Return type
MetadataRequest

`get_num_boosting_rounds()`

Gets the number of xgboost boosting rounds.

Return type
int

`get_params(deep=True)`

Get parameters.

Parameters
deep (*bool*)

Return type
Dict[str, *Any*]

`get_xgb_params()`

Get xgboost specific parameters.

Return type
Dict[str, *Any*]

`property intercept_: ndarray`

Intercept (bias) property

For tree-based model, the returned value is the *base_score*.

Returns
intercept_

Return type
array of shape `(1,)` or `[n_classes]`

`load_model(fname)`

Load the model from a file or a bytearray.

The model is saved in an XGBoost internal format which is universal among the various XGBoost interfaces. Auxiliary attributes of the Python Booster object (such as feature_names) are only saved when using JSON or UBJSON (default) format. Also, parameters that are not part of the model (like metrics, *max_depth*, etc) are not saved, see *Model IO* for more info.

```
model.save_model("model.json")
model.load_model("model.json")

# or
model.save_model("model.ubj")
model.load_model("model.ubj")

# or
buf = model.save_raw()
model.load_model(buf)
```

Parameters
fname (*PathLike | bytearray | str*) – Input file name or memory buffer(see also save_raw)

Return type
None

property n_features_in_: int

Number of features seen during *fit()*.

predict(*X*, *, *output_margin=False*, *validate_features=True*, *base_margin=None*, *iteration_range=None*)

Predict with *X*. If the model is trained with early stopping, then *best_iteration* is used automatically. The estimator uses *inplace_predict* by default and falls back to using *DMatrix* if devices between the data and the estimator don't match.

> **Note**
>
> This function is only thread safe for *gbtree* and *dart*.

Parameters

- **X** (*Any*) – Data to predict with. See *Markers* for a list of supported types.
- **output_margin** (*bool*) – Whether to output the raw untransformed margin value.
- **validate_features** (*bool*) – When this is True, validate that the Booster's and data's feature_names are identical. Otherwise, it is assumed that the feature_names are the same.
- **base_margin** (*Any | None*) – Global bias for each instance. See *Intercept* for details.
- **iteration_range** (*Tuple[int | integer, int | integer] | None*) – Specifies which layer of trees are used in prediction. For example, if a random forest is trained with 100 rounds. Specifying `iteration_range=(10, 20)`, then only the forests built during [10, 20) (half open set) rounds are used in this prediction.

  Added in version 1.4.0.

Return type

prediction

save_model(*fname*)

Save the model to a file.

The model is saved in an XGBoost internal format which is universal among the various XGBoost interfaces. Auxiliary attributes of the Python Booster object (such as feature_names) are only saved when using JSON or UBJSON (default) format. Also, parameters that are not part of the model (like metrics, *max_depth*, etc) are not saved, see *Model IO* for more info.

```
model.save_model("model.json")
# or
model.save_model("model.ubj")
```

Parameters

fname (*str | PathLike*) – Output file name

Return type

None

score(*X*, *y*, *sample_weight=None*)

Return coefficient of determination on test data.

The coefficient of determination, R^2, is defined as $(1 - \frac{u}{v})$, where u is the residual sum of squares `((y_true - y_pred)** 2).sum()` and v is the total sum of squares `((y_true - y_true.mean()) ** 2).sum()`. The best possible score is 1.0 and it can be negative (because the model can be arbitrarily

worse). A constant model that always predicts the expected value of *y*, disregarding the input features, would get a R^2 score of 0.0.

Parameters

- **X** (*array-like of shape (n_samples, n_features)*) – Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead with shape `(n_samples, n_samples_fitted)`, where `n_samples_fitted` is the number of samples used in the fitting for the estimator.
- **y** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – True values for *X*.
- **sample_weight** (*array-like of shape (n_samples,), default=None*) – Sample weights.

Returns
score – R^2 of `self.predict(X)` w.r.t. *y*.

Return type
float

Notes

The R^2 score used when calling `score` on a regressor uses `multioutput='uniform_average'` from version 0.23 to keep consistent with default value of `r2_score()`. This influences the `score` method of all the multioutput regressors (except for `MultiOutputRegressor`).

set_fit_request(**, base_margin='$UNCHANGED$', base_margin_eval_set='$UNCHANGED$', eval_set='$UNCHANGED$', feature_weights='$UNCHANGED$', sample_weight='$UNCHANGED$', sample_weight_eval_set='$UNCHANGED$', verbose='$UNCHANGED$', xgb_model='$UNCHANGED$'*)

Configure whether metadata should be requested to be passed to the `fit` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a meta-estimator and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

- **base_margin** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `base_margin` parameter in `fit`.

- **base_margin_eval_set** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for base_margin_eval_set parameter in fit.
- **eval_set** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for eval_set parameter in fit.
- **feature_weights** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for feature_weights parameter in fit.
- **sample_weight** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for sample_weight parameter in fit.
- **sample_weight_eval_set** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for sample_weight_eval_set parameter in fit.
- **verbose** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for verbose parameter in fit.
- **xgb_model** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for xgb_model parameter in fit.
- **self** (XGBRegressor)

Returns
self – The updated object.

Return type
object

set_params(***params*)

Set the parameters of this estimator. Modification of the sklearn method to allow unknown kwargs. This allows using the full range of xgboost parameters that are not defined as member variables in sklearn grid search.

Return type
self

Parameters
params (*Any*)

set_predict_request(*, *base_margin='$UNCHANGED$'*, *iteration_range='$UNCHANGED$'*, *output_margin='$UNCHANGED$'*, *validate_features='$UNCHANGED$'*)

Configure whether metadata should be requested to be passed to the predict method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a meta-estimator and metadata routing is enabled with enable_metadata_routing=True (see sklearn.set_config()). Please check the User Guide on how the routing mechanism works.

The options for each parameter are:

- True: metadata is requested, and passed to predict if provided. The request is ignored if metadata is not provided.
- False: metadata is not requested and the meta-estimator will not pass it to predict.

- None: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- str: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (sklearn.utils.metadata_routing.UNCHANGED) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

- **base_margin** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for base_margin parameter in predict.
- **iteration_range** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for iteration_range parameter in predict.
- **output_margin** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for output_margin parameter in predict.
- **validate_features** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for validate_features parameter in predict.
- **self** (XGBRegressor)

Returns
self – The updated object.

Return type
object

set_score_request(**, sample_weight='$UNCHANGED$'*)

Configure whether metadata should be requested to be passed to the score method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a meta-estimator and metadata routing is enabled with enable_metadata_routing=True (see sklearn.set_config()). Please check the User Guide on how the routing mechanism works.

The options for each parameter are:

- True: metadata is requested, and passed to score if provided. The request is ignored if metadata is not provided.
- False: metadata is not requested and the meta-estimator will not pass it to score.
- None: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- str: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (sklearn.utils.metadata_routing.UNCHANGED) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

- **sample_weight** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for sample_weight parameter in score.

- **self** (XGBRegressor)

Returns
self – The updated object.

Return type
object

class xgboost.**XGBClassifier**(*, *objective='binary:logistic'*, ***kwargs*)

Bases: ClassifierMixin, XGBModel

Implementation of the scikit-learn API for XGBoost classification. See *Using the Scikit-Learn Estimator Interface* for more information.

Parameters

- **n_estimators** (*Optional[int]*) – Number of boosting rounds.
- **max_depth** (*Optional[int]*) – Maximum tree depth for base learners.
- **max_leaves** (*Optional[int]*) – Maximum number of leaves; 0 indicates no limit.
- **max_bin** (*Optional[int]*) – If using histogram-based algorithm, maximum number of bins per feature
- **grow_policy** (*Optional[str]*) – Tree growing policy.
  - depthwise: Favors splitting at nodes closest to the node,
  - lossguide: Favors splitting at nodes with highest loss change.
- **learning_rate** (*Optional[float]*) – Boosting learning rate (xgb's "eta")
- **verbosity** (*Optional[int]*) – The degree of verbosity. Valid values are 0 (silent) - 3 (debug).
- **objective** (*Union[str, xgboost.sklearn._SklObjWProto, Callable[[Any, Any], Tuple[numpy.ndarray, numpy.ndarray]], NoneType]*) – Specify the learning task and the corresponding learning objective or a custom objective function to be used.

  For custom objective, see *Custom Objective and Evaluation Metric* and *Custom objective and metric* for more information, along with the end note for function signatures.
- **booster** (*Optional[str]*) – Specify which booster to use: gbtree, gblinear or dart.

  Deprecated since version 3.3.0: gblinear is deprecated and support will be removed in a future release.
- **tree_method** (*Optional[str]*) – Specify which tree method to use. Default to auto. If this parameter is set to default, XGBoost will choose the most conservative option available. It's recommended to study this option from the parameters document *tree method*
- **n_jobs** (*Optional[int]*) – Number of parallel threads used to run xgboost. When used with other Scikit-Learn algorithms like grid search, you may choose which algorithm to parallelize and balance the threads. Creating thread contention will significantly slow down both algorithms.
- **gamma** (*Optional[float]*) – (min_split_loss) Minimum loss reduction required to make a further partition on a leaf node of the tree.
- **min_child_weight** (*Optional[float]*) – Minimum sum of instance weight(hessian) needed in a child.

- **max_delta_step** (*Optional[float]*) – Maximum delta step we allow each tree's weight estimation to be.
- **subsample** (*Optional[float]*) – Subsample ratio of the training instance.
- **sampling_method** (*Optional[str]*) – Sampling method. Used only by the GPU version of `hist` tree method.
  - `uniform`: Select random training instances uniformly.
  - **gradient_based: Select random training instances with higher probability**
    when the gradient and hessian are larger. (cf. CatBoost)
- **colsample_bytree** (*Optional[float]*) – Subsample ratio of columns when constructing each tree.
- **colsample_bylevel** (*Optional[float]*) – Subsample ratio of columns for each level.
- **colsample_bynode** (*Optional[float]*) – Subsample ratio of columns for each split.
- **reg_alpha** (*Optional[float]*) – L1 regularization term on weights (xgb's alpha).
- **reg_lambda** (*Optional[float]*) – L2 regularization term on weights (xgb's lambda).
- **scale_pos_weight** (*Optional[float]*) – Balancing of positive and negative weights.
- **base_score** (*Union[float, List[float], NoneType]*) – The initial prediction score of all instances, global bias.
- **random_state** (*Union[numpy.random.mtrand.RandomState, numpy.random._generator.Generator, int, NoneType]*) – Random number seed.

  > **Note**
  >
  > Using gblinear booster with shotgun updater is nondeterministic as it uses Hogwild algorithm.

- **missing** (*float*) – Value in the data which needs to be present as a missing value. Default to `numpy.nan`.
- **num_parallel_tree** (*Optional[int]*) – Used for boosting random forest.
- **monotone_constraints** (*Union[Dict[str, int], str, NoneType]*) – Constraint of variable monotonicity. See *tutorial* for more information.
- **interaction_constraints** (*Union[str, List[Tuple[str]], NoneType]*) – Constraints for interaction representing permitted interactions. The constraints must be specified in the form of a nested list, e.g. `[[0, 1], [2, 3, 4]]`, where each inner list is a group of indices of features that are allowed to interact with each other. See *tutorial* for more information
- **importance_type** (*Optional[str]*) – The feature importance type for the feature_importances_ property:
  - For tree model, it's either "gain", "weight", "cover", "total_gain" or "total_cover".
  - For linear model, only "weight" is defined and it's the normalized coefficients without bias.
- **device** (*Optional[str]*) – Added in version 2.0.0.

  Device ordinal, available options are *cpu*, *cuda*, and *gpu*.

- **validate_parameters** (*Optional[bool]*) – Give warnings for unknown parameter.
- **enable_categorical** (*bool*) – See the same parameter of *DMatrix* for details.
- **feature_types** (*Optional[Sequence[str]]*) – Added in version 1.7.0.

  Used for specifying feature types without constructing a dataframe. See the *DMatrix* for details.
- **feature_weights** (*Optional[ArrayLike]*) – Weight for each feature, defines the probability of each feature being selected when colsample is being used. All values must be greater than 0, otherwise a *ValueError* is thrown.
- **max_cat_to_onehot** (*Optional[int]*) – Added in version 1.6.0.

  Note

  This parameter is experimental

  A threshold for deciding whether XGBoost should use one-hot encoding based split for categorical data. When number of categories is lesser than the threshold then one-hot encoding is chosen, otherwise the categories will be partitioned into children nodes. Also, *enable_categorical* needs to be set to have categorical feature support. See *Categorical Data* and *Parameters for Categorical Feature* for details.
- **max_cat_threshold** (*Optional[int]*) – Added in version 1.7.0.

  Note

  This parameter is experimental

  Maximum number of categories considered for each split. Used only by partition-based splits for preventing over-fitting. Also, *enable_categorical* needs to be set to have categorical feature support. See *Categorical Data* and *Parameters for Categorical Feature* for details.
- **multi_strategy** (*Optional[str]*) – Added in version 2.0.0.

  Note

  This parameter is working-in-progress.

  The strategy used for training multi-target models, including multi-target regression and multi-class classification. See *Multiple Outputs* for more information.

  – `one_output_per_tree`: One model for each target.

  – `multi_output_tree`: Use multi-target trees.
- **eval_metric** (*Union[str, List[Union[str, Callable]], Callable, NoneType]*) – Added in version 1.6.0.

  Metric used for monitoring the training result and early stopping. It can be a string or list of strings as names of predefined metric in XGBoost (See *XGBoost Parameters*), one of the metrics in `sklearn.metrics`, or any other user defined metric that looks like *sklearn.metrics*.

  If custom objective is also provided, then custom metric should implement the corresponding reverse link function.

Unlike the *scoring* parameter commonly used in scikit-learn, when a callable object is provided, it's assumed to be a cost function and by default XGBoost will minimize the result during early stopping.

For advanced usage on Early stopping like directly choosing to maximize instead of minimize, see `xgboost.callback.EarlyStopping`.

See *Custom Objective and Evaluation Metric* and *Custom objective and metric* for more information.

```
from sklearn.datasets import load_diabetes
from sklearn.metrics import mean_absolute_error
X, y = load_diabetes(return_X_y=True)
reg = xgb.XGBRegressor(
    tree_method="hist",
    eval_metric=mean_absolute_error,
)
reg.fit(X, y, eval_set=[(X, y)])
```

- **early_stopping_rounds** (*Optional[int]*) – Added in version 1.6.0.
  - Activates early stopping. Validation metric needs to improve at least once in every **early_stopping_rounds** round(s) to continue training. Requires at least one item in **eval_set** in `fit()`.
  - If early stopping occurs, the model will have two additional attributes: `best_score` and `best_iteration`. These are used by the `predict()` and `apply()` methods to determine the optimal number of trees during inference. If users want to access the full model (including trees built after early stopping), they can specify the *iteration_range* in these inference methods. In addition, other utilities like model plotting can also use the entire model.
  - If you prefer to discard the trees after *best_iteration*, consider using the callback function `xgboost.callback.EarlyStopping`.
  - If there's more than one item in **eval_set**, the last entry will be used for early stopping. If there's more than one metric in **eval_metric**, the last metric will be used for early stopping.
- **callbacks** (*Optional[List[xgboost.callback.TrainingCallback]]*) – List of callback functions that are applied at end of each iteration. It is possible to use predefined callbacks by using *Callback API*.

  Note

  States in callback are not preserved during training, which means callback objects can not be reused for multiple training sessions without reinitialization or deepcopy.

```
for params in parameters_grid:
    # be sure to (re)initialize the callbacks before each run
    callbacks = [xgb.callback.LearningRateScheduler(custom_rates)]
    reg = xgboost.XGBRegressor(**params, callbacks=callbacks)
    reg.fit(X, y)
```

- **kwargs** (*Optional[Any]*) – Keyword arguments for XGBoost Booster object. Full documentation of parameters can be found *here*. Attempting to set a parameter via the constructor args and **kwargs dict simultaneously will result in a TypeError.

> **Note**
>
> **kwargs unsupported by scikit-learn
>
> **kwargs is unsupported by scikit-learn. We do not guarantee that parameters passed via this argument will interact properly with scikit-learn.

> **Note**
>
> Custom objective function
>
> A custom objective function can be provided for the `objective` parameter. In this case, it should have the signature `objective(y_true, y_pred) -> [grad, hess]` or `objective(y_true, y_pred, *, sample_weight) -> [grad, hess]`:
>
> **y_true: array_like of shape [n_samples]**
> The target values
>
> **y_pred: array_like of shape [n_samples]**
> The predicted values
>
> **sample_weight :**
> Optional sample weights.
>
> **grad: array_like of shape [n_samples]**
> The value of the gradient for each sample point.
>
> **hess: array_like of shape [n_samples]**
> The value of the second derivative for each sample point
>
> Note that, if the custom objective produces negative values for the Hessian, these will be clipped. If the objective is non-convex, one might also consider using the expected Hessian (Fisher information) instead.

`apply`(*X*, *iteration_range=None*)

Return the predicted leaf every tree for each sample. If the model is trained with early stopping, then *best_iteration* is used automatically.

Parameters

- **X** (*Any*) – Input features matrix. See *Markers* for a list of supported types.
- **iteration_range** (*Tuple[int | integer, int | integer] | None*) – See *predict()*.

Returns

X_leaves – For each datapoint x in X and for each tree, return the index of the leaf x ends up in. Leaves are numbered within `[0; 2**(self.max_depth+1))`, possibly with gaps in the numbering.

Return type

array_like, shape=[n_samples, n_trees]

`property best_iteration: int`

The best iteration obtained by early stopping. This attribute is 0-based, for instance if the best iteration is the first round, then best_iteration is 0.

property best_score: float

The best score obtained by early stopping.

property coef_: ndarray

Coefficients property

> **Note**
>
> Coefficients are defined only for linear learners
>
> Coefficients are only defined when the linear model is chosen as base learner (*booster=gblinear*). It is not defined for other base learner types, such as tree learners (*booster=gbtree*).

Returns
coef_

Return type
array of shape `[n_features]` or `[n_classes, n_features]`

evals_result()

Return the evaluation results.

If **eval_set** is passed to the `fit()` function, you can call `evals_result()` to get evaluation results for all passed **eval_sets**. When **eval_metric** is also passed to the `fit()` function, the **evals_result** will contain the **eval_metrics** passed to the `fit()` function.

The returned evaluation result is a dictionary:

```
{'validation_0': {'logloss': ['0.604835', '0.531479']},
 'validation_1': {'logloss': ['0.41965', '0.17686']}}
```

Return type
evals_result

property feature_importances_: ndarray

Feature importances property, return depends on *importance_type* parameter. When model trained with multi-class/multi-label/multi-target dataset, the feature importance is "averaged" over all targets. The "average" is defined based on the importance type. For instance, if the importance type is "total_gain", then the score is sum of loss change for each split from all trees.

Returns

- **feature_importances_** (array of shape `[n_features]` except for multi-class)
- linear model, which returns an array with shape *(n_features, n_classes)*

property feature_names_in_: ndarray

Names of features seen during `fit()`. Defined only when *X* has feature names that are all strings.

fit(*X*, *y*, ***, *sample_weight=None*, *base_margin=None*, *eval_set=None*, *verbose=True*, *xgb_model=None*, *sample_weight_eval_set=None*, *base_margin_eval_set=None*, *feature_weights=None*)

Fit gradient boosting classifier.

Note that calling `fit()` multiple times will cause the model object to be re-fit from scratch. To resume training from a previous checkpoint, explicitly pass `xgb_model` argument.

Parameters

- **X** (*Any*) – Input feature matrix. See *Markers* for a list of supported types.

  When the `tree_method` is set to `hist`, internally, the *QuantileDMatrix* will be used instead of the *DMatrix* for conserving memory. However, this has performance implications when the device of input data is not matched with algorithm. For instance, if the input is a numpy array on CPU but `cuda` is used for training, then the data is first processed on CPU then transferred to GPU.
- **y** (*Any*) – Labels
- **sample_weight** (*Any | None*) – instance weights
- **base_margin** (*Any | None*) – Global bias for each instance. See *Intercept* for details.
- **eval_set** (*Sequence[Tuple[Any, Any]] | None*) – A list of (X, y) tuple pairs to use as validation sets, for which metrics will be computed. Validation metrics will help us track the performance of the model.
- **verbose** (*bool | int | None*) – If *verbose* is True and an evaluation set is used, the evaluation metric measured on the validation set is printed to stdout at each boosting stage. If *verbose* is an integer, the evaluation metric is printed at each *verbose* boosting stage. The last boosting stage / the boosting stage found by using *early_stopping_rounds* is also printed.
- **xgb_model** (Booster *| str | XGBModel | None*) – file name of stored XGBoost model or 'Booster' instance XGBoost model to be loaded before training (allows training continuation).
- **sample_weight_eval_set** (*Sequence[Any] | None*) – A list of the form [L_1, L_2, ..., L_n], where each L_i is an array like object storing instance weights for the i-th validation set.
- **base_margin_eval_set** (*Sequence[Any] | None*) – A list of the form [M_1, M_2, ..., M_n], where each M_i is an array like object storing base margin for the i-th validation set.
- **feature_weights** (*Any | None*) – Deprecated since version 3.0.0.

  Use *feature_weights* in `__init__()` or *set_params()* instead.

Return type
XGBClassifier

`get_booster()`

Get the underlying xgboost Booster of this model.

This will raise an exception when fit was not called

Returns
booster

Return type
a xgboost booster of underlying model

`get_metadata_routing()`

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns
routing – A `MetadataRequest` encapsulating routing information.

Return type
MetadataRequest

`get_num_boosting_rounds()`

Gets the number of xgboost boosting rounds.

Return type
int

`get_params(deep=True)`

Get parameters.

Parameters
deep (*bool*)

Return type
Dict[str, *Any*]

`get_xgb_params()`

Get xgboost specific parameters.

Return type
Dict[str, *Any*]

`property intercept_: ndarray`

Intercept (bias) property

For tree-based model, the returned value is the *base_score*.

Returns
intercept_

Return type
array of shape `(1,)` or `[n_classes]`

`load_model(fname)`

Load the model from a file or a bytearray.

The model is saved in an XGBoost internal format which is universal among the various XGBoost interfaces. Auxiliary attributes of the Python Booster object (such as feature_names) are only saved when using JSON or UBJSON (default) format. Also, parameters that are not part of the model (like metrics, *max_depth*, etc) are not saved, see *Model IO* for more info.

```
model.save_model("model.json")
model.load_model("model.json")

# or
model.save_model("model.ubj")
model.load_model("model.ubj")

# or
buf = model.save_raw()
model.load_model(buf)
```

Parameters
fname (*PathLike | bytearray | str*) – Input file name or memory buffer(see also save_raw)

Return type
None

property n_features_in_: int

Number of features seen during `fit()`.

predict(*X*, ***, *output_margin=False*, *validate_features=True*, *base_margin=None*, *iteration_range=None*)

Predict with *X*. If the model is trained with early stopping, then `best_iteration` is used automatically. The estimator uses *inplace_predict* by default and falls back to using `DMatrix` if devices between the data and the estimator don't match.

Note

This function is only thread safe for *gbtree* and *dart*.

Parameters

- **X** (*Any*) – Data to predict with. See *Markers* for a list of supported types.
- **output_margin** (*bool*) – Whether to output the raw untransformed margin value.
- **validate_features** (*bool*) – When this is True, validate that the Booster's and data's feature_names are identical. Otherwise, it is assumed that the feature_names are the same.
- **base_margin** (*Any | None*) – Global bias for each instance. See *Intercept* for details.
- **iteration_range** (*Tuple[int | integer, int | integer] | None*) – Specifies which layer of trees are used in prediction. For example, if a random forest is trained with 100 rounds. Specifying `iteration_range=(10, 20)`, then only the forests built during [10, 20) (half open set) rounds are used in this prediction.

  Added in version 1.4.0.

Return type

prediction

predict_proba(*X*, *validate_features=True*, *base_margin=None*, *iteration_range=None*)

Predict the probability of each *X* example being of a given class. If the model is trained with early stopping, then `best_iteration` is used automatically. The estimator uses *inplace_predict* by default and falls back to using `DMatrix` if devices between the data and the estimator don't match.

Note

This function is only thread safe for *gbtree* and *dart*.

Parameters

- **X** (*Any*) – Feature matrix. See *Markers* for a list of supported types.
- **validate_features** (*bool*) – When this is True, validate that the Booster's and data's feature_names are identical. Otherwise, it is assumed that the feature_names are the same.
- **base_margin** (*Any | None*) – Global bias for each instance. See *Intercept* for details.
- **iteration_range** (*Tuple[int | integer, int | integer] | None*) – Specifies which layer of trees are used in prediction. For example, if a random forest is trained with 100 rounds. Specifying *iteration_range=(10, 20)*, then only the forests built during [10, 20) (half open set) rounds are used in this prediction.

Returns
a numpy array of shape array-like of shape (n_samples, n_classes) with the probability of each data example being of a given class.

Return type
prediction

save_model(*fname*)

Save the model to a file.

The model is saved in an XGBoost internal format which is universal among the various XGBoost interfaces. Auxiliary attributes of the Python Booster object (such as feature_names) are only saved when using JSON or UBJSON (default) format. Also, parameters that are not part of the model (like metrics, *max_depth*, etc) are not saved, see *Model IO* for more info.

```
model.save_model("model.json")
# or
model.save_model("model.ubj")
```

Parameters
fname (*str | PathLike*) – Output file name

Return type
None

score(*X*, *y*, *sample_weight=None*)

Return accuracy on provided data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

- **X** (*array-like of shape (n_samples, n_features)*) – Test samples.
- **y** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – True labels for *X*.
- **sample_weight** (*array-like of shape (n_samples,), default=None*) – Sample weights.

Returns
score – Mean accuracy of `self.predict(X)` w.r.t. *y*.

Return type
float

set_fit_request(**, base_margin='$UNCHANGED$', base_margin_eval_set='$UNCHANGED$', eval_set='$UNCHANGED$', feature_weights='$UNCHANGED$', sample_weight='$UNCHANGED$', sample_weight_eval_set='$UNCHANGED$', verbose='$UNCHANGED$', xgb_model='$UNCHANGED$'*)

Configure whether metadata should be requested to be passed to the `fit` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a meta-estimator and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

- **base_margin** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `base_margin` parameter in `fit`.
- **base_margin_eval_set** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `base_margin_eval_set` parameter in `fit`.
- **eval_set** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `eval_set` parameter in `fit`.
- **feature_weights** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `feature_weights` parameter in `fit`.
- **sample_weight** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `sample_weight` parameter in `fit`.
- **sample_weight_eval_set** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `sample_weight_eval_set` parameter in `fit`.
- **verbose** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `verbose` parameter in `fit`.
- **xgb_model** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `xgb_model` parameter in `fit`.
- **self** (`XGBClassifier`)

Returns

self – The updated object.

Return type

object

`set_params`(***params*)

Set the parameters of this estimator. Modification of the sklearn method to allow unknown kwargs. This allows using the full range of xgboost parameters that are not defined as member variables in sklearn grid search.

Return type
self

Parameters
params (*Any*)

set_predict_proba_request(*, *base_margin='$UNCHANGED$'*, *iteration_range='$UNCHANGED$'*, *validate_features='$UNCHANGED$'*)

Configure whether metadata should be requested to be passed to the `predict_proba` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a meta-estimator and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `predict_proba` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `predict_proba`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

- **base_margin** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `base_margin` parameter in `predict_proba`.
- **iteration_range** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `iteration_range` parameter in `predict_proba`.
- **validate_features** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `validate_features` parameter in `predict_proba`.
- **self** (`XGBClassifier`)

Returns
self – The updated object.

Return type
object

set_predict_request(*, *base_margin='$UNCHANGED$'*, *iteration_range='$UNCHANGED$'*, *output_margin='$UNCHANGED$'*, *validate_features='$UNCHANGED$'*)

Configure whether metadata should be requested to be passed to the `predict` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a meta-estimator and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `predict` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `predict`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

- **base_margin** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `base_margin` parameter in `predict`.
- **iteration_range** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `iteration_range` parameter in `predict`.
- **output_margin** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `output_margin` parameter in `predict`.
- **validate_features** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `validate_features` parameter in `predict`.
- **self** (`XGBClassifier`)

Returns
self – The updated object.

Return type
object

set_score_request(**, sample_weight='$UNCHANGED$'*)

Configure whether metadata should be requested to be passed to the `score` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a meta-estimator and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

- **sample_weight** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `sample_weight` parameter in `score`.
- **self** (`XGBClassifier`)

Returns
self – The updated object.

Return type
object

`class xgboost.XGBRanker(*, objective='rank:ndcg', **kwargs)`

Bases: `XGBRankerMixIn`, `XGBModel`

Implementation of the Scikit-Learn API for XGBoost Ranking.

See *Learning to Rank* for an introducion.

See *Using the Scikit-Learn Estimator Interface* for more information.

Parameters

- **n_estimators** (*Optional[int]*) – Number of gradient boosted trees. Equivalent to number of boosting rounds.
- **max_depth** (*Optional[int]*) – Maximum tree depth for base learners.
- **max_leaves** (*Optional[int]*) – Maximum number of leaves; 0 indicates no limit.
- **max_bin** (*Optional[int]*) – If using histogram-based algorithm, maximum number of bins per feature
- **grow_policy** (*Optional[str]*) – Tree growing policy.
  - depthwise: Favors splitting at nodes closest to the node,
  - lossguide: Favors splitting at nodes with highest loss change.
- **learning_rate** (*Optional[float]*) – Boosting learning rate (xgb's "eta")
- **verbosity** (*Optional[int]*) – The degree of verbosity. Valid values are 0 (silent) - 3 (debug).
- **objective** (*Union[str, xgboost.sklearn._SklObjWProto, Callable[[Any, Any], Tuple[numpy.ndarray, numpy.ndarray]], NoneType]*) – Specify the learning task and the corresponding learning objective or a custom objective function to be used.

  For custom objective, see *Custom Objective and Evaluation Metric* and *Custom objective and metric* for more information, along with the end note for function signatures.
- **booster** (*Optional[str]*) – Specify which booster to use: `gbtree`, `gblinear` or `dart`.

  Deprecated since version 3.3.0: `gblinear` is deprecated and support will be removed in a future release.
- **tree_method** (*Optional[str]*) – Specify which tree method to use. Default to auto. If this parameter is set to default, XGBoost will choose the most conservative option available. It's recommended to study this option from the parameters document *tree method*
- **n_jobs** (*Optional[int]*) – Number of parallel threads used to run xgboost. When used with other Scikit-Learn algorithms like grid search, you may choose which algorithm to

parallelize and balance the threads. Creating thread contention will significantly slow down both algorithms.

- **gamma** (*Optional[float]*) – (min_split_loss) Minimum loss reduction required to make a further partition on a leaf node of the tree.
- **min_child_weight** (*Optional[float]*) – Minimum sum of instance weight(hessian) needed in a child.
- **max_delta_step** (*Optional[float]*) – Maximum delta step we allow each tree's weight estimation to be.
- **subsample** (*Optional[float]*) – Subsample ratio of the training instance.
- **sampling_method** (*Optional[str]*) – Sampling method. Used only by the GPU version of `hist` tree method.
  - `uniform`: Select random training instances uniformly.
  - **gradient_based: Select random training instances with higher probability**
    when the gradient and hessian are larger. (cf. CatBoost)
- **colsample_bytree** (*Optional[float]*) – Subsample ratio of columns when constructing each tree.
- **colsample_bylevel** (*Optional[float]*) – Subsample ratio of columns for each level.
- **colsample_bynode** (*Optional[float]*) – Subsample ratio of columns for each split.
- **reg_alpha** (*Optional[float]*) – L1 regularization term on weights (xgb's alpha).
- **reg_lambda** (*Optional[float]*) – L2 regularization term on weights (xgb's lambda).
- **scale_pos_weight** (*Optional[float]*) – Balancing of positive and negative weights.
- **base_score** (*Union[float, List[float], NoneType]*) – The initial prediction score of all instances, global bias.
- **random_state** (*Union[numpy.random.mtrand.RandomState, numpy.random._generator.Generator, int, NoneType]*) – Random number seed.

> **Note**
>
> Using gblinear booster with shotgun updater is nondeterministic as it uses Hogwild algorithm.

- **missing** (*float*) – Value in the data which needs to be present as a missing value. Default to `numpy.nan`.
- **num_parallel_tree** (*Optional[int]*) – Used for boosting random forest.
- **monotone_constraints** (*Union[Dict[str, int], str, NoneType]*) – Constraint of variable monotonicity. See *tutorial* for more information.
- **interaction_constraints** (*Union[str, List[Tuple[str]], NoneType]*) – Constraints for interaction representing permitted interactions. The constraints must be specified in the form of a nested list, e.g. `[[0, 1], [2, 3, 4]]`, where each inner list is a group of indices of features that are allowed to interact with each other. See *tutorial* for more information
- **importance_type** (*Optional[str]*) – The feature importance type for the feature_importances_ property:

  - For tree model, it's either "gain", "weight", "cover", "total_gain" or "total_cover".
  - For linear model, only "weight" is defined and it's the normalized coefficients without bias.

- **device** (*Optional[str]*) – Added in version 2.0.0.

  Device ordinal, available options are *cpu*, *cuda*, and *gpu*.

- **validate_parameters** (*Optional[bool]*) – Give warnings for unknown parameter.
- **enable_categorical** (*bool*) – See the same parameter of *DMatrix* for details.
- **feature_types** (*Optional[Sequence[str]]*) – Added in version 1.7.0.

  Used for specifying feature types without constructing a dataframe. See the *DMatrix* for details.

- **feature_weights** (*Optional[ArrayLike]*) – Weight for each feature, defines the probability of each feature being selected when colsample is being used. All values must be greater than 0, otherwise a *ValueError* is thrown.
- **max_cat_to_onehot** (*Optional[int]*) – Added in version 1.6.0.

  Note

  This parameter is experimental

  A threshold for deciding whether XGBoost should use one-hot encoding based split for categorical data. When number of categories is lesser than the threshold then one-hot encoding is chosen, otherwise the categories will be partitioned into children nodes. Also, *enable_categorical* needs to be set to have categorical feature support. See *Categorical Data* and *Parameters for Categorical Feature* for details.

- **max_cat_threshold** (*Optional[int]*) – Added in version 1.7.0.

  Note

  This parameter is experimental

  Maximum number of categories considered for each split. Used only by partition-based splits for preventing over-fitting. Also, *enable_categorical* needs to be set to have categorical feature support. See *Categorical Data* and *Parameters for Categorical Feature* for details.

- **multi_strategy** (*Optional[str]*) – Added in version 2.0.0.

  Note

  This parameter is working-in-progress.

  The strategy used for training multi-target models, including multi-target regression and multi-class classification. See *Multiple Outputs* for more information.

  - `one_output_per_tree`: One model for each target.
  - `multi_output_tree`: Use multi-target trees.

- **eval_metric** (*Union[str, List[Union[str, Callable]], Callable, NoneType]*) – Added in version 1.6.0.

  Metric used for monitoring the training result and early stopping. It can be a string or list of strings as names of predefined metric in XGBoost (See *XGBoost Parameters*), one of the metrics in `sklearn.metrics`, or any other user defined metric that looks like *sklearn.metrics*.

  If custom objective is also provided, then custom metric should implement the corresponding reverse link function.

  Unlike the *scoring* parameter commonly used in scikit-learn, when a callable object is provided, it's assumed to be a cost function and by default XGBoost will minimize the result during early stopping.

  For advanced usage on Early stopping like directly choosing to maximize instead of minimize, see *xgboost.callback.EarlyStopping*.

  See *Custom Objective and Evaluation Metric* and *Custom objective and metric* for more information.

  ```
  from sklearn.datasets import load_diabetes
  from sklearn.metrics import mean_absolute_error
  X, y = load_diabetes(return_X_y=True)
  reg = xgb.XGBRegressor(
      tree_method="hist",
      eval_metric=mean_absolute_error,
  )
  reg.fit(X, y, eval_set=[(X, y)])
  ```

- **early_stopping_rounds** (*Optional[int]*) – Added in version 1.6.0.
  - Activates early stopping. Validation metric needs to improve at least once in every **early_stopping_rounds** round(s) to continue training. Requires at least one item in **eval_set** in *fit()*.
  - If early stopping occurs, the model will have two additional attributes: *best_score* and *best_iteration*. These are used by the *predict()* and *apply()* methods to determine the optimal number of trees during inference. If users want to access the full model (including trees built after early stopping), they can specify the *iteration_range* in these inference methods. In addition, other utilities like model plotting can also use the entire model.
  - If you prefer to discard the trees after *best_iteration*, consider using the callback function *xgboost.callback.EarlyStopping*.
  - If there's more than one item in **eval_set**, the last entry will be used for early stopping. If there's more than one metric in **eval_metric**, the last metric will be used for early stopping.
- **callbacks** (*Optional[List[*`xgboost.callback.TrainingCallback`*]]*) – List of callback functions that are applied at end of each iteration. It is possible to use predefined callbacks by using *Callback API*.

  > **Note**
  >
  > States in callback are not preserved during training, which means callback objects can not be reused for multiple training sessions without reinitialization or deepcopy.

```
for params in parameters_grid:
    # be sure to (re)initialize the callbacks before each run
    callbacks = [xgb.callback.LearningRateScheduler(custom_rates)]
    reg = xgboost.XGBRegressor(**params, callbacks=callbacks)
    reg.fit(X, y)
```

- **kwargs** (*Optional[Any]*) – Keyword arguments for XGBoost Booster object. Full documentation of parameters can be found *here*. Attempting to set a parameter via the constructor args and **kwargs dict simultaneously will result in a TypeError.

> **Note**
>
> **kwargs unsupported by scikit-learn
>
> **kwargs is unsupported by scikit-learn. We do not guarantee that parameters passed via this argument will interact properly with scikit-learn.

> **Note**
>
> A custom objective function is currently not supported by XGBRanker.

> **Note**
>
> Query group information is only required for ranking training but not prediction. Multiple groups can be predicted on a single call to `predict()`.

When fitting the model with the *group* parameter, your data need to be sorted by the query group first. *group* is an array that contains the size of each query group.

Similarly, when fitting the model with the *qid* parameter, the data should be sorted according to query index and *qid* is an array that contains the query index for each training sample.

For example, if your original data look like:

qid	label	features
1	0	x_1
1	1	x_2
1	0	x_3
2	0	x_4
2	1	x_5
2	1	x_6
2	1	x_7

then `fit()` method can be called with either *group* array as `[3, 4]` or with *qid* as `[1, 1, 1, 2, 2, 2, 2]`, that is the qid column. Also, the *qid* can be a special column of input *X* instead of a separated parameter, see `fit()` for more info.

apply(*X*, *iteration_range=None*)

Return the predicted leaf every tree for each sample. If the model is trained with early stopping, then `best_iteration` is used automatically.

Parameters

- **X** (*Any*) – Input features matrix. See *Markers* for a list of supported types.
- **iteration_range** (*Tuple[int | integer, int | integer] | None*) – See `predict()`.

Returns
X_leaves – For each datapoint x in X and for each tree, return the index of the leaf x ends up in. Leaves are numbered within `[0; 2**(self.max_depth+1))`, possibly with gaps in the numbering.

Return type
array_like, shape=[n_samples, n_trees]

property best_iteration: int

The best iteration obtained by early stopping. This attribute is 0-based, for instance if the best iteration is the first round, then best_iteration is 0.

property best_score: float

The best score obtained by early stopping.

property coef_: ndarray

Coefficients property

> **Note**
>
> Coefficients are defined only for linear learners
>
> Coefficients are only defined when the linear model is chosen as base learner (*booster=gblinear*). It is not defined for other base learner types, such as tree learners (*booster=gbtree*).

Returns
coef_

Return type
array of shape `[n_features]` or `[n_classes, n_features]`

evals_result()

Return the evaluation results.

If **eval_set** is passed to the `fit()` function, you can call `evals_result()` to get evaluation results for all passed **eval_sets**. When **eval_metric** is also passed to the `fit()` function, the **evals_result** will contain the **eval_metrics** passed to the `fit()` function.

The returned evaluation result is a dictionary:

```
{'validation_0': {'logloss': ['0.604835', '0.531479']},
 'validation_1': {'logloss': ['0.41965', '0.17686']}}
```

Return type
evals_result

property feature_importances_: ndarray

Feature importances property, return depends on *importance_type* parameter. When model trained with

multi-class/multi-label/multi-target dataset, the feature importance is "averaged" over all targets. The "average" is defined based on the importance type. For instance, if the importance type is "total_gain", then the score is sum of loss change for each split from all trees.

Returns

- **feature_importances_** (array of shape `[n_features]` except for multi-class)
- linear model, which returns an array with shape *(n_features, n_classes)*

property feature_names_in_: ndarray

Names of features seen during *fit()*. Defined only when *X* has feature names that are all strings.

fit(*X*, *y*, ***, *group=None*, *qid=None*, *sample_weight=None*, *base_margin=None*, *eval_set=None*, *eval_group=None*, *eval_qid=None*, *verbose=True*, *xgb_model=None*, *sample_weight_eval_set=None*, *base_margin_eval_set=None*, *feature_weights=None*)

Fit gradient boosting ranker

Note that calling `fit()` multiple times will cause the model object to be re-fit from scratch. To resume training from a previous checkpoint, explicitly pass `xgb_model` argument.

Parameters

- **X** (*Any*) – Feature matrix. See *Markers* for a list of supported types.

  When this is a `pandas.DataFrame` or a `cudf.DataFrame`, it may contain a special column called `qid` for specifying the query index. Using a special column is the same as using the *qid* parameter, except for being compatible with sklearn utility functions like `sklearn.model_selection.cross_validation()`. The same convention applies to the *XGBRanker.score()* and *XGBRanker.predict()*.

qid	feat_0	feat_1
0	x_{00}	x_{01}
1	x_{10}	x_{11}
1	x_{20}	x_{21}

  When the `tree_method` is set to `hist`, internally, the *QuantileDMatrix* will be used instead of the *DMatrix* for conserving memory. However, this has performance implications when the device of input data is not matched with algorithm. For instance, if the input is a numpy array on CPU but `cuda` is used for training, then the data is first processed on CPU then transferred to GPU.

- **y** (*Any*) – Labels
- **group** (*Any | None*) – Size of each query group of training data. Should have as many elements as the query groups in the training data. If this is set to None, then user must provide qid.
- **qid** (*Any | None*) – Query ID for each training sample. Should have the size of n_samples. If this is set to None, then user must provide group or a special column in X.
- **sample_weight** (*Any | None*) – Query group weights

  Note

  Weights are per-group for ranking tasks

> In ranking task, one weight is assigned to each query group/id (not each data point). This is because we only care about the relative ordering of data points within each group, so it doesn't make sense to assign weights to individual data points.

- **base_margin** (*Any | None*) – Global bias for each instance. See *Intercept* for details.
- **eval_set** (*Sequence[Tuple[Any, Any]] | None*) – A list of (X, y) tuple pairs to use as validation sets, for which metrics will be computed. Validation metrics will help us track the performance of the model.
- **eval_group** (*Sequence[Any] | None*) – A list in which `eval_group[i]` is the list containing the sizes of all query groups in the `i`-th pair in **eval_set**.
- **eval_qid** (*Sequence[Any] | None*) – A list in which `eval_qid[i]` is the array containing query ID of `i`-th pair in **eval_set**. The special column convention in *X* applies to validation datasets as well.
- **verbose** (*bool | int | None*) – If *verbose* is True and an evaluation set is used, the evaluation metric measured on the validation set is printed to stdout at each boosting stage. If *verbose* is an integer, the evaluation metric is printed at each *verbose* boosting stage. The last boosting stage / the boosting stage found by using *early_stopping_rounds* is also printed.
- **xgb_model** (Booster *| str | XGBModel | None*) – file name of stored XGBoost model or 'Booster' instance XGBoost model to be loaded before training (allows training continuation).
- **sample_weight_eval_set** (*Sequence[Any] | None*) – A list of the form [L_1, L_2, . . . , L_n], where each L_i is a list of group weights on the i-th validation set.

> **Note**
>
> Weights are per-group for ranking tasks
>
> In ranking task, one weight is assigned to each query group (not each data point). This is because we only care about the relative ordering of data points within each group, so it doesn't make sense to assign weights to individual data points.

- **base_margin_eval_set** (*Sequence[Any] | None*) – A list of the form [M_1, M_2, . . . , M_n], where each M_i is an array like object storing base margin for the i-th validation set.
- **feature_weights** (*Any | None*) – Weight for each feature, defines the probability of each feature being selected when colsample is being used. All values must be greater than 0, otherwise a *ValueError* is thrown.

Return type
XGBRanker

`get_booster()`

Get the underlying xgboost Booster of this model.

This will raise an exception when fit was not called

Returns
booster

Return type
a xgboost booster of underlying model

get_metadata_routing()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns
routing – A `MetadataRequest` encapsulating routing information.

Return type
MetadataRequest

get_num_boosting_rounds()

Gets the number of xgboost boosting rounds.

Return type
int

get_params(*deep=True*)

Get parameters.

Parameters
deep (*bool*)

Return type
Dict[str, *Any*]

get_xgb_params()

Get xgboost specific parameters.

Return type
Dict[str, *Any*]

property intercept_: ndarray

Intercept (bias) property

For tree-based model, the returned value is the *base_score*.

Returns
intercept_

Return type
array of shape `(1,)` or `[n_classes]`

load_model(*fname*)

Load the model from a file or a bytearray.

The model is saved in an XGBoost internal format which is universal among the various XGBoost interfaces. Auxiliary attributes of the Python Booster object (such as feature_names) are only saved when using JSON or UBJSON (default) format. Also, parameters that are not part of the model (like metrics, *max_depth*, etc) are not saved, see *Model IO* for more info.

```
model.save_model("model.json")
model.load_model("model.json")

# or
model.save_model("model.ubj")
model.load_model("model.ubj")
```

(continues on next page)

(continued from previous page)

```
# or
buf = model.save_raw()
model.load_model(buf)
```

Parameters
fname (*PathLike | bytearray | str*) – Input file name or memory buffer(see also save_raw)

Return type
None

property n_features_in_: int

Number of features seen during *fit()*.

predict(*X*, ***, *output_margin=False*, *validate_features=True*, *base_margin=None*, *iteration_range=None*)

Predict with *X*. If the model is trained with early stopping, then *best_iteration* is used automatically. The estimator uses *inplace_predict* by default and falls back to using *DMatrix* if devices between the data and the estimator don't match.

Note

This function is only thread safe for *gbtree* and *dart*.

Parameters

- **X** (*Any*) – Data to predict with. See *Markers* for a list of supported types.
- **output_margin** (*bool*) – Whether to output the raw untransformed margin value.
- **validate_features** (*bool*) – When this is True, validate that the Booster's and data's feature_names are identical. Otherwise, it is assumed that the feature_names are the same.
- **base_margin** (*Any | None*) – Global bias for each instance. See *Intercept* for details.
- **iteration_range** (*Tuple[int | integer, int | integer] | None*) – Specifies which layer of trees are used in prediction. For example, if a random forest is trained with 100 rounds. Specifying `iteration_range=(10, 20)`, then only the forests built during [10, 20) (half open set) rounds are used in this prediction.

  Added in version 1.4.0.

Return type
prediction

save_model(*fname*)

Save the model to a file.

The model is saved in an XGBoost internal format which is universal among the various XGBoost interfaces. Auxiliary attributes of the Python Booster object (such as feature_names) are only saved when using JSON or UBJSON (default) format. Also, parameters that are not part of the model (like metrics, *max_depth*, etc) are not saved, see *Model IO* for more info.

```
model.save_model("model.json")
# or
model.save_model("model.ubj")
```

Parameters
fname (*str | PathLike*) – Output file name

Return type
None

score(*X*, *y*)

Evaluate score for data using the last evaluation metric. If the model is trained with early stopping, then `best_iteration` is used automatically.

Parameters

- **X** (*Union[pd.DataFrame, cudf.DataFrame]*) – Feature matrix. A DataFrame with a special *qid* column.
- **y** (*Any*) – Labels

Returns
The result of the first evaluation metric for the ranker.

Return type
score

set_fit_request(**, base_margin='$UNCHANGED$', base_margin_eval_set='$UNCHANGED$', eval_group='$UNCHANGED$', eval_qid='$UNCHANGED$', eval_set='$UNCHANGED$', feature_weights='$UNCHANGED$', group='$UNCHANGED$', qid='$UNCHANGED$', sample_weight='$UNCHANGED$', sample_weight_eval_set='$UNCHANGED$', verbose='$UNCHANGED$', xgb_model='$UNCHANGED$'*)

Configure whether metadata should be requested to be passed to the `fit` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a meta-estimator and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

- **base_margin** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `base_margin` parameter in `fit`.

- **base_margin_eval_set** (*str*, *True*, *False*, *or None*, *default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for base_margin_eval_set parameter in fit.
- **eval_group** (*str*, *True*, *False*, *or None*, *default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for eval_group parameter in fit.
- **eval_qid** (*str*, *True*, *False*, *or None*, *default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for eval_qid parameter in fit.
- **eval_set** (*str*, *True*, *False*, *or None*, *default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for eval_set parameter in fit.
- **feature_weights** (*str*, *True*, *False*, *or None*, *default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for feature_weights parameter in fit.
- **group** (*str*, *True*, *False*, *or None*, *default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for group parameter in fit.
- **qid** (*str*, *True*, *False*, *or None*, *default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for qid parameter in fit.
- **sample_weight** (*str*, *True*, *False*, *or None*, *default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for sample_weight parameter in fit.
- **sample_weight_eval_set** (*str*, *True*, *False*, *or None*, *default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for sample_weight_eval_set parameter in fit.
- **verbose** (*str*, *True*, *False*, *or None*, *default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for verbose parameter in fit.
- **xgb_model** (*str*, *True*, *False*, *or None*, *default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for xgb_model parameter in fit.
- **self** (XGBRanker)

Returns
self – The updated object.

Return type
object

set_params(***params*)

Set the parameters of this estimator. Modification of the sklearn method to allow unknown kwargs. This allows using the full range of xgboost parameters that are not defined as member variables in sklearn grid search.

Return type
self

Parameters
params (*Any*)

set_predict_request(**, base_margin='$UNCHANGED$', iteration_range='$UNCHANGED$', output_margin='$UNCHANGED$', validate_features='$UNCHANGED$'*)

Configure whether metadata should be requested to be passed to the `predict` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a meta-estimator and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `predict` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `predict`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

- **base_margin** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `base_margin` parameter in `predict`.
- **iteration_range** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `iteration_range` parameter in `predict`.
- **output_margin** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `output_margin` parameter in `predict`.
- **validate_features** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `validate_features` parameter in `predict`.
- **self** (`XGBRanker`)

Returns

self – The updated object.

Return type

object

class xgboost.**XGBRFRegressor**(**, learning_rate=1.0, subsample=0.8, colsample_bynode=0.8, reg_lambda=1e-05, **kwargs*)

Bases: *XGBRegressor*

scikit-learn API for XGBoost random forest regression. See *Using the Scikit-Learn Estimator Interface* for more information.

Parameters

- **n_estimators** (*Optional[int]*) – Number of trees in random forest to fit.
- **max_depth** (*Optional[int]*) – Maximum tree depth for base learners.

- **max_leaves** (*Optional[int]*) – Maximum number of leaves; 0 indicates no limit.
- **max_bin** (*Optional[int]*) – If using histogram-based algorithm, maximum number of bins per feature
- **grow_policy** (*Optional[str]*) – Tree growing policy.
  - depthwise: Favors splitting at nodes closest to the node,
  - lossguide: Favors splitting at nodes with highest loss change.
- **learning_rate** (*Optional[float]*) – Boosting learning rate (xgb's "eta")
- **verbosity** (*Optional[int]*) – The degree of verbosity. Valid values are 0 (silent) - 3 (debug).
- **objective** (*Union[str, xgboost.sklearn._SklObjWProto, Callable[[Any, Any], Tuple[numpy.ndarray, numpy.ndarray]], NoneType]*) – Specify the learning task and the corresponding learning objective or a custom objective function to be used.

  For custom objective, see *Custom Objective and Evaluation Metric* and *Custom objective and metric* for more information, along with the end note for function signatures.
- **booster** (*Optional[str]*) – Specify which booster to use: `gbtree`, `gblinear` or `dart`.

  Deprecated since version 3.3.0: `gblinear` is deprecated and support will be removed in a future release.
- **tree_method** (*Optional[str]*) – Specify which tree method to use. Default to auto. If this parameter is set to default, XGBoost will choose the most conservative option available. It's recommended to study this option from the parameters document *tree method*
- **n_jobs** (*Optional[int]*) – Number of parallel threads used to run xgboost. When used with other Scikit-Learn algorithms like grid search, you may choose which algorithm to parallelize and balance the threads. Creating thread contention will significantly slow down both algorithms.
- **gamma** (*Optional[float]*) – (min_split_loss) Minimum loss reduction required to make a further partition on a leaf node of the tree.
- **min_child_weight** (*Optional[float]*) – Minimum sum of instance weight(hessian) needed in a child.
- **max_delta_step** (*Optional[float]*) – Maximum delta step we allow each tree's weight estimation to be.
- **subsample** (*Optional[float]*) – Subsample ratio of the training instance.
- **sampling_method** (*Optional[str]*) – Sampling method. Used only by the GPU version of `hist` tree method.
  - `uniform`: Select random training instances uniformly.
  - **gradient_based: Select random training instances with higher probability**
    when the gradient and hessian are larger. (cf. CatBoost)
- **colsample_bytree** (*Optional[float]*) – Subsample ratio of columns when constructing each tree.
- **colsample_bylevel** (*Optional[float]*) – Subsample ratio of columns for each level.
- **colsample_bynode** (*Optional[float]*) – Subsample ratio of columns for each split.
- **reg_alpha** (*Optional[float]*) – L1 regularization term on weights (xgb's alpha).

- **reg_lambda** (*Optional[float]*) – L2 regularization term on weights (xgb's lambda).
- **scale_pos_weight** (*Optional[float]*) – Balancing of positive and negative weights.
- **base_score** (*Union[float, List[float], NoneType]*) – The initial prediction score of all instances, global bias.
- **random_state** (*Union[numpy.random.mtrand.RandomState, numpy.random._generator.Generator, int, NoneType]*) – Random number seed.

Note

Using gblinear booster with shotgun updater is nondeterministic as it uses Hogwild algorithm.

- **missing** (*float*) – Value in the data which needs to be present as a missing value. Default to `numpy.nan`.
- **num_parallel_tree** (*Optional[int]*) – Used for boosting random forest.
- **monotone_constraints** (*Union[Dict[str, int], str, NoneType]*) – Constraint of variable monotonicity. See *tutorial* for more information.
- **interaction_constraints** (*Union[str, List[Tuple[str]], NoneType]*) – Constraints for interaction representing permitted interactions. The constraints must be specified in the form of a nested list, e.g. `[[0, 1], [2, 3, 4]]`, where each inner list is a group of indices of features that are allowed to interact with each other. See *tutorial* for more information
- **importance_type** (*Optional[str]*) – The feature importance type for the feature_importances_ property:
  - For tree model, it's either "gain", "weight", "cover", "total_gain" or "total_cover".
  - For linear model, only "weight" is defined and it's the normalized coefficients without bias.
- **device** (*Optional[str]*) – Added in version 2.0.0.

  Device ordinal, available options are *cpu*, *cuda*, and *gpu*.
- **validate_parameters** (*Optional[bool]*) – Give warnings for unknown parameter.
- **enable_categorical** (*bool*) – See the same parameter of `DMatrix` for details.
- **feature_types** (*Optional[Sequence[str]]*) – Added in version 1.7.0.

  Used for specifying feature types without constructing a dataframe. See the `DMatrix` for details.
- **feature_weights** (*Optional[ArrayLike]*) – Weight for each feature, defines the probability of each feature being selected when colsample is being used. All values must be greater than 0, otherwise a *ValueError* is thrown.
- **max_cat_to_onehot** (*Optional[int]*) – Added in version 1.6.0.

Note

This parameter is experimental

A threshold for deciding whether XGBoost should use one-hot encoding based split for categorical data. When number of categories is lesser than the threshold then one-hot encoding is chosen, otherwise the categories will be partitioned into children nodes. Also, *enable_categorical* needs to be set to have categorical feature support. See *Categorical Data* and *Parameters for Categorical Feature* for details.

- **max_cat_threshold** (*Optional[int]*) – Added in version 1.7.0.

  Note

  This parameter is experimental

  Maximum number of categories considered for each split. Used only by partition-based splits for preventing over-fitting. Also, *enable_categorical* needs to be set to have categorical feature support. See *Categorical Data* and *Parameters for Categorical Feature* for details.

- **multi_strategy** (*Optional[str]*) – Added in version 2.0.0.

  Note

  This parameter is working-in-progress.

  The strategy used for training multi-target models, including multi-target regression and multi-class classification. See *Multiple Outputs* for more information.

  – `one_output_per_tree`: One model for each target.

  – `multi_output_tree`: Use multi-target trees.

- **eval_metric** (*Union[str, List[Union[str, Callable]], Callable, NoneType]*) – Added in version 1.6.0.

  Metric used for monitoring the training result and early stopping. It can be a string or list of strings as names of predefined metric in XGBoost (See *XGBoost Parameters*), one of the metrics in `sklearn.metrics`, or any other user defined metric that looks like *sklearn.metrics*.

  If custom objective is also provided, then custom metric should implement the corresponding reverse link function.

  Unlike the *scoring* parameter commonly used in scikit-learn, when a callable object is provided, it's assumed to be a cost function and by default XGBoost will minimize the result during early stopping.

  For advanced usage on Early stopping like directly choosing to maximize instead of minimize, see `xgboost.callback.EarlyStopping`.

  See *Custom Objective and Evaluation Metric* and *Custom objective and metric* for more information.

```
from sklearn.datasets import load_diabetes
from sklearn.metrics import mean_absolute_error
```

(continues on next page)

(continued from previous page)

```
X, y = load_diabetes(return_X_y=True)
reg = xgb.XGBRegressor(
    tree_method="hist",
    eval_metric=mean_absolute_error,
)
reg.fit(X, y, eval_set=[(X, y)])
```

- **early_stopping_rounds** (*Optional[int]*) – Added in version 1.6.0.
  - Activates early stopping. Validation metric needs to improve at least once in every **early_stopping_rounds** round(s) to continue training. Requires at least one item in **eval_set** in `fit()`.
  - If early stopping occurs, the model will have two additional attributes: `best_score` and `best_iteration`. These are used by the `predict()` and `apply()` methods to determine the optimal number of trees during inference. If users want to access the full model (including trees built after early stopping), they can specify the *iteration_range* in these inference methods. In addition, other utilities like model plotting can also use the entire model.
  - If you prefer to discard the trees after *best_iteration*, consider using the callback function `xgboost.callback.EarlyStopping`.
  - If there's more than one item in **eval_set**, the last entry will be used for early stopping. If there's more than one metric in **eval_metric**, the last metric will be used for early stopping.
- **callbacks** (*Optional[List[xgboost.callback.TrainingCallback]]*) – List of callback functions that are applied at end of each iteration. It is possible to use predefined callbacks by using *Callback API*.

> **Note**
>
> States in callback are not preserved during training, which means callback objects can not be reused for multiple training sessions without reinitialization or deepcopy.

```
for params in parameters_grid:
    # be sure to (re)initialize the callbacks before each run
    callbacks = [xgb.callback.LearningRateScheduler(custom_rates)]
    reg = xgboost.XGBRegressor(**params, callbacks=callbacks)
    reg.fit(X, y)
```

- **kwargs** (*Optional[Any]*) – Keyword arguments for XGBoost Booster object. Full documentation of parameters can be found *here*. Attempting to set a parameter via the constructor args and **kwargs dict simultaneously will result in a TypeError.

> **Note**
>
> **kwargs unsupported by scikit-learn
>
> **kwargs is unsupported by scikit-learn. We do not guarantee that parameters passed via this argument will interact properly with scikit-learn.

> **Note**
>
> Custom objective function
>
> A custom objective function can be provided for the `objective` parameter. In this case, it should have the signature `objective(y_true, y_pred) -> [grad, hess]` or `objective(y_true, y_pred, *, sample_weight) -> [grad, hess]`:
>
> **y_true: array_like of shape [n_samples]**
> The target values
>
> **y_pred: array_like of shape [n_samples]**
> The predicted values
>
> **sample_weight :**
> Optional sample weights.
>
> **grad: array_like of shape [n_samples]**
> The value of the gradient for each sample point.
>
> **hess: array_like of shape [n_samples]**
> The value of the second derivative for each sample point
>
> Note that, if the custom objective produces negative values for the Hessian, these will be clipped. If the objective is non-convex, one might also consider using the expected Hessian (Fisher information) instead.

`apply`(*X*, *iteration_range=None*)

Return the predicted leaf every tree for each sample. If the model is trained with early stopping, then *best_iteration* is used automatically.

Parameters

- **X** (*Any*) – Input features matrix. See *Markers* for a list of supported types.
- **iteration_range** (*Tuple[int | integer, int | integer] | None*) – See *predict()*.

Returns

X_leaves – For each datapoint x in X and for each tree, return the index of the leaf x ends up in. Leaves are numbered within `[0; 2**(self.max_depth+1))`, possibly with gaps in the numbering.

Return type

array_like, shape=[n_samples, n_trees]

property best_iteration: int

The best iteration obtained by early stopping. This attribute is 0-based, for instance if the best iteration is the first round, then best_iteration is 0.

property best_score: float

The best score obtained by early stopping.

property coef_: ndarray

Coefficients property

> **Note**
>
> Coefficients are defined only for linear learners

Coefficients are only defined when the linear model is chosen as base learner (*booster=gblinear*). It is not defined for other base learner types, such as tree learners (*booster=gbtree*).

Returns
coef_

Return type
array of shape `[n_features]` or `[n_classes, n_features]`

`evals_result()`

Return the evaluation results.

If **eval_set** is passed to the `fit()` function, you can call `evals_result()` to get evaluation results for all passed **eval_sets**. When **eval_metric** is also passed to the `fit()` function, the **evals_result** will contain the **eval_metrics** passed to the `fit()` function.

The returned evaluation result is a dictionary:

```
{'validation_0': {'logloss': ['0.604835', '0.531479']},
 'validation_1': {'logloss': ['0.41965', '0.17686']}}
```

Return type
evals_result

`property feature_importances_: ndarray`

Feature importances property, return depends on *importance_type* parameter. When model trained with multi-class/multi-label/multi-target dataset, the feature importance is “averaged” over all targets. The “average” is defined based on the importance type. For instance, if the importance type is “total_gain”, then the score is sum of loss change for each split from all trees.

Returns

- **feature_importances_** (array of shape `[n_features]` except for multi-class)
- linear model, which returns an array with shape *(n_features, n_classes)*

`property feature_names_in_: ndarray`

Names of features seen during `fit()`. Defined only when *X* has feature names that are all strings.

`fit`(*X*, *y*, *, *sample_weight=None*, *base_margin=None*, *eval_set=None*, *verbose=True*, *xgb_model=None*, *sample_weight_eval_set=None*, *base_margin_eval_set=None*, *feature_weights=None*)

Fit gradient boosting model.

Note that calling `fit()` multiple times will cause the model object to be re-fit from scratch. To resume training from a previous checkpoint, explicitly pass `xgb_model` argument.

Parameters

- **X** (*Any*) – Input feature matrix. See *Markers* for a list of supported types.

  When the `tree_method` is set to `hist`, internally, the `QuantileDMatrix` will be used instead of the `DMatrix` for conserving memory. However, this has performance implications when the device of input data is not matched with algorithm. For instance, if the input is a numpy array on CPU but `cuda` is used for training, then the data is first processed on CPU then transferred to GPU.

- **y** (*Any*) – Labels

- **sample_weight** (*Any | None*) – instance weights
- **base_margin** (*Any | None*) – Global bias for each instance. See *Intercept* for details.
- **eval_set** (*Sequence[Tuple[Any, Any]] | None*) – A list of (X, y) tuple pairs to use as validation sets, for which metrics will be computed. Validation metrics will help us track the performance of the model.
- **verbose** (*bool | int | None*) – If *verbose* is True and an evaluation set is used, the evaluation metric measured on the validation set is printed to stdout at each boosting stage. If *verbose* is an integer, the evaluation metric is printed at each *verbose* boosting stage. The last boosting stage / the boosting stage found by using *early_stopping_rounds* is also printed.
- **xgb_model** (Booster *| str | XGBModel | None*) – file name of stored XGBoost model or 'Booster' instance XGBoost model to be loaded before training (allows training continuation).
- **sample_weight_eval_set** (*Sequence[Any] | None*) – A list of the form [L_1, L_2, . . . , L_n], where each L_i is an array like object storing instance weights for the i-th validation set.
- **base_margin_eval_set** (*Sequence[Any] | None*) – A list of the form [M_1, M_2, . . . , M_n], where each M_i is an array like object storing base margin for the i-th validation set.
- **feature_weights** (*Any | None*) – Deprecated since version 3.0.0.

  Use *feature_weights* in `__init__()` or `set_params()` instead.

Return type
XGBRFRegressor

`get_booster()`

Get the underlying xgboost Booster of this model.

This will raise an exception when fit was not called

Returns
booster

Return type
a xgboost booster of underlying model

`get_metadata_routing()`

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns
routing – A `MetadataRequest` encapsulating routing information.

Return type
MetadataRequest

`get_num_boosting_rounds()`

Gets the number of xgboost boosting rounds.

Return type
int

get_params(*deep=True*)

Get parameters.

Parameters
deep (*bool*)

Return type
Dict[str, *Any*]

get_xgb_params()

Get xgboost specific parameters.

Return type
Dict[str, *Any*]

property intercept_: ndarray

Intercept (bias) property

For tree-based model, the returned value is the *base_score*.

Returns
intercept_

Return type
array of shape `(1,)` or `[n_classes]`

load_model(*fname*)

Load the model from a file or a bytearray.

The model is saved in an XGBoost internal format which is universal among the various XGBoost interfaces. Auxiliary attributes of the Python Booster object (such as feature_names) are only saved when using JSON or UBJSON (default) format. Also, parameters that are not part of the model (like metrics, *max_depth*, etc) are not saved, see *Model IO* for more info.

```
model.save_model("model.json")
model.load_model("model.json")

# or
model.save_model("model.ubj")
model.load_model("model.ubj")

# or
buf = model.save_raw()
model.load_model(buf)
```

Parameters
fname (*PathLike | bytearray | str*) – Input file name or memory buffer(see also save_raw)

Return type
None

property n_features_in_: int

Number of features seen during `fit()`.

predict(*X*, *, *output_margin=False*, *validate_features=True*, *base_margin=None*, *iteration_range=None*)

Predict with *X*. If the model is trained with early stopping, then `best_iteration` is used automatically. The estimator uses *inplace_predict* by default and falls back to using `DMatrix` if devices between the data and the estimator don't match.

Note

This function is only thread safe for *gbtree* and *dart*.

Parameters

- **X** (*Any*) – Data to predict with. See *Markers* for a list of supported types.
- **output_margin** (*bool*) – Whether to output the raw untransformed margin value.
- **validate_features** (*bool*) – When this is True, validate that the Booster's and data's feature_names are identical. Otherwise, it is assumed that the feature_names are the same.
- **base_margin** (*Any | None*) – Global bias for each instance. See *Intercept* for details.
- **iteration_range** (*Tuple[int | integer, int | integer] | None*) – Specifies which layer of trees are used in prediction. For example, if a random forest is trained with 100 rounds. Specifying `iteration_range=(10, 20)`, then only the forests built during [10, 20) (half open set) rounds are used in this prediction.

  Added in version 1.4.0.

Return type
prediction

save_model(*fname*)

Save the model to a file.

The model is saved in an XGBoost internal format which is universal among the various XGBoost interfaces. Auxiliary attributes of the Python Booster object (such as feature_names) are only saved when using JSON or UBJSON (default) format. Also, parameters that are not part of the model (like metrics, *max_depth*, etc) are not saved, see *Model IO* for more info.

```
model.save_model("model.json")
# or
model.save_model("model.ubj")
```

Parameters
fname (*str | PathLike*) – Output file name

Return type
None

score(*X*, *y*, *sample_weight=None*)

Return coefficient of determination on test data.

The coefficient of determination, R^2, is defined as $(1 - \frac{u}{v})$, where u is the residual sum of squares `((y_true - y_pred)** 2).sum()` and v is the total sum of squares `((y_true - y_true.mean()) ** 2).sum()`. The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of *y*, disregarding the input features, would get a R^2 score of 0.0.

Parameters

- **X** (*array-like of shape (n_samples, n_features)*) – Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead with shape `(n_samples, n_samples_fitted)`, where `n_samples_fitted` is the number of samples used in the fitting for the estimator.

- **y** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – True values for *X*.
- **sample_weight** (*array-like of shape (n_samples,), default=None*) – Sample weights.

Returns
score – R^2 of `self.predict(X)` w.r.t. *y*.

Return type
float

Notes

The R^2 score used when calling `score` on a regressor uses `multioutput='uniform_average'` from version 0.23 to keep consistent with default value of `r2_score()`. This influences the `score` method of all the multioutput regressors (except for `MultiOutputRegressor`).

set_fit_request(**, base_margin='$UNCHANGED$', base_margin_eval_set='$UNCHANGED$', eval_set='$UNCHANGED$', feature_weights='$UNCHANGED$', sample_weight='$UNCHANGED$', sample_weight_eval_set='$UNCHANGED$', verbose='$UNCHANGED$', xgb_model='$UNCHANGED$'*)

Configure whether metadata should be requested to be passed to the `fit` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a meta-estimator and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

- **base_margin** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `base_margin` parameter in `fit`.
- **base_margin_eval_set** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `base_margin_eval_set` parameter in `fit`.
- **eval_set** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `eval_set` parameter in `fit`.
- **feature_weights** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `feature_weights` parameter in `fit`.

- **sample_weight** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `sample_weight` parameter in `fit`.
- **sample_weight_eval_set** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `sample_weight_eval_set` parameter in `fit`.
- **verbose** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `verbose` parameter in `fit`.
- **xgb_model** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `xgb_model` parameter in `fit`.
- **self** (`XGBRFRegressor`)

Returns
self – The updated object.

Return type
object

`set_params`(***params*)

Set the parameters of this estimator. Modification of the sklearn method to allow unknown kwargs. This allows using the full range of xgboost parameters that are not defined as member variables in sklearn grid search.

Return type
self

Parameters
params (*Any*)

`set_predict_request`(**, base_margin='$UNCHANGED$', iteration_range='$UNCHANGED$', output_margin='$UNCHANGED$', validate_features='$UNCHANGED$'*)

Configure whether metadata should be requested to be passed to the `predict` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a meta-estimator and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `predict` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `predict`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

- **base_margin** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `base_margin` parameter in `predict`.
- **iteration_range** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `iteration_range` parameter in `predict`.
- **output_margin** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `output_margin` parameter in `predict`.
- **validate_features** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `validate_features` parameter in `predict`.
- **self** (`XGBRFRegressor`)

Returns
self – The updated object.

Return type
object

set_score_request(**, sample_weight='$UNCHANGED$'*)

Configure whether metadata should be requested to be passed to the `score` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a meta-estimator and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

- **sample_weight** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `sample_weight` parameter in `score`.
- **self** (`XGBRFRegressor`)

Returns
self – The updated object.

Return type
object

class xgboost.**XGBRFClassifier**(*, *learning_rate=1.0*, *subsample=0.8*, *colsample_bynode=0.8*, *reg_lambda=1e-05*, ***kwargs*)

Bases: *XGBClassifier*

scikit-learn API for XGBoost random forest classification. See *Using the Scikit-Learn Estimator Interface* for more information.

Parameters

- **n_estimators** (*Optional[int]*) – Number of trees in random forest to fit.
- **max_depth** (*Optional[int]*) – Maximum tree depth for base learners.
- **max_leaves** (*Optional[int]*) – Maximum number of leaves; 0 indicates no limit.
- **max_bin** (*Optional[int]*) – If using histogram-based algorithm, maximum number of bins per feature
- **grow_policy** (*Optional[str]*) – Tree growing policy.
  - depthwise: Favors splitting at nodes closest to the node,
  - lossguide: Favors splitting at nodes with highest loss change.
- **learning_rate** (*Optional[float]*) – Boosting learning rate (xgb's "eta")
- **verbosity** (*Optional[int]*) – The degree of verbosity. Valid values are 0 (silent) - 3 (debug).
- **objective** (*Union[str, xgboost.sklearn._SklObjWProto, Callable[[Any, Any], Tuple[numpy.ndarray, numpy.ndarray]], NoneType]*) – Specify the learning task and the corresponding learning objective or a custom objective function to be used.

  For custom objective, see *Custom Objective and Evaluation Metric* and *Custom objective and metric* for more information, along with the end note for function signatures.
- **booster** (*Optional[str]*) – Specify which booster to use: `gbtree`, `gblinear` or `dart`.

  Deprecated since version 3.3.0: `gblinear` is deprecated and support will be removed in a future release.
- **tree_method** (*Optional[str]*) – Specify which tree method to use. Default to auto. If this parameter is set to default, XGBoost will choose the most conservative option available. It's recommended to study this option from the parameters document *tree method*
- **n_jobs** (*Optional[int]*) – Number of parallel threads used to run xgboost. When used with other Scikit-Learn algorithms like grid search, you may choose which algorithm to parallelize and balance the threads. Creating thread contention will significantly slow down both algorithms.
- **gamma** (*Optional[float]*) – (min_split_loss) Minimum loss reduction required to make a further partition on a leaf node of the tree.
- **min_child_weight** (*Optional[float]*) – Minimum sum of instance weight(hessian) needed in a child.
- **max_delta_step** (*Optional[float]*) – Maximum delta step we allow each tree's weight estimation to be.
- **subsample** (*Optional[float]*) – Subsample ratio of the training instance.
- **sampling_method** (*Optional[str]*) – Sampling method. Used only by the GPU version of `hist` tree method.

  - uniform: Select random training instances uniformly.
  - **gradient_based: Select random training instances with higher probability** when the gradient and hessian are larger. (cf. CatBoost)
- **colsample_bytree** (*Optional[float]*) – Subsample ratio of columns when constructing each tree.
- **colsample_bylevel** (*Optional[float]*) – Subsample ratio of columns for each level.
- **colsample_bynode** (*Optional[float]*) – Subsample ratio of columns for each split.
- **reg_alpha** (*Optional[float]*) – L1 regularization term on weights (xgb's alpha).
- **reg_lambda** (*Optional[float]*) – L2 regularization term on weights (xgb's lambda).
- **scale_pos_weight** (*Optional[float]*) – Balancing of positive and negative weights.
- **base_score** (*Union[float, List[float], NoneType]*) – The initial prediction score of all instances, global bias.
- **random_state** (*Union[numpy.random.mtrand.RandomState, numpy.random._generator.Generator, int, NoneType]*) – Random number seed.

> **Note**
>
> Using gblinear booster with shotgun updater is nondeterministic as it uses Hogwild algorithm.

- **missing** (*float*) – Value in the data which needs to be present as a missing value. Default to numpy.nan.
- **num_parallel_tree** (*Optional[int]*) – Used for boosting random forest.
- **monotone_constraints** (*Union[Dict[str, int], str, NoneType]*) – Constraint of variable monotonicity. See *tutorial* for more information.
- **interaction_constraints** (*Union[str, List[Tuple[str]], NoneType]*) – Constraints for interaction representing permitted interactions. The constraints must be specified in the form of a nested list, e.g. [[0, 1], [2, 3, 4]], where each inner list is a group of indices of features that are allowed to interact with each other. See *tutorial* for more information
- **importance_type** (*Optional[str]*) – The feature importance type for the feature_importances_ property:
  - For tree model, it's either "gain", "weight", "cover", "total_gain" or "total_cover".
  - For linear model, only "weight" is defined and it's the normalized coefficients without bias.
- **device** (*Optional[str]*) – Added in version 2.0.0.

  Device ordinal, available options are *cpu*, *cuda*, and *gpu*.
- **validate_parameters** (*Optional[bool]*) – Give warnings for unknown parameter.
- **enable_categorical** (*bool*) – See the same parameter of DMatrix for details.
- **feature_types** (*Optional[Sequence[str]]*) – Added in version 1.7.0.

  Used for specifying feature types without constructing a dataframe. See the DMatrix for details.

- **feature_weights** (*Optional[ArrayLike]*) – Weight for each feature, defines the probability of each feature being selected when colsample is being used. All values must be greater than 0, otherwise a *ValueError* is thrown.
- **max_cat_to_onehot** (*Optional[int]*) – Added in version 1.6.0.

  Note

  This parameter is experimental

  A threshold for deciding whether XGBoost should use one-hot encoding based split for categorical data. When number of categories is lesser than the threshold then one-hot encoding is chosen, otherwise the categories will be partitioned into children nodes. Also, *enable_categorical* needs to be set to have categorical feature support. See *Categorical Data* and *Parameters for Categorical Feature* for details.
- **max_cat_threshold** (*Optional[int]*) – Added in version 1.7.0.

  Note

  This parameter is experimental

  Maximum number of categories considered for each split. Used only by partition-based splits for preventing over-fitting. Also, *enable_categorical* needs to be set to have categorical feature support. See *Categorical Data* and *Parameters for Categorical Feature* for details.
- **multi_strategy** (*Optional[str]*) – Added in version 2.0.0.

  Note

  This parameter is working-in-progress.

  The strategy used for training multi-target models, including multi-target regression and multi-class classification. See *Multiple Outputs* for more information.

  – `one_output_per_tree`: One model for each target.

  – `multi_output_tree`: Use multi-target trees.
- **eval_metric** (*Union[str, List[Union[str, Callable]], Callable, NoneType]*) – Added in version 1.6.0.

  Metric used for monitoring the training result and early stopping. It can be a string or list of strings as names of predefined metric in XGBoost (See *XGBoost Parameters*), one of the metrics in `sklearn.metrics`, or any other user defined metric that looks like *sklearn.metrics*.

  If custom objective is also provided, then custom metric should implement the corresponding reverse link function.

  Unlike the *scoring* parameter commonly used in scikit-learn, when a callable object is provided, it's assumed to be a cost function and by default XGBoost will minimize the result during early stopping.

  For advanced usage on Early stopping like directly choosing to maximize instead of minimize, see `xgboost.callback.EarlyStopping`.

See *Custom Objective and Evaluation Metric* and *Custom objective and metric* for more information.

```
from sklearn.datasets import load_diabetes
from sklearn.metrics import mean_absolute_error
X, y = load_diabetes(return_X_y=True)
reg = xgb.XGBRegressor(
    tree_method="hist",
    eval_metric=mean_absolute_error,
)
reg.fit(X, y, eval_set=[(X, y)])
```

- **early_stopping_rounds** (*Optional[int]*) – Added in version 1.6.0.
  - Activates early stopping. Validation metric needs to improve at least once in every **early_stopping_rounds** round(s) to continue training. Requires at least one item in **eval_set** in *fit()*.
  - If early stopping occurs, the model will have two additional attributes: *best_score* and *best_iteration*. These are used by the *predict()* and *apply()* methods to determine the optimal number of trees during inference. If users want to access the full model (including trees built after early stopping), they can specify the *iteration_range* in these inference methods. In addition, other utilities like model plotting can also use the entire model.
  - If you prefer to discard the trees after *best_iteration*, consider using the callback function *xgboost.callback.EarlyStopping*.
  - If there's more than one item in **eval_set**, the last entry will be used for early stopping. If there's more than one metric in **eval_metric**, the last metric will be used for early stopping.
- **callbacks** (*Optional[List[xgboost.callback.TrainingCallback]]*) – List of callback functions that are applied at end of each iteration. It is possible to use predefined callbacks by using *Callback API*.

> **Note**
>
> States in callback are not preserved during training, which means callback objects can not be reused for multiple training sessions without reinitialization or deepcopy.

```
for params in parameters_grid:
    # be sure to (re)initialize the callbacks before each run
    callbacks = [xgb.callback.LearningRateScheduler(custom_rates)]
    reg = xgboost.XGBRegressor(**params, callbacks=callbacks)
    reg.fit(X, y)
```

- **kwargs** (*Optional[Any]*) – Keyword arguments for XGBoost Booster object. Full documentation of parameters can be found *here*. Attempting to set a parameter via the constructor args and **kwargs dict simultaneously will result in a TypeError.

> **Note**
>
> **kwargs unsupported by scikit-learn

**kwargs is unsupported by scikit-learn. We do not guarantee that parameters passed via this argument will interact properly with scikit-learn.

Note

Custom objective function

A custom objective function can be provided for the `objective` parameter. In this case, it should have the signature `objective(y_true, y_pred) -> [grad, hess]` or `objective(y_true, y_pred, *, sample_weight) -> [grad, hess]`:

y_true: array_like of shape [n_samples]
The target values

y_pred: array_like of shape [n_samples]
The predicted values

sample_weight :
Optional sample weights.

grad: array_like of shape [n_samples]
The value of the gradient for each sample point.

hess: array_like of shape [n_samples]
The value of the second derivative for each sample point

Note that, if the custom objective produces negative values for the Hessian, these will be clipped. If the objective is non-convex, one might also consider using the expected Hessian (Fisher information) instead.

apply(*X*, *iteration_range=None*)

Return the predicted leaf every tree for each sample. If the model is trained with early stopping, then *best_iteration* is used automatically.

Parameters

- **X** (*Any*) – Input features matrix. See *Markers* for a list of supported types.
- **iteration_range** (*Tuple[int | integer, int | integer] | None*) – See *predict()*.

Returns
X_leaves – For each datapoint x in X and for each tree, return the index of the leaf x ends up in. Leaves are numbered within `[0; 2**(self.max_depth+1))`, possibly with gaps in the numbering.

Return type
array_like, shape=[n_samples, n_trees]

property best_iteration: int

The best iteration obtained by early stopping. This attribute is 0-based, for instance if the best iteration is the first round, then best_iteration is 0.

property best_score: float

The best score obtained by early stopping.

property coef_: ndarray

Coefficients property

> **Note**
>
> Coefficients are defined only for linear learners
>
> Coefficients are only defined when the linear model is chosen as base learner (*booster=gblinear*). It is not defined for other base learner types, such as tree learners (*booster=gbtree*).

Returns
coef_

Return type
array of shape `[n_features]` or `[n_classes, n_features]`

`evals_result()`

Return the evaluation results.

If **eval_set** is passed to the `fit()` function, you can call `evals_result()` to get evaluation results for all passed **eval_sets**. When **eval_metric** is also passed to the `fit()` function, the **evals_result** will contain the **eval_metrics** passed to the `fit()` function.

The returned evaluation result is a dictionary:

```
{'validation_0': {'logloss': ['0.604835', '0.531479']},
 'validation_1': {'logloss': ['0.41965', '0.17686']}}
```

Return type
evals_result

`property feature_importances_: ndarray`

Feature importances property, return depends on *importance_type* parameter. When model trained with multi-class/multi-label/multi-target dataset, the feature importance is "averaged" over all targets. The "average" is defined based on the importance type. For instance, if the importance type is "total_gain", then the score is sum of loss change for each split from all trees.

Returns

- **feature_importances_** (array of shape `[n_features]` except for multi-class)
- linear model, which returns an array with shape *(n_features, n_classes)*

`property feature_names_in_: ndarray`

Names of features seen during `fit()`. Defined only when *X* has feature names that are all strings.

`fit`(*X*, *y*, **, *sample_weight=None*, *base_margin=None*, *eval_set=None*, *verbose=True*, *xgb_model=None*, *sample_weight_eval_set=None*, *base_margin_eval_set=None*, *feature_weights=None*)

Fit gradient boosting classifier.

Note that calling `fit()` multiple times will cause the model object to be re-fit from scratch. To resume training from a previous checkpoint, explicitly pass `xgb_model` argument.

Parameters

- **X** (*Any*) – Input feature matrix. See *Markers* for a list of supported types.

  When the `tree_method` is set to `hist`, internally, the `QuantileDMatrix` will be used instead of the `DMatrix` for conserving memory. However, this has performance implications when the device of input data is not matched with algorithm. For instance, if the input is a

numpy array on CPU but cuda is used for training, then the data is first processed on CPU then transferred to GPU.

- **y** (*Any*) – Labels
- **sample_weight** (*Any | None*) – instance weights
- **base_margin** (*Any | None*) – Global bias for each instance. See *Intercept* for details.
- **eval_set** (*Sequence[Tuple[Any, Any]] | None*) – A list of (X, y) tuple pairs to use as validation sets, for which metrics will be computed. Validation metrics will help us track the performance of the model.
- **verbose** (*bool | int | None*) – If *verbose* is True and an evaluation set is used, the evaluation metric measured on the validation set is printed to stdout at each boosting stage. If *verbose* is an integer, the evaluation metric is printed at each *verbose* boosting stage. The last boosting stage / the boosting stage found by using *early_stopping_rounds* is also printed.
- **xgb_model** (Booster | *str* | *XGBModel* | *None*) – file name of stored XGBoost model or 'Booster' instance XGBoost model to be loaded before training (allows training continuation).
- **sample_weight_eval_set** (*Sequence[Any] | None*) – A list of the form [L_1, L_2, . . . , L_n], where each L_i is an array like object storing instance weights for the i-th validation set.
- **base_margin_eval_set** (*Sequence[Any] | None*) – A list of the form [M_1, M_2, . . . , M_n], where each M_i is an array like object storing base margin for the i-th validation set.
- **feature_weights** (*Any | None*) – Deprecated since version 3.0.0.

  Use *feature_weights* in `__init__()` or `set_params()` instead.

Return type
XGBRFClassifier

`get_booster()`

Get the underlying xgboost Booster of this model.

This will raise an exception when fit was not called

Returns
booster

Return type
a xgboost booster of underlying model

`get_metadata_routing()`

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns
routing – A `MetadataRequest` encapsulating routing information.

Return type
MetadataRequest

`get_num_boosting_rounds()`

Gets the number of xgboost boosting rounds.

Return type
int

`get_params(deep=True)`

Get parameters.

Parameters
deep (*bool*)

Return type
Dict[str, *Any*]

`get_xgb_params()`

Get xgboost specific parameters.

Return type
Dict[str, *Any*]

`property intercept_: ndarray`

Intercept (bias) property

For tree-based model, the returned value is the *base_score*.

Returns
intercept_

Return type
array of shape `(1,)` or `[n_classes]`

`load_model(fname)`

Load the model from a file or a bytearray.

The model is saved in an XGBoost internal format which is universal among the various XGBoost interfaces. Auxiliary attributes of the Python Booster object (such as feature_names) are only saved when using JSON or UBJSON (default) format. Also, parameters that are not part of the model (like metrics, *max_depth*, etc) are not saved, see *Model IO* for more info.

```
model.save_model("model.json")
model.load_model("model.json")

# or
model.save_model("model.ubj")
model.load_model("model.ubj")

# or
buf = model.save_raw()
model.load_model(buf)
```

Parameters
fname (*PathLike | bytearray | str*) – Input file name or memory buffer(see also save_raw)

Return type
None

`property n_features_in_: int`

Number of features seen during `fit()`.

predict(*X*, *, *output_margin=False*, *validate_features=True*, *base_margin=None*, *iteration_range=None*)

Predict with *X*. If the model is trained with early stopping, then `best_iteration` is used automatically. The estimator uses *inplace_predict* by default and falls back to using `DMatrix` if devices between the data and the estimator don't match.

Note

This function is only thread safe for *gbtree* and *dart*.

Parameters

- **X** (*Any*) – Data to predict with. See *Markers* for a list of supported types.
- **output_margin** (*bool*) – Whether to output the raw untransformed margin value.
- **validate_features** (*bool*) – When this is True, validate that the Booster's and data's feature_names are identical. Otherwise, it is assumed that the feature_names are the same.
- **base_margin** (*Any | None*) – Global bias for each instance. See *Intercept* for details.
- **iteration_range** (*Tuple[int | integer, int | integer] | None*) – Specifies which layer of trees are used in prediction. For example, if a random forest is trained with 100 rounds. Specifying `iteration_range=(10, 20)`, then only the forests built during [10, 20) (half open set) rounds are used in this prediction.

  Added in version 1.4.0.

Return type

prediction

predict_proba(*X*, *validate_features=True*, *base_margin=None*, *iteration_range=None*)

Predict the probability of each *X* example being of a given class. If the model is trained with early stopping, then `best_iteration` is used automatically. The estimator uses *inplace_predict* by default and falls back to using `DMatrix` if devices between the data and the estimator don't match.

Note

This function is only thread safe for *gbtree* and *dart*.

Parameters

- **X** (*Any*) – Feature matrix. See *Markers* for a list of supported types.
- **validate_features** (*bool*) – When this is True, validate that the Booster's and data's feature_names are identical. Otherwise, it is assumed that the feature_names are the same.
- **base_margin** (*Any | None*) – Global bias for each instance. See *Intercept* for details.
- **iteration_range** (*Tuple[int | integer, int | integer] | None*) – Specifies which layer of trees are used in prediction. For example, if a random forest is trained with 100 rounds. Specifying *iteration_range=(10, 20)*, then only the forests built during [10, 20) (half open set) rounds are used in this prediction.

Returns

a numpy array of shape array-like of shape (n_samples, n_classes) with the probability of each data example being of a given class.

Return type
prediction

save_model(*fname*)

Save the model to a file.

The model is saved in an XGBoost internal format which is universal among the various XGBoost interfaces. Auxiliary attributes of the Python Booster object (such as feature_names) are only saved when using JSON or UBJSON (default) format. Also, parameters that are not part of the model (like metrics, *max_depth*, etc) are not saved, see *Model IO* for more info.

```
model.save_model("model.json")
# or
model.save_model("model.ubj")
```

Parameters
fname (*str | PathLike*) – Output file name

Return type
None

score(*X*, *y*, *sample_weight=None*)

Return accuracy on provided data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

- **X** (*array-like of shape (n_samples, n_features)*) – Test samples.
- **y** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – True labels for *X*.
- **sample_weight** (*array-like of shape (n_samples,), default=None*) – Sample weights.

Returns
score – Mean accuracy of `self.predict(X)` w.r.t. *y*.

Return type
float

set_fit_request(*, *base_margin='$UNCHANGED$'*, *base_margin_eval_set='$UNCHANGED$'*, *eval_set='$UNCHANGED$'*, *feature_weights='$UNCHANGED$'*, *sample_weight='$UNCHANGED$'*, *sample_weight_eval_set='$UNCHANGED$'*, *verbose='$UNCHANGED$'*, *xgb_model='$UNCHANGED$'*)

Configure whether metadata should be requested to be passed to the `fit` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a meta-estimator and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.

- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

- **base_margin** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `base_margin` parameter in `fit`.
- **base_margin_eval_set** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `base_margin_eval_set` parameter in `fit`.
- **eval_set** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `eval_set` parameter in `fit`.
- **feature_weights** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `feature_weights` parameter in `fit`.
- **sample_weight** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `sample_weight` parameter in `fit`.
- **sample_weight_eval_set** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `sample_weight_eval_set` parameter in `fit`.
- **verbose** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `verbose` parameter in `fit`.
- **xgb_model** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `xgb_model` parameter in `fit`.
- **self** (`XGBRFClassifier`)

Returns

self – The updated object.

Return type

object

`set_params`(***params*)

Set the parameters of this estimator. Modification of the sklearn method to allow unknown kwargs. This allows using the full range of xgboost parameters that are not defined as member variables in sklearn grid search.

Return type

self

Parameters

params (*Any*)

set_predict_proba_request(**, base_margin='$UNCHANGED$', iteration_range='$UNCHANGED$', validate_features='$UNCHANGED$'*)

Configure whether metadata should be requested to be passed to the `predict_proba` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a meta-estimator and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `predict_proba` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `predict_proba`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

- **base_margin** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `base_margin` parameter in `predict_proba`.
- **iteration_range** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `iteration_range` parameter in `predict_proba`.
- **validate_features** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `validate_features` parameter in `predict_proba`.
- **self** (`XGBRFClassifier`)

Returns
self – The updated object.

Return type
object

set_predict_request(**, base_margin='$UNCHANGED$', iteration_range='$UNCHANGED$', output_margin='$UNCHANGED$', validate_features='$UNCHANGED$'*)

Configure whether metadata should be requested to be passed to the `predict` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a meta-estimator and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `predict` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `predict`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.

- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

- **base_margin** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `base_margin` parameter in `predict`.
- **iteration_range** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `iteration_range` parameter in `predict`.
- **output_margin** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `output_margin` parameter in `predict`.
- **validate_features** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `validate_features` parameter in `predict`.
- **self** (`XGBRFClassifier`)

Returns
self – The updated object.

Return type
object

set_score_request(**, sample_weight='$UNCHANGED$'*)

Configure whether metadata should be requested to be passed to the `score` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a meta-estimator and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

- **sample_weight** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `sample_weight` parameter in `score`.
- **self** (`XGBRFClassifier`)

Returns
self – The updated object.

Return type
object

Plotting API

Plotting Library.

`xgboost.plot_importance`(*booster*, **, *ax=None*, *height=0.2*, *xlim=None*, *ylim=None*, *title='Feature importance'*, *xlabel='Importance score'*, *ylabel='Features'*, *fmap=''*, *importance_type='weight'*, *max_num_features=None*, *grid=True*, *show_values=True*, *values_format='{v}'*, ***kwargs*)

Plot importance based on fitted trees.

Parameters

- **booster** (*XGBModel | Booster | dict*) – Booster or XGBModel instance, or dict taken by Booster.get_fscore()
- **ax** (*matplotlib Axes*) – Target axes instance. If None, new figure and axes will be created.
- **grid** (*bool*) – Turn the axes grids on or off. Default is True (On).
- **importance_type** (*str*) – How the importance is calculated: either "weight", "gain", or "cover"
  - "weight" is the number of times a feature appears in a tree
  - "gain" is the average gain of splits which use the feature
  - "cover" is the average coverage of splits which use the feature where coverage is defined as the number of samples affected by the split
- **max_num_features** (*int | None*) – Maximum number of top features displayed on plot. If None, all features will be displayed.
- **height** (*float*) – Bar height, passed to ax.barh()
- **xlim** (*tuple | None*) – Tuple passed to axes.xlim()
- **ylim** (*tuple | None*) – Tuple passed to axes.ylim()
- **title** (*str*) – Axes title. To disable, pass None.
- **xlabel** (*str*) – X axis title label. To disable, pass None.
- **ylabel** (*str*) – Y axis title label. To disable, pass None.
- **fmap** (*str | PathLike*) – The name of feature map file.
- **show_values** (*bool*) – Show values on plot. To disable, pass False.
- **values_format** (*str*) – Format string for values. "v" will be replaced by the value of the feature importance. e.g. Pass "{v:.2f}" in order to limit the number of digits after the decimal point to two, for each value printed on the graph.
- **kwargs** (*Any*) – Other keywords passed to ax.barh()

Returns
ax

Return type
matplotlib Axes

`xgboost.plot_tree`(*booster*, ***, *fmap=''*, *num_trees=None*, *rankdir=None*, *ax=None*, *with_stats=False*, *tree_idx=0*, ***kwargs*)

Plot specified tree.

Parameters

- **booster** (`Booster | XGBModel`) – Booster or XGBModel instance
- **fmap** (`str (optional)`) – The name of feature map file
- **num_trees** (`int | None`) – Deprecated since version 3.0.
- **rankdir** (`str, default "TB"`) – Passed to graphviz via graph_attr
- **ax** (`matplotlib Axes, default None`) – Target axes instance. If None, new figure and axes will be created.
- **with_stats** (`bool`) – Added in version 3.0.

  See `to_graphviz()`.
- **tree_idx** (`int`) – Added in version 3.0.

  See `to_graphviz()`.
- **kwargs** (`Any`) – Other keywords passed to `to_graphviz()`

Returns
ax

Return type
matplotlib Axes

`xgboost.to_graphviz`(*booster*, ***, *fmap=''*, *num_trees=None*, *rankdir=None*, *yes_color=None*, *no_color=None*, *condition_node_params=None*, *leaf_node_params=None*, *with_stats=False*, *tree_idx=0*, ***kwargs*)

Convert specified tree to graphviz instance. IPython can automatically plot the returned graphviz instance. Otherwise, you should call .render() method of the returned graphviz instance.

Parameters

- **booster** (`Booster | XGBModel`) – Booster or XGBModel instance
- **fmap** (`str | PathLike`) – The name of feature map file
- **num_trees** (`int | None`) – Deprecated since version 3.0.

  Specify the ordinal number of target tree
- **rankdir** (`str | None`) – Passed to graphviz via graph_attr
- **yes_color** (`str | None`) – Edge color when meets the node condition.
- **no_color** (`str | None`) – Edge color when doesn't meet the node condition.
- **condition_node_params** (`dict | None`) – Condition node configuration for for graphviz. Example:

```
{'shape': 'box',
 'style': 'filled,rounded',
 'fillcolor': '#78bceb'}
```

- **leaf_node_params** (`dict | None`) – Leaf node configuration for graphviz. Example:

```
{'shape': 'box',
 'style': 'filled',
 'fillcolor': '#e48038'}
```

- **with_stats** (*bool*) – Added in version 3.0.

  Controls whether the split statistics should be included.

- **tree_idx** (*int*) – Added in version 3.0.

  Specify the ordinal index of target tree.

- **kwargs** (*Any*) – Other keywords passed to graphviz graph_attr, e.g. `graph [ {key} = {value} ]`

Returns
graph

Return type
graphviz.Source

Callback API

Callback library containing training routines. See *Callback Functions* for a quick introduction.

class xgboost.callback.**TrainingCallback**

Interface for training callback.

Added in version 1.3.0.

after_iteration(*model*, *epoch*, *evals_log*)

Run after each iteration. Returns *True* when training should stop.

Parameters

- **model** (*Any*) – Eeither a `Booster` object or a CVPack if the cv function in xgboost is being used.
- **epoch** (*int*) – The current training iteration.
- **evals_log** (*Dict[str, Dict[str, List[float] | List[Tuple[float, float]]]]*) – A dictionary containing the evaluation history:

```
{"data_name": {"metric_name": [0.5, ...]}}
```

Return type
bool

after_training(*model*)

Run after training is finished.

Parameters
model (*Any*)

Return type
Any

before_iteration(*model*, *epoch*, *evals_log*)

Run before each iteration. Returns True when training should stop. See *after_iteration()* for details.

Parameters

- **model** (*Any*)
- **epoch** (*int*)
- **evals_log** (*Dict[str, Dict[str, List[float] | List[Tuple[float, float]]]]*)

Return type
bool

before_training(*model*)

Run before training starts.

Parameters
model (*Any*)

Return type
Any

class xgboost.callback.**EvaluationMonitor**(*rank=0*, *period=1*, *show_stdv=False*, *logger=<function communicator_print>*)

Bases: *TrainingCallback*

Print the evaluation result at each iteration.

Added in version 1.3.0.

Parameters

- **rank** (*int*) – Which worker should be used for printing the result.
- **period** (*int*) – How many epoches between printing.
- **show_stdv** (*bool*) – Used in cv to show standard deviation. Users should not specify it.
- **logger** (*Callable[[str], None]*) – A callable used for logging evaluation result.

after_iteration(*model*, *epoch*, *evals_log*)

Run after each iteration. Returns *True* when training should stop.

Parameters

- **model** (*Any*) – Eeither a *Booster* object or a CVPack if the cv function in xgboost is being used.
- **epoch** (*int*) – The current training iteration.
- **evals_log** (*Dict[str, Dict[str, List[float] | List[Tuple[float, float]]]]*) – A dictionary containing the evaluation history:

```
{"data_name": {"metric_name": [0.5, ...]}}
```

Return type
bool

after_training(*model*)

Run after training is finished.

Parameters
`model` (*Any*)

Return type
Any

class `xgboost.callback.EarlyStopping`(*, *rounds*, *metric_name=None*, *data_name=None*, *maximize=None*, *save_best=False*, *min_delta=0.0*)

Bases: `TrainingCallback`

Callback function for early stopping

Added in version 1.3.0.

Parameters

- **rounds** (*int*) – Early stopping rounds.
- **metric_name** (*str | None*) – Name of metric that is used for early stopping.
- **data_name** (*str | None*) – Name of dataset that is used for early stopping.
- **maximize** (*bool | None*) – Whether to maximize evaluation metric. None means auto (discouraged).
- **save_best** (*bool | None*) – Whether training should return the best model or the last model. If set to *True*, it will only keep the boosting rounds up to the detected best iteration, discarding the ones that come after. This is only supported with tree methods (not *gblinear*). Also, the *cv* function doesn't return a model, the parameter is not applicable.
- **min_delta** (*float*) – Added in version 1.5.0.

  Minimum absolute change in score to be qualified as an improvement.

Examples

```
es = xgboost.callback.EarlyStopping(
    rounds=2,
    min_delta=1e-3,
    save_best=True,
    maximize=False,
    data_name="validation_0",
    metric_name="mlogloss",
)
clf = xgboost.XGBClassifier(tree_method="hist", device="cuda", callbacks=[es])

X, y = load_digits(return_X_y=True)
clf.fit(X, y, eval_set=[(X, y)])
```

`after_iteration`(*model*, *epoch*, *evals_log*)

Run after each iteration. Returns *True* when training should stop.

Parameters

- **model** (*Any*) – Eeither a `Booster` object or a CVPack if the cv function in xgboost is being used.
- **epoch** (*int*) – The current training iteration.
- **evals_log** (*Dict[str, Dict[str, List[float] | List[Tuple[float, float]]]]*) – A dictionary containing the evaluation history:

```
{"data_name": {"metric_name": [0.5, ...]}}
```

Return type
bool

after_training(*model*)

Run after training is finished.

Parameters
model (*Any*)

Return type
Any

before_training(*model*)

Run before training starts.

Parameters
model (*Any*)

Return type
Any

class xgboost.callback.**LearningRateScheduler**(*learning_rates*)

Bases: *TrainingCallback*

Callback function for scheduling learning rate.

Added in version 1.3.0.

Parameters
learning_rates (*Callable[[int], float] | Sequence[float]*) – If it's a callable object, then it should accept an integer parameter *epoch* and returns the corresponding learning rate. Otherwise it should be a sequence like list or tuple with the same size of boosting rounds.

after_iteration(*model*, *epoch*, *evals_log*)

Run after each iteration. Returns *True* when training should stop.

Parameters

- **model** (*Any*) – Eeither a *Booster* object or a CVPack if the cv function in xgboost is being used.
- **epoch** (*int*) – The current training iteration.
- **evals_log** (*Dict[str, Dict[str, List[float] | List[Tuple[float, float]]]]*) – A dictionary containing the evaluation history:

```
{"data_name": {"metric_name": [0.5, ...]}}
```

Return type
bool

class xgboost.callback.**TrainingCheckPoint**(*directory*, *name='model'*, *as_pickle=False*, *interval=100*)

Bases: *TrainingCallback*

Checkpointing operation. Users are encouraged to create their own callbacks for checkpoint as XGBoost doesn't handle distributed file systems. When checkpointing on distributed systems, be sure to know the rank of the worker to avoid multiple workers checkpointing to the same place.

Added in version 1.3.0.

Since XGBoost 2.1.0, the default format is changed to UBJSON.

Parameters

- **directory** (*str | PathLike*) – Output model directory.
- **name** (*str*) – pattern of output model file. Models will be saved as name_0.ubj, name_1.ubj, name_2.ubj ….
- **as_pickle** (*bool*) – When set to True, all training parameters will be saved in pickle format, instead of saving only the model.
- **interval** (*int*) – Interval of checkpointing. Checkpointing is slow so setting a larger number can reduce performance hit.

after_iteration(*model*, *epoch*, *evals_log*)

Run after each iteration. Returns *True* when training should stop.

Parameters

- **model** (*Any*) – Eeither a Booster object or a CVPack if the cv function in xgboost is being used.
- **epoch** (*int*) – The current training iteration.
- **evals_log** (*Dict[str, Dict[str, List[float] | List[Tuple[float, float]]]]*) – A dictionary containing the evaluation history:

```
{"data_name": {"metric_name": [0.5, ...]}}
```

Return type

bool

before_training(*model*)

Run before training starts.

Parameters

model (*Any*)

Return type

Any

Dask API

PySpark API

PySpark XGBoost integration interface

class xgboost.spark.**SparkXGBClassifier**(**, features_col='features', label_col='label', prediction_col='prediction', probability_col='probability', raw_prediction_col='rawPrediction', pred_contrib_col=None, validation_indicator_col=None, weight_col=None, base_margin_col=None, num_workers=1, device=None, force_repartition=False, repartition_random_shuffle=False, enable_sparse_data_optim=False, launch_tracker_on_driver=None, coll_cfg=None, **kwargs*)

Bases: `_SparkXGBEstimator`, `HasProbabilityCol`, `HasRawPredictionCol`

SparkXGBClassifier is a PySpark ML estimator. It implements the XGBoost classification algorithm based on XGBoost python library, and it can be used in PySpark Pipeline and PySpark ML meta algorithms like - `CrossValidator`/ - `TrainValidationSplit`/ - `OneVsRest`

SparkXGBClassifier automatically supports most of the parameters in `xgboost.XGBClassifier` constructor and most of the parameters used in `xgboost.XGBClassifier.fit()` and `xgboost.XGBClassifier.predict()` method.

To enable GPU support, set *device* to *cuda* or *gpu*.

SparkXGBClassifier doesn't support setting *base_margin* explicitly as well, but support another param called *base_margin_col*. see doc below for more details.

SparkXGBClassifier doesn't support setting *output_margin*, but we can get output margin from the raw prediction column. See *raw_prediction_col* param doc below for more details.

SparkXGBClassifier doesn't support *validate_features* and *output_margin* param.

SparkXGBClassifier doesn't support setting *nthread* xgboost param, instead, the *nthread* param for each xgboost worker will be set equal to *spark.task.cpus* config value.

Parameters

- **features_col** (*str | List[str]*) – When the value is string, it requires the features column name to be vector type. When the value is a list of string, it requires all the feature columns to be numeric types.
- **label_col** (*str*) – Label column name. Default to "label".
- **prediction_col** (*str*) – Prediction column name. Default to "prediction"
- **probability_col** (*str*) – Column name for predicted class conditional probabilities. Default to probabilityCol
- **raw_prediction_col** (*str*) – The *output_margin=True* is implicitly supported by the *rawPredictionCol* output column, which is always returned with the predicted margin values.
- **pred_contrib_col** (*pyspark.ml.param.Param[str]*) – Contribution prediction column name.
- **validation_indicator_col** (*str | None*) – For params related to *xgboost.XGBClassifier* training with evaluation dataset's supervision, set `xgboost.spark.SparkXGBClassifier.validation_indicator_col` parameter instead of setting the *eval_set* parameter in *xgboost.XGBClassifier* fit method.
- **weight_col** (*str | None*) – To specify the weight of the training and validation dataset, set `xgboost.spark.SparkXGBClassifier.weight_col` parameter instead of setting *sample_weight* and *sample_weight_eval_set* parameter in *xgboost.XGBClassifier* fit method.
- **base_margin_col** (*str | None*) – To specify the base margins of the training and validation dataset, set `xgboost.spark.SparkXGBClassifier.base_margin_col` parameter instead of setting *base_margin* and *base_margin_eval_set* in the *xgboost.XGBClassifier* fit method.
- **num_workers** (*int*) – How many XGBoost workers to be used to train. Each XGBoost worker corresponds to one spark task.
- **device** (*str | None*) – Added in version 2.0.0.

  Device for XGBoost workers, available options are *cpu*, *cuda*, and *gpu*.
- **force_repartition** (*bool*) – Boolean value to specify if forcing the input dataset to be repartitioned before XGBoost training.
- **repartition_random_shuffle** (*bool*) – Boolean value to specify if randomly shuffling the dataset when repartitioning is required.

- **enable_sparse_data_optim** (*bool*) – Boolean value to specify if enabling sparse data optimization, if True, Xgboost DMatrix object will be constructed from sparse matrix instead of dense matrix.
- **launch_tracker_on_driver** (*bool | None*) – Boolean value to indicate whether the tracker should be launched on the driver side or the executor side.
- **coll_cfg** (`Config` | *None*) – The collective configuration. See `Config`
- **kwargs** (*Any*) – A dictionary of xgboost parameters, please refer to https://xgboost.readthedocs.io/en/stable/parameter.html

Note

The Parameters chart above contains parameters that need special handling. For a full list of parameters, see entries with *Param(parent=…* below.

This API is experimental.

Examples

```
>>> from xgboost.spark import SparkXGBClassifier
>>> from pyspark.ml.linalg import Vectors
>>> df_train = spark.createDataFrame([
...     (Vectors.dense(1.0, 2.0, 3.0), 0, False, 1.0),
...     (Vectors.sparse(3, {1: 1.0, 2: 5.5}), 1, False, 2.0),
...     (Vectors.dense(4.0, 5.0, 6.0), 0, True, 1.0),
...     (Vectors.sparse(3, {1: 6.0, 2: 7.5}), 1, True, 2.0),
... ], ["features", "label", "isVal", "weight"])
>>> df_test = spark.createDataFrame([
...     (Vectors.dense(1.0, 2.0, 3.0), ),
... ], ["features"])
>>> xgb_classifier = SparkXGBClassifier(max_depth=5, missing=0.0,
...     validation_indicator_col='isVal', weight_col='weight',
...     early_stopping_rounds=1, eval_metric='logloss')
>>> xgb_clf_model = xgb_classifier.fit(df_train)
>>> xgb_clf_model.transform(df_test).show()
```

clear(*param*)

Clears a param from the param map if it has been explicitly set.

Parameters
param (*Param*)

Return type
None

copy(*extra=None*)

Creates a copy of this instance with the same uid and some extra params. The default implementation creates a shallow copy using `copy.copy()`, and then copies the embedded and extra parameters over and returns the copy. Subclasses should override this method if the default approach is not sufficient.

Parameters

- **extra** (*dict, optional*) – Extra parameters to copy to the new instance
- **self** (*P*)

Returns
Copy of this instance

Return type
Params

`explainParam`(*param*)

Explains a single param and returns its name, doc, and optional default value and user-supplied value in a string.

Parameters
param (*str | Param*)

Return type
str

`explainParams`()

Returns the documentation of all params with their optionally default values and user-supplied values.

Return type
str

`extractParamMap`(*extra=None*)

Extracts the embedded default param values and user-supplied values, and then merges them with extra values from input into a flat param map, where the latter value is used if there exist conflicts, i.e., with ordering: default param values < user-supplied values < extra.

Parameters
extra (*dict, optional*) – extra param values

Returns
merged param map

Return type
dict

`fit`(*dataset*, *params=None*)

Fits a model to the input dataset with optional parameters.

Added in version 1.3.0.

Parameters

- **dataset** (`pyspark.sql.DataFrame`) – input dataset.
- **params** (*dict or list or tuple, optional*) – an optional param map that overrides embedded params. If a list/tuple of param maps is given, this calls fit on each param map and returns a list of models.

Returns
fitted model(s)

Return type
`Transformer` or a list of `Transformer`

`fitMultiple`(*dataset*, *paramMaps*)

Fits a model to the input dataset for each param map in *paramMaps*.

Added in version 2.3.0.

Parameters

- **dataset** (`pyspark.sql.DataFrame`) – input dataset.

- **paramMaps** (`collections.abc.Sequence`) – A Sequence of param maps.

Returns
A thread safe iterable which contains one model for each param map. Each call to *next(modelIterator)* will return *(index, model)* where model was fit using *paramMaps[index]*. *index* values may not be sequential.

Return type
`_FitMultipleIterator`

`getFeaturesCol()`

Gets the value of featuresCol or its default value.

Return type
str

`getLabelCol()`

Gets the value of labelCol or its default value.

Return type
str

`getOrDefault(param)`

Gets the value of a param in the user-supplied param map or its default value. Raises an error if neither is set.

Parameters
param (*str* | *Param[T]*)

Return type
Any | *T*

`getParam(paramName)`

Gets a param by its name.

Parameters
paramName (*str*)

Return type
Param

`getPredictionCol()`

Gets the value of predictionCol or its default value.

Return type
str

`getProbabilityCol()`

Gets the value of probabilityCol or its default value.

Return type
str

`getRawPredictionCol()`

Gets the value of rawPredictionCol or its default value.

Return type
str

`getValidationIndicatorCol()`

Gets the value of validationIndicatorCol or its default value.

Return type
str

`getWeightCol()`

Gets the value of weightCol or its default value.

Return type
str

`hasDefault(param)`

Checks whether a param has a default value.

Parameters
param (*str | Param[Any]*)

Return type
bool

`hasParam(paramName)`

Tests whether this instance contains a param with a given (string) name.

Parameters
paramName (*str*)

Return type
bool

`isDefined(param)`

Checks whether a param is explicitly set by user or has a default value.

Parameters
param (*str | Param[Any]*)

Return type
bool

`isSet(param)`

Checks whether a param is explicitly set by user.

Parameters
param (*str | Param[Any]*)

Return type
bool

`classmethod load(path)`

Reads an ML instance from the input path, a shortcut of *read().load(path)*.

Parameters
path (*str*)

Return type
RL

`property params: List[Param]`

Returns all params ordered by name. The default implementation uses `dir()` to get all attributes of type `Param`.

`classmethod read()`

Return the reader for loading the estimator.

Return type
SparkXGBReader

`save(path)`
Save this ML instance to the given path, a shortcut of 'write().save(path)'.

Parameters
`path` (*str*)

Return type
None

`set(param, value)`
Sets a parameter in the embedded param map.

Parameters
- `param` (*Param*)
- `value` (*Any*)

Return type
None

`setParams(**kwargs)`
Set params for the estimator.

Parameters
`kwargs` (*Any*)

Return type
None

`set_coll_cfg(value)`
Set collective configuration

Parameters
`value` (`Config`)

Return type
_SparkXGBParams

`set_device(value)`
Set device, optional value: cpu, cuda, gpu

Parameters
`value` (*str*)

Return type
_SparkXGBParams

`uid`
A unique id for the object.

`write()`
Return the writer for saving the estimator.

Return type
SparkXGBWriter

class xgboost.spark.**SparkXGBClassifierModel**(*xgb_sklearn_model=None*, *training_summary=None*)

Bases: `_ClassificationModel`

The model returned by *xgboost.spark.SparkXGBClassifier.fit()*

Note

This API is experimental.

Parameters

- **xgb_sklearn_model** (*XGBModel | None*)
- **training_summary** (*XGBoostTrainingSummary | None*)

clear(*param*)

Clears a param from the param map if it has been explicitly set.

Parameters
param (*Param*)

Return type
None

copy(*extra=None*)

Creates a copy of this instance with the same uid and some extra params. The default implementation creates a shallow copy using `copy.copy()`, and then copies the embedded and extra parameters over and returns the copy. Subclasses should override this method if the default approach is not sufficient.

Parameters

- **extra** (*dict, optional*) – Extra parameters to copy to the new instance
- **self** (*P*)

Returns
Copy of this instance

Return type
`Params`

explainParam(*param*)

Explains a single param and returns its name, doc, and optional default value and user-supplied value in a string.

Parameters
param (*str | Param*)

Return type
str

explainParams()

Returns the documentation of all params with their optionally default values and user-supplied values.

Return type
str

extractParamMap(*extra=None*)

Extracts the embedded default param values and user-supplied values, and then merges them with extra values from input into a flat param map, where the latter value is used if there exist conflicts, i.e., with ordering: default param values < user-supplied values < extra.

Parameters
extra (*dict, optional*) – extra param values

Returns
merged param map

Return type
dict

getFeaturesCol()

Gets the value of featuresCol or its default value.

Return type
str

getLabelCol()

Gets the value of labelCol or its default value.

Return type
str

getOrDefault(*param*)

Gets the value of a param in the user-supplied param map or its default value. Raises an error if neither is set.

Parameters
param (*str | Param[T]*)

Return type
Any | *T*

getParam(*paramName*)

Gets a param by its name.

Parameters
paramName (*str*)

Return type
Param

getPredictionCol()

Gets the value of predictionCol or its default value.

Return type
str

getProbabilityCol()

Gets the value of probabilityCol or its default value.

Return type
str

getRawPredictionCol()

Gets the value of rawPredictionCol or its default value.

Return type
str

getValidationIndicatorCol()

Gets the value of validationIndicatorCol or its default value.

Return type
str

getWeightCol()

Gets the value of weightCol or its default value.

Return type
str

get_booster()

Return the *xgboost.core.Booster* instance.

Return type
Booster

get_feature_importances(*importance_type='weight'*)

Get feature importance of each feature. Importance type can be defined as:

- 'weight': the number of times a feature is used to split the data across all trees.
- 'gain': the average gain across all splits the feature is used in.
- 'cover': the average coverage across all splits the feature is used in.
- 'total_gain': the total gain across all splits the feature is used in.
- 'total_cover': the total coverage across all splits the feature is used in.

Parameters
importance_type (*str, default 'weight'*) – One of the importance types defined above.

Return type
Dict[str, float | *List*[float]]

hasDefault(*param*)

Checks whether a param has a default value.

Parameters
param (*str | Param[Any]*)

Return type
bool

hasParam(*paramName*)

Tests whether this instance contains a param with a given (string) name.

Parameters
paramName (*str*)

Return type
bool

isDefined(*param*)

Checks whether a param is explicitly set by user or has a default value.

Parameters
param (*str | Param[Any]*)

Return type
bool

`isSet`(*param*)

Checks whether a param is explicitly set by user.

Parameters
param (*str | Param[Any]*)

Return type
bool

`classmethod load`(*path*)

Reads an ML instance from the input path, a shortcut of *read().load(path)*.

Parameters
path (*str*)

Return type
RL

`property params: List[Param]`

Returns all params ordered by name. The default implementation uses `dir()` to get all attributes of type `Param`.

`classmethod read`()

Return the reader for loading the model.

Return type
SparkXGBModelReader

`save`(*path*)

Save this ML instance to the given path, a shortcut of 'write().save(path)'.

Parameters
path (*str*)

Return type
None

`set`(*param*, *value*)

Sets a parameter in the embedded param map.

Parameters

- **param** (*Param*)
- **value** (*Any*)

Return type
None

`set_coll_cfg`(*value*)

Set collective configuration

Parameters
value (`Config`)

Return type
_SparkXGBParams

`set_device`(*value*)

Set device, optional value: cpu, cuda, gpu

Parameters
`value` (*str*)

Return type
_SparkXGBParams

`transform`(*dataset*, *params=None*)

Transforms the input dataset with optional parameters.

Added in version 1.3.0.

Parameters

- `dataset` (`pyspark.sql.DataFrame`) – input dataset
- `params` (*dict, optional*) – an optional param map that overrides embedded params.

Returns
transformed dataset

Return type
`pyspark.sql.DataFrame`

`uid`

A unique id for the object.

`write`()

Return the writer for saving the model.

Return type
SparkXGBModelWriter

class xgboost.spark.**SparkXGBRegressor**(*, *features_col='features'*, *label_col='label'*, *prediction_col='prediction'*, *pred_contrib_col=None*, *validation_indicator_col=None*, *weight_col=None*, *base_margin_col=None*, *num_workers=1*, *device=None*, *force_repartition=False*, *repartition_random_shuffle=False*, *enable_sparse_data_optim=False*, *launch_tracker_on_driver=None*, *coll_cfg=None*, ***kwargs*)

Bases: `_SparkXGBEstimator`

SparkXGBRegressor is a PySpark ML estimator. It implements the XGBoost regression algorithm based on XGBoost python library, and it can be used in PySpark Pipeline and PySpark ML meta algorithms like - `CrossValidator`/ - `TrainValidationSplit`/ - `OneVsRest`

SparkXGBRegressor automatically supports most of the parameters in `xgboost.XGBRegressor` constructor and most of the parameters used in `xgboost.XGBRegressor.fit()` and `xgboost.XGBRegressor.predict()` method.

To enable GPU support, set *device* to *cuda* or *gpu*.

SparkXGBRegressor doesn't support setting *base_margin* explicitly as well, but support another param called *base_margin_col*. see doc below for more details.

SparkXGBRegressor doesn't support *validate_features* and *output_margin* param.

SparkXGBRegressor doesn't support setting *nthread* xgboost param, instead, the *nthread* param for each xgboost worker will be set equal to *spark.task.cpus* config value.

Parameters

- **features_col** (*str* | *List[str]*) – When the value is string, it requires the features column name to be vector type. When the value is a list of string, it requires all the feature columns to be numeric types.
- **label_col** (*str*) – Label column name. Default to "label".
- **prediction_col** (*str*) – Prediction column name. Default to "prediction"
- **pred_contrib_col** (*str* | *None*) – Contribution prediction column name.
- **validation_indicator_col** (*str* | *None*) – For params related to *xgboost.XGBRegressor* training with evaluation dataset's supervision, set `xgboost.spark.SparkXGBRegressor.validation_indicator_col` parameter instead of setting the *eval_set* parameter in *xgboost.XGBRegressor* fit method.
- **weight_col** (*str* | *None*) – To specify the weight of the training and validation dataset, set `xgboost.spark.SparkXGBRegressor.weight_col` parameter instead of setting *sample_weight* and *sample_weight_eval_set* parameter in *xgboost.XGBRegressor* fit method.
- **base_margin_col** (*str* | *None*) – To specify the base margins of the training and validation dataset, set `xgboost.spark.SparkXGBRegressor.base_margin_col` parameter instead of setting *base_margin* and *base_margin_eval_set* in the *xgboost.XGBRegressor* fit method.
- **num_workers** (*int*) – How many XGBoost workers to be used to train. Each XGBoost worker corresponds to one spark task.
- **device** (*str* | *None*) – Added in version 2.0.0.

  Device for XGBoost workers, available options are *cpu*, *cuda*, and *gpu*.
- **force_repartition** (*bool*) – Boolean value to specify if forcing the input dataset to be repartitioned before XGBoost training.
- **repartition_random_shuffle** (*bool*) – Boolean value to specify if randomly shuffling the dataset when repartitioning is required.
- **enable_sparse_data_optim** (*bool*) – Boolean value to specify if enabling sparse data optimization, if True, Xgboost DMatrix object will be constructed from sparse matrix instead of dense matrix.
- **launch_tracker_on_driver** (*bool* | *None*) – Boolean value to indicate whether the tracker should be launched on the driver side or the executor side.
- **coll_cfg** (`Config` | *None*) – The collective configuration. See `Config`
- **kwargs** (*Any*) – A dictionary of xgboost parameters, please refer to https://xgboost.readthedocs.io/en/stable/parameter.html

> **Note**
>
> The Parameters chart above contains parameters that need special handling. For a full list of parameters, see entries with *Param(parent=...* below.
>
> This API is experimental.

Examples

```
>>> from xgboost.spark import SparkXGBRegressor
>>> from pyspark.ml.linalg import Vectors
>>> df_train = spark.createDataFrame([
...     (Vectors.dense(1.0, 2.0, 3.0), 0, False, 1.0),
...     (Vectors.sparse(3, {1: 1.0, 2: 5.5}), 1, False, 2.0),
...     (Vectors.dense(4.0, 5.0, 6.0), 2, True, 1.0),
...     (Vectors.sparse(3, {1: 6.0, 2: 7.5}), 3, True, 2.0),
... ], ["features", "label", "isVal", "weight"])
>>> df_test = spark.createDataFrame([
...     (Vectors.dense(1.0, 2.0, 3.0), ),
...     (Vectors.sparse(3, {1: 1.0, 2: 5.5}), )
... ], ["features"])
>>> xgb_regressor = SparkXGBRegressor(max_depth=5, missing=0.0,
... validation_indicator_col='isVal', weight_col='weight',
... early_stopping_rounds=1, eval_metric='rmse')
>>> xgb_reg_model = xgb_regressor.fit(df_train)
>>> xgb_reg_model.transform(df_test)
```

clear(*param*)

Clears a param from the param map if it has been explicitly set.

Parameters
param (*Param*)

Return type
None

copy(*extra=None*)

Creates a copy of this instance with the same uid and some extra params. The default implementation creates a shallow copy using `copy.copy()`, and then copies the embedded and extra parameters over and returns the copy. Subclasses should override this method if the default approach is not sufficient.

Parameters

- **extra** (*dict, optional*) – Extra parameters to copy to the new instance
- **self** (*P*)

Returns
Copy of this instance

Return type
`Params`

explainParam(*param*)

Explains a single param and returns its name, doc, and optional default value and user-supplied value in a string.

Parameters
`param` (*str* | *Param*)

Return type
str

`explainParams()`

Returns the documentation of all params with their optionally default values and user-supplied values.

Return type
str

`extractParamMap`(*extra=None*)

Extracts the embedded default param values and user-supplied values, and then merges them with extra values from input into a flat param map, where the latter value is used if there exist conflicts, i.e., with ordering: default param values < user-supplied values < extra.

Parameters
`extra` (*dict*, *optional*) – extra param values

Returns
merged param map

Return type
dict

`fit`(*dataset*, *params=None*)

Fits a model to the input dataset with optional parameters.

Added in version 1.3.0.

Parameters

- `dataset` (`pyspark.sql.DataFrame`) – input dataset.
- `params` (*dict or list or tuple*, *optional*) – an optional param map that overrides embedded params. If a list/tuple of param maps is given, this calls fit on each param map and returns a list of models.

Returns
fitted model(s)

Return type
`Transformer` or a list of `Transformer`

`fitMultiple`(*dataset*, *paramMaps*)

Fits a model to the input dataset for each param map in *paramMaps*.

Added in version 2.3.0.

Parameters

- `dataset` (`pyspark.sql.DataFrame`) – input dataset.
- `paramMaps` (`collections.abc.Sequence`) – A Sequence of param maps.

Returns
A thread safe iterable which contains one model for each param map. Each call to *next(modelIterator)* will return *(index, model)* where model was fit using *paramMaps[index]*. *index* values may not be sequential.

Return type
`_FitMultipleIterator`

getFeaturesCol()

Gets the value of featuresCol or its default value.

Return type
str

getLabelCol()

Gets the value of labelCol or its default value.

Return type
str

getOrDefault(*param*)

Gets the value of a param in the user-supplied param map or its default value. Raises an error if neither is set.

Parameters
param (*str | Param[T]*)

Return type
Any | *T*

getParam(*paramName*)

Gets a param by its name.

Parameters
paramName (*str*)

Return type
Param

getPredictionCol()

Gets the value of predictionCol or its default value.

Return type
str

getValidationIndicatorCol()

Gets the value of validationIndicatorCol or its default value.

Return type
str

getWeightCol()

Gets the value of weightCol or its default value.

Return type
str

hasDefault(*param*)

Checks whether a param has a default value.

Parameters
param (*str | Param[Any]*)

Return type
bool

hasParam(*paramName*)

Tests whether this instance contains a param with a given (string) name.

Parameters
: **paramName** (*str*)

Return type
: bool

`isDefined`(*param*)
: Checks whether a param is explicitly set by user or has a default value.

Parameters
: **param** (*str | Param[Any]*)

Return type
: bool

`isSet`(*param*)
: Checks whether a param is explicitly set by user.

Parameters
: **param** (*str | Param[Any]*)

Return type
: bool

`classmethod load`(*path*)
: Reads an ML instance from the input path, a shortcut of *read().load(path)*.

Parameters
: **path** (*str*)

Return type
: *RL*

`property params: List[Param]`
: Returns all params ordered by name. The default implementation uses `dir()` to get all attributes of type `Param`.

`classmethod read`()
: Return the reader for loading the estimator.

Return type
: *SparkXGBReader*

`save`(*path*)
: Save this ML instance to the given path, a shortcut of 'write().save(path)'.

Parameters
: **path** (*str*)

Return type
: None

`set`(*param*, *value*)
: Sets a parameter in the embedded param map.

Parameters

- **param** (*Param*)
- **value** (*Any*)

Return type
: None

setParams(***kwargs*)

Set params for the estimator.

Parameters
kwargs (*Any*)

Return type
None

set_coll_cfg(*value*)

Set collective configuration

Parameters
value (Config)

Return type
_SparkXGBParams

set_device(*value*)

Set device, optional value: cpu, cuda, gpu

Parameters
value (*str*)

Return type
_SparkXGBParams

uid

A unique id for the object.

write()

Return the writer for saving the estimator.

Return type
SparkXGBWriter

class xgboost.spark.**SparkXGBRegressorModel**(*xgb_sklearn_model=None*, *training_summary=None*)

Bases: `_SparkXGBModel`

The model returned by *xgboost.spark.SparkXGBRegressor.fit()*

> **Note**
>
> This API is experimental.

Parameters

- **xgb_sklearn_model** (*XGBModel | None*)
- **training_summary** (*XGBoostTrainingSummary | None*)

clear(*param*)

Clears a param from the param map if it has been explicitly set.

Parameters
param (*Param*)

Return type
None

copy(*extra=None*)

Creates a copy of this instance with the same uid and some extra params. The default implementation creates a shallow copy using `copy.copy()`, and then copies the embedded and extra parameters over and returns the copy. Subclasses should override this method if the default approach is not sufficient.

Parameters

- **extra** (*dict, optional*) – Extra parameters to copy to the new instance
- **self** (*P*)

Returns
Copy of this instance

Return type
Params

explainParam(*param*)

Explains a single param and returns its name, doc, and optional default value and user-supplied value in a string.

Parameters
param (*str | Param*)

Return type
str

explainParams()

Returns the documentation of all params with their optionally default values and user-supplied values.

Return type
str

extractParamMap(*extra=None*)

Extracts the embedded default param values and user-supplied values, and then merges them with extra values from input into a flat param map, where the latter value is used if there exist conflicts, i.e., with ordering: default param values < user-supplied values < extra.

Parameters
extra (*dict, optional*) – extra param values

Returns
merged param map

Return type
dict

getFeaturesCol()

Gets the value of featuresCol or its default value.

Return type
str

getLabelCol()

Gets the value of labelCol or its default value.

Return type
str

getOrDefault(*param*)

Gets the value of a param in the user-supplied param map or its default value. Raises an error if neither is set.

Parameters
param (*str | Param[T]*)

Return type
Any | *T*

getParam(*paramName*)

Gets a param by its name.

Parameters
paramName (*str*)

Return type
Param

getPredictionCol()

Gets the value of predictionCol or its default value.

Return type
str

getValidationIndicatorCol()

Gets the value of validationIndicatorCol or its default value.

Return type
str

getWeightCol()

Gets the value of weightCol or its default value.

Return type
str

get_booster()

Return the *xgboost.core.Booster* instance.

Return type
Booster

get_feature_importances(*importance_type='weight'*)

Get feature importance of each feature. Importance type can be defined as:

- 'weight': the number of times a feature is used to split the data across all trees.
- 'gain': the average gain across all splits the feature is used in.
- 'cover': the average coverage across all splits the feature is used in.
- 'total_gain': the total gain across all splits the feature is used in.
- 'total_cover': the total coverage across all splits the feature is used in.

Parameters
importance_type (*str, default 'weight'*) – One of the importance types defined above.

Return type
Dict[str, float | *List*[float]]

hasDefault(*param*)

Checks whether a param has a default value.

Parameters
: **param** (*str | Param[Any]*)

Return type
: bool

hasParam(*paramName*)

Tests whether this instance contains a param with a given (string) name.

Parameters
: **paramName** (*str*)

Return type
: bool

isDefined(*param*)

Checks whether a param is explicitly set by user or has a default value.

Parameters
: **param** (*str | Param[Any]*)

Return type
: bool

isSet(*param*)

Checks whether a param is explicitly set by user.

Parameters
: **param** (*str | Param[Any]*)

Return type
: bool

classmethod load(*path*)

Reads an ML instance from the input path, a shortcut of *read().load(path)*.

Parameters
: **path** (*str*)

Return type
: *RL*

property params: List[Param]

Returns all params ordered by name. The default implementation uses `dir()` to get all attributes of type `Param`.

classmethod read()

Return the reader for loading the model.

Return type
: *SparkXGBModelReader*

save(*path*)

Save this ML instance to the given path, a shortcut of 'write().save(path)'.

Parameters
: **path** (*str*)

Return type
None

`set`(*param*, *value*)

Sets a parameter in the embedded param map.

Parameters

- **param** (*Param*)
- **value** (*Any*)

Return type
None

`set_coll_cfg`(*value*)

Set collective configuration

Parameters
value (Config)

Return type
_SparkXGBParams

`set_device`(*value*)

Set device, optional value: cpu, cuda, gpu

Parameters
value (*str*)

Return type
_SparkXGBParams

`transform`(*dataset*, *params=None*)

Transforms the input dataset with optional parameters.

Added in version 1.3.0.

Parameters

- **dataset** (pyspark.sql.DataFrame) – input dataset
- **params** (*dict, optional*) – an optional param map that overrides embedded params.

Returns
transformed dataset

Return type
pyspark.sql.DataFrame

`uid`

A unique id for the object.

`write`()

Return the writer for saving the model.

Return type
SparkXGBModelWriter

class xgboost.spark.**SparkXGBRanker**(*, *features_col='features'*, *label_col='label'*, *prediction_col='prediction'*, *pred_contrib_col=None*, *validation_indicator_col=None*, *weight_col=None*, *base_margin_col=None*, *qid_col=None*, *num_workers=1*, *device=None*, *force_repartition=False*, *repartition_random_shuffle=False*, *enable_sparse_data_optim=False*, *launch_tracker_on_driver=None*, *coll_cfg=None*, ***kwargs*)

Bases: `_SparkXGBEstimator`

SparkXGBRanker is a PySpark ML estimator. It implements the XGBoost ranking algorithm based on XGBoost python library, and it can be used in PySpark Pipeline and PySpark ML meta algorithms like `CrossValidator`/ `TrainValidationSplit`/ `OneVsRest`

SparkXGBRanker automatically supports most of the parameters in `xgboost.XGBRanker` constructor and most of the parameters used in `xgboost.XGBRanker.fit()` and `xgboost.XGBRanker.predict()` method.

To enable GPU support, set *device* to *cuda* or *gpu*.

SparkXGBRanker doesn't support setting *base_margin* explicitly as well, but support another param called *base_margin_col*. see doc below for more details.

SparkXGBRanker doesn't support setting *output_margin*, but we can get output margin from the raw prediction column. See *raw_prediction_col* param doc below for more details.

SparkXGBRanker doesn't support *validate_features* and *output_margin* param.

SparkXGBRanker doesn't support setting *nthread* xgboost param, instead, the *nthread* param for each xgboost worker will be set equal to *spark.task.cpus* config value.

Parameters

- **features_col** (*str | List[str]*) – When the value is string, it requires the features column name to be vector type. When the value is a list of string, it requires all the feature columns to be numeric types.
- **label_col** (*str*) – Label column name. Default to "label".
- **prediction_col** (*str*) – Prediction column name. Default to "prediction"
- **pred_contrib_col** (*str | None*) – Contribution prediction column name.
- **validation_indicator_col** (*str | None*) – For params related to *xgboost.XGBRanker* training with evaluation dataset's supervision, set `xgboost.spark.SparkXGBRanker.validation_indicator_col` parameter instead of setting the *eval_set* parameter in `xgboost.XGBRanker` fit method.
- **weight_col** (*str | None*) – To specify the weight of the training and validation dataset, set `xgboost.spark.SparkXGBRanker.weight_col` parameter instead of setting *sample_weight* and *sample_weight_eval_set* parameter in `xgboost.XGBRanker` fit method.
- **base_margin_col** (*str | None*) – To specify the base margins of the training and validation dataset, set `xgboost.spark.SparkXGBRanker.base_margin_col` parameter instead of setting *base_margin* and *base_margin_eval_set* in the `xgboost.XGBRanker` fit method.
- **qid_col** (*str | None*) – Query id column name.
- **num_workers** (*int*) – How many XGBoost workers to be used to train. Each XGBoost worker corresponds to one spark task.
- **device** (*str | None*) – Added in version 2.0.0.

  Device for XGBoost workers, available options are *cpu*, *cuda*, and *gpu*.

- **force_repartition** (*bool*) – Boolean value to specify if forcing the input dataset to be repartitioned before XGBoost training.
- **repartition_random_shuffle** (*bool*) – Boolean value to specify if randomly shuffling the dataset when repartitioning is required.
- **enable_sparse_data_optim** (*bool*) – Boolean value to specify if enabling sparse data optimization, if True, Xgboost DMatrix object will be constructed from sparse matrix instead of dense matrix.
- **launch_tracker_on_driver** (*bool | None*) – Boolean value to indicate whether the tracker should be launched on the driver side or the executor side.
- **coll_cfg** (Config | *None*) – The collective configuration. See *Config*
- **kwargs** (*Any*) – A dictionary of xgboost parameters, please refer to https://xgboost.readthedocs.io/en/stable/parameter.html
- **Note:** (*..*) – The Parameters chart above contains parameters that need special handling.: For a full list of parameters, see entries with *Param(parent=…* below.
- **Note:** – This API is experimental.:

Examples

```
>>> from xgboost.spark import SparkXGBRanker
>>> from pyspark.ml.linalg import Vectors
>>> ranker = SparkXGBRanker(qid_col="qid")
>>> df_train = spark.createDataFrame(
...     [
...         (Vectors.dense(1.0, 2.0, 3.0), 0, 0),
...         (Vectors.dense(4.0, 5.0, 6.0), 1, 0),
...         (Vectors.dense(9.0, 4.0, 8.0), 2, 0),
...         (Vectors.sparse(3, {1: 1.0, 2: 5.5}), 0, 1),
...         (Vectors.sparse(3, {1: 6.0, 2: 7.5}), 1, 1),
...         (Vectors.sparse(3, {1: 8.0, 2: 9.5}), 2, 1),
...     ],
...     ["features", "label", "qid"],
... )
>>> df_test = spark.createDataFrame(
...     [
...         (Vectors.dense(1.5, 2.0, 3.0), 0),
...         (Vectors.dense(4.5, 5.0, 6.0), 0),
...         (Vectors.dense(9.0, 4.5, 8.0), 0),
...         (Vectors.sparse(3, {1: 1.0, 2: 6.0}), 1),
...         (Vectors.sparse(3, {1: 6.0, 2: 7.0}), 1),
...         (Vectors.sparse(3, {1: 8.0, 2: 10.5}), 1),
...     ],
...     ["features", "qid"],
... )
>>> model = ranker.fit(df_train)
>>> model.transform(df_test).show()
```

clear(*param*)

Clears a param from the param map if it has been explicitly set.

Parameters

param (*Param*)

Return type
None

`copy`(*extra=None*)

Creates a copy of this instance with the same uid and some extra params. The default implementation creates a shallow copy using `copy.copy()`, and then copies the embedded and extra parameters over and returns the copy. Subclasses should override this method if the default approach is not sufficient.

Parameters

- **extra** (`dict, optional`) – Extra parameters to copy to the new instance
- **self** (*P*)

Returns
Copy of this instance

Return type
`Params`

`explainParam`(*param*)

Explains a single param and returns its name, doc, and optional default value and user-supplied value in a string.

Parameters
param (`str | Param`)

Return type
str

`explainParams`()

Returns the documentation of all params with their optionally default values and user-supplied values.

Return type
str

`extractParamMap`(*extra=None*)

Extracts the embedded default param values and user-supplied values, and then merges them with extra values from input into a flat param map, where the latter value is used if there exist conflicts, i.e., with ordering: default param values < user-supplied values < extra.

Parameters
extra (`dict, optional`) – extra param values

Returns
merged param map

Return type
dict

`fit`(*dataset*, *params=None*)

Fits a model to the input dataset with optional parameters.

Added in version 1.3.0.

Parameters

- **dataset** (`pyspark.sql.DataFrame`) – input dataset.
- **params** (`dict or list or tuple, optional`) – an optional param map that overrides embedded params. If a list/tuple of param maps is given, this calls fit on each param map and returns a list of models.

Returns
fitted model(s)

Return type
`Transformer` or a list of `Transformer`

fitMultiple(*dataset*, *paramMaps*)

Fits a model to the input dataset for each param map in *paramMaps*.

Added in version 2.3.0.

Parameters

- **dataset** (`pyspark.sql.DataFrame`) – input dataset.
- **paramMaps** (`collections.abc.Sequence`) – A Sequence of param maps.

Returns
A thread safe iterable which contains one model for each param map. Each call to *next(modelIterator)* will return *(index, model)* where model was fit using *paramMaps[index]*. *index* values may not be sequential.

Return type
`_FitMultipleIterator`

getFeaturesCol()

Gets the value of featuresCol or its default value.

Return type
str

getLabelCol()

Gets the value of labelCol or its default value.

Return type
str

getOrDefault(*param*)

Gets the value of a param in the user-supplied param map or its default value. Raises an error if neither is set.

Parameters
param (*str* | *Param*[*T*])

Return type
Any | *T*

getParam(*paramName*)

Gets a param by its name.

Parameters
paramName (*str*)

Return type
Param

getPredictionCol()

Gets the value of predictionCol or its default value.

Return type
str

getValidationIndicatorCol()

Gets the value of validationIndicatorCol or its default value.

Return type
str

getWeightCol()

Gets the value of weightCol or its default value.

Return type
str

hasDefault(*param*)

Checks whether a param has a default value.

Parameters
param (*str | Param[Any]*)

Return type
bool

hasParam(*paramName*)

Tests whether this instance contains a param with a given (string) name.

Parameters
paramName (*str*)

Return type
bool

isDefined(*param*)

Checks whether a param is explicitly set by user or has a default value.

Parameters
param (*str | Param[Any]*)

Return type
bool

isSet(*param*)

Checks whether a param is explicitly set by user.

Parameters
param (*str | Param[Any]*)

Return type
bool

classmethod load(*path*)

Reads an ML instance from the input path, a shortcut of *read().load(path)*.

Parameters
path (*str*)

Return type
RL

property params: List[Param]

Returns all params ordered by name. The default implementation uses `dir()` to get all attributes of type `Param`.

classmethod read()

Return the reader for loading the estimator.

Return type
SparkXGBReader

save(*path*)

Save this ML instance to the given path, a shortcut of 'write().save(path)'.

Parameters
path (*str*)

Return type
None

set(*param*, *value*)

Sets a parameter in the embedded param map.

Parameters

- **param** (*Param*)
- **value** (*Any*)

Return type
None

setParams(***kwargs*)

Set params for the estimator.

Parameters
kwargs (*Any*)

Return type
None

set_coll_cfg(*value*)

Set collective configuration

Parameters
value (Config)

Return type
_SparkXGBParams

set_device(*value*)

Set device, optional value: cpu, cuda, gpu

Parameters
value (*str*)

Return type
_SparkXGBParams

uid

A unique id for the object.

write()

Return the writer for saving the estimator.

Return type
SparkXGBWriter

class xgboost.spark.**SparkXGBRankerModel**(*xgb_sklearn_model=None*, *training_summary=None*)

Bases: `_SparkXGBModel`

The model returned by `xgboost.spark.SparkXGBRanker.fit()`

> **Note**
>
> This API is experimental.

Parameters

- **xgb_sklearn_model** (*XGBModel | None*)
- **training_summary** (*XGBoostTrainingSummary | None*)

clear(*param*)

Clears a param from the param map if it has been explicitly set.

Parameters
param (*Param*)

Return type
None

copy(*extra=None*)

Creates a copy of this instance with the same uid and some extra params. The default implementation creates a shallow copy using `copy.copy()`, and then copies the embedded and extra parameters over and returns the copy. Subclasses should override this method if the default approach is not sufficient.

Parameters

- **extra** (*dict, optional*) – Extra parameters to copy to the new instance
- **self** (*P*)

Returns
Copy of this instance

Return type
`Params`

explainParam(*param*)

Explains a single param and returns its name, doc, and optional default value and user-supplied value in a string.

Parameters
param (*str | Param*)

Return type
str

explainParams()

Returns the documentation of all params with their optionally default values and user-supplied values.

Return type
str

`extractParamMap`(*extra=None*)

Extracts the embedded default param values and user-supplied values, and then merges them with extra values from input into a flat param map, where the latter value is used if there exist conflicts, i.e., with ordering: default param values < user-supplied values < extra.

Parameters
`extra` (*dict, optional*) – extra param values

Returns
merged param map

Return type
dict

`getFeaturesCol`()

Gets the value of featuresCol or its default value.

Return type
str

`getLabelCol`()

Gets the value of labelCol or its default value.

Return type
str

`getOrDefault`(*param*)

Gets the value of a param in the user-supplied param map or its default value. Raises an error if neither is set.

Parameters
`param` (*str | Param[T]*)

Return type
Any | *T*

`getParam`(*paramName*)

Gets a param by its name.

Parameters
`paramName` (*str*)

Return type
Param

`getPredictionCol`()

Gets the value of predictionCol or its default value.

Return type
str

`getValidationIndicatorCol`()

Gets the value of validationIndicatorCol or its default value.

Return type
str

`getWeightCol`()

Gets the value of weightCol or its default value.

Return type
str

get_booster()

Return the *xgboost.core.Booster* instance.

Return type
Booster

get_feature_importances(*importance_type='weight'*)

Get feature importance of each feature. Importance type can be defined as:

- 'weight': the number of times a feature is used to split the data across all trees.
- 'gain': the average gain across all splits the feature is used in.
- 'cover': the average coverage across all splits the feature is used in.
- 'total_gain': the total gain across all splits the feature is used in.
- 'total_cover': the total coverage across all splits the feature is used in.

Parameters
importance_type (*str, default 'weight'*) – One of the importance types defined above.

Return type
Dict[str, float | *List*[float]]

hasDefault(*param*)

Checks whether a param has a default value.

Parameters
param (*str | Param[Any]*)

Return type
bool

hasParam(*paramName*)

Tests whether this instance contains a param with a given (string) name.

Parameters
paramName (*str*)

Return type
bool

isDefined(*param*)

Checks whether a param is explicitly set by user or has a default value.

Parameters
param (*str | Param[Any]*)

Return type
bool

isSet(*param*)

Checks whether a param is explicitly set by user.

Parameters
param (*str | Param[Any]*)

Return type
bool

classmethod load(*path*)

Reads an ML instance from the input path, a shortcut of *read().load(path)*.

Parameters
path (*str*)

Return type
RL

property params: List[Param]

Returns all params ordered by name. The default implementation uses `dir()` to get all attributes of type `Param`.

classmethod read()

Return the reader for loading the model.

Return type
SparkXGBModelReader

save(*path*)

Save this ML instance to the given path, a shortcut of 'write().save(path)'.

Parameters
path (*str*)

Return type
None

set(*param*, *value*)

Sets a parameter in the embedded param map.

Parameters

- **param** (*Param*)
- **value** (*Any*)

Return type
None

set_coll_cfg(*value*)

Set collective configuration

Parameters
value (Config)

Return type
_SparkXGBParams

set_device(*value*)

Set device, optional value: cpu, cuda, gpu

Parameters
value (*str*)

Return type
_SparkXGBParams

transform(*dataset*, *params=None*)

Transforms the input dataset with optional parameters.

Added in version 1.3.0.

Parameters

- **dataset** (pyspark.sql.DataFrame) – input dataset
- **params** (*dict, optional*) – an optional param map that overrides embedded params.

Returns
transformed dataset

Return type
pyspark.sql.DataFrame

uid

A unique id for the object.

write()

Return the writer for saving the model.

Return type
SparkXGBModelWriter

Collective

XGBoost collective communication related API.

class xgboost.collective.**Config**(*retry=None*, *timeout=None*, *tracker_host_ip=None*, *tracker_port=None*, *tracker_timeout=None*, *worker_port=None*)

User configuration for the communicator context. This is used for easier integration with distributed frameworks. Users of the collective module can pass the parameters directly into tracker and the communicator.

Added in version 3.0.

Parameters

- **retry** (*int | None*)
- **timeout** (*int | None*)
- **tracker_host_ip** (*str | None*)
- **tracker_port** (*int | None*)
- **tracker_timeout** (*int | None*)
- **worker_port** (*Callable[[], int] | int | None*)

retry

Type
See *dmlc_retry* in `init()`.

timeout

See *dmlc_timeout* in `init()`. This is only used for communicators, not the tracker. They are different parameters since the timeout for tracker limits only the time for starting and finalizing the communication group, whereas the timeout for communicators limits the time used for collective operations, like `allreduce()`.

Type
int | None

tracker_host_ip

Type
See *RabitTracker*.

tracker_port

Type
See *RabitTracker*.

tracker_timeout

Type
See *RabitTracker*.

worker_port

The port each worker listens to for peer-to-peer connections. By default, workers use an available port assigned by the OS. This option can be used in restricted network environments where only specific ports are open.

This can be an integer for a fixed port used by all workers, or a callback function that takes no arguments and returns a port number. The callback is invoked per-worker at the worker side.

Note

The option does not affect the NCCL communicator group, which must be configured via NCCL's own environment variables.

Type
Callable[[], int] | int | None

`xgboost.collective.init(**args)`

Initialize the collective library with arguments.

Parameters
args (*int | str | None*) – Keyword arguments representing the parameters and their values.

Accepted parameters:

- dmlc_communicator: The type of the communicator. * rabit: Use Rabit. This is the default if the type is unspecified. * federated: Use the gRPC interface for Federated Learning.

Only applicable to the Rabit communicator:

- dmlc_tracker_uri: Hostname of the tracker.
- dmlc_tracker_port: Port number of the tracker.
- dmlc_task_id: ID of the current task, can be used to obtain deterministic
- dmlc_retry: The number of retry when handling network errors.
- dmlc_timeout: Timeout in seconds.
- dmlc_nccl_path: Path to load (dlopen) nccl for GPU-based communication.

Only applicable to the Federated communicator:

- federated_server_address: Address of the federated server.
- federated_world_size: Number of federated workers.

- federated_rank: Rank of the current worker.
- federated_server_cert: Server certificate file path. Only needed for the SSL mode.
- federated_client_key: Client key file path. Only needed for the SSL mode.
- federated_client_cert: Client certificate file path. Only needed for the SSL mode.

Use upper case for environment variables, use lower case for runtime configuration.

Return type
None

`xgboost.collective.finalize()`

Finalize the communicator.

Return type
None

`xgboost.collective.get_rank()`

Get rank of current process.

Returns
rank – Rank of current process.

Return type
int

`xgboost.collective.get_world_size()`

Get total number workers.

Returns
Total number of process.

Return type
n

`class xgboost.collective.CommunicatorContext(**args)`

A context controlling collective communicator initialization and finalization.

Parameters
args (*int | str | None*)

Tracker for XGBoost collective.

`class xgboost.tracker.RabitTracker(n_workers, host_ip, port=0, *, sortby='host', timeout=0)`

Tracker for the collective used in XGBoost, acting as a coordinator between workers.

Parameters

- **n_workers** (*int*) – The total number of workers in the communication group.
- **host_ip** (*str | None*) – The IP address of the tracker node. XGBoost can try to guess one by probing with sockets. But it's best to explicitly pass an address.
- **port** (*int*) – The port this tracker should listen to. XGBoost can query an available port from the OS, this configuration is useful for restricted network environments.
- **sortby** (*str*) – How to sort the workers for rank assignment. The default is host, but users can set the *DMLC_TASK_ID* via arguments of `init()` and obtain deterministic rank assignment through sorting by task name. Available options are:
  - host
  - task

- **timeout** (*int*) – Timeout for constructing (bootstrap) and shutting down the communication group, doesn't apply to communication when the group is up and running.

  The timeout value should take the time of data loading and pre-processing into account, due to potential lazy execution. By default the Tracker doesn't have any timeout to avoid pre-mature aborting.

  The `wait_for()` method has a different timeout parameter that can stop the tracker even if the tracker is still being used. A value error is raised when timeout is reached.

Examples

```
from xgboost.tracker import RabitTracker
from xgboost import collective as coll

tracker = RabitTracker(host_ip="127.0.0.1", n_workers=2)
tracker.start()

with coll.CommunicatorContext(**tracker.worker_args()):
    ret = coll.broadcast("msg", 0)
    assert str(ret) == "msg"
```

Supported Python data structures

This page is a support matrix for various input types.

Markers

- T: Supported.
- F: Not supported.
- NE: Invalid type for the use case. For instance, `pandas.Series` can not be multi-target label.
- NPA: Support with the help of numpy array.
- AT: Support with the help of arrow table.
- CPA: Support with the help of cupy array.
- SciCSR: Support with the help of scipy sparse CSR `scipy.sparse.csr_matrix`. The conversion to scipy CSR may or may not be possible. Raise a type error if conversion fails.
- FF: We can look forward to having its support in recent future if requested.
- empty: To be filled in.

Table Header

- *X* means predictor matrix.
- Meta info: label, weight, etc.
- Multi Label: 2-dim label for multi-target.
- Others: Anything else that we don't list here explicitly including formats like *lil*, *dia*, *bsr*. XGBoost will try to convert it into scipy csr.

Support Matrix

Name	DMatrix X	QuantileDMatrix X	Sklearn X	Meta Info	Inplace prediction	Multi Label
numpy.ndarray	T	T	T	T	T	T
scipy.sparse.csr	T	T	T	NE	T	F
scipy.sparse.csc	T	F	T	NE	F	F
scipy.sparse.coo	SciCSR	F	SciCSR	NE	F	F
uri	T	F	F	F	NE	F
list	NPA	NPA	NPA	NPA	NPA	T
tuple	NPA	NPA	NPA	NPA	NPA	T
pandas.DataFrame	NPA	NPA	NPA	NPA	NPA	NPA
pandas.Series	NPA	NPA	NPA	NPA	NPA	NE
cudf.DataFrame	T	T	T	T	T	T
cudf.Series	T	T	T	T	FF	NE
cupy.ndarray	T	T	T	T	T	T
torch.Tensor	T	T	T	T	T	T
dlpack	CPA	CPA		CPA	FF	FF
modin.DataFrame	NPA	FF	NPA	NPA	FF	
modin.Series	NPA	FF	NPA	NPA	FF	
pyarrow.Table	T	T	T	T	T	T
polars.DataFrame	AT	AT	AT	AT	AT	AT
polars.LazyFrame (WARN)	AT	AT	AT	AT	AT	AT
polars.Series	AT	AT	AT	AT	AT	NE
__array__	NPA	F	NPA	NPA	H	
Others	SciCSR	F		F	F	

The polars `LazyFrame.collect` supports many configurations, ranging from the choice of query engine to type coercion. XGBoost simply uses the default parameter. Please run `collect` to obtain the `DataFrame` before passing it into XGBoost for finer control over the behaviour.

Callback Functions

This document gives a basic walkthrough of *callback API* used in XGBoost Python package. In XGBoost 1.3, a new callback interface is designed for Python package, which provides the flexibility of designing various extension for training. Also, XGBoost has a number of pre-defined callbacks for supporting early stopping, checkpoints etc.

Using builtin callbacks

By default, training methods in XGBoost have parameters like `early_stopping_rounds` and `verbose/verbose_eval`, when specified the training procedure will define the corresponding callbacks internally. For example, when `early_stopping_rounds` is specified, `EarlyStopping` callback is invoked inside iteration loop. You can also pass this callback function directly into XGBoost:

```
D_train = xgb.DMatrix(X_train, y_train)
D_valid = xgb.DMatrix(X_valid, y_valid)

# Define a custom evaluation metric used for early stopping.
def eval_error_metric(predt, dtrain: xgb.DMatrix):
    label = dtrain.get_label()
    r = np.zeros(predt.shape)
```

(continues on next page)

(continued from previous page)

```
    gt = predt > 0.5
    r[gt] = 1 - label[gt]
    le = predt <= 0.5
    r[le] = label[le]
    return 'CustomErr', np.sum(r)

# Specify which dataset and which metric should be used for early stopping.
early_stop = xgb.callback.EarlyStopping(rounds=early_stopping_rounds,
                                        metric_name='CustomErr',
                                        data_name='Valid')

booster = xgb.train(
    {'objective': 'binary:logistic',
     'eval_metric': ['error', 'rmse'],
     'tree_method': 'hist'}, D_train,
    evals=[(D_train, 'Train'), (D_valid, 'Valid')],
    feval=eval_error_metric,
    num_boost_round=1000,
    callbacks=[early_stop],
    verbose_eval=False)

dump = booster.get_dump(dump_format='json')
assert len(early_stop.stopping_history['Valid']['CustomErr']) == len(dump)
```

Defining your own callback

XGBoost provides an callback interface class: *TrainingCallback*, user defined callbacks should inherit this class and override corresponding methods. There's a working example in *Demo for using and defining callback functions*.

XGBoost Python Feature Walkthrough

This is a collection of examples for using the XGBoost Python package.

Demo for obtaining leaf index

```
import os

import xgboost as xgb

# load data in do training
CURRENT_DIR = os.path.dirname(__file__)
dtrain = xgb.DMatrix(
    os.path.join(CURRENT_DIR, "../data/agaricus.txt.train?format=libsvm")
)
dtest = xgb.DMatrix(
    os.path.join(CURRENT_DIR, "../data/agaricus.txt.test?format=libsvm")
)
param = {"max_depth": 2, "eta": 1, "objective": "binary:logistic"}
watchlist = [(dtest, "eval"), (dtrain, "train")]
num_round = 3
bst = xgb.train(param, dtrain, num_round, watchlist)
```

(continues on next page)

(continued from previous page)

```
print("start testing predict the leaf indices")
# predict using first 2 tree
leafindex = bst.predict(
    dtest, iteration_range=(0, 2), pred_leaf=True, strict_shape=True
)
print(leafindex.shape)
print(leafindex)
# predict all trees
leafindex = bst.predict(dtest, pred_leaf=True)
print(leafindex.shape)
```

Demo for using xgboost with sklearn

```
import multiprocessing
from urllib.error import HTTPError

from sklearn.datasets import fetch_california_housing, make_regression
from sklearn.model_selection import GridSearchCV

import xgboost as xgb

if __name__ == "__main__":
    print("Parallel Parameter optimization")
    try:
        X, y = fetch_california_housing(return_X_y=True)
    except HTTPError:
        # Use a synthetic dataset instead if we couldn't
        X, y = make_regression(n_samples=20640, n_features=8, random_state=1234)
    # Make sure the number of threads is balanced.
    xgb_model = xgb.XGBRegressor(
        n_jobs=multiprocessing.cpu_count() // 2, tree_method="hist"
    )
    clf = GridSearchCV(
        xgb_model,
        {"max_depth": [2, 4, 6], "n_estimators": [50, 100, 200]},
        verbose=1,
        n_jobs=2,
    )
    clf.fit(X, y)
    print(clf.best_score_)
    print(clf.best_params_)
```

Using xgboost on GPU devices

Shows how to train a model on the forest cover type dataset using GPU acceleration. The forest cover type dataset has 581,012 rows and 54 features, making it time consuming to process. We compare the run-time and accuracy of the GPU and CPU histogram algorithms.

In addition, The demo showcases using GPU with other GPU-related libraries including cupy and cuml. These libraries are not strictly required.

```
import time

import cupy as cp
from cuml.model_selection import train_test_split
from sklearn.datasets import fetch_covtype

import xgboost as xgb

# Fetch dataset using sklearn
X, y = fetch_covtype(return_X_y=True)
X = cp.array(X)
y = cp.array(y)
y -= y.min()

# Create 0.75/0.25 train/test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.25, train_size=0.75, random_state=42
)

# Specify sufficient boosting iterations to reach a minimum
num_round = 3000

# Leave most parameters as default
clf = xgb.XGBClassifier(device="cuda", n_estimators=num_round)
# Train model
start = time.time()
clf.fit(X_train, y_train, eval_set=[(X_test, y_test)])
gpu_res = clf.evals_result()
print("GPU Training Time: %s seconds" % (str(time.time() - start)))

# Repeat for CPU algorithm
clf = xgb.XGBClassifier(device="cpu", n_estimators=num_round)
start = time.time()
clf.fit(X_train, y_train, eval_set=[(X_test, y_test)])
cpu_res = clf.evals_result()
print("CPU Training Time: %s seconds" % (str(time.time() - start)))
```

Demo for gamma regression

```
import numpy as np

import xgboost as xgb

#  this script demonstrates how to fit gamma regression model (with log link function)
#  in xgboost, before running the demo you need to generate the autoclaims dataset
#  by running gen_autoclaims.R located in xgboost/demo/data.

data = np.genfromtxt('../data/autoclaims.csv', delimiter=',')
dtrain = xgb.DMatrix(data[0:4741, 0:34], data[0:4741, 34])
dtest = xgb.DMatrix(data[4741:6773, 0:34], data[4741:6773, 34])

# for gamma regression, we need to set the objective to 'reg:gamma', it also suggests
```

(continues on next page)

(continued from previous page)

```
# to set the base_score to a value between 1 to 5 if the number of iteration is small
param = {'objective':'reg:gamma', 'booster':'gbtree', 'base_score':3}

# the rest of settings are the same
watchlist = [(dtest, 'eval'), (dtrain, 'train')]
num_round = 30

# training and evaluation
bst = xgb.train(param, dtrain, num_round, watchlist)
preds = bst.predict(dtest)
labels = dtest.get_label()
print('test deviance=%f' % (2 * np.sum((labels - preds) / preds - np.log(labels) + np.
↪log(preds))))
```

This script demonstrate how to access the eval metrics

```
import os
from typing import Any, Dict

import xgboost as xgb

CURRENT_DIR = os.path.dirname(__file__)
dtrain = xgb.DMatrix(
    os.path.join(CURRENT_DIR, "../data/agaricus.txt.train?format=libsvm")
)
dtest = xgb.DMatrix(
    os.path.join(CURRENT_DIR, "../data/agaricus.txt.test?format=libsvm")
)

param = [
    ("max_depth", 2),
    ("objective", "binary:logistic"),
    ("eval_metric", "logloss"),
    ("eval_metric", "error"),
]

num_round = 2
watchlist = [(dtest, "eval"), (dtrain, "train")]

evals_result: Dict[str, Any] = {}
bst = xgb.train(param, dtrain, num_round, watchlist, evals_result=evals_result)

print("Access logloss metric directly from evals_result:")
print(evals_result["eval"]["logloss"])

print("")
print("Access metrics through a loop:")
for e_name, e_mtrs in evals_result.items():
    print("- {}".format(e_name))
    for e_mtr_name, e_mtr_vals in e_mtrs.items():
        print("    - {}".format(e_mtr_name))
```

(continues on next page)

(continued from previous page)

```
        print("       - {}".format(e_mtr_vals))

print("")
print("Access complete dictionary:")
print(evals_result)
```

Demo for boosting from prediction

```
import os

import xgboost as xgb

CURRENT_DIR = os.path.dirname(__file__)
dtrain = xgb.DMatrix(
    os.path.join(CURRENT_DIR, "../data/agaricus.txt.train?format=libsvm")
)
dtest = xgb.DMatrix(
    os.path.join(CURRENT_DIR, "../data/agaricus.txt.test?format=libsvm")
)
watchlist = [(dtest, "eval"), (dtrain, "train")]
###
# advanced: start from a initial base prediction
#
print("start running example to start from a initial prediction")
# specify parameters via map, definition are same as c++ version
param = {"max_depth": 2, "eta": 1, "objective": "binary:logistic"}
# train xgboost for 1 round
bst = xgb.train(param, dtrain, 1, watchlist)
# Note: we need the margin value instead of transformed prediction in
# set_base_margin
# do predict with output_margin=True, will always give you margin values
# before logistic transformation
ptrain = bst.predict(dtrain, output_margin=True)
ptest = bst.predict(dtest, output_margin=True)
dtrain.set_base_margin(ptrain)
dtest.set_base_margin(ptest)

print("this is result of running from initial prediction")
bst = xgb.train(param, dtrain, 1, watchlist)
```

Demo for accessing the xgboost eval metrics by using sklearn interface

```
import numpy as np
from sklearn.datasets import make_hastie_10_2

import xgboost as xgb

X, y = make_hastie_10_2(n_samples=2000, random_state=42)

# Map labels from {-1, 1} to {0, 1}
labels, y = np.unique(y, return_inverse=True)
```

(continues on next page)

(continued from previous page)

```
X_train, X_test = X[:1600], X[1600:]
y_train, y_test = y[:1600], y[1600:]

param_dist = {"objective": "binary:logistic", "n_estimators": 2}

clf = xgb.XGBModel(
    **param_dist,
    eval_metric="logloss",
)
# Or you can use: clf = xgb.XGBClassifier(**param_dist)

clf.fit(
    X_train,
    y_train,
    eval_set=[(X_train, y_train), (X_test, y_test)],
    verbose=True,
)

# Load evals result by calling the evals_result() function
evals_result = clf.evals_result()

print("Access logloss metric directly from validation_0:")
print(evals_result["validation_0"]["logloss"])

print("")
print("Access metrics through a loop:")
for e_name, e_mtrs in evals_result.items():
    print("- {}".format(e_name))
    for e_mtr_name, e_mtr_vals in e_mtrs.items():
        print("    - {}".format(e_mtr_name))
        print("        - {}".format(e_mtr_vals))

print("")
print("Access complete dict:")
print(evals_result)
```

Demo for using feature weight to change column sampling

Added in version 1.3.0.

```
import argparse

import numpy as np
from matplotlib import pyplot as plt

import xgboost


def main(args: argparse.Namespace) -> None:
    rng = np.random.RandomState(1994)
```

(continues on next page)

(continued from previous page)

```
    kRows = 4196
    kCols = 10

    X = rng.randn(kRows, kCols)
    y = rng.randn(kRows)
    fw = np.ones(shape=(kCols,))
    for i in range(kCols):
        fw[i] *= float(i)

    dtrain = xgboost.DMatrix(X, y)
    dtrain.set_info(feature_weights=fw)

    # Perform column sampling for each node split evaluation, the sampling process is
    # weighted by feature weights.
    bst = xgboost.train(
        {"tree_method": "hist", "colsample_bynode": 0.2},
        dtrain,
        num_boost_round=10,
        evals=[(dtrain, "d")],
    )
    feature_map = bst.get_fscore()

    # feature zero has 0 weight
    assert feature_map.get("f0", None) is None
    assert max(feature_map.values()) == feature_map.get("f9")

    if args.plot:
        xgboost.plot_importance(bst)
        plt.show()


if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument(
        "--plot",
        type=int,
        default=1,
        help="Set to 0 to disable plotting the evaluation history.",
    )
    args = parser.parse_args()
    main(args)
```

Demo for GLM

```
import os

import xgboost as xgb

##
#  this script demonstrate how to fit generalized linear model in xgboost
#  basically, we are using linear model, instead of tree for our boosters
```

(continues on next page)

(continued from previous page)

```
##
CURRENT_DIR = os.path.dirname(__file__)
dtrain = xgb.DMatrix(
    os.path.join(CURRENT_DIR, "../data/agaricus.txt.train?format=libsvm")
)
dtest = xgb.DMatrix(
    os.path.join(CURRENT_DIR, "../data/agaricus.txt.test?format=libsvm")
)
# change booster to gblinear, so that we are fitting a linear model
# alpha is the L1 regularizer
# lambda is the L2 regularizer
# you can also set lambda_bias which is L2 regularizer on the bias term
param = {
    "objective": "binary:logistic",
    "booster": "gblinear",
    "alpha": 0.0001,
    "lambda": 1,
}

# normally, you do not need to set eta (step_size)
# XGBoost uses a parallel coordinate descent algorithm (shotgun),
# there could be affection on convergence with parallelization on certain cases
# setting eta to be smaller value, e.g 0.5 can make the optimization more stable
# param['eta'] = 1

##
# the rest of settings are the same
##
watchlist = [(dtest, "eval"), (dtrain, "train")]
num_round = 4
bst = xgb.train(param, dtrain, num_round, watchlist)
preds = bst.predict(dtest)
labels = dtest.get_label()
print(
    "error=%f"
    % (
        sum(1 for i in range(len(preds)) if int(preds[i] > 0.5) != labels[i])
        / float(len(preds))
    )
)
```

Use GPU to speedup SHAP value computation

Demonstrates using GPU acceleration to compute SHAP values for feature importance.

```
from urllib.error import HTTPError

import shap
from sklearn.datasets import fetch_california_housing, make_regression

import xgboost as xgb
```

(continues on next page)

(continued from previous page)

```
# Fetch dataset using sklearn
try:
    _data = fetch_california_housing(return_X_y=True)
    X = _data.data
    y = _data.target
    feature_names = _data.feature_names
    print(_data.DESCR)
except HTTPError:
    # Use a synthetic dataset instead if we couldn't
    X, y = make_regression(n_samples=20640, n_features=8, random_state=1234)
    feature_names = [f"f{i}" for i in range(8)]

num_round = 500

param = {
    "eta": 0.05,
    "max_depth": 10,
    "tree_method": "hist",
    "device": "cuda",
}

# GPU accelerated training
dtrain = xgb.DMatrix(X, label=y, feature_names=feature_names)
model = xgb.train(param, dtrain, num_round)

# Compute shap values using GPU with xgboost
model.set_param({"device": "cuda"})
shap_values = model.predict(dtrain, pred_contribs=True)

# Compute shap interaction values using GPU
shap_interaction_values = model.predict(dtrain, pred_interactions=True)


# shap will call the GPU accelerated version as long as the device parameter is set to
# "cuda"
explainer = shap.TreeExplainer(model)
shap_values = explainer.shap_values(X)

# visualize the first prediction's explanation
shap.force_plot(
    explainer.expected_value,
    shap_values[0, :],
    X[0, :],
    feature_names=feature_names,
    matplotlib=True,
)

# Show a summary of feature importance
shap.summary_plot(shap_values, X, plot_type="bar", feature_names=feature_names)
```

Demo for prediction using number of trees

```
import os

import numpy as np
from sklearn.datasets import load_svmlight_file

import xgboost as xgb

CURRENT_DIR = os.path.dirname(__file__)
train = os.path.join(CURRENT_DIR, "../data/agaricus.txt.train")
test = os.path.join(CURRENT_DIR, "../data/agaricus.txt.test")


def native_interface() -> None:
    # load data in do training
    dtrain = xgb.DMatrix(train + "?format=libsvm")
    dtest = xgb.DMatrix(test + "?format=libsvm")
    param = {"max_depth": 2, "eta": 1, "objective": "binary:logistic"}
    watchlist = [(dtest, "eval"), (dtrain, "train")]
    num_round = 3
    bst = xgb.train(param, dtrain, num_round, watchlist)

    print("start testing prediction from first n trees")
    # predict using first 1 tree
    label = dtest.get_label()
    ypred1 = bst.predict(dtest, iteration_range=(0, 1))
    # by default, we predict using all the trees
    ypred2 = bst.predict(dtest)

    print("error of ypred1=%f" % (np.sum((ypred1 > 0.5) != label) / float(len(label))))
    print("error of ypred2=%f" % (np.sum((ypred2 > 0.5) != label) / float(len(label))))


def sklearn_interface() -> None:
    X_train, y_train = load_svmlight_file(train)
    X_test, y_test = load_svmlight_file(test)
    clf = xgb.XGBClassifier(n_estimators=3, max_depth=2, eta=1)
    clf.fit(X_train, y_train, eval_set=[(X_test, y_test)])
    assert clf.n_classes_ == 2

    print("start testing prediction from first n trees")
    # predict using first 1 tree
    ypred1 = clf.predict(X_test, iteration_range=(0, 1))
    # by default, we predict using all the trees
    ypred2 = clf.predict(X_test)

    print(
        "error of ypred1=%f" % (np.sum((ypred1 > 0.5) != y_test) / float(len(y_test)))
    )
    print(
        "error of ypred2=%f" % (np.sum((ypred2 > 0.5) != y_test) / float(len(y_test)))
    )
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    native_interface()
    sklearn_interface()
```

Getting started with XGBoost

This is a simple example of using the native XGBoost interface, there are other interfaces in the Python package like scikit-learn interface and Dask interface.

See *Python Package Introduction* and *XGBoost Tutorials* for other references.

```
import os
import pickle

import numpy as np
from sklearn.datasets import load_svmlight_file

import xgboost as xgb

# Make sure the demo knows where to load the data.
CURRENT_DIR = os.path.dirname(os.path.abspath(__file__))
XGBOOST_ROOT_DIR = os.path.dirname(os.path.dirname(CURRENT_DIR))
DEMO_DIR = os.path.join(XGBOOST_ROOT_DIR, "demo")

# X is a scipy csr matrix, XGBoost supports many other input types,
X, y = load_svmlight_file(os.path.join(DEMO_DIR, "data", "agaricus.txt.train"))
dtrain = xgb.DMatrix(X, y)
# validation set
X_test, y_test = load_svmlight_file(os.path.join(DEMO_DIR, "data", "agaricus.txt.test"))
dtest = xgb.DMatrix(X_test, y_test)

# specify parameters via map, definition are same as c++ version
param = {"max_depth": 2, "eta": 1, "objective": "binary:logistic"}

# specify validations set to watch performance
watchlist = [(dtest, "eval"), (dtrain, "train")]
# number of boosting rounds
num_round = 2
bst = xgb.train(param, dtrain, num_boost_round=num_round, evals=watchlist)

# run prediction
preds = bst.predict(dtest)
labels = dtest.get_label()
print(
    "error=%f"
    % (
        sum(1 for i in range(len(preds)) if int(preds[i] > 0.5) != labels[i])
        / float(len(preds))
    )
)
```

(continues on next page)

(continued from previous page)

```
bst.save_model("model-0.json")
# dump model
bst.dump_model("dump.raw.txt")
# dump model with feature map
bst.dump_model("dump.nice.txt", os.path.join(DEMO_DIR, "data/featmap.txt"))

# save dmatrix into binary buffer
dtest.save_binary("dtest.dmatrix")
# save model
bst.save_model("model-1.json")
# load model and data in
bst2 = xgb.Booster(model_file="model-1.json")
dtest2 = xgb.DMatrix("dtest.dmatrix")
preds2 = bst2.predict(dtest2)
# assert they are the same
assert np.sum(np.abs(preds2 - preds)) == 0

# alternatively, you can pickle the booster
pks = pickle.dumps(bst2)
# load model and data in
bst3 = pickle.loads(pks)
preds3 = bst3.predict(dtest2)
# assert they are the same
assert np.sum(np.abs(preds3 - preds)) == 0
```

Getting started with categorical data

Experimental support for categorical data.

In before, users need to run an encoder themselves before passing the data into XGBoost, which creates a sparse matrix and potentially increase memory usage. This demo showcases the experimental categorical data support, more advanced features are planned.

Added in version 1.5.0.

See Also

- *Tutorial*
- *Train XGBoost with cat_in_the_dat dataset*
- *Feature engineering pipeline for categorical data*

```
from typing import Tuple

import numpy as np
import pandas as pd

import xgboost as xgb


def make_categorical(
    n_samples: int, n_features: int, n_categories: int, onehot: bool
) -> Tuple[pd.DataFrame, pd.Series]:
```

(continues on next page)

(continued from previous page)

```
    """Make some random data for demo."""
    rng = np.random.RandomState(1994)

    pd_dict = {}
    for i in range(n_features + 1):
        c = rng.randint(low=0, high=n_categories, size=n_samples)
        pd_dict[str(i)] = pd.Series(c, dtype=np.int64)

    df = pd.DataFrame(pd_dict)
    label = df.iloc[:, 0]
    df = df.iloc[:, 1:]
    for i in range(0, n_features):
        label += df.iloc[:, i]
    label += 1

    df = df.astype("category")
    categories = np.arange(0, n_categories)
    for col in df.columns:
        df[col] = df[col].cat.set_categories(categories)

    if onehot:
        return pd.get_dummies(df), label
    return df, label


def main() -> None:
    # Use builtin categorical data support

    # For scikit-learn interface, the input data should be pandas DataFrame or cudf
    # DataFrame with categorical features. If an numpy/cupy array is used instead, the
    # `feature_types` for `XGBRegressor` should be set accordingly.
    X, y = make_categorical(100, 10, 4, False)
    # Specify `enable_categorical` to True, also we use onehot-encoding-based split here
    # for demonstration. For details see the document of `max_cat_to_onehot`.
    reg = xgb.XGBRegressor(
        tree_method="hist", enable_categorical=True, max_cat_to_onehot=5, device="cuda"
    )
    reg.fit(X, y, eval_set=[(X, y)])

    # Pass in already encoded data
    X_enc, y_enc = make_categorical(100, 10, 4, True)
    reg_enc = xgb.XGBRegressor(tree_method="hist", device="cuda")
    reg_enc.fit(X_enc, y_enc, eval_set=[(X_enc, y_enc)])

    reg_results = np.array(reg.evals_result()["validation_0"]["rmse"])
    reg_enc_results = np.array(reg_enc.evals_result()["validation_0"]["rmse"])

    # Check that they have same results
    np.testing.assert_allclose(reg_results, reg_enc_results)

    # Convert to DMatrix for SHAP value
    booster: xgb.Booster = reg.get_booster()
```

(continues on next page)

(continued from previous page)

```
    m = xgb.DMatrix(X, enable_categorical=True)  # specify categorical data support.
    SHAP = booster.predict(m, pred_contribs=True)
    margin = booster.predict(m, output_margin=True)
    np.testing.assert_allclose(
        np.sum(SHAP, axis=len(SHAP.shape) - 1), margin, rtol=1e-3
    )


if __name__ == "__main__":
    main()
```

Collection of examples for using sklearn interface

For an introduction to XGBoost's scikit-learn estimator interface, see *Using the Scikit-Learn Estimator Interface*.

Created on 1 Apr 2015

@author: Jamie Hall

```
import pickle
from urllib.error import HTTPError

import numpy as np
from sklearn.datasets import (
    fetch_california_housing,
    load_digits,
    load_iris,
    make_regression,
)
from sklearn.metrics import confusion_matrix, mean_squared_error
from sklearn.model_selection import GridSearchCV, KFold, train_test_split

import xgboost as xgb

rng = np.random.RandomState(31337)

print("Zeros and Ones from the Digits dataset: binary classification")
digits = load_digits(n_class=2)
y = digits["target"]
X = digits["data"]
kf = KFold(n_splits=2, shuffle=True, random_state=rng)
for train_index, test_index in kf.split(X):
    xgb_model = xgb.XGBClassifier(n_jobs=1).fit(X[train_index], y[train_index])
    predictions = xgb_model.predict(X[test_index])
    actuals = y[test_index]
    print(confusion_matrix(actuals, predictions))

print("Iris: multiclass classification")
iris = load_iris()
y = iris["target"]
X = iris["data"]
kf = KFold(n_splits=2, shuffle=True, random_state=rng)
for train_index, test_index in kf.split(X):
```

(continues on next page)

(continued from previous page)

```
    xgb_model = xgb.XGBClassifier(n_jobs=1).fit(X[train_index], y[train_index])
    predictions = xgb_model.predict(X[test_index])
    actuals = y[test_index]
    print(confusion_matrix(actuals, predictions))

print("California Housing: regression")

try:
    X, y = fetch_california_housing(return_X_y=True)
except HTTPError:
    # Use a synthetic dataset instead if we couldn't
    X, y = make_regression(n_samples=20640, n_features=8, random_state=1234)

kf = KFold(n_splits=2, shuffle=True, random_state=rng)
for train_index, test_index in kf.split(X):
    xgb_model = xgb.XGBRegressor(n_jobs=1).fit(X[train_index], y[train_index])
    predictions = xgb_model.predict(X[test_index])
    actuals = y[test_index]
    print(mean_squared_error(actuals, predictions))

print("Parameter optimization")
xgb_model = xgb.XGBRegressor(n_jobs=1)
clf = GridSearchCV(
    xgb_model,
    {"max_depth": [2, 4], "n_estimators": [50, 100]},
    verbose=1,
    n_jobs=1,
    cv=3,
)
clf.fit(X, y)
print(clf.best_score_)
print(clf.best_params_)

# The sklearn API models are picklable
print("Pickling sklearn API models")
# must open in binary format to pickle
pickle.dump(clf, open("best_calif.pkl", "wb"))
clf2 = pickle.load(open("best_calif.pkl", "rb"))
print(np.allclose(clf.predict(X), clf2.predict(X)))

# Early-stopping

X = digits["data"]
y = digits["target"]
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
clf = xgb.XGBClassifier(n_jobs=1, early_stopping_rounds=10, eval_metric="auc")
clf.fit(X_train, y_train, eval_set=[(X_test, y_test)])
```

Demo for using cross validation

```
import os
from typing import Any, Dict, Tuple

import numpy as np

import xgboost as xgb

# load data in do training
CURRENT_DIR = os.path.dirname(__file__)
dtrain = xgb.DMatrix(
    os.path.join(CURRENT_DIR, "../data/agaricus.txt.train?format=libsvm")
)
param = {"max_depth": 2, "eta": 1, "objective": "binary:logistic"}
num_round = 2

print("running cross validation")
# do cross validation, this will print result out as
# [iteration]  metric_name:mean_value+std_value
# std_value is standard deviation of the metric
xgb.cv(
    param,
    dtrain,
    num_round,
    nfold=5,
    metrics={"error"},
    seed=0,
    callbacks=[xgb.callback.EvaluationMonitor(show_stdv=True)],
)

print("running cross validation, disable standard deviation display")
# do cross validation, this will print result out as
# [iteration]  metric_name:mean_value
res = xgb.cv(
    param,
    dtrain,
    num_boost_round=10,
    nfold=5,
    metrics={"error"},
    seed=0,
    callbacks=[
        xgb.callback.EvaluationMonitor(show_stdv=False),
        xgb.callback.EarlyStopping(3),
    ],
)
print(res)
print("running cross validation, with preprocessing function")


# define the preprocessing function
# used to return the preprocessed training, test data, and parameter
# we can use this to do weight rescale, etc.
```

(continues on next page)

(continued from previous page)

```
# as a example, we try to set scale_pos_weight
def fpreproc(
    dtrain: xgb.DMatrix, dtest: xgb.DMatrix, param: Any
) -> Tuple[xgb.DMatrix, xgb.DMatrix, Dict[str, Any]]:
    label = dtrain.get_label()
    ratio = float(np.sum(label == 0)) / np.sum(label == 1)
    param["scale_pos_weight"] = ratio
    return (dtrain, dtest, param)


# do cross validation, for each fold
# the dtrain, dtest, param will be passed into fpreproc
# then the return value of fpreproc will be used to generate
# results of that fold
xgb.cv(param, dtrain, num_round, nfold=5, metrics={"auc"}, seed=0, fpreproc=fpreproc)

###
# you can also do cross validation with customized loss function
# See custom_objective.py
##
print("running cross validation, with customized loss function")


def logregobj(preds: np.ndarray, dtrain: xgb.DMatrix) -> Tuple[np.ndarray, np.ndarray]:
    labels = dtrain.get_label()
    preds = 1.0 / (1.0 + np.exp(-preds))
    grad = preds - labels
    hess = preds * (1.0 - preds)
    return grad, hess


def evalerror(preds: np.ndarray, dtrain: xgb.DMatrix) -> Tuple[str, float]:
    labels = dtrain.get_label()
    preds = 1.0 / (1.0 + np.exp(-preds))
    return "error", float(sum(labels != (preds > 0.0))) / len(labels)


param = {"max_depth": 2, "eta": 1}
# train with customized objective
xgb.cv(
    param, dtrain, num_round, nfold=5, seed=0, obj=logregobj, custom_metric=evalerror
)
```

A demo for multi-output regression using reduced gradient

See *Multiple Outputs* for more information.

Added in version 3.2.0.

Note

The implementation is experimental and many features are missing.

See also

A demo for multi-output regression

```
import argparse
from typing import Tuple

import numpy as np
from sklearn.base import BaseEstimator
from sklearn.datasets import make_regression

import xgboost as xgb
from xgboost.objective import TreeObjective


class LsObjMean(TreeObjective):
    """Least squared error. Reduce the size of the gradient using mean value."""

    def __init__(self, device: str) -> None:
        self.device = device

    def __call__(
        self, iteration: int, y_pred: np.ndarray, dtrain: xgb.DMatrix
    ) -> Tuple[np.ndarray, np.ndarray]:
        y_true = dtrain.get_label()
        grad = y_pred - y_true
        if self.device == "cpu":
            hess = np.ones(grad.shape)
            return grad, hess
        else:
            import cupy as cp

            hess = cp.ones(grad.shape)

            return cp.array(grad), cp.array(hess)

    def split_grad(
        self, iteration: int, grad: np.ndarray, hess: np.ndarray
    ) -> Tuple[np.ndarray, np.ndarray]:
        if self.device == "cpu":
            from numpy import mean
        else:
            from cupy import mean  # type: ignore[no-redef]

        sgrad = mean(grad, axis=1)
        shess = mean(hess, axis=1)
        return sgrad, shess


def svd_class(device: str) -> BaseEstimator:
    """One of the methods in the sketch boost paper."""
    from sklearn.decomposition import TruncatedSVD
```

(continues on next page)

(continued from previous page)

```
    svd_params = {"algorithm": "arpack", "n_components": 2, "n_iter": 8}
    svd = TruncatedSVD(**svd_params)
    return svd


class LsObjSvd(LsObjMean):
    """Reduce the size of the gradient using SVD."""

    def __init__(self, device: str) -> None:
        super().__init__(device=device)

    def split_grad(
        self, iteration: int, grad: np.ndarray, hess: np.ndarray
    ) -> Tuple[np.ndarray, np.ndarray]:
        svd = svd_class(self.device)
        if self.device == "cuda":
            grad = grad.get()  # type: ignore
            hess = hess.get()  # type: ignore

        svd.fit(grad)
        grad = svd.transform(grad)
        hess = svd.transform(hess)
        if self.device == "cpu":
            hess = np.clip(hess, 0.01, None)
        else:
            import cupy as cp

            hess = cp.clip(hess, 0.01, None)
        return grad, hess


def main() -> None:
    parser = argparse.ArgumentParser()
    parser.add_argument("--device", choices=["cpu", "cuda"], default="cpu")
    args = parser.parse_args()

    X, y = make_regression(
        n_samples=8192, n_features=32, n_targets=8, random_state=2026
    )
    Xy = xgb.QuantileDMatrix(X, y)

    for obj in (LsObjMean(args.device), LsObjSvd(args.device)):
        xgb.train(
            {
                "device": args.device,
                "multi_strategy": "multi_output_tree",
            },
            Xy,
            evals=[(Xy, "Train")],
            obj=obj,
            num_boost_round=16,
```

(continues on next page)

(continued from previous page)

```
    )


if __name__ == "__main__":
    main()
```

Demo for using *process_type* with *prune* and *refresh*

Modifying existing trees is not a well established use for XGBoost, so feel free to experiment.

```
from urllib.error import HTTPError

import numpy as np
from sklearn.datasets import fetch_california_housing, make_regression

import xgboost as xgb


def main() -> None:
    n_rounds = 32

    try:
        X, y = fetch_california_housing(return_X_y=True)
    except HTTPError:
        # Use a synthetic dataset instead if we couldn't
        X, y = make_regression(n_samples=20640, n_features=8, random_state=1234)

    # Train a model first
    X_train = X[: X.shape[0] // 2]
    y_train = y[: y.shape[0] // 2]
    Xy = xgb.DMatrix(X_train, y_train)
    evals_result: xgb.callback.EvaluationMonitor.EvalsLog = {}
    booster = xgb.train(
        {"tree_method": "hist", "max_depth": 6, "device": "cuda"},
        Xy,
        num_boost_round=n_rounds,
        evals=[(Xy, "Train")],
        evals_result=evals_result,
    )
    SHAP = booster.predict(Xy, pred_contribs=True)

    # Refresh the leaf value and tree statistic
    X_refresh = X[X.shape[0] // 2 :]
    y_refresh = y[y.shape[0] // 2 :]
    Xy_refresh = xgb.DMatrix(X_refresh, y_refresh)
    # The model will adapt to other half of the data by changing leaf value (no change in
    # split condition) with refresh_leaf set to True.
    refresh_result: xgb.callback.EvaluationMonitor.EvalsLog = {}
    refreshed = xgb.train(
        {"process_type": "update", "updater": "refresh", "refresh_leaf": True},
        Xy_refresh,
        num_boost_round=n_rounds,
```

(continues on next page)

(continued from previous page)

```
        xgb_model=booster,
        evals=[(Xy, "Original"), (Xy_refresh, "Train")],
        evals_result=refresh_result,
    )

    # Refresh the model without changing the leaf value, but tree statistic including
    # cover and weight are refreshed.
    refresh_result = {}
    refreshed = xgb.train(
        {"process_type": "update", "updater": "refresh", "refresh_leaf": False},
        Xy_refresh,
        num_boost_round=n_rounds,
        xgb_model=booster,
        evals=[(Xy, "Original"), (Xy_refresh, "Train")],
        evals_result=refresh_result,
    )
    # Without refreshing the leaf value, resulting trees should be the same with original
    # model except for accumulated statistic.  The rtol is for floating point error in
    # prediction.
    np.testing.assert_allclose(
        refresh_result["Original"]["rmse"], evals_result["Train"]["rmse"], rtol=1e-5
    )
    # But SHAP value is changed as cover in tree nodes are changed.
    refreshed_SHAP = refreshed.predict(Xy, pred_contribs=True)
    assert not np.allclose(SHAP, refreshed_SHAP, rtol=1e-3)

    # Prune the trees with smaller max_depth
    X_update = X_train
    y_update = y_train
    Xy_update = xgb.DMatrix(X_update, y_update)

    prune_result: xgb.callback.EvaluationMonitor.EvalsLog = {}
    pruned = xgb.train(
        {"process_type": "update", "updater": "prune", "max_depth": 2},
        Xy_update,
        num_boost_round=n_rounds,
        xgb_model=booster,
        evals=[(Xy, "Original"), (Xy_update, "Train")],
        evals_result=prune_result,
    )
    # Have a smaller model, but similar accuracy.
    np.testing.assert_allclose(
        np.array(prune_result["Original"]["rmse"]),
        np.array(prune_result["Train"]["rmse"]),
        atol=1e-5,
    )


if __name__ == "__main__":
    main()
```

Demo for prediction using individual trees and model slices

```
import os

import numpy as np
from scipy.special import logit
from sklearn.datasets import load_svmlight_file

import xgboost as xgb

CURRENT_DIR = os.path.dirname(__file__)
train = os.path.join(CURRENT_DIR, "../data/agaricus.txt.train")
test = os.path.join(CURRENT_DIR, "../data/agaricus.txt.test")


def individual_tree() -> None:
    """Get prediction from each individual tree and combine them together."""
    X_train, y_train = load_svmlight_file(train)
    X_test, y_test = load_svmlight_file(test)
    Xy_train = xgb.QuantileDMatrix(X_train, y_train)

    n_rounds = 4
    # Specify the base score, otherwise xgboost will estimate one from the training
    # data.
    base_score = 0.5
    params = {
        "max_depth": 2,
        "eta": 1,
        "objective": "reg:logistic",
        "tree_method": "hist",
        "base_score": base_score,
    }
    booster = xgb.train(params, Xy_train, num_boost_round=n_rounds)

    # Use logit to inverse the base score back to raw leaf value (margin)
    scores = np.full((X_test.shape[0],), logit(base_score))
    for i in range(n_rounds):
        # - Use output_margin to get raw leaf values
        # - Use iteration_range to get prediction for only one tree
        # - Use previous prediction as base margin for the model
        Xy_test = xgb.DMatrix(X_test, base_margin=scores)

        if i == n_rounds - 1:
            # last round, get the transformed prediction
            scores = booster.predict(
                Xy_test, iteration_range=(i, i + 1), output_margin=False
            )
        else:
            # get raw leaf value for accumulation
            scores = booster.predict(
                Xy_test, iteration_range=(i, i + 1), output_margin=True
            )
```

(continues on next page)

(continued from previous page)

```
    full = booster.predict(xgb.DMatrix(X_test), output_margin=False)
    np.testing.assert_allclose(scores, full)


def model_slices() -> None:
    """Inference with each individual tree using model slices."""
    X_train, y_train = load_svmlight_file(train)
    X_test, y_test = load_svmlight_file(test)
    Xy_train = xgb.QuantileDMatrix(X_train, y_train)

    n_rounds = 4
    # Specify the base score, otherwise xgboost will estimate one from the training
    # data.
    base_score = 0.5
    params = {
        "max_depth": 2,
        "eta": 1,
        "objective": "reg:logistic",
        "tree_method": "hist",
        "base_score": base_score,
    }
    booster = xgb.train(params, Xy_train, num_boost_round=n_rounds)
    trees = [booster[t] for t in range(n_rounds)]

    # Use logit to inverse the base score back to raw leaf value (margin)
    scores = np.full((X_test.shape[0],), logit(base_score))
    for i, t in enumerate(trees):
        # Feed previous scores into base margin.
        Xy_test = xgb.DMatrix(X_test, base_margin=scores)

        if i == n_rounds - 1:
            # last round, get the transformed prediction
            scores = t.predict(Xy_test, output_margin=False)
        else:
            # get raw leaf value for accumulation
            scores = t.predict(Xy_test, output_margin=True)

    full = booster.predict(xgb.DMatrix(X_test), output_margin=False)
    np.testing.assert_allclose(scores, full)


if __name__ == "__main__":
    individual_tree()
    model_slices()
```

Demo for using data iterator with Quantile DMatrix

Added in version 1.2.0.

The demo that defines a customized iterator for passing batches of data into `xgboost.QuantileDMatrix` and use this `QuantileDMatrix` for training. The feature is primarily designed to reduce the required GPU memory for training on distributed environment.

Aftering going through the demo, one might ask why don't we use more native Python iterator? That's because XGBoost requires a *reset* function, while using *itertools.tee* might incur significant memory usage according to:

> https://docs.python.org/3/library/itertools.html#itertools.tee.

See also

Experimental support for external memory

```
from typing import Callable

import cupy
import numpy

import xgboost

COLS = 64
ROWS_PER_BATCH = 1000  # data is splited by rows
BATCHES = 32


class IterForDMatrixDemo(xgboost.core.DataIter):
    """A data iterator for XGBoost DMatrix.

    `reset` and `next` are required for any data iterator, other functions here
    are utilites for demonstration's purpose.

    """

    def __init__(self) -> None:
        """Generate some random data for demostration.

        Actual data can be anything that is currently supported by XGBoost.
        """
        self.rows = ROWS_PER_BATCH
        self.cols = COLS
        rng = cupy.random.RandomState(numpy.uint64(1994))
        self._data = [rng.randn(self.rows, self.cols)] * BATCHES
        self._labels = [rng.randn(self.rows)] * BATCHES
        self._weights = [rng.uniform(size=self.rows)] * BATCHES

        self.it = 0  # set iterator to 0
        super().__init__()

    def as_array(self) -> cupy.ndarray:
        return cupy.concatenate(self._data)

    def as_array_labels(self) -> cupy.ndarray:
        return cupy.concatenate(self._labels)

    def as_array_weights(self) -> cupy.ndarray:
        return cupy.concatenate(self._weights)
```

(continues on next page)

(continued from previous page)

```
    def data(self) -> cupy.ndarray:
        """Utility function for obtaining current batch of data."""
        return self._data[self.it]

    def labels(self) -> cupy.ndarray:
        """Utility function for obtaining current batch of label."""
        return self._labels[self.it]

    def weights(self) -> cupy.ndarray:
        return self._weights[self.it]

    def reset(self) -> None:
        """Reset the iterator"""
        self.it = 0

    def next(self, input_data: Callable) -> bool:
        """Yield the next batch of data."""
        if self.it == len(self._data):
            # Return False to let XGBoost know this is the end of iteration
            return False

        # input_data is a keyword-only function passed in by XGBoost and has the similar
        # signature to the ``DMatrix`` constructor.
        input_data(data=self.data(), label=self.labels(), weight=self.weights())
        self.it += 1
        return True


def main() -> None:
    rounds = 100
    it = IterForDMatrixDemo()

    # Use iterator, must be `QuantileDMatrix`.

    # In this demo, the input batches are created using cupy, and the data processing
    # (quantile sketching) will be performed on GPU. If data is loaded with CPU based
    # data structures like numpy or pandas, then the processing step will be performed
    # on CPU instead.
    m_with_it = xgboost.QuantileDMatrix(it)

    # Use regular DMatrix.
    m = xgboost.DMatrix(
        it.as_array(), it.as_array_labels(), weight=it.as_array_weights()
    )

    assert m_with_it.num_col() == m.num_col()
    assert m_with_it.num_row() == m.num_row()
    # Tree method must be `hist`.
    reg_with_it = xgboost.train(
        {"tree_method": "hist", "device": "cuda"},
        m_with_it,
```

(continues on next page)

(continued from previous page)

```
        num_boost_round=rounds,
        evals=[(m_with_it, "Train")],
    )
    predict_with_it = reg_with_it.predict(m_with_it)

    reg = xgboost.train(
        {"tree_method": "hist", "device": "cuda"},
        m,
        num_boost_round=rounds,
        evals=[(m, "Train")],
    )
    predict = reg.predict(m)


if __name__ == "__main__":
    main()
```

Collection of examples for using xgboost.spark estimator interface

@author: Weichen Xu

```
import numpy as np
import sklearn.datasets
from pyspark.ml.evaluation import MulticlassClassificationEvaluator, RegressionEvaluator
from pyspark.ml.linalg import Vectors
from pyspark.sql import DataFrame, SparkSession
from pyspark.sql.functions import rand
from sklearn.model_selection import train_test_split

from xgboost.spark import SparkXGBClassifier, SparkXGBRegressor

spark = SparkSession.builder.master("local[*]").getOrCreate()


def create_spark_df(X: np.ndarray, y: np.ndarray) -> DataFrame:
    return spark.createDataFrame(
        spark.sparkContext.parallelize(
            [(Vectors.dense(features), float(label)) for features, label in zip(X, y)]
        ),
        ["features", "label"],
    )


# load diabetes dataset (regression dataset)
diabetes_X, diabetes_y = sklearn.datasets.load_diabetes(return_X_y=True)
diabetes_X_train, diabetes_X_test, diabetes_y_train, diabetes_y_test = train_test_split(
    diabetes_X, diabetes_y, test_size=0.3, shuffle=True
)

diabetes_train_spark_df = create_spark_df(diabetes_X_train, diabetes_y_train)
diabetes_test_spark_df = create_spark_df(diabetes_X_test, diabetes_y_test)
```

(continues on next page)

(continued from previous page)

```
# train xgboost regressor model
xgb_regressor = SparkXGBRegressor(max_depth=5)
xgb_regressor_model = xgb_regressor.fit(diabetes_train_spark_df)

transformed_diabetes_test_spark_df = xgb_regressor_model.transform(
    diabetes_test_spark_df
)
regressor_evaluator = RegressionEvaluator(metricName="rmse")
print(
    f"regressor rmse={regressor_evaluator.evaluate(transformed_diabetes_test_spark_df)}"
)

diabetes_train_spark_df2 = diabetes_train_spark_df.withColumn(
    "validationIndicatorCol", rand(1) > 0.7
)

# train xgboost regressor model with validation dataset
xgb_regressor2 = SparkXGBRegressor(
    max_depth=5, validation_indicator_col="validationIndicatorCol"
)
xgb_regressor_model2 = xgb_regressor2.fit(diabetes_train_spark_df2)
transformed_diabetes_test_spark_df2 = xgb_regressor_model2.transform(
    diabetes_test_spark_df
)
print(
    f"regressor2 rmse={regressor_evaluator.evaluate(transformed_diabetes_test_spark_df2)}
↪"
)


# load iris dataset (classification dataset)
iris_X, iris_y = sklearn.datasets.load_iris(return_X_y=True)
iris_X_train, iris_X_test, iris_y_train, iris_y_test = train_test_split(
    iris_X, iris_y, test_size=0.3, shuffle=True
)

iris_train_spark_df = create_spark_df(iris_X_train, iris_y_train)
iris_test_spark_df = create_spark_df(iris_X_test, iris_y_test)

# train xgboost classifier model
xgb_classifier = SparkXGBClassifier(max_depth=5)
xgb_classifier_model = xgb_classifier.fit(iris_train_spark_df)

transformed_iris_test_spark_df = xgb_classifier_model.transform(iris_test_spark_df)
classifier_evaluator = MulticlassClassificationEvaluator(metricName="f1")
print(f"classifier f1={classifier_evaluator.evaluate(transformed_iris_test_spark_df)}")

iris_train_spark_df2 = iris_train_spark_df.withColumn(
    "validationIndicatorCol", rand(1) > 0.7
)

# train xgboost classifier model with validation dataset
```

(continues on next page)

(continued from previous page)

```
xgb_classifier2 = SparkXGBClassifier(
    max_depth=5, validation_indicator_col="validationIndicatorCol"
)
xgb_classifier_model2 = xgb_classifier2.fit(iris_train_spark_df2)
transformed_iris_test_spark_df2 = xgb_classifier_model2.transform(iris_test_spark_df)
print(
    f"classifier2 f1={classifier_evaluator.evaluate(transformed_iris_test_spark_df2)}"
)

spark.stop()
```

Train XGBoost with cat_in_the_dat dataset

A simple demo for categorical data support using dataset from Kaggle categorical data tutorial.

The excellent tutorial is at: https://www.kaggle.com/shahules/an-overview-of-encoding-techniques

And the data can be found at: https://www.kaggle.com/shahules/an-overview-of-encoding-techniques/data

Added in version 1.6.0.

See Also

- *Tutorial*
- *Getting started with categorical data*
- *Feature engineering pipeline for categorical data*

```
from __future__ import annotations

import os
from tempfile import TemporaryDirectory
from time import time

import pandas as pd
from sklearn.metrics import roc_auc_score
from sklearn.model_selection import train_test_split

import xgboost as xgb


def load_cat_in_the_dat() -> tuple[pd.DataFrame, pd.Series]:
    """Assuming you have already downloaded the data into `input` directory."""

    df_train = pd.read_csv("./input/cat-in-the-dat/train.csv")

    print(
        "train data set has got {} rows and {} columns".format(
            df_train.shape[0], df_train.shape[1]
        )
    )
    X = df_train.drop(["target"], axis=1)
    y = df_train["target"]
```

(continues on next page)

(continued from previous page)

```
    for i in range(0, 5):
        X["bin_" + str(i)] = X["bin_" + str(i)].astype("category")

    for i in range(0, 5):
        X["nom_" + str(i)] = X["nom_" + str(i)].astype("category")

    for i in range(5, 10):
        X["nom_" + str(i)] = X["nom_" + str(i)].apply(int, base=16)

    for i in range(0, 6):
        X["ord_" + str(i)] = X["ord_" + str(i)].astype("category")

    print(
        "train data set has got {} rows and {} columns".format(X.shape[0], X.shape[1])
    )
    return X, y


params = {
    "tree_method": "hist",
    "device": "cuda",
    "n_estimators": 32,
    "colsample_bylevel": 0.7,
}


def categorical_model(X: pd.DataFrame, y: pd.Series, output_dir: str) -> None:
    """Train using builtin categorical data support from XGBoost"""
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, random_state=1994, test_size=0.2
    )
    # Be aware that the encoding for X_train and X_test are the same here. In practice,
    # we should try to use an encoder like (sklearn OrdinalEncoder) to obtain the
    # categorical values.

    # Specify `enable_categorical` to True.
    clf = xgb.XGBClassifier(
        **params,
        eval_metric="auc",
        enable_categorical=True,
        max_cat_to_onehot=1,  # We use optimal partitioning exclusively
    )
    clf.fit(X_train, y_train, eval_set=[(X_test, y_test), (X_train, y_train)])
    clf.save_model(os.path.join(output_dir, "categorical.json"))

    y_score = clf.predict_proba(X_test)[:, 1]  # proba of positive samples
    auc = roc_auc_score(y_test, y_score)
    print("AUC of using builtin categorical data support:", auc)


def onehot_encoding_model(X: pd.DataFrame, y: pd.Series, output_dir: str) -> None:
```

(continues on next page)

(continued from previous page)

```
    """Train using one-hot encoded data."""
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, random_state=42, test_size=0.2
    )
    # Specify `enable_categorical` to False as we are using encoded data.
    clf = xgb.XGBClassifier(**params, eval_metric="auc", enable_categorical=False)
    clf.fit(
        X_train,
        y_train,
        eval_set=[(X_test, y_test), (X_train, y_train)],
    )
    clf.save_model(os.path.join(output_dir, "one-hot.json"))

    y_score = clf.predict_proba(X_test)[:, 1]  # proba of positive samples
    auc = roc_auc_score(y_test, y_score)
    print("AUC of using onehot encoding:", auc)


if __name__ == "__main__":
    X, y = load_cat_in_the_dat()

    with TemporaryDirectory() as tmpdir:
        start = time()
        categorical_model(X, y, tmpdir)
        end = time()
        print("Duration:categorical", end - start)

        X = pd.get_dummies(X)
        start = time()
        onehot_encoding_model(X, y, tmpdir)
        end = time()
        print("Duration:onehot", end - start)
```

Demo for training continuation

```
import os
import pickle
import tempfile

from sklearn.datasets import load_breast_cancer

import xgboost


def training_continuation(tmpdir: str, use_pickle: bool) -> None:
    """Basic training continuation."""
    # Train 128 iterations in 1 session
    X, y = load_breast_cancer(return_X_y=True)
    clf = xgboost.XGBClassifier(n_estimators=128, eval_metric="logloss")
    clf.fit(X, y, eval_set=[(X, y)])
    print("Total boosted rounds:", clf.get_booster().num_boosted_rounds())
```

(continues on next page)

(continued from previous page)

```
    # Train 128 iterations in 2 sessions, with the first one runs for 32 iterations and
    # the second one runs for 96 iterations
    clf = xgboost.XGBClassifier(n_estimators=32, eval_metric="logloss")
    clf.fit(X, y, eval_set=[(X, y)])
    assert clf.get_booster().num_boosted_rounds() == 32

    # load back the model, this could be a checkpoint
    if use_pickle:
        path = os.path.join(tmpdir, "model-first-32.pkl")
        with open(path, "wb") as fd:
            pickle.dump(clf, fd)
        with open(path, "rb") as fd:
            loaded = pickle.load(fd)
    else:
        path = os.path.join(tmpdir, "model-first-32.json")
        clf.save_model(path)
        loaded = xgboost.XGBClassifier()
        loaded.load_model(path)

    clf = xgboost.XGBClassifier(n_estimators=128 - 32, eval_metric="logloss")
    clf.fit(X, y, eval_set=[(X, y)], xgb_model=loaded)

    print("Total boosted rounds:", clf.get_booster().num_boosted_rounds())

    assert clf.get_booster().num_boosted_rounds() == 128


def training_continuation_early_stop(tmpdir: str, use_pickle: bool) -> None:
    """Training continuation with early stopping."""
    early_stopping_rounds = 5
    early_stop = xgboost.callback.EarlyStopping(
        rounds=early_stopping_rounds, save_best=True
    )
    n_estimators = 512

    X, y = load_breast_cancer(return_X_y=True)
    clf = xgboost.XGBClassifier(
        n_estimators=n_estimators, eval_metric="logloss", callbacks=[early_stop]
    )
    clf.fit(X, y, eval_set=[(X, y)])
    print("Total boosted rounds:", clf.get_booster().num_boosted_rounds())
    best = clf.best_iteration

    # Train 512 iterations in 2 sessions, with the first one runs for 128 iterations and
    # the second one runs until early stop.
    clf = xgboost.XGBClassifier(
        n_estimators=128, eval_metric="logloss", callbacks=[early_stop]
    )
    # Reinitialize the early stop callback
    early_stop = xgboost.callback.EarlyStopping(
        rounds=early_stopping_rounds, save_best=True
```

(continues on next page)

(continued from previous page)

```
    )
    clf.set_params(callbacks=[early_stop])
    clf.fit(X, y, eval_set=[(X, y)])
    assert clf.get_booster().num_boosted_rounds() == 128

    # load back the model, this could be a checkpoint
    if use_pickle:
        path = os.path.join(tmpdir, "model-first-128.pkl")
        with open(path, "wb") as fd:
            pickle.dump(clf, fd)
        with open(path, "rb") as fd:
            loaded = pickle.load(fd)
    else:
        path = os.path.join(tmpdir, "model-first-128.json")
        clf.save_model(path)
        loaded = xgboost.XGBClassifier()
        loaded.load_model(path)

    early_stop = xgboost.callback.EarlyStopping(
        rounds=early_stopping_rounds, save_best=True
    )
    clf = xgboost.XGBClassifier(
        n_estimators=n_estimators - 128, eval_metric="logloss", callbacks=[early_stop]
    )
    clf.fit(
        X,
        y,
        eval_set=[(X, y)],
        xgb_model=loaded,
    )

    print("Total boosted rounds:", clf.get_booster().num_boosted_rounds())
    assert clf.best_iteration == best


if __name__ == "__main__":
    with tempfile.TemporaryDirectory() as tmpdir:
        training_continuation_early_stop(tmpdir, False)
        training_continuation_early_stop(tmpdir, True)

        training_continuation(tmpdir, True)
        training_continuation(tmpdir, False)
```

A demo for multi-output regression

The demo is adopted from scikit-learn:

https://scikit-learn.org/stable/auto_examples/ensemble/plot_random_forest_regression_multioutput.html#sphx-glr-auto-examples-ensemble-plot-random-forest-regression-multioutput-py

See *Multiple Outputs* for more information.

Note

The feature is experimental. For the *multi_output_tree* strategy, many features are missing.

See also

A demo for multi-output regression using reduced gradient

```
import argparse
from typing import Dict, List, Optional, Tuple

import matplotlib
import numpy as np
from matplotlib import pyplot as plt

import xgboost as xgb


def plot_predt(
    y: np.ndarray, y_predt: np.ndarray, name: str, ax: matplotlib.axes.Axes
) -> None:
    s = 25
    ax.scatter(y[:, 0], y[:, 1], c="navy", s=s, edgecolor="black", label=name)
    ax.scatter(y_predt[:, 0], y_predt[:, 1], c="cornflowerblue", s=s, edgecolor="black")
    ax.legend()


def gen_circle() -> Tuple[np.ndarray, np.ndarray]:
    "Generate a sample dataset that y is a 2 dim circle."
    rng = np.random.RandomState(1994)
    X = np.sort(200 * rng.rand(100, 1) - 100, axis=0)
    y = np.array([np.pi * np.sin(X).ravel(), np.pi * np.cos(X).ravel()]).T
    y[::5, :] += 0.5 - rng.rand(20, 2)
    y = y - y.min()
    y = y / y.max()
    return X, y


def rmse_model(strategy: str, ax: Optional[matplotlib.axes.Axes]) -> None:
    """Draw a circle with 2-dim coordinate as target variables."""
    X, y = gen_circle()
    # Train a regressor on it
    reg = xgb.XGBRegressor(
        tree_method="hist",
        n_estimators=128,
        n_jobs=16,
        max_depth=8,
        multi_strategy=strategy,
        subsample=0.6,
    )
```

(continues on next page)

(continued from previous page)

```
    reg.fit(X, y, eval_set=[(X, y)])

    y_predt = reg.predict(X)
    if ax:
        plot_predt(y, y_predt, f"RMSE-{strategy}", ax)


def custom_rmse_model(strategy: str, ax: Optional[matplotlib.axes.Axes]) -> None:
    """Train using Python implementation of Squared Error."""

    def gradient(predt: np.ndarray, dtrain: xgb.DMatrix) -> np.ndarray:
        """Compute the gradient squared error."""
        y = dtrain.get_label().reshape(predt.shape)
        return predt - y

    def hessian(predt: np.ndarray, dtrain: xgb.DMatrix) -> np.ndarray:
        """Compute the hessian for squared error."""
        return np.ones(predt.shape)

    def squared_log(
        predt: np.ndarray, dtrain: xgb.DMatrix
    ) -> Tuple[np.ndarray, np.ndarray]:
        grad = gradient(predt, dtrain)
        hess = hessian(predt, dtrain)
        # both numpy.ndarray and cupy.ndarray works.
        return grad, hess

    def rmse(predt: np.ndarray, dtrain: xgb.DMatrix) -> Tuple[str, float]:
        y = dtrain.get_label().reshape(predt.shape)
        v = np.sqrt(np.mean(np.power(y - predt, 2)))
        return "PyRMSE", v

    X, y = gen_circle()
    Xy = xgb.DMatrix(X, y)
    results: Dict[str, Dict[str, List[float]]] = {}
    # Make sure the `num_target` is passed to XGBoost when custom objective is used.
    # When builtin objective is used, XGBoost can figure out the number of targets
    # automatically.
    booster = xgb.train(
        {
            "tree_method": "hist",
            "num_target": y.shape[1],
            "multi_strategy": strategy,
        },
        dtrain=Xy,
        num_boost_round=128,
        obj=squared_log,
        evals=[(Xy, "Train")],
        evals_result=results,
        custom_metric=rmse,
    )
```

(continues on next page)

(continued from previous page)

```
    y_predt = booster.inplace_predict(X)
    if ax:
        plot_predt(y, y_predt, f"PyRMSE-{strategy}", ax)

    np.testing.assert_allclose(
        results["Train"]["rmse"], results["Train"]["PyRMSE"], rtol=1e-2
    )


if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--plot", choices=[0, 1], type=int, default=1)
    args = parser.parse_args()
    if args.plot == 1:
        _, axs = plt.subplots(2, 2)
    else:
        axs = np.full(shape=(2, 2), fill_value=None)
    assert isinstance(axs, np.ndarray)

    # Train with builtin RMSE objective
    # - One model per output.
    rmse_model("one_output_per_tree", axs[0, 0])
    # - One model for all outputs, this is still working in progress, many features are
    # missing.
    rmse_model("multi_output_tree", axs[0, 1])

    # Train with custom objective.
    # - One model per output.
    custom_rmse_model("one_output_per_tree", axs[1, 0])
    # - One model for all outputs, this is still working in progress, many features are
    # missing.
    custom_rmse_model("multi_output_tree", axs[1, 1])
    if args.plot == 1:
        plt.show()
```

Feature engineering pipeline for categorical data

The script showcases how to keep the categorical data encoding consistent across training and inference. There are many ways to attain the same goal, this script can be used as a starting point.

Changed in version 3.1: Start with 3.1, users don't need this for most of the cases. See *Auto-recoding (Data Consistency)* for more info.

See Also

- *Tutorial*
- *Getting started with categorical data*
- *Train XGBoost with cat_in_the_dat dataset*

```
from typing import List, Tuple
```

(continues on next page)

(continued from previous page)

```
import numpy as np
import pandas as pd
from sklearn.compose import make_column_selector, make_column_transformer
from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import OrdinalEncoder

import xgboost as xgb


def make_example_data() -> Tuple[pd.DataFrame, pd.Series, List[str]]:
    """Generate data for demo."""
    n_samples = 2048
    rng = np.random.default_rng(1994)

    # We have three categorical features, while the rest are numerical.
    categorical_features = ["brand_id", "retailer_id", "category_id"]

    df = pd.DataFrame(
        np.random.randint(32, 96, size=(n_samples, 3)),
        columns=categorical_features,
    )

    df["price"] = rng.integers(100, 200, size=(n_samples,))
    df["stock_status"] = rng.choice([True, False], n_samples)
    df["on_sale"] = rng.choice([True, False], n_samples)
    df["label"] = rng.normal(loc=0.0, scale=1.0, size=n_samples)

    X = df.drop(["label"], axis=1)
    y = df["label"]

    return X, y, categorical_features


def native() -> None:
    """Using the native XGBoost interface."""
    X, y, cat_feats = make_example_data()

    X_train, X_test, y_train, y_test = train_test_split(
        X, y, random_state=1994, test_size=0.2
    )

    # Create an encoder based on training data.
    enc = OrdinalEncoder(handle_unknown="use_encoded_value", unknown_value=np.nan)
    enc.set_output(transform="pandas")
    enc = enc.fit(X_train[cat_feats])

    def enc_transform(X: pd.DataFrame) -> pd.DataFrame:
        # don't make change inplace so that we can have demonstrations for encoding
        X = X.copy()
        cat_cols = enc.transform(X[cat_feats])
        for i, name in enumerate(cat_feats):
```

(continues on next page)

(continued from previous page)

```
            # create pd.Series based on the encoder
            cat_cols[name] = pd.Categorical.from_codes(
                codes=cat_cols[name].astype(np.int32), categories=enc.categories_[i]
            )
        X[cat_feats] = cat_cols
        return X

    # Encode the data based on fitted encoder.
    X_train_enc = enc_transform(X_train)
    X_test_enc = enc_transform(X_test)
    # Train XGBoost model using the native interface.
    Xy_train = xgb.QuantileDMatrix(X_train_enc, y_train, enable_categorical=True)
    Xy_test = xgb.QuantileDMatrix(
        X_test_enc, y_test, enable_categorical=True, ref=Xy_train
    )
    booster = xgb.train({}, Xy_train)
    booster.predict(Xy_test)

    # Following shows that data are encoded consistently.

    # We first obtain result from newly encoded data
    predt0 = booster.inplace_predict(enc_transform(X_train.head(16)))
    # then we obtain result from already encoded data from training.
    predt1 = booster.inplace_predict(X_train_enc.head(16))

    np.testing.assert_allclose(predt0, predt1)


def pipeline() -> None:
    """Using the sklearn pipeline."""
    X, y, cat_feats = make_example_data()

    X_train, X_test, y_train, y_test = train_test_split(
        X, y, random_state=3, test_size=0.2
    )

    enc = make_column_transformer(
        (
            OrdinalEncoder(handle_unknown="use_encoded_value", unknown_value=np.nan),
            # all categorical feature names end with "_id"
            make_column_selector(pattern=".*_id"),
        ),
        remainder="passthrough",
        verbose_feature_names_out=False,
    )
    # No need to set pandas output, we use `feature_types` to indicate the type of
    # features.

    # enc.set_output(transform="pandas")

    feature_types = ["c" if fn in cat_feats else "q" for fn in X_train.columns]
    reg = xgb.XGBRegressor(
```

(continues on next page)

(continued from previous page)

```
        feature_types=feature_types, enable_categorical=True, n_estimators=10
    )
    p = make_pipeline(enc, reg)
    p.fit(X_train, y_train)
    # check XGBoost is using the feature type correctly.
    model_types = reg.get_booster().feature_types
    assert model_types is not None
    for a, b in zip(model_types, feature_types):
        assert a == b

    # Following shows that data are encoded consistently.

    # We first create a slice of data that doesn't contain all the categories
    predt0 = p.predict(X_train.iloc[:16, :])
    # Then we use the dataframe that contains all the categories
    predt1 = p.predict(X_train)[:16]

    # The resulting encoding is the same
    np.testing.assert_allclose(predt0, predt1)


if __name__ == "__main__":
    pipeline()
    native()
```

Demo for using and defining callback functions

Added in version 1.3.0.

```
import argparse
import os
import tempfile
from typing import Dict

import numpy as np
from matplotlib import pyplot as plt
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split

import xgboost as xgb


class Plotting(xgb.callback.TrainingCallback):
    """Plot evaluation result during training.  Only for demonstration purpose as it's
    quite slow to draw using matplotlib.

    """

    def __init__(self, rounds: int) -> None:
        self.fig = plt.figure()
        self.ax = self.fig.add_subplot(111)
        self.rounds = rounds
```

(continues on next page)

(continued from previous page)

```
        self.lines: Dict[str, plt.Line2D] = {}
        self.fig.show()
        self.x = np.linspace(0, self.rounds, self.rounds)
        plt.ion()

    def _get_key(self, data: str, metric: str) -> str:
        return f"{data}-{metric}"

    def after_iteration(
        self, model: xgb.Booster, epoch: int, evals_log: Dict[str, dict]
    ) -> bool:
        """Update the plot."""
        if not self.lines:
            for data, metric in evals_log.items():
                for metric_name, log in metric.items():
                    key = self._get_key(data, metric_name)
                    expanded = log + [0] * (self.rounds - len(log))
                    (self.lines[key],) = self.ax.plot(self.x, expanded, label=key)
                    self.ax.legend()
        else:
            # https://pythonspot.com/matplotlib-update-plot/
            for data, metric in evals_log.items():
                for metric_name, log in metric.items():
                    key = self._get_key(data, metric_name)
                    expanded = log + [0] * (self.rounds - len(log))
                    self.lines[key].set_ydata(expanded)
            self.fig.canvas.draw()
        # False to indicate training should not stop.
        return False


def custom_callback() -> None:
    """Demo for defining a custom callback function that plots evaluation result during
    training."""
    X, y = load_breast_cancer(return_X_y=True)
    X_train, X_valid, y_train, y_valid = train_test_split(X, y, random_state=0)

    D_train = xgb.DMatrix(X_train, y_train)
    D_valid = xgb.DMatrix(X_valid, y_valid)

    num_boost_round = 100
    plotting = Plotting(num_boost_round)

    # Pass it to the `callbacks` parameter as a list.
    xgb.train(
        {
            "objective": "binary:logistic",
            "eval_metric": ["error", "rmse"],
            "tree_method": "hist",
            "device": "cuda",
        },
        D_train,
```

(continues on next page)

(continued from previous page)

```
        evals=[(D_train, "Train"), (D_valid, "Valid")],
        num_boost_round=num_boost_round,
        callbacks=[plotting],
    )


def check_point_callback() -> None:
    """Demo for using the checkpoint callback. Custom logic for handling output is
    usually required and users are encouraged to define their own callback for
    checkpointing operations. The builtin one can be used as a starting point.

    """
    # Only for demo, set a larger value (like 100) in practice as checkpointing is quite
    # slow.
    rounds = 2

    def check(as_pickle: bool) -> None:
        for i in range(0, 10, rounds):
            if i == 0:
                continue
            if as_pickle:
                path = os.path.join(tmpdir, "model_" + str(i) + ".pkl")
            else:
                path = os.path.join(
                    tmpdir,
                    f"model_{i}.{xgb.callback.TrainingCheckPoint.default_format}",
                )
            assert os.path.exists(path)

    X, y = load_breast_cancer(return_X_y=True)
    m = xgb.DMatrix(X, y)
    # Check point to a temporary directory for demo
    with tempfile.TemporaryDirectory() as tmpdir:
        # Use callback class from xgboost.callback
        # Feel free to subclass/customize it to suit your need.
        check_point = xgb.callback.TrainingCheckPoint(
            directory=tmpdir, interval=rounds, name="model"
        )
        xgb.train(
            {"objective": "binary:logistic"},
            m,
            num_boost_round=10,
            verbose_eval=False,
            callbacks=[check_point],
        )
        check(False)

        # This version of checkpoint saves everything including parameters and
        # model.  See: doc/tutorials/saving_model.rst
        check_point = xgb.callback.TrainingCheckPoint(
            directory=tmpdir, interval=rounds, as_pickle=True, name="model"
        )
```

(continues on next page)

(continued from previous page)

```
        xgb.train(
            {"objective": "binary:logistic"},
            m,
            num_boost_round=10,
            verbose_eval=False,
            callbacks=[check_point],
        )
        check(True)


if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--plot", default=1, type=int)
    args = parser.parse_args()

    check_point_callback()

    if args.plot:
        custom_callback()
```

Getting started with learning to rank

Added in version 2.0.0.

This is a demonstration of using XGBoost for learning to rank tasks using the MSLR_10k_letor dataset. For more infomation about the dataset, please visit its description page.

This is a two-part demo, the first one contains a basic example of using XGBoost to train on relevance degree, and the second part simulates click data and enable the position debiasing training.

For an overview of learning to rank in XGBoost, please see *Learning to Rank*.

```
from __future__ import annotations

import argparse
import json
import os
import pickle as pkl

import numpy as np
import pandas as pd
from sklearn.datasets import load_svmlight_file

import xgboost as xgb
from xgboost.testing.data import RelDataCV, simulate_clicks, sort_ltr_samples


def load_mslr_10k(data_path: str, cache_path: str) -> RelDataCV:
    """Load the MSLR10k dataset from data_path and cache a pickle object in cache_path.

    Returns
    -------
```

(continues on next page)

(continued from previous page)

```
    A list of tuples [(X, y, qid), ...].

    """
    root_path = os.path.expanduser(args.data)
    cacheroot_path = os.path.expanduser(args.cache)
    cache_path = os.path.join(cacheroot_path, "MSLR_10K_LETOR.pkl")

    # Use only the Fold1 for demo:
    # Train,      Valid, Test
    # {S1,S2,S3}, S4,    S5
    fold = 1

    if not os.path.exists(cache_path):
        fold_path = os.path.join(root_path, f"Fold{fold}")
        train_path = os.path.join(fold_path, "train.txt")
        valid_path = os.path.join(fold_path, "vali.txt")
        test_path = os.path.join(fold_path, "test.txt")
        X_train, y_train, qid_train = load_svmlight_file(
            train_path, query_id=True, dtype=np.float32
        )
        y_train = y_train.astype(np.int32)
        qid_train = qid_train.astype(np.int32)

        X_valid, y_valid, qid_valid = load_svmlight_file(
            valid_path, query_id=True, dtype=np.float32
        )
        y_valid = y_valid.astype(np.int32)
        qid_valid = qid_valid.astype(np.int32)

        X_test, y_test, qid_test = load_svmlight_file(
            test_path, query_id=True, dtype=np.float32
        )
        y_test = y_test.astype(np.int32)
        qid_test = qid_test.astype(np.int32)

        data = RelDataCV(
            train=(X_train, y_train, qid_train),
            test=(X_test, y_test, qid_test),
            max_rel=4,
        )

        with open(cache_path, "wb") as fd:
            pkl.dump(data, fd)

    with open(cache_path, "rb") as fd:
        data = pkl.load(fd)

    return data


def ranking_demo(args: argparse.Namespace) -> None:
    """Demonstration for learning to rank with relevance degree."""
```

(continues on next page)

(continued from previous page)

```
    data = load_mslr_10k(args.data, args.cache)

    # Sort data according to query index
    X_train, y_train, qid_train = data.train
    sorted_idx = np.argsort(qid_train)
    X_train = X_train[sorted_idx]
    y_train = y_train[sorted_idx]
    qid_train = qid_train[sorted_idx]

    X_test, y_test, qid_test = data.test
    sorted_idx = np.argsort(qid_test)
    X_test = X_test[sorted_idx]
    y_test = y_test[sorted_idx]
    qid_test = qid_test[sorted_idx]

    ranker = xgb.XGBRanker(
        tree_method="hist",
        device="cuda",
        lambdarank_pair_method="topk",
        lambdarank_num_pair_per_sample=13,
        eval_metric=["ndcg@1", "ndcg@8"],
    )
    ranker.fit(
        X_train,
        y_train,
        qid=qid_train,
        eval_set=[(X_test, y_test)],
        eval_qid=[qid_test],
        verbose=True,
    )


def click_data_demo(args: argparse.Namespace) -> None:
    """Demonstration for learning to rank with click data."""
    data = load_mslr_10k(args.data, args.cache)
    train, test = simulate_clicks(data)
    assert test is not None

    assert train.X.shape[0] == train.click.size
    assert test.X.shape[0] == test.click.size
    assert test.score.dtype == np.float32
    assert test.click.dtype == np.int32

    X_train, clicks_train, y_train, qid_train = sort_ltr_samples(
        train.X,
        train.y,
        train.qid,
        train.click,
        train.pos,
    )
    X_test, clicks_test, y_test, qid_test = sort_ltr_samples(
        test.X,
```

(continues on next page)

(continued from previous page)

```
    test.y,
    test.qid,
    test.click,
    test.pos,
)

class ShowPosition(xgb.callback.TrainingCallback):
    def after_iteration(
        self,
        model: xgb.Booster,
        epoch: int,
        evals_log: xgb.callback.TrainingCallback.EvalsLog,
    ) -> bool:
        config = json.loads(model.save_config())
        ti_plus = np.array(config["learner"]["objective"]["ti+"])
        tj_minus = np.array(config["learner"]["objective"]["tj-"])
        df = pd.DataFrame({"ti+": ti_plus, "tj-": tj_minus})
        print(df)
        return False

ranker = xgb.XGBRanker(
    n_estimators=512,
    tree_method="hist",
    device="cuda",
    learning_rate=0.01,
    reg_lambda=1.5,
    subsample=0.8,
    sampling_method="gradient_based",
    # LTR specific parameters
    objective="rank:ndcg",
    # - Enable bias estimation
    lambdarank_unbiased=True,
    # - normalization (1 / (norm + 1))
    lambdarank_bias_norm=1,
    # - Focus on the top 12 documents
    lambdarank_num_pair_per_sample=12,
    lambdarank_pair_method="topk",
    ndcg_exp_gain=True,
    eval_metric=["ndcg@1", "ndcg@3", "ndcg@5", "ndcg@10"],
    callbacks=[ShowPosition()],
)
ranker.fit(
    X_train,
    clicks_train,
    qid=qid_train,
    eval_set=[(X_test, y_test), (X_test, clicks_test)],
    eval_qid=[qid_test, qid_test],
    verbose=True,
)
ranker.predict(X_test)
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    parser = argparse.ArgumentParser(
        description="Demonstration of learning to rank using XGBoost."
    )
    parser.add_argument(
        "--data",
        type=str,
        help="Root directory of the MSLR-WEB10K data.",
        required=True,
    )
    parser.add_argument(
        "--cache",
        type=str,
        help="Directory for caching processed data.",
        required=True,
    )
    args = parser.parse_args()

    ranking_demo(args)
    click_data_demo(args)
```

Demo for defining a custom regression objective and metric

Demo for defining customized metric and objective. Notice that for simplicity reason weight is not used in following example. In this script, we implement the Squared Log Error (SLE) objective and RMSLE metric as customized functions, then compare it with native implementation in XGBoost.

See *Custom Objective and Evaluation Metric* for a step by step walkthrough, with other details.

The *SLE* objective reduces impact of outliers in training dataset, hence here we also compare its performance with standard squared error.

```
import argparse
from time import time
from typing import Dict, List, Tuple

import numpy as np
from matplotlib import pyplot as plt

import xgboost as xgb

# shape of generated data.
kRows = 4096
kCols = 16

kOutlier = 10000                # mean of generated outliers
kNumberOfOutliers = 64

kRatio = 0.7
kSeed = 1994

kBoostRound = 20
```

(continues on next page)

(continued from previous page)

```
np.random.seed(seed=kSeed)


def generate_data() -> Tuple[xgb.DMatrix, xgb.DMatrix]:
    '''Generate data containing outliers.'''
    x = np.random.randn(kRows, kCols)
    y = np.random.randn(kRows)
    y += np.abs(np.min(y))

    # Create outliers
    for i in range(0, kNumberOfOutliers):
        ind = np.random.randint(0, len(y)-1)
        y[ind] += np.random.randint(0, kOutlier)

    train_portion = int(kRows * kRatio)

    # rmsle requires all label be greater than -1.
    assert np.all(y > -1.0)

    train_x: np.ndarray = x[: train_portion]
    train_y: np.ndarray = y[: train_portion]
    dtrain = xgb.DMatrix(train_x, label=train_y)

    test_x = x[train_portion:]
    test_y = y[train_portion:]
    dtest = xgb.DMatrix(test_x, label=test_y)
    return dtrain, dtest


def native_rmse(dtrain: xgb.DMatrix,
                dtest: xgb.DMatrix) -> Dict[str, Dict[str, List[float]]]:
    '''Train using native implementation of Root Mean Squared Loss.'''
    print('Squared Error')
    squared_error = {
        'objective': 'reg:squarederror',
        'eval_metric': 'rmse',
        'tree_method': 'hist',
        'seed': kSeed
    }
    start = time()
    results: Dict[str, Dict[str, List[float]]] = {}
    xgb.train(squared_error,
              dtrain=dtrain,
              num_boost_round=kBoostRound,
              evals=[(dtrain, 'dtrain'), (dtest, 'dtest')],
              evals_result=results)
    print('Finished Squared Error in:', time() - start, '\n')
    return results


def native_rmsle(dtrain: xgb.DMatrix,
                 dtest: xgb.DMatrix) -> Dict[str, Dict[str, List[float]]]:
```

(continues on next page)

(continued from previous page)

```
    '''Train using native implementation of Squared Log Error.'''
    print('Squared Log Error')
    results: Dict[str, Dict[str, List[float]]] = {}
    squared_log_error = {
        'objective': 'reg:squaredlogerror',
        'eval_metric': 'rmsle',
        'tree_method': 'hist',
        'seed': kSeed
    }
    start = time()
    xgb.train(squared_log_error,
              dtrain=dtrain,
              num_boost_round=kBoostRound,
              evals=[(dtrain, 'dtrain'), (dtest, 'dtest')],
              evals_result=results)
    print('Finished Squared Log Error in:', time() - start)
    return results


def py_rmsle(dtrain: xgb.DMatrix, dtest: xgb.DMatrix) -> Dict:
    '''Train using Python implementation of Squared Log Error.'''
    def gradient(predt: np.ndarray, dtrain: xgb.DMatrix) -> np.ndarray:
        '''Compute the gradient squared log error.'''
        y = dtrain.get_label()
        return (np.log1p(predt) - np.log1p(y)) / (predt + 1)

    def hessian(predt: np.ndarray, dtrain: xgb.DMatrix) -> np.ndarray:
        '''Compute the hessian for squared log error.'''
        y = dtrain.get_label()
        return ((-np.log1p(predt) + np.log1p(y) + 1) /
                np.power(predt + 1, 2))

    def squared_log(predt: np.ndarray,
                    dtrain: xgb.DMatrix) -> Tuple[np.ndarray, np.ndarray]:
        '''Squared Log Error objective. A simplified version for RMSLE used as
        objective function.

        :math:`\frac{1}{2}[log(pred + 1) - log(label + 1)]^2`

        '''
        predt[predt < -1] = -1 + 1e-6
        grad = gradient(predt, dtrain)
        hess = hessian(predt, dtrain)
        return grad, hess

    def rmsle(predt: np.ndarray, dtrain: xgb.DMatrix) -> Tuple[str, float]:
        ''' Root mean squared log error metric.

        :math:`\\sqrt{\frac{1}{N}[log(pred + 1) - log(label + 1)]^2}`
        '''
        y = dtrain.get_label()
        predt[predt < -1] = -1 + 1e-6
```

(continues on next page)

(continued from previous page)

```
        elements = np.power(np.log1p(y) - np.log1p(predt), 2)
        return 'PyRMSLE', float(np.sqrt(np.sum(elements) / len(y)))

    results: Dict[str, Dict[str, List[float]]] = {}
    xgb.train({'tree_method': 'hist', 'seed': kSeed,
               'disable_default_eval_metric': 1},
              dtrain=dtrain,
              num_boost_round=kBoostRound,
              obj=squared_log,
              custom_metric=rmsle,
              evals=[(dtrain, 'dtrain'), (dtest, 'dtest')],
              evals_result=results)

    return results


def plot_history(
    rmse_evals: Dict[str, Dict],
    rmsle_evals: Dict[str, Dict],
    py_rmsle_evals: Dict[str, Dict]
) -> None:
    fig, axs = plt.subplots(3, 1)
    assert isinstance(axs, np.ndarray)
    ax0 = axs[0]
    ax1 = axs[1]
    ax2 = axs[2]

    x = np.arange(0, kBoostRound, 1)

    ax0.plot(x, rmse_evals['dtrain']['rmse'], label='train-RMSE')
    ax0.plot(x, rmse_evals['dtest']['rmse'], label='test-RMSE')
    ax0.legend()

    ax1.plot(x, rmsle_evals['dtrain']['rmsle'], label='train-native-RMSLE')
    ax1.plot(x, rmsle_evals['dtest']['rmsle'], label='test-native-RMSLE')
    ax1.legend()

    ax2.plot(x, py_rmsle_evals['dtrain']['PyRMSLE'], label='train-PyRMSLE')
    ax2.plot(x, py_rmsle_evals['dtest']['PyRMSLE'], label='test-PyRMSLE')
    ax2.legend()


def main(args: argparse.Namespace) -> None:
    dtrain, dtest = generate_data()
    rmse_evals = native_rmse(dtrain, dtest)
    rmsle_evals = native_rmsle(dtrain, dtest)
    py_rmsle_evals = py_rmsle(dtrain, dtest)

    if args.plot != 0:
        plot_history(rmse_evals, rmsle_evals, py_rmsle_evals)
        plt.show()
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    parser = argparse.ArgumentParser(
        description='Arguments for custom RMSLE objective function demo.')
    parser.add_argument(
        '--plot',
        type=int,
        default=1,
        help='Set to 0 to disable plotting the evaluation history.')
    args = parser.parse_args()
    main(args)
```

Demo for creating customized multi-class objective function

This demo is only applicable after (excluding) XGBoost 1.0.0, as before this version XGBoost returns transformed prediction for multi-class objective function. More details in comments.

See *Custom Objective and Evaluation Metric* and *Advanced Usage of Custom Objectives* for detailed tutorial and notes.

```
# pylint: disable=missing-function-docstring,redefined-outer-name,unused-variable

import argparse
from typing import Dict, Tuple

import numpy as np
import xgboost as xgb
from matplotlib import pyplot as plt

np.random.seed(1994)

kRows = 100
kCols = 10
kClasses = 4  # number of classes

kRounds = 10  # number of boosting rounds.

# Generate some random data for demo.
X = np.random.randn(kRows, kCols)
y = np.random.randint(0, 4, size=kRows)

m = xgb.DMatrix(X, y)


def softmax(x: np.ndarray) -> np.ndarray:
    """Softmax function with x as input vector."""
    e = np.exp(x)
    return e / np.sum(e)


def softprob_obj(predt: np.ndarray, data: xgb.DMatrix) -> Tuple[np.ndarray, np.ndarray]:
    """Loss function. Computing the gradient and upper bound on the
    Hessian with a diagonal structure for XGBoost (note that this is
```

(continues on next page)

(continued from previous page)

```
    not the true Hessian).
    Reimplements the `multi:softprob` inside XGBoost.

    """
    labels = data.get_label()
    if data.get_weight().size == 0:
        # Use 1 as weight if we don't have custom weight.
        weights = np.ones(kRows, dtype=float)
    else:
        weights = data.get_weight()

    # The prediction is of shape (rows, classes), each element in a row
    # represents a raw prediction (leaf weight, hasn't gone through softmax
    # yet).  In XGBoost 1.0.0, the prediction is transformed by a softmax
    # function, fixed in later versions.
    assert predt.shape == (kRows, kClasses)

    grad = np.zeros((kRows, kClasses), dtype=float)
    hess = np.zeros((kRows, kClasses), dtype=float)

    eps = 1e-6

    # compute the gradient and hessian upper bound, slow iterations in Python, only
    # suitable for demo.  Also the one in native XGBoost core is more robust to
    # numeric overflow as we don't do anything to mitigate the `exp` in
    # `softmax` here.
    for r in range(predt.shape[0]):
        target = int(labels[r])
        weight = float(weights[r])
        p = softmax(predt[r, :])
        for c in range(predt.shape[1]):
            assert 0 <= target < kClasses
            pc = float(p[c])
            g = pc - 1.0 if c == target else pc
            g = g * weight
            h = max(2.0 * pc * (1.0 - pc) * weight, eps)
            grad[r, c] = g
            hess[r, c] = h

    # After 2.1.0, pass the gradient as it is.
    return grad, hess


def predict(booster: xgb.Booster, X: xgb.DMatrix) -> np.ndarray:
    """A customized prediction function that converts raw prediction to
    target class.

    """
    # Output margin means we want to obtain the raw prediction obtained from
    # tree leaf weight.
    predt = booster.predict(X, output_margin=True)
    out = np.zeros(kRows)
```

(continues on next page)

(continued from previous page)

```
    for r in range(predt.shape[0]):
        # the class with maximum prob (not strictly prob as it haven't gone
        # through softmax yet so it doesn't sum to 1, but result is the same
        # for argmax).
        i = np.argmax(predt[r])
        out[r] = i
    return out


def merror(predt: np.ndarray, dtrain: xgb.DMatrix) -> Tuple[str, np.float64]:
    y = dtrain.get_label()
    # Like custom objective, the predt is untransformed leaf weight when custom
    # objective is provided.

    # With the use of `custom_metric` parameter in train function, custom metric
    # receives raw input only when custom objective is also being used.  Otherwise
    # custom metric will receive transformed prediction.
    assert predt.shape == (kRows, kClasses)
    out = np.zeros(kRows)
    for r in range(predt.shape[0]):
        i = np.argmax(predt[r])
        out[r] = i

    assert y.shape == out.shape

    errors = np.zeros(kRows)
    errors[y != out] = 1.0
    return "PyMError", np.sum(errors) / kRows


def plot_history(
    custom_results: Dict[str, Dict], native_results: Dict[str, Dict]
) -> None:
    axs: np.ndarray
    fig, axs = plt.subplots(2, 1)  # type: ignore
    ax0 = axs[0]
    ax1 = axs[1]

    pymerror = custom_results["train"]["PyMError"]
    merror = native_results["train"]["merror"]

    x = np.arange(0, kRounds, 1)
    ax0.plot(x, pymerror, label="Custom objective")
    ax0.legend()
    ax1.plot(x, merror, label="multi:softmax")
    ax1.legend()

    plt.show()


def main(args: argparse.Namespace) -> None:
    # Since 3.1, XGBoost can estimate the base_score automatically for built-in
```

(continues on next page)

(continued from previous page)

```
    # multi-class objectives.
    #
    # We explicitly specify it here to disable the automatic estimation to have a proper
    # comparison between the custom implementation and the built-in implementation.
    intercept = np.full(shape=(kClasses,), fill_value=1 / kClasses)

    custom_results: Dict[str, Dict] = {}
    # Use our custom objective function
    booster_custom = xgb.train(
        {
            "num_class": kClasses,
            "base_score": intercept,
            "disable_default_eval_metric": True,
        },
        m,
        num_boost_round=kRounds,
        obj=softprob_obj,
        custom_metric=merror,
        evals_result=custom_results,
        evals=[(m, "train")],
    )

    predt_custom = predict(booster_custom, m)

    native_results: Dict[str, Dict] = {}
    # Use the same objective function defined in XGBoost.
    booster_native = xgb.train(
        {
            "num_class": kClasses,
            "base_score": intercept,
            "objective": "multi:softmax",
            "eval_metric": "merror",
        },
        m,
        num_boost_round=kRounds,
        evals_result=native_results,
        evals=[(m, "train")],
    )
    predt_native = booster_native.predict(m)

    # We are reimplementing the loss function in XGBoost, so it should
    # be the same for normal cases.
    assert np.all(predt_custom == predt_native)
    np.testing.assert_allclose(
        custom_results["train"]["PyMError"], native_results["train"]["merror"]
    )

    if args.plot != 0:
        plot_history(custom_results, native_results)


if __name__ == "__main__":
```

(continues on next page)

(continued from previous page)

```
    parser = argparse.ArgumentParser(
        description="Arguments for custom softmax objective function demo."
    )
    parser.add_argument(
        "--plot",
        type=int,
        default=1,
        help="Set to 0 to disable plotting the evaluation history.",
    )
    args = parser.parse_args()
    main(args)
```

Prediction Intervals with Quantile and Expectile Regression

Added in version 2.0.0.

The script is inspired by this awesome example in sklearn: https://scikit-learn.org/stable/auto_examples/ensemble/plot_gradient_boosting_quantile.html

Note

The feature is only supported using the Python, R, and C packages. In addition, quantile crossing can happen due to limitation in the algorithm.

This example also trains `reg:expectileerror`. Expectiles are asymmetric means, not percentiles, but they can be used to construct tail-sensitive bands around the conditional mean.

Added in version 3.3.0.

```
import argparse
from typing import Dict, Tuple

import numpy as np
import xgboost as xgb
from sklearn.model_selection import train_test_split

Data = Tuple[xgb.QuantileDMatrix, xgb.QuantileDMatrix]
Predictions = Dict[str, np.ndarray]


def f(x: np.ndarray) -> np.ndarray:
    """The function to predict."""
    return x * np.sin(x)


def make_dataset(rng: np.random.RandomState) -> Tuple[np.ndarray, np.ndarray]:
    """Generate heteroscedastic data with asymmetric noise."""
    features = np.atleast_2d(rng.uniform(0, 10.0, size=1000)).T
    expected_y = f(features).ravel()

    sigma = 0.5 + features.ravel() / 10.0
    noise = rng.lognormal(sigma=sigma) - np.exp(sigma**2.0 / 2.0)
```

(continues on next page)

(continued from previous page)

```
    target = expected_y + noise
    return features, target


def train_interval_model(
    params: Dict[str, object],
    alpha_name: str,
    alpha: np.ndarray,
    data: Data,
) -> Tuple[xgb.Booster, Dict[str, Dict]]:
    """Train a multi-output interval model."""
    dtrain, dtest = data
    evals_result: Dict[str, Dict] = {}
    params = params.copy()
    params[alpha_name] = alpha
    booster = xgb.train(
        params,
        dtrain,
        num_boost_round=64,
        early_stopping_rounds=4,
        evals=[(dtrain, "Train"), (dtest, "Test")],
        evals_result=evals_result,
    )
    return booster, evals_result


def squared_error_model(params: Dict[str, object], data: Data) -> xgb.Booster:
    """Train a squared-error model for comparison."""
    dtrain, dtest = data
    params = params.copy()
    params["objective"] = "reg:squarederror"
    return xgb.train(
        params,
        dtrain,
        num_boost_round=64,
        early_stopping_rounds=4,
        evals=[(dtrain, "Train"), (dtest, "Test")],
    )


def base_params(cli_args: argparse.Namespace) -> Dict[str, object]:
    """Parameters shared by the three models in the demo."""
    return {
        "tree_method": "hist",
        "learning_rate": 0.04,
        "max_depth": 5,
        "multi_strategy": cli_args.multi_strategy,
        "device": cli_args.device,
    }


def train_models(
```

(continues on next page)

(continued from previous page)

```
    data: Data, alpha: np.ndarray, params: Dict[str, object]
) -> Tuple[Dict[str, xgb.Booster], Dict[str, Dict[str, Dict]]]:
    """Train quantile, expectile, and squared-error models."""
    quantile_model, quantile_evals = train_interval_model(
        {**params, "objective": "reg:quantileerror"}, "quantile_alpha", alpha, data
    )
    expectile_model, expectile_evals = train_interval_model(
        {**params, "objective": "reg:expectileerror"}, "expectile_alpha", alpha, data
    )
    models = {
        "quantile": quantile_model,
        "expectile": expectile_model,
        "mean": squared_error_model(params, data),
    }
    return models, {"quantile": quantile_evals, "expectile": expectile_evals}


def make_predictions(models: Dict[str, xgb.Booster]) -> Tuple[np.ndarray, Predictions]:
    """Predict on a dense grid for plotting."""
    grid = np.atleast_2d(np.linspace(0, 10, 1000)).T
    predictions = {
        "quantile": models["quantile"].inplace_predict(grid),
        "expectile": models["expectile"].inplace_predict(grid),
        "mean": models["mean"].inplace_predict(grid),
    }
    return grid, predictions


def make_matrices(
    rng: np.random.RandomState,
) -> Tuple[Data, Tuple[np.ndarray, np.ndarray]]:
    """Create train/test DMatrices and return held-out data for plotting."""
    features, target = make_dataset(rng)
    split = train_test_split(features, target, random_state=rng)
    train_features, test_features, train_target, test_target = split
    train = xgb.QuantileDMatrix(train_features, train_target)
    test = xgb.QuantileDMatrix(test_features, test_target, ref=train)
    return (train, test), (test_features, test_target)


def plot_prediction_intervals(
    grid: np.ndarray,
    test_data: Tuple[np.ndarray, np.ndarray],
    predictions: Predictions,
    output: str,
) -> None:
    """Plot quantile interval and expectile band."""
    from matplotlib import pyplot as plt

    features, target = test_data
    fig, axes = plt.subplots(2, 1, figsize=(10, 12), sharex=True, sharey=True)
```

(continues on next page)

(continued from previous page)

```
    def plot_band(
        ax: "plt.Axes", pred: np.ndarray, title: str, center_label: str
    ) -> None:
        ax.plot(grid, f(grid), "g:", linewidth=3, label=r"$f(x) = x\,\sin(x)$")
        ax.plot(
            features,
            target,
            "b.",
            markersize=5,
            alpha=0.35,
            label="Test observations",
        )
        ax.plot(grid, pred[:, 1], "r-", label=center_label)
        ax.plot(grid, predictions["mean"], "m--", label="Squared-error mean")
        ax.plot(grid, pred[:, 0], "k-")
        ax.plot(grid, pred[:, 2], "k-")
        ax.fill_between(
            grid.ravel(), pred[:, 0], pred[:, 2], alpha=0.3, label="90% band"
        )
        ax.set_title(title)
        ax.set_ylabel("$y$")
        ax.set_ylim(-10, 25)
        ax.legend(loc="upper left")

    plot_band(
        axes[0],
        predictions["quantile"],
        "Quantile regression interval",
        "Predicted median",
    )
    plot_band(
        axes[1],
        predictions["expectile"],
        "Expectile regression band",
        "Predicted 0.5 expectile",
    )
    axes[1].set_xlabel("$x$")
    fig.tight_layout()

    if output:
        fig.savefig(output, dpi=150)
    else:
        plt.show()


def prediction_intervals(cli_args: argparse.Namespace) -> None:
    """Train quantile and expectile interval models."""
    rng = np.random.RandomState(1994)
    alpha = np.array([0.05, 0.5, 0.95])

    data, test_data = make_matrices(rng)
    models, evals = train_models(data, alpha, base_params(cli_args))
```

(continues on next page)

(continued from previous page)

```
    grid, predictions = make_predictions(models)

    assert predictions["quantile"].shape == (grid.shape[0], alpha.shape[0])
    assert predictions["expectile"].shape == (grid.shape[0], alpha.shape[0])

    print("Quantile test metric:", evals["quantile"]["Test"]["quantile"][-1])
    print("Expectile test metric:", evals["expectile"]["Test"]["expectile"][-1])

    if cli_args.plot:
        plot_prediction_intervals(grid, test_data, predictions, cli_args.output)


if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument(
        "--plot",
        action="store_true",
        help="Specify it to enable plotting the outputs.",
    )
    parser.add_argument(
        "--output",
        type=str,
        default="",
        help="Optional path for saving the plot instead of displaying it.",
    )
    parser.add_argument(
        "--multi_strategy",
        choices=["multi_output_tree", "one_output_per_tree"],
        default="one_output_per_tree",
        help="See the parameter `multi_strategy` for more info. (Experimental)",
    )
    parser.add_argument("--device", choices=["cpu", "cuda"], default="cpu")
    prediction_intervals(parser.parse_args())
```

Experimental support for external memory

This is similar to the one in *quantile_data_iterator.py*, but for external memory instead of Quantile DMatrix. The feature is not ready for production use yet.

Added in version 1.5.0.

See *the tutorial* for more details.

Changed in version 3.0.0: Added `ExtMemQuantileDMatrix`.

To run the example, following packages in addition to XGBoost native dependencies are required:

- scikit-learn

If *device* is *cuda*, following are also needed:

- cupy
- rmm
- cuda-python

See also

Experimental support for distributed training with external memory

Not shown in this example, but you should pay attention to NUMA configuration as discussed in the tutorial.

```
import argparse
import os
import tempfile
from typing import TYPE_CHECKING, Callable, List, Literal, Tuple

import numpy as np
import xgboost
from sklearn.datasets import make_regression

if TYPE_CHECKING:
    from cuda.bindings.runtime import cudaError_t


def _checkcu(status: "cudaError_t") -> None:
    import cuda.bindings.runtime as cudart

    if status != cudart.cudaError_t.cudaSuccess:
        raise RuntimeError(cudart.cudaGetErrorString(status))


def device_mem_total() -> int:
    """The total number of bytes of memory this GPU has."""
    import cuda.bindings.runtime as cudart

    status, _, total = cudart.cudaMemGetInfo()
    _checkcu(status)
    return total


def make_batches(
    n_samples_per_batch: int,
    n_features: int,
    n_batches: int,
    work_dir: str,
) -> List[Tuple[str, str]]:
    """Write ``n_batches`` synthetic regression batches as `.npy` pairs under ``work_dir``.""
↪"
    files: List[Tuple[str, str]] = []
    rng = np.random.RandomState(1994)
    for i in range(n_batches):
        X, y = make_regression(n_samples_per_batch, n_features, random_state=rng)
        X_path = os.path.join(work_dir, "X-" + str(i) + ".npy")
        y_path = os.path.join(work_dir, "y-" + str(i) + ".npy")
        np.save(X_path, X)
        np.save(y_path, y)
        files.append((X_path, y_path))
```

(continues on next page)

(continued from previous page)

```
    return files


class Iterator(xgboost.DataIter):
    """A custom iterator for loading files in batches."""

    def __init__(
        self, device: Literal["cpu", "cuda"], file_paths: List[Tuple[str, str]]
    ) -> None:
        self.device = device

        self._file_paths = file_paths
        self._it = 0
        # XGBoost will generate some cache files under the current directory with the
        # prefix "cache"
        super().__init__(cache_prefix=os.path.join(".", "cache"))

    def load_file(self) -> Tuple[np.ndarray, np.ndarray]:
        """Load a single batch of data."""
        X_path, y_path = self._file_paths[self._it]
        # When the `ExtMemQuantileDMatrix` is used, the device must match. GPU cannot
        # consume CPU input data and vice-versa.
        if self.device == "cpu":
            X = np.load(X_path)
            y = np.load(y_path)
        else:
            import cupy as cp  # pylint: disable=import-outside-toplevel

            X = cp.load(X_path)
            y = cp.load(y_path)

        assert X.shape[0] == y.shape[0]
        return X, y

    def next(self, input_data: Callable) -> bool:
        """Advance the iterator by 1 step and pass the data to XGBoost.  This function
        is called by XGBoost during the construction of ``DMatrix``

        """
        if self._it == len(self._file_paths):
            # return False to let XGBoost know this is the end of iteration
            return False

        # input_data is a keyword-only function passed in by XGBoost and has the similar
        # signature to the ``DMatrix`` constructor.
        X, y = self.load_file()
        input_data(data=X, label=y)
        self._it += 1
        return True

    def reset(self) -> None:
        """Reset the iterator to its beginning"""
```

(continues on next page)

(continued from previous page)

```
        self._it = 0


def hist_train(it: Iterator) -> None:
    """The hist tree method can use a special data structure `ExtMemQuantileDMatrix` for
    faster initialization and lower memory usage (recommended).

    .. versionadded:: 3.0.0

    """
    # For non-data arguments, specify it here once instead of passing them by the `next`
    # method.
    Xy = xgboost.ExtMemQuantileDMatrix(it, missing=np.nan, enable_categorical=False)
    booster = xgboost.train(
        {"tree_method": "hist", "max_depth": 4, "device": it.device},
        Xy,
        evals=[(Xy, "Train")],
        num_boost_round=10,
    )
    booster.predict(Xy)


def approx_train(it: Iterator) -> None:
    """The approx tree method uses the basic `DMatrix` (not recommended)."""

    # For non-data arguments, specify it here once instead of passing them by the `next`
    # method.
    Xy = xgboost.DMatrix(it, missing=np.nan, enable_categorical=False)
    # ``approx`` is also supported, but less efficient due to sketching. It's
    # recommended to use `hist` instead.
    booster = xgboost.train(
        {"tree_method": "approx", "max_depth": 4, "device": it.device},
        Xy,
        evals=[(Xy, "Train")],
        num_boost_round=10,
    )
    booster.predict(Xy)


def main(work_dir: str, cli_args: argparse.Namespace) -> None:
    """Entry point for training."""

    # generate some random data for demo
    files = make_batches(
        n_samples_per_batch=1024, n_features=17, n_batches=31, work_dir=work_dir
    )
    it = Iterator(cli_args.device, files)

    hist_train(it)
    approx_train(it)
```

(continues on next page)

(continued from previous page)

```
def setup_async_pool() -> None:
    """Setup CUDA async pool. As an alternative, the RMM plugin can be used as well. See
    the `setup_rmm`. This is the same as using the `CudaAsyncMemoryResource` from RMM,
    but without the RMM dependency.

    .. versionadded:: 3.2.0

    """
    import cuda.bindings.runtime as cudart
    import cupy as cp  # pylint: disable=import-outside-toplevel
    from cuda.bindings import driver
    from cupy.cuda import MemoryAsyncPool

    status, dft_pool = cudart.cudaDeviceGetDefaultMemPool(0)
    _checkcu(status)

    total = device_mem_total()

    v = driver.cuuint64_t(int(total * 0.9))
    (status,) = cudart.cudaMemPoolSetAttribute(
        dft_pool,
        cudart.cudaMemPoolAttr.cudaMemPoolAttrReleaseThreshold,
        v,
    )
    _checkcu(status)
    # Set the allocator for cupy as well.
    cp.cuda.set_allocator(MemoryAsyncPool().malloc)


def setup_rmm() -> None:
    """Setup RMM for GPU-based external memory training.

    It's important to use RMM with `CudaAsyncMemoryResource` or `ArenaMemoryResource`
    for GPU-based external memory to improve performance. If XGBoost is not built with
    RMM support, a warning is raised when constructing the `DMatrix`.

    """

    import rmm
    from rmm.allocators.cupy import rmm_cupy_allocator
    from rmm.mr import ArenaMemoryResource

    if not xgboost.build_info()["USE_RMM"]:
        return

    import cupy as cp  # pylint: disable=import-outside-toplevel

    total = device_mem_total()

    mr: rmm.mr.DeviceMemoryResource = rmm.mr.CudaMemoryResource()
    mr = ArenaMemoryResource(mr, arena_size=int(total * 0.9))
```

(continues on next page)

(continued from previous page)

```
    rmm.mr.set_current_device_resource(mr)
    # Set the allocator for cupy as well.
    cp.cuda.set_allocator(rmm_cupy_allocator)


if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--device", choices=["cpu", "cuda"], default="cpu")
    parser.add_argument(
        "--memory_pool",
        choices=["rmm", "cuda"],
        default="rmm",
        help="Use a memory pool for asynchronous memory allocation in XGBoost.",
    )
    args = parser.parse_args()
    if args.device == "cuda":
        if args.memory_pool == "rmm":
            setup_rmm()
        elif args.memory_pool == "cuda":
            setup_async_pool()
        # Make sure XGBoost is using RMM for all allocations.
        with xgboost.config_context(
            use_rmm=args.memory_pool == "rmm",
            use_cuda_async_pool=args.memory_pool == "cuda",
        ):
            with tempfile.TemporaryDirectory() as tmpdir:
                main(tmpdir, args)
    else:
        with tempfile.TemporaryDirectory() as tmpdir:
            main(tmpdir, args)
```

Experimental support for distributed training with external memory

Added in version 3.0.0.

See *the tutorial* for more details. To run the example, following packages in addition to XGBoost native dependencies are required:

- scikit-learn
- loky

If *device* is *cuda*, following are also needed:

- cupy
- cuda-python
- pyhwloc

```
import argparse
import multiprocessing as mp
import os
import sys
import tempfile
```

(continues on next page)

(continued from previous page)

```
import traceback
from functools import partial, update_wrapper, wraps
from typing import TYPE_CHECKING, Callable, List, ParamSpec, Tuple, TypeVar

import numpy as np
import xgboost
from loky import get_reusable_executor
from sklearn.datasets import make_regression
from xgboost import collective as coll
from xgboost.tracker import RabitTracker

if TYPE_CHECKING:
    from cuda.bindings.runtime import cudaError_t


def _checkcu(status: "cudaError_t") -> None:
    import cuda.bindings.runtime as cudart

    if status != cudart.cudaError_t.cudaSuccess:
        raise RuntimeError(cudart.cudaGetErrorString(status))


def device_mem_total() -> int:
    """The total number of bytes of memory this GPU has."""
    import cuda.bindings.runtime as cudart

    status, _, total = cudart.cudaMemGetInfo()
    _checkcu(status)
    return total


def make_batches(
    n_samples_per_batch: int, n_features: int, n_batches: int, tmpdir: str, rank: int
) -> List[Tuple[str, str]]:
    """Create multiple batches of synthetic data and return their file paths."""
    files: List[Tuple[str, str]] = []
    rng = np.random.RandomState(rank)
    for i in range(n_batches):
        X, y = make_regression(n_samples_per_batch, n_features, random_state=rng)
        X_path = os.path.join(tmpdir, f"X-r{rank}-{i}.npy")
        y_path = os.path.join(tmpdir, f"y-r{rank}-{i}.npy")
        np.save(X_path, X)
        np.save(y_path, y)
        files.append((X_path, y_path))
    return files


class Iterator(xgboost.DataIter):
    """A custom iterator for loading files in batches."""

    def __init__(self, device: str, file_paths: List[Tuple[str, str]]) -> None:
        self.device = device
```

(continues on next page)

(continued from previous page)

```
        self._file_paths = file_paths
        self._it = 0
        # XGBoost will generate some cache files under the current directory with the
        # prefix "cache"
        super().__init__(cache_prefix=os.path.join(".", "cache"))

    def load_file(self) -> Tuple[np.ndarray, np.ndarray]:
        """Load a single batch of data."""
        X_path, y_path = self._file_paths[self._it]
        # When the `ExtMemQuantileDMatrix` is used, the device must match. GPU cannot
        # consume CPU input data and vice-versa.
        if self.device == "cpu":
            X = np.load(X_path)
            y = np.load(y_path)
        else:
            import cupy as cp

            X = cp.load(X_path)
            y = cp.load(y_path)

        assert X.shape[0] == y.shape[0]
        return X, y

    def next(self, input_data: Callable) -> bool:
        """Advance the iterator by 1 step and pass the data to XGBoost.  This function
        is called by XGBoost during the construction of ``DMatrix``

        """
        if self._it == len(self._file_paths):
            # return False to let XGBoost know this is the end of iteration
            return False

        # input_data is a keyword-only function passed in by XGBoost and has the similar
        # signature to the ``DMatrix`` constructor.
        X, y = self.load_file()
        input_data(data=X, label=y)
        self._it += 1
        return True

    def reset(self) -> None:
        """Reset the iterator to its beginning"""
        self._it = 0


def setup_numa() -> None:
    """Set correct NUMA binding for GPU-based external memory training."""
    from pyhwloc import from_this_system
    from pyhwloc.cuda_runtime import get_device
    from pyhwloc.topology import MemBindFlags, MemBindPolicy, TypeFilter

    devices = os.getenv("CUDA_VISIBLE_DEVICES", None)
```

(continues on next page)

(continued from previous page)

```
    assert devices is not None, "CUDA_VISIBLE_DEVICES must be set."

    with from_this_system().set_io_types_filter(TypeFilter.KEEP_ALL) as topo:
        # Get CPU affinity for this GPU. Device ordinal 0 is used because
        # CUDA_VISIBLE_DEVICES has already reordered the devices.
        dev = get_device(topo, device=0)
        cpuset = dev.get_affinity()

        # Set CPU binding
        topo.set_cpubind(cpuset)
        # Set memory binding with STRICT policy - ensures all memory allocations come
        # from the local NUMA node. hwloc determines the NUMA nodes from cpuset.
        topo.set_membind(cpuset, MemBindPolicy.BIND, MemBindFlags.STRICT)


def setup_async_pool() -> None:
    """Setup CUDA async pool. As an alternative, the RMM plugin can be used as well.
    This is the same as using the `CudaAsyncMemoryResource` from RMM, but without the
    RMM dependency.

    .. versionadded:: 3.2.0

    """
    import cuda.bindings.runtime as cudart
    from cuda.bindings import driver
    from cupy.cuda import MemoryAsyncPool

    status, dft_pool = cudart.cudaDeviceGetDefaultMemPool(0)
    _checkcu(status)

    total = device_mem_total()

    v = driver.cuuint64_t(int(total * 0.9))
    (status,) = cudart.cudaMemPoolSetAttribute(
        dft_pool,
        cudart.cudaMemPoolAttr.cudaMemPoolAttrReleaseThreshold,
        v,
    )
    _checkcu(status)
    # Set the allocator for cupy as well.
    import cupy as cp

    cp.cuda.set_allocator(MemoryAsyncPool().malloc)


R = TypeVar("R")
P = ParamSpec("P")


def try_run(fn: Callable[P, R]) -> Callable[P, R]:
    """Loky aborts the process without printing out any error message if there's an
    exception.
```

(continues on next page)

(continued from previous page)

```
    """

    @wraps(fn)
    def inner(*args: P.args, **kwargs: P.kwargs) -> R:
        try:
            return fn(*args, **kwargs)
        except Exception as e:
            print(traceback.format_exc(), file=sys.stderr)
            raise RuntimeError("Running into exception in worker.") from e

    return inner


@try_run
def hist_train(
    worker_idx: int,
    tmpdir: str,
    device: str,
    rabit_args: dict,
) -> None:
    """The hist tree method can use a special data structure `ExtMemQuantileDMatrix` for
    faster initialization and lower memory usage.

    """

    # Make sure XGBoost is using the configured memory pool for all allocations.
    with (
        coll.CommunicatorContext(**rabit_args),
        xgboost.config_context(
            use_cuda_async_pool=device == "cuda",
        ),
    ):
        print("Worker: ", worker_idx)
        # Generate the data for demonstration. The synthetic data is sharded by workers.
        files = make_batches(
            n_samples_per_batch=4096,
            n_features=16,
            n_batches=17,
            tmpdir=tmpdir,
            rank=coll.get_rank(),
        )
        # Since we are running two workers on a single node, we should divide the number
        # of threads between workers.
        n_threads = os.cpu_count()
        assert n_threads is not None
        n_threads = max(n_threads // coll.get_world_size(), 1)
        it = Iterator(device, files)
        Xy = xgboost.ExtMemQuantileDMatrix(
            it, missing=np.nan, enable_categorical=False, nthread=n_threads
        )
        # Check the device is correctly set.
```

(continues on next page)

(continued from previous page)

```
        if device == "cuda":
            # Check the first device
            assert (
                int(os.environ["CUDA_VISIBLE_DEVICES"].split(",")[0])
                < coll.get_world_size()
            )
        booster = xgboost.train(
            {
                "tree_method": "hist",
                "max_depth": 4,
                "device": it.device,
                "nthread": n_threads,
            },
            Xy,
            evals=[(Xy, "Train")],
            num_boost_round=10,
        )
        booster.predict(Xy)


def launch_workers(tmpdir: str, args: argparse.Namespace) -> None:
    """Client function to launch workers."""
    n_workers = 2

    tracker = RabitTracker(host_ip="127.0.0.1", n_workers=n_workers)
    tracker.start()
    rabit_args = tracker.worker_args()

    def initializer(device: str) -> None:
        # Set CUDA device before launching child processes.
        if device == "cuda":
            # name: LokyProcess-1
            _, sidx = mp.current_process().name.split("-")
            idx = int(sidx) - 1  # 1-based indexing from loky
            # Assuming two workers for demo.
            devices = ",".join([str(idx), str((idx + 1) % n_workers)])
            # P0: CUDA_VISIBLE_DEVICES=0,1
            # P1: CUDA_VISIBLE_DEVICES=1,0
            os.environ["CUDA_VISIBLE_DEVICES"] = devices
            setup_numa()
            setup_async_pool()

    with get_reusable_executor(
        max_workers=n_workers,
        initargs=(args.device,),
        initializer=initializer,
    ) as pool:
        # Poor man's currying
        fn = update_wrapper(
            partial(
                hist_train,
                tmpdir=tmpdir,
```

(continues on next page)

(continued from previous page)

```
                device=args.device,
                rabit_args=rabit_args,
            ),
            hist_train,
        )
        pool.map(fn, range(n_workers))


def main() -> None:
    """Demo for distributed training from scratch."""
    parser = argparse.ArgumentParser()
    parser.add_argument("--device", choices=["cpu", "cuda"], default="cpu")
    args = parser.parse_args()
    with tempfile.TemporaryDirectory() as tmpdir:
        launch_workers(tmpdir, args)


if __name__ == "__main__":
    main()
```

Demonstration for parsing JSON/UBJSON tree model files

See *Introduction to Model IO* for details about the model serialization.

```
import argparse
import json
from dataclasses import dataclass
from enum import IntEnum, unique
from typing import Any, Dict, List, Sequence, Union

import numpy as np

try:
    import ubjson
except ImportError:
    ubjson = None


ParamT = Dict[str, str]


def to_integers(data: Union[bytes, List[int]]) -> List[int]:
    """Convert a sequence of bytes to a list of Python integer"""
    return [v for v in data]


@unique
class SplitType(IntEnum):
    numerical = 0
    categorical = 1


```

(continues on next page)

(continued from previous page)

```
@dataclass
class Node:
    # properties
    left: int
    right: int
    parent: int
    split_idx: int
    split_cond: float
    default_left: bool
    split_type: SplitType
    categories: List[int]
    # statistic
    base_weight: float
    loss_chg: float
    sum_hess: float


class Tree:
    """A tree built by XGBoost."""

    def __init__(self, tree_id: int, nodes: Sequence[Node]) -> None:
        self.tree_id = tree_id
        self.nodes = nodes

    def loss_change(self, node_id: int) -> float:
        """Loss gain of a node."""
        return self.nodes[node_id].loss_chg

    def sum_hessian(self, node_id: int) -> float:
        """Sum Hessian of a node."""
        return self.nodes[node_id].sum_hess

    def base_weight(self, node_id: int) -> float:
        """Base weight of a node."""
        return self.nodes[node_id].base_weight

    def split_index(self, node_id: int) -> int:
        """Split feature index of node."""
        return self.nodes[node_id].split_idx

    def split_condition(self, node_id: int) -> float:
        """Split value of a node."""
        return self.nodes[node_id].split_cond

    def split_categories(self, node_id: int) -> List[int]:
        """Categories in a node."""
        return self.nodes[node_id].categories

    def is_categorical(self, node_id: int) -> bool:
        """Whether a node has categorical split."""
        return self.nodes[node_id].split_type == SplitType.categorical
```

(continues on next page)

(continued from previous page)

```
    def is_numerical(self, node_id: int) -> bool:
        return not self.is_categorical(node_id)

    def parent(self, node_id: int) -> int:
        """Parent ID of a node."""
        return self.nodes[node_id].parent

    def left_child(self, node_id: int) -> int:
        """Left child ID of a node."""
        return self.nodes[node_id].left

    def right_child(self, node_id: int) -> int:
        """Right child ID of a node."""
        return self.nodes[node_id].right

    def is_leaf(self, node_id: int) -> bool:
        """Whether a node is leaf."""
        return self.nodes[node_id].left == -1

    def is_deleted(self, node_id: int) -> bool:
        """Whether a node is deleted."""
        return self.split_index(node_id) == np.iinfo(np.uint32).max

    def __str__(self) -> str:
        stack = [0]
        nodes = []
        while stack:
            node: Dict[str, Union[float, int, List[int]]] = {}
            nid = stack.pop()

            node["node id"] = nid
            node["gain"] = self.loss_change(nid)
            node["cover"] = self.sum_hessian(nid)
            nodes.append(node)

            if not self.is_leaf(nid) and not self.is_deleted(nid):
                left = self.left_child(nid)
                right = self.right_child(nid)
                stack.append(left)
                stack.append(right)
                categories = self.split_categories(nid)
                if categories:
                    assert self.is_categorical(nid)
                    node["categories"] = categories
                else:
                    assert self.is_numerical(nid)
                    node["condition"] = self.split_condition(nid)
            if self.is_leaf(nid):
                node["weight"] = self.split_condition(nid)

        string = "\n".join(map(lambda x: "  " + str(x), nodes))
        return string
```

(continues on next page)

(continued from previous page)

```
class Model:
    """Gradient boosted tree model."""

    def __init__(self, model: dict) -> None:
        """Construct the Model from a JSON object.

        parameters
        ----------
         model : A dictionary loaded by json representing a XGBoost boosted tree model.
        """
        # Basic properties of a model
        self.learner_model_shape: ParamT = model["learner"]["learner_model_param"]
        self.num_output_group = int(self.learner_model_shape["num_class"])
        self.num_feature = int(self.learner_model_shape["num_feature"])
        self.base_score: List[float] = json.loads(
            self.learner_model_shape["base_score"]
        )
        # A field encoding which output group a tree belongs
        self.tree_info = model["learner"]["gradient_booster"]["model"]["tree_info"]

        model_shape: ParamT = model["learner"]["gradient_booster"]["model"][
            "gbtree_model_param"
        ]

        # JSON representation of trees
        j_trees = model["learner"]["gradient_booster"]["model"]["trees"]

        # Load the trees
        self.num_trees = int(model_shape["num_trees"])

        trees: List[Tree] = []
        for i in range(self.num_trees):
            tree: Dict[str, Any] = j_trees[i]
            tree_id = int(tree["id"])
            assert tree_id == i, (tree_id, i)
            # - properties
            left_children: List[int] = tree["left_children"]
            right_children: List[int] = tree["right_children"]
            parents: List[int] = tree["parents"]
            split_conditions: List[float] = tree["split_conditions"]
            split_indices: List[int] = tree["split_indices"]
            # when ubjson is used, this is a byte array with each element as uint8
            default_left = to_integers(tree["default_left"])

            # - categorical features
            # when ubjson is used, this is a byte array with each element as uint8
            split_types = to_integers(tree["split_type"])
            # categories for each node is stored in a CSR style storage with segment as
            # the begin ptr and the `categories' as values.
            cat_segments: List[int] = tree["categories_segments"]
```

(continues on next page)

(continued from previous page)

```
        cat_sizes: List[int] = tree["categories_sizes"]
        # node index for categorical nodes
        cat_nodes: List[int] = tree["categories_nodes"]
        assert len(cat_segments) == len(cat_sizes) == len(cat_nodes)
        cats = tree["categories"]
        assert len(left_children) == len(split_types)

        # The storage for categories is only defined for categorical nodes to
        # prevent unnecessary overhead for numerical splits, we track the
        # categorical node that are processed using a counter.
        cat_cnt = 0
        if cat_nodes:
            last_cat_node = cat_nodes[cat_cnt]
        else:
            last_cat_node = -1
        node_categories: List[List[int]] = []
        for node_id in range(len(left_children)):
            if node_id == last_cat_node:
                beg = cat_segments[cat_cnt]
                size = cat_sizes[cat_cnt]
                end = beg + size
                node_cats = cats[beg:end]
                # categories are unique for each node
                assert len(set(node_cats)) == len(node_cats)
                cat_cnt += 1
                if cat_cnt == len(cat_nodes):
                    last_cat_node = -1  # continue to process the rest of the nodes
                else:
                    last_cat_node = cat_nodes[cat_cnt]
                assert node_cats
                node_categories.append(node_cats)
            else:
                # append an empty node, it's either a numerical node or a leaf.
                node_categories.append([])

        # - stats
        base_weights: List[float] = tree["base_weights"]
        loss_changes: List[float] = tree["loss_changes"]
        sum_hessian: List[float] = tree["sum_hessian"]

        # Construct a list of nodes that have complete information
        nodes: List[Node] = [
            Node(
                left_children[node_id],
                right_children[node_id],
                parents[node_id],
                split_indices[node_id],
                split_conditions[node_id],
                default_left[node_id] == 1,  # to boolean
                SplitType(split_types[node_id]),
                node_categories[node_id],
                base_weights[node_id],
```

(continues on next page)

(continued from previous page)

```
                    loss_changes[node_id],
                    sum_hessian[node_id],
                )
                for node_id in range(len(left_children))
            ]

            pytree = Tree(tree_id, nodes)
            trees.append(pytree)

        self.trees = trees

    def print_model(self) -> None:
        for i, tree in enumerate(self.trees):
            print("\ntree_id:", i)
            print(tree)


if __name__ == "__main__":
    parser = argparse.ArgumentParser(
        description="Demonstration for loading XGBoost JSON/UBJSON model."
    )
    parser.add_argument(
        "--model", type=str, required=True, help="Path to .json/.ubj model file."
    )
    args = parser.parse_args()
    if args.model.endswith("json"):
        # use json format
        with open(args.model, "r") as fd:
            model = json.load(fd)
    elif args.model.endswith("ubj"):
        if ubjson is None:
            raise ImportError("ubjson is not installed.")
        # use ubjson format
        with open(args.model, "rb") as bfd:
            model = ubjson.load(bfd)
    else:
        raise ValueError(
            "Unexpected file extension. Supported file extension are json and ubj."
        )
    model = Model(model)
    model.print_model()
```

XGBoost Dask Feature Walkthrough

This directory contains some demonstrations for using *dask* with *XGBoost*. For an overview, see *Distributed XGBoost with Dask*

Use scikit-learn regressor interface with CPU histogram tree method

```
from dask import array as da
from dask.distributed import Client, LocalCluster
```

(continues on next page)

(continued from previous page)

```python
from xgboost import dask as dxgb


def main(client: Client) -> dxgb.Booster:
    # generate some random data for demonstration
    n = 100
    m = 10000
    partition_size = 100
    X = da.random.random((m, n), partition_size)
    y = da.random.random(m, partition_size)

    regressor = dxgb.DaskXGBRegressor(verbosity=1, n_estimators=2)
    regressor.set_params(tree_method="hist")
    # assigning client here is optional
    regressor.client = client

    regressor.fit(X, y, eval_set=[(X, y)])
    prediction = regressor.predict(X)

    bst = regressor.get_booster()
    history = regressor.evals_result()

    print("Evaluation history:", history)
    # returned prediction is always a dask array.
    assert isinstance(prediction, da.Array)
    return bst  # returning the trained model


if __name__ == "__main__":
    # or use other clusters for scaling
    with LocalCluster(n_workers=4, threads_per_worker=1) as cluster:
        with Client(cluster) as client:
            main(client)
```

Use scikit-learn regressor interface with GPU histogram tree method

```python
import dask
from dask import array as da
from dask.distributed import Client

# It's recommended to use dask_cuda for GPU assignment
from dask_cuda import LocalCUDACluster

from xgboost import dask as dxgb


def main(client: Client) -> dxgb.Booster:
    # Generate some random data for demonstration
    rng = da.random.default_rng(1)

    m = 2**18
```

(continues on next page)

(continued from previous page)

```
    n = 100
    X = rng.uniform(size=(m, n), chunks=(128**2, -1))
    y = X.sum(axis=1)

    regressor = dxgb.DaskXGBRegressor(verbosity=1)
    # Set the device to CUDA
    regressor.set_params(tree_method="hist", device="cuda")
    # Assigning client here is optional
    regressor.client = client

    regressor.fit(X, y, eval_set=[(X, y)])
    prediction = regressor.predict(X)

    bst = regressor.get_booster()
    history = regressor.evals_result()

    print("Evaluation history:", history)
    # returned prediction is always a dask array.
    assert isinstance(prediction, da.Array)
    return bst  # returning the trained model


if __name__ == "__main__":
    # With dask cuda, one can scale up XGBoost to arbitrary GPU clusters.
    # `LocalCUDACluster` used here is only for demonstration purpose.
    with LocalCUDACluster() as cluster:
        # Create client from cluster, set the backend to GPU array (cupy).
        with Client(cluster) as client, dask.config.set({"array.backend": "cupy"}):
            main(client)
```

Example of training with Dask on CPU

```
from dask import array as da
from dask.distributed import Client, LocalCluster

from xgboost import dask as dxgb
from xgboost.dask import DaskDMatrix


def main(client: Client) -> None:
    # generate some random data for demonstration
    m = 100000
    n = 100
    rng = da.random.default_rng(1)
    X = rng.normal(size=(m, n), chunks=(10000, -1))
    y = X.sum(axis=1)

    # DaskDMatrix acts like normal DMatrix, works as a proxy for local
    # DMatrix scatter around workers.
    dtrain = DaskDMatrix(client, X, y)
```

(continues on next page)

(continued from previous page)

```
    # Use train method from xgboost.dask instead of xgboost.  This
    # distributed version of train returns a dictionary containing the
    # resulting booster and evaluation history obtained from
    # evaluation metrics.
    output = dxgb.train(
        client,
        {"verbosity": 1, "tree_method": "hist"},
        dtrain,
        num_boost_round=4,
        evals=[(dtrain, "train")],
    )
    bst = output["booster"]
    history = output["history"]

    # you can pass output directly into `predict` too.
    prediction = dxgb.predict(client, bst, dtrain)
    print("Evaluation history:", history)
    print("Error:", da.sqrt((prediction - y) ** 2).mean().compute())


if __name__ == "__main__":
    # or use other clusters for scaling
    with LocalCluster(n_workers=7, threads_per_worker=4) as cluster:
        with Client(cluster) as client:
            main(client)
```

Example of training survival model with Dask on CPU

```
import os

import dask.array as da
import dask.dataframe as dd
from dask.distributed import Client, LocalCluster

from xgboost import dask as dxgb
from xgboost.dask import DaskDMatrix


def main(client: Client) -> da.Array:
    # Load an example survival data from CSV into a Dask data frame.
    # The Veterans' Administration Lung Cancer Trial
    # The Statistical Analysis of Failure Time Data by Kalbfleisch J. and Prentice R␣
↪(1980)
    CURRENT_DIR = os.path.dirname(__file__)
    df = dd.read_csv(
        os.path.join(CURRENT_DIR, os.pardir, "data", "veterans_lung_cancer.csv")
    )

    # DaskDMatrix acts like normal DMatrix, works as a proxy for local
    # DMatrix scatter around workers.
    # For AFT survival, you'd need to extract the lower and upper bounds for the label
```

(continues on next page)

(continued from previous page)

```
    # and pass them as arguments to DaskDMatrix.
    y_lower_bound = df["Survival_label_lower_bound"]
    y_upper_bound = df["Survival_label_upper_bound"]
    X = df.drop(["Survival_label_lower_bound", "Survival_label_upper_bound"], axis=1)
    dtrain = DaskDMatrix(
        client, X, label_lower_bound=y_lower_bound, label_upper_bound=y_upper_bound
    )

    # Use train method from xgboost.dask instead of xgboost.  This
    # distributed version of train returns a dictionary containing the
    # resulting booster and evaluation history obtained from
    # evaluation metrics.
    params = {
        "verbosity": 1,
        "objective": "survival:aft",
        "eval_metric": "aft-nloglik",
        "learning_rate": 0.05,
        "aft_loss_distribution_scale": 1.20,
        "aft_loss_distribution": "normal",
        "max_depth": 6,
        "lambda": 0.01,
        "alpha": 0.02,
    }
    output = dxgb.train(
        client, params, dtrain, num_boost_round=100, evals=[(dtrain, "train")]
    )
    bst = output["booster"]
    history = output["history"]

    # you can pass output directly into `predict` too.
    prediction = dxgb.predict(client, bst, dtrain)
    print("Evaluation history: ", history)

    # Uncomment the following line to save the model to the disk
    # bst.save_model('survival_model.json')

    return prediction


if __name__ == "__main__":
    # or use other clusters for scaling
    with LocalCluster(n_workers=7, threads_per_worker=4) as cluster:
        with Client(cluster) as client:
            main(client)
```

Example of using callbacks with Dask

```
from typing import Any

import numpy as np
from dask.distributed import Client, LocalCluster
```

(continues on next page)

(continued from previous page)

```
from dask_ml.datasets import make_regression
from dask_ml.model_selection import train_test_split

import xgboost as xgb
import xgboost.dask as dxgb
from xgboost.dask import DaskDMatrix


def probability_for_going_backward(epoch: int) -> float:
    return 0.999 / (1.0 + 0.05 * np.log(1.0 + epoch))


# All callback functions must inherit from TrainingCallback
class CustomEarlyStopping(xgb.callback.TrainingCallback):
    """A custom early stopping class where early stopping is determined stochastically.
    In the beginning, allow the metric to become worse with a probability of 0.999.
    As boosting progresses, the probability should be adjusted downward"""

    def __init__(
        self, *, validation_set: str, target_metric: str, maximize: bool, seed: int
    ) -> None:
        self.validation_set = validation_set
        self.target_metric = target_metric
        self.maximize = maximize
        self.seed = seed
        self.rng = np.random.default_rng(seed=seed)
        if maximize:
            self.better = lambda x, y: x > y
        else:
            self.better = lambda x, y: x < y

    def after_iteration(
        self, model: Any, epoch: int, evals_log: xgb.callback.TrainingCallback.EvalsLog
    ) -> bool:
        metric_history = evals_log[self.validation_set][self.target_metric]
        if len(metric_history) < 2 or self.better(
            metric_history[-1], metric_history[-2]
        ):
            return False  # continue training
        p = probability_for_going_backward(epoch)
        go_backward = self.rng.choice(2, size=(1,), replace=True, p=[1 - p, p]).astype(
            np.bool_
        )[0]
        print(
            "The validation metric went into the wrong direction. "
            + f"Stopping training with probability {1 - p}..."
        )
        if go_backward:
            return False  # continue training
        else:
            return True  # stop training
```

(continues on next page)

(continued from previous page)

```
def main(client: Client) -> None:
    m = 100000
    n = 100
    X, y = make_regression(n_samples=m, n_features=n, chunks=200, random_state=0)
    X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

    dtrain = DaskDMatrix(client, X_train, y_train)
    dtest = DaskDMatrix(client, X_test, y_test)

    output = dxgb.train(
        client,
        {
            "verbosity": 1,
            "tree_method": "hist",
            "objective": "reg:squarederror",
            "eval_metric": "rmse",
            "max_depth": 6,
            "learning_rate": 1.0,
        },
        dtrain,
        num_boost_round=1000,
        evals=[(dtrain, "train"), (dtest, "test")],
        callbacks=[
            CustomEarlyStopping(
                validation_set="test", target_metric="rmse", maximize=False, seed=0
            )
        ],
    )


if __name__ == "__main__":
    # or use other clusters for scaling
    with LocalCluster(n_workers=4, threads_per_worker=1) as cluster:
        with Client(cluster) as client:
            main(client)
```

Example of training with Dask on GPU

```
import dask
import dask_cudf
from dask import array as da
from dask import dataframe as dd
from dask.distributed import Client
from dask_cuda import LocalCUDACluster

from xgboost import dask as dxgb
from xgboost.dask import DaskDMatrix


def using_dask_matrix(client: Client, X: da.Array, y: da.Array) -> da.Array:
```

(continues on next page)

(continued from previous page)

```
    # DaskDMatrix acts like normal DMatrix, works as a proxy for local DMatrix scatter
    # around workers.
    dtrain = DaskDMatrix(client, X, y)

    # Use train method from xgboost.dask instead of xgboost.  This distributed version
    # of train returns a dictionary containing the resulting booster and evaluation
    # history obtained from evaluation metrics.
    output = dxgb.train(
        client,
        # Make sure the device is set to CUDA.
        {"tree_method": "hist", "device": "cuda"},
        dtrain,
        num_boost_round=4,
        evals=[(dtrain, "train")],
    )
    bst = output["booster"]
    history = output["history"]

    # you can pass output directly into `predict` too.
    prediction = dxgb.predict(client, bst, dtrain)
    print("Evaluation history:", history)
    return prediction


def using_quantile_device_dmatrix(client: Client, X: da.Array, y: da.Array) -> da.Array:
    """`DaskQuantileDMatrix` is a data type specialized for `hist` tree methods for
     reducing memory usage.

    .. versionadded:: 1.2.0

    """
    # `DaskQuantileDMatrix` is used instead of `DaskDMatrix`, be careful that it can not
    # be used for anything else other than training unless a reference is specified. See
    # the `ref` argument of `DaskQuantileDMatrix`.
    dtrain = dxgb.DaskQuantileDMatrix(client, X, y)
    output = dxgb.train(
        client,
        # Make sure the device is set to CUDA.
        {"tree_method": "hist", "device": "cuda"},
        dtrain,
        num_boost_round=4,
        evals=[(dtrain, "train")],
    )

    prediction = dxgb.predict(client, output, X)
    return prediction


if __name__ == "__main__":
    # `LocalCUDACluster` is used for assigning GPU to XGBoost processes.  Here
    # `n_workers` represents the number of GPUs since we use one GPU per worker process.
    with LocalCUDACluster(n_workers=2, threads_per_worker=4) as cluster:
```

(continues on next page)

(continued from previous page)

```
        # Create client from cluster, set the backend to GPU array (cupy).
        with Client(cluster) as client, dask.config.set({"array.backend": "cupy"}):
            # Generate some random data for demonstration
            rng = da.random.default_rng(1)

            m = 2**18
            n = 100
            X = rng.uniform(size=(m, n), chunks=(128**2, -1))
            y = X.sum(axis=1)

            X = dd.from_dask_array(X)
            y = dd.from_dask_array(y)
            # XGBoost can take arrays. This is to show that DataFrame uses the GPU
            # backend as well.
            assert isinstance(X, dask_cudf.DataFrame)
            assert isinstance(y, dask_cudf.Series)

            print("Using DaskQuantileDMatrix")
            from_ddqdm = using_quantile_device_dmatrix(client, X, y).compute()
            print("Using DMatrix")
            from_dmatrix = using_dask_matrix(client, X, y).compute()
```

Example of forwarding evaluation logs to the client

The example runs on GPU. Two classes are defined to show how to use Dask builtins to forward the logs to the client process.

```
import logging

import dask
import distributed
from dask import array as da
from dask_cuda import LocalCUDACluster
from distributed import Client

from xgboost import dask as dxgb
from xgboost.callback import EvaluationMonitor


def _get_logger() -> logging.Logger:
    logger = logging.getLogger("[xgboost.dask]")
    logger.setLevel(logging.INFO)
    if not logger.hasHandlers():
        handler = logging.StreamHandler()
        logger.addHandler(handler)
    return logger


class ForwardLoggingMonitor(EvaluationMonitor):
    def __init__(
        self,
        client: Client,
```

(continues on next page)

(continued from previous page)

```
        rank: int = 0,
        period: int = 1,
    ) -> None:
        """Print the evaluation result at each iteration. The default monitor in the
        native interface logs the result to the Dask scheduler process. This class can
        be used to forward the logging to the client process. Important: see the
        `client` parameter for more info.

        Parameters
        ----------
        client :
            Distributed client. This must be the top-level client. The class uses
            :py:meth:`distributed.Client.forward_logging` in conjunction with the Python
            :py:mod:`logging` module to forward the evaluation results to the client
            process. It has undefined behaviour if called in a nested task. As a result,
            client-side logging is not enabled by default.

        """
        client.forward_logging(_get_logger().name)

        super().__init__(
            rank=rank,
            period=period,
            logger=lambda msg: _get_logger().info(msg.strip()),
        )


class WorkerEventMonitor(EvaluationMonitor):
    """Use :py:meth:`distributed.print` to forward the log. A downside is that not only
    all clients connected to the cluster can see the log, the logs are also printed on
    the worker. If you use a local cluster, the log is duplicated.

    """

    def __init__(self, rank: int = 0, period: int = 1) -> None:
        super().__init__(
            rank=rank, period=period, logger=lambda msg: distributed.print(msg.strip())
        )


def hist_train(
    client: Client, X: da.Array, y: da.Array, monitor: EvaluationMonitor
) -> da.Array:
    # `DaskQuantileDMatrix` is used instead of `DaskDMatrix`, be careful that it can not
    # be used for anything else other than as a training DMatrix, unless a reference is
    # specified. See the `ref` argument of `DaskQuantileDMatrix`.
    dtrain = dxgb.DaskQuantileDMatrix(client, X, y)
    output = dxgb.train(
        client,
        # Make sure the device is set to CUDA.
        {"tree_method": "hist", "device": "cuda"},
        dtrain,
```

(continues on next page)

(continued from previous page)

```
        num_boost_round=4,
        evals=[(dtrain, "train")],
        # Use the monitor to forward the log.
        callbacks=[monitor],
        # Disable the internal logging and prefer the client-side `EvaluationMonitor`.
        verbose_eval=False,
    )
    bst = output["booster"]
    history = output["history"]

    prediction = dxgb.predict(client, bst, X)
    print("Evaluation history:", history)
    return prediction


if __name__ == "__main__":
    # `LocalCUDACluster` is used for assigning GPU to XGBoost processes.  Here
    # `n_workers` represents the number of GPUs since we use one GPU per worker process.
    with LocalCUDACluster(n_workers=2, threads_per_worker=4) as cluster:
        # Create client from cluster, set the backend to GPU array (cupy).
        with Client(cluster) as client, dask.config.set({"array.backend": "cupy"}):
            # Generate some random data for demonstration
            rng = da.random.default_rng(1)

            m = 2**18
            n = 100
            X = rng.uniform(size=(m, n), chunks=(128**2, -1))
            y = X.sum(axis=1)

            # Use forwarding, the client must be the top client.
            monitor: EvaluationMonitor = ForwardLoggingMonitor(client)
            hist_train(client, X, y, monitor).compute()

            # Use distributed.print, the logs in this demo are duplicated as the same
            # log is printed in all workers along with the client.
            monitor = WorkerEventMonitor()
            hist_train(client, X, y, monitor).compute()
```

Learning to rank with the Dask Interface

Added in version 3.0.0.

This is a demonstration of using XGBoost for learning to rank tasks using the MSLR_10k_letor dataset. For more infomation about the dataset, please visit its description page.

See *Distributed Training* for a general description for distributed learning to rank and *Learning to Rank* for Dask-specific features.

```
from __future__ import annotations

import argparse
import os
from contextlib import contextmanager
```

(continues on next page)

(continued from previous page)

```
from typing import Generator

import dask
import numpy as np
from dask import dataframe as dd
from distributed import Client, LocalCluster, wait
from sklearn.datasets import load_svmlight_file

from xgboost import dask as dxgb


def load_mslr_10k(
    device: str, data_path: str, cache_path: str
) -> tuple[dd.DataFrame, dd.DataFrame, dd.DataFrame]:
    """Load the MSLR10k dataset from data_path and save parquet files in the cache_path."
↪""
    root_path = os.path.expanduser(args.data)
    cache_path = os.path.expanduser(args.cache)

    # Use only the Fold1 for demo:
    # Train,      Valid, Test
    # {S1,S2,S3}, S4,    S5
    fold = 1

    if not os.path.exists(cache_path):
        os.mkdir(cache_path)
        fold_path = os.path.join(root_path, f"Fold{fold}")
        train_path = os.path.join(fold_path, "train.txt")
        valid_path = os.path.join(fold_path, "vali.txt")
        test_path = os.path.join(fold_path, "test.txt")

        X_train, y_train, qid_train = load_svmlight_file(
            train_path, query_id=True, dtype=np.float32
        )
        columns = [f"f{i}" for i in range(X_train.shape[1])]
        X_train = dd.from_array(X_train.toarray(), columns=columns)
        y_train = y_train.astype(np.int32)
        qid_train = qid_train.astype(np.int32)

        X_train["y"] = dd.from_array(y_train)
        X_train["qid"] = dd.from_array(qid_train)
        X_train.to_parquet(os.path.join(cache_path, "train"), engine="pyarrow")

        X_valid, y_valid, qid_valid = load_svmlight_file(
            valid_path, query_id=True, dtype=np.float32
        )
        X_valid = dd.from_array(X_valid.toarray(), columns=columns)
        y_valid = y_valid.astype(np.int32)
        qid_valid = qid_valid.astype(np.int32)

        X_valid["y"] = dd.from_array(y_valid)
        X_valid["qid"] = dd.from_array(qid_valid)
```

(continues on next page)

(continued from previous page)

```
        X_valid.to_parquet(os.path.join(cache_path, "valid"), engine="pyarrow")

        X_test, y_test, qid_test = load_svmlight_file(
            test_path, query_id=True, dtype=np.float32
        )

        X_test = dd.from_array(X_test.toarray(), columns=columns)
        y_test = y_test.astype(np.int32)
        qid_test = qid_test.astype(np.int32)

        X_test["y"] = dd.from_array(y_test)
        X_test["qid"] = dd.from_array(qid_test)
        X_test.to_parquet(os.path.join(cache_path, "test"), engine="pyarrow")

    df_train = dd.read_parquet(
        os.path.join(cache_path, "train"), calculate_divisions=True
    )
    df_valid = dd.read_parquet(
        os.path.join(cache_path, "valid"), calculate_divisions=True
    )
    df_test = dd.read_parquet(
        os.path.join(cache_path, "test"), calculate_divisions=True
    )

    return df_train, df_valid, df_test


def ranking_demo(client: Client, args: argparse.Namespace) -> None:
    """Learning to rank with data sorted locally."""
    df_tr, df_va, _ = load_mslr_10k(args.device, args.data, args.cache)

    X_train: dd.DataFrame = df_tr[df_tr.columns.difference(["y", "qid"])]
    y_train = df_tr[["y", "qid"]]
    Xy_train = dxgb.DaskQuantileDMatrix(client, X_train, y_train.y, qid=y_train.qid)

    X_valid: dd.DataFrame = df_va[df_va.columns.difference(["y", "qid"])]
    y_valid = df_va[["y", "qid"]]
    Xy_valid = dxgb.DaskQuantileDMatrix(
        client, X_valid, y_valid.y, qid=y_valid.qid, ref=Xy_train
    )
    # Upon training, you will see a performance warning about sorting data based on
    # query groups.
    dxgb.train(
        client,
        {"objective": "rank:ndcg", "device": args.device},
        Xy_train,
        evals=[(Xy_train, "Train"), (Xy_valid, "Valid")],
        num_boost_round=100,
    )


def ranking_wo_split_demo(client: Client, args: argparse.Namespace) -> None:
```

(continues on next page)

(continued from previous page)

```
    """Learning to rank with data partitioned according to query groups."""
    df_tr, df_va, df_te = load_mslr_10k(args.device, args.data, args.cache)

    X_tr = df_tr[df_tr.columns.difference(["y", "qid"])]
    X_va = df_va[df_va.columns.difference(["y", "qid"])]

    # `allow_group_split=False` makes sure data is partitioned according to the query
    # groups.
    ltr = dxgb.DaskXGBRanker(allow_group_split=False, device=args.device)
    ltr.client = client
    ltr = ltr.fit(
        X_tr,
        df_tr.y,
        qid=df_tr.qid,
        eval_set=[(X_tr, df_tr.y), (X_va, df_va.y)],
        eval_qid=[df_tr.qid, df_va.qid],
        verbose=True,
    )

    df_te = df_te.persist()
    wait([df_te])

    X_te = df_te[df_te.columns.difference(["y", "qid"])]
    predt = ltr.predict(X_te)
    y = client.compute(df_te.y)
    wait([predt, y])


@contextmanager
def gen_client(device: str) -> Generator[Client, None, None]:
    match device:
        case "cuda":
            from dask_cuda import LocalCUDACluster

            with LocalCUDACluster() as cluster:
                with Client(cluster) as client:
                    with dask.config.set(
                        {
                            "array.backend": "cupy",
                            "dataframe.backend": "cudf",
                        }
                    ):
                        yield client
        case "cpu":
            with LocalCluster() as cluster:
                with Client(cluster) as client:
                    yield client


if __name__ == "__main__":
    parser = argparse.ArgumentParser(
        description="Demonstration of learning to rank using XGBoost."
```

(continues on next page)

(continued from previous page)

```
    )
    parser.add_argument(
        "--data",
        type=str,
        help="Root directory of the MSLR-WEB10K data.",
        required=True,
    )
    parser.add_argument(
        "--cache",
        type=str,
        help="Directory for caching processed data.",
        required=True,
    )
    parser.add_argument("--device", choices=["cpu", "cuda"], default="cpu")
    parser.add_argument(
        "--no-split",
        action="store_true",
        help="Flag to indicate query groups should not be split.",
    )
    args = parser.parse_args()

    with gen_client(args.device) as client:
        if args.no_split:
            ranking_wo_split_demo(client, args)
        else:
            ranking_demo(client, args)
```

Survival Analysis Walkthrough

This is a collection of examples for using the XGBoost Python package for training survival models. For an introduction, see *Survival Analysis with Accelerated Failure Time*

Demo for survival analysis (regression).

Demo for survival analysis (regression). using Accelerated Failure Time (AFT) model.

```
import os

import numpy as np
import pandas as pd
from sklearn.model_selection import ShuffleSplit

import xgboost as xgb

# The Veterans' Administration Lung Cancer Trial
# The Statistical Analysis of Failure Time Data by Kalbfleisch J. and Prentice R (1980)
CURRENT_DIR = os.path.dirname(__file__)
df = pd.read_csv(os.path.join(CURRENT_DIR, '../data/veterans_lung_cancer.csv'))
print('Training data:')
print(df)

# Split features and labels
```

(continues on next page)

(continued from previous page)

```
y_lower_bound = df['Survival_label_lower_bound']
y_upper_bound = df['Survival_label_upper_bound']
X = df.drop(['Survival_label_lower_bound', 'Survival_label_upper_bound'], axis=1)

# Split data into training and validation sets
rs = ShuffleSplit(n_splits=2, test_size=.7, random_state=0)
train_index, valid_index = next(rs.split(X))
dtrain = xgb.DMatrix(X.values[train_index, :])
dtrain.set_float_info('label_lower_bound', y_lower_bound[train_index])
dtrain.set_float_info('label_upper_bound', y_upper_bound[train_index])
dvalid = xgb.DMatrix(X.values[valid_index, :])
dvalid.set_float_info('label_lower_bound', y_lower_bound[valid_index])
dvalid.set_float_info('label_upper_bound', y_upper_bound[valid_index])

# Train gradient boosted trees using AFT loss and metric
params = {'verbosity': 0,
          'objective': 'survival:aft',
          'eval_metric': 'aft-nloglik',
          'tree_method': 'hist',
          'learning_rate': 0.05,
          'aft_loss_distribution': 'normal',
          'aft_loss_distribution_scale': 1.20,
          'max_depth': 6,
          'lambda': 0.01,
          'alpha': 0.02}
bst = xgb.train(params, dtrain, num_boost_round=10000,
                evals=[(dtrain, 'train'), (dvalid, 'valid')],
                early_stopping_rounds=50)

# Run prediction on the validation set
df = pd.DataFrame({'Label (lower bound)': y_lower_bound[valid_index],
                   'Label (upper bound)': y_upper_bound[valid_index],
                   'Predicted label': bst.predict(dvalid)})
print(df)
# Show only data points with right-censored labels
print(df[np.isinf(df['Label (upper bound)'])])

# Save trained model
bst.save_model('aft_model.json')
```

Demo for survival analysis (regression) with Optuna.

Demo for survival analysis (regression) using Accelerated Failure Time (AFT) model, using Optuna to tune hyperparameters

```
import numpy as np
import optuna
import pandas as pd
from sklearn.model_selection import ShuffleSplit

import xgboost as xgb
```

(continues on next page)

(continued from previous page)

```
# The Veterans' Administration Lung Cancer Trial
# The Statistical Analysis of Failure Time Data by Kalbfleisch J. and Prentice R (1980)
df = pd.read_csv('../data/veterans_lung_cancer.csv')
print('Training data:')
print(df)

# Split features and labels
y_lower_bound = df['Survival_label_lower_bound']
y_upper_bound = df['Survival_label_upper_bound']
X = df.drop(['Survival_label_lower_bound', 'Survival_label_upper_bound'], axis=1)

# Split data into training and validation sets
rs = ShuffleSplit(n_splits=2, test_size=.7, random_state=0)
train_index, valid_index = next(rs.split(X))
dtrain = xgb.DMatrix(X.values[train_index, :])
dtrain.set_float_info('label_lower_bound', y_lower_bound[train_index])
dtrain.set_float_info('label_upper_bound', y_upper_bound[train_index])
dvalid = xgb.DMatrix(X.values[valid_index, :])
dvalid.set_float_info('label_lower_bound', y_lower_bound[valid_index])
dvalid.set_float_info('label_upper_bound', y_upper_bound[valid_index])

# Define hyperparameter search space
base_params = {'verbosity': 0,
              'objective': 'survival:aft',
              'eval_metric': 'aft-nloglik',
              'tree_method': 'hist'}  # Hyperparameters common to all trials
def objective(trial):
    params = {'learning_rate': trial.suggest_loguniform('learning_rate', 0.01, 1.0),
              'aft_loss_distribution': trial.suggest_categorical('aft_loss_distribution',
                                                                  ['normal', 'logistic',
↪'extreme']),
              'aft_loss_distribution_scale': trial.suggest_loguniform('aft_loss_
↪distribution_scale', 0.1, 10.0),
              'max_depth': trial.suggest_int('max_depth', 3, 8),
              'lambda': trial.suggest_loguniform('lambda', 1e-8, 1.0),
              'alpha': trial.suggest_loguniform('alpha', 1e-8, 1.0)}  # Search space
    params.update(base_params)
    pruning_callback = optuna.integration.XGBoostPruningCallback(trial, 'valid-aft-
↪nloglik')
    bst = xgb.train(params, dtrain, num_boost_round=10000,
                    evals=[(dtrain, 'train'), (dvalid, 'valid')],
                    early_stopping_rounds=50, verbose_eval=False, callbacks=[pruning_
↪callback])
    if bst.best_iteration >= 25:
        return bst.best_score
    else:
        return np.inf  # Reject models with < 25 trees

# Run hyperparameter search
study = optuna.create_study(direction='minimize')
study.optimize(objective, n_trials=200)
print('Completed hyperparameter tuning with best aft-nloglik = {}.'.format(study.best_
```

(continues on next page)

(continued from previous page)

```
↪trial.value))
params = {}
params.update(base_params)
params.update(study.best_trial.params)

# Re-run training with the best hyperparameter combination
print('Re-running the best trial... params = {}'.format(params))
bst = xgb.train(params, dtrain, num_boost_round=10000,
                evals=[(dtrain, 'train'), (dvalid, 'valid')],
                early_stopping_rounds=50)

# Run prediction on the validation set
df = pd.DataFrame({'Label (lower bound)': y_lower_bound[valid_index],
                   'Label (upper bound)': y_upper_bound[valid_index],
                   'Predicted label': bst.predict(dvalid)})
print(df)
# Show only data points with right-censored labels
print(df[np.isinf(df['Label (upper bound)'])])

# Save trained model
bst.save_model('aft_best_model.json')
```

Visual demo for survival analysis (regression) with Accelerated Failure Time (AFT) model.

This demo uses 1D toy data and visualizes how XGBoost fits a tree ensemble. The ensemble model starts out as a flat line and evolves into a step function in order to account for all ranged labels.

```
import matplotlib.pyplot as plt
import numpy as np
import xgboost as xgb

plt.rcParams.update({"font.size": 13})


# pylint: disable=redefined-outer-name
def plot_censored_labels(
    X: np.ndarray, y_lower: np.ndarray, y_upper: np.ndarray
) -> None:
    """Function to visualize censored labels"""

    def replace_inf(x: np.ndarray, target_value: float) -> np.ndarray:
        x[np.isinf(x)] = target_value
        return x

    plt.plot(X, y_lower, "o", label="y_lower", color="blue")
    plt.plot(X, y_upper, "o", label="y_upper", color="fuchsia")
    plt.vlines(
        X,
        ymin=replace_inf(y_lower, 0.01),
        ymax=replace_inf(y_upper, 1000.0),
        label="Range for y",
        color="gray",
```

(continues on next page)

(continued from previous page)

```
    )


# Toy data
X = np.array([1, 2, 3, 4, 5]).reshape((-1, 1))
INF = np.inf
y_lower = np.array([10, 15, -INF, 30, 100])
y_upper = np.array([INF, INF, 20, 50, INF])

# Visualize toy data
plt.figure(figsize=(5, 4))
plot_censored_labels(X, y_lower, y_upper)
plt.ylim((6, 200))
plt.legend(loc="lower right")
plt.title("Toy data")
plt.xlabel("Input feature")
plt.ylabel("Label")
plt.yscale("log")
plt.tight_layout()
plt.show(block=True)

# Will be used to visualize XGBoost model
grid_pts = np.linspace(0.8, 5.2, 1000).reshape((-1, 1))

# Train AFT model using XGBoost
dmat = xgb.DMatrix(X)
dmat.set_float_info("label_lower_bound", y_lower)
dmat.set_float_info("label_upper_bound", y_upper)
params = {"max_depth": 3, "objective": "survival:aft", "min_child_weight": 0}

accuracy_history: list[np.float64] = []


class PlotIntermediateModel(xgb.callback.TrainingCallback):
    """Custom callback to plot intermediate models."""

    def after_iteration(
        self,
        model: xgb.Booster,
        epoch: int,
        evals_log: xgb.callback.TrainingCallback.EvalsLog,
    ) -> bool:
        """Run after training is finished."""
        # Compute y_pred = prediction using the intermediate model, at current boosting
        # iteration
        y_pred = model.predict(dmat)
        # "Accuracy" = the number of data points whose ranged label (y_lower, y_upper)
        #              includes the corresponding predicted label (y_pred)
        acc = np.sum(
            np.logical_and(y_pred >= y_lower, y_pred <= y_upper) / len(X) * 100
        )
        accuracy_history.append(acc)
```

(continues on next page)

(continued from previous page)

```
        # Plot ranged labels as well as predictions by the model
        plt.subplot(5, 3, epoch + 1)
        plot_censored_labels(X, y_lower, y_upper)
        y_pred_grid_pts = model.predict(xgb.DMatrix(grid_pts))
        plt.plot(
            grid_pts, y_pred_grid_pts, "r-", label="XGBoost AFT model", linewidth=4
        )
        plt.title(f"Iteration {epoch}", x=0.5, y=0.8)
        plt.xlim((0.8, 5.2))
        plt.ylim((1 if np.min(y_pred) < 6 else 6, 200))
        plt.yscale("log")
        return False


res: xgb.callback.TrainingCallback.EvalsLog = {}
plt.figure(figsize=(12, 13))
bst = xgb.train(
    params,
    dmat,
    num_boost_round=15,
    evals=[(dmat, "train")],
    evals_result=res,
    callbacks=[PlotIntermediateModel()],
)
plt.tight_layout()
plt.legend(
    loc="lower center",
    ncol=4,
    bbox_to_anchor=(0.5, 0),
    bbox_transform=plt.gcf().transFigure,
)
plt.tight_layout()

# Plot negative log likelihood over boosting iterations
plt.figure(figsize=(8, 3))
plt.subplot(1, 2, 1)
plt.plot(res["train"]["aft-nloglik"], "b-o", label="aft-nloglik")
plt.xlabel("# Boosting Iterations")
plt.legend(loc="best")

# Plot "accuracy" over boosting iterations
# "Accuracy" = the number of data points whose ranged label (y_lower, y_upper) includes
#              the corresponding predicted label (y_pred)
plt.subplot(1, 2, 2)
plt.plot(accuracy_history, "r-o", label="Accuracy (%)")
plt.xlabel("# Boosting Iterations")
plt.legend(loc="best")
plt.tight_layout()

plt.show()
```

Using XGBoost with RAPIDS Memory Manager (RMM) plugin

RAPIDS Memory Manager (RMM) library provides a collection of efficient memory allocators for NVIDIA GPUs. It is now possible to use XGBoost with memory allocators provided by RMM, by enabling the RMM integration plugin.

The demos in this directory highlights one RMM allocator in particular: **the pool sub-allocator**. This allocator addresses the slow speed of `cudaMalloc()` by allocating a large chunk of memory upfront. Subsequent allocations will draw from the pool of already allocated memory and thus avoid the overhead of calling `cudaMalloc()` directly. See this GTC talk slides for more details.

Before running the demos, ensure that XGBoost is compiled with the RMM plugin enabled. To do this, run CMake with option `-DPLUGIN_RMM=ON` (`-DUSE_CUDA=ON` also required):

```
cmake -B build -S . -DUSE_CUDA=ON -DUSE_NCCL=ON -DPLUGIN_RMM=ON
cmake --build build -j$(nproc)
```

CMake will attempt to locate the RMM library in your build environment. You may choose to build RMM from the source, or install it using the Conda package manager. If CMake cannot find RMM, you should specify the location of RMM with the CMake prefix:

```
# If using Conda:
cmake -B build -S . -DUSE_CUDA=ON -DUSE_NCCL=ON -DPLUGIN_RMM=ON -DCMAKE_PREFIX_PATH=
↪$CONDA_PREFIX
# If using RMM installed with a custom location
cmake -B build -S . -DUSE_CUDA=ON -DUSE_NCCL=ON -DPLUGIN_RMM=ON -DCMAKE_PREFIX_PATH=/
↪path/to/rmm
```

Informing XGBoost about RMM pool

When XGBoost is compiled with RMM, most of the large size allocation will go through RMM allocators, but some small allocations in performance critical areas are using a different caching allocator so that we can have better control over memory allocation behavior. Users can override this behavior and force the use of rmm for all allocations by setting the global configuration `use_rmm`:

```python
with xgb.config_context(use_rmm=True):
  clf = xgb.XGBClassifier(tree_method="hist", device="cuda")
```

Depending on the choice of memory pool size and the type of the allocator, this can add more consistency to memory usage but with slightly degraded performance impact.

No Device Ordinal for Multi-GPU

Since with RMM the memory pool is pre-allocated on a specific device, changing the CUDA device ordinal in XGBoost can result in memory error `cudaErrorIllegalAddress`. Use the `CUDA_VISIBLE_DEVICES` environment variable instead of the `device="cuda:1"` parameter for selecting device. For distributed training, the distributed computing frameworks like `dask-cuda` are responsible for device management. For Scala-Spark, see *XGBoost4J-Spark-GPU Tutorial* for more info.

Memory Over-Subscription

> **Warning**
>
> This feature is still experimental and is under active development.

The newer NVIDIA platforms like Grace-Hopper use NVLink-C2C, which allows the CPU and GPU to have a coherent memory model. Users can use the *SamHeadroomMemoryResource* in the latest RMM to utilize system memory for storing data. This can help XGBoost utilize memory from the host for GPU computation, but it may reduce performance due to slower CPU memory speed and page migration overhead.

Using rmm on a single node device

```
import rmm
from sklearn.datasets import make_classification

import xgboost as xgb

# Initialize RMM pool allocator
rmm.reinitialize(pool_allocator=True)
# Optionally force XGBoost to use RMM for all GPU memory allocation, see ./README.md
# xgb.set_config(use_rmm=True)

X, y = make_classification(n_samples=10000, n_informative=5, n_classes=3)
dtrain = xgb.DMatrix(X, label=y)

params = {
    "max_depth": 8,
    "eta": 0.01,
    "objective": "multi:softprob",
    "num_class": 3,
    "tree_method": "hist",
    "device": "cuda",
}
# XGBoost will automatically use the RMM pool allocator
bst = xgb.train(params, dtrain, num_boost_round=100, evals=[(dtrain, "train")])
```

Using rmm with Dask

```
import dask
from dask.distributed import Client
from dask_cuda import LocalCUDACluster
from sklearn.datasets import make_classification

import xgboost as xgb


def main(client):
    # Optionally force XGBoost to use RMM for all GPU memory allocation, see ./README.md
    # xgb.set_config(use_rmm=True)

    X, y = make_classification(n_samples=10000, n_informative=5, n_classes=3)
    # In pratice one should prefer loading the data with dask collections instead of
```

(continues on next page)

(continued from previous page)

```
    # using `from_array`.
    X = dask.array.from_array(X)
    y = dask.array.from_array(y)
    dtrain = xgb.dask.DaskDMatrix(client, X, label=y)

    params = {
        "max_depth": 8,
        "eta": 0.01,
        "objective": "multi:softprob",
        "num_class": 3,
        "tree_method": "hist",
        "eval_metric": "merror",
        "device": "cuda",
    }
    output = xgb.dask.train(
        client, params, dtrain, num_boost_round=100, evals=[(dtrain, "train")]
    )
    bst = output["booster"]
    history = output["history"]
    for i, e in enumerate(history["train"]["merror"]):
        print(f"[{i}] train-merror: {e}")


if __name__ == "__main__":
    # To use RMM pool allocator with a GPU Dask cluster, just add rmm_pool_size option
    # to LocalCUDACluster constructor.
    with LocalCUDACluster(rmm_pool_size="2GB") as cluster:
        with Client(cluster) as client:
            main(client)
```

1.11 XGBoost R Package

You have found the XGBoost R Package!

1.11.1 Get Started

Since XGBoost 3.0.0, the latest R package is available on R-universe in addition to the CRAN package.

- Check out the *Installation Guide* for instructions on how to install xgboost, and *Tutorials* for examples on how to use XGBoost for various tasks.
- Read the latest API documentation.
- Read the CRAN documentation.

1.11.2 Vignettes

XGBoost for R introduction

Introduction

XGBoost is an optimized distributed gradient boosting library designed to be highly **efficient**, **flexible** and **portable**. It implements machine learning algorithms under the `gradient boosting` framework. XGBoost provides a parallel

tree boosting (also known as GBDT, GBM) that solve many data science problems in a fast and accurate way. The same code runs on major distributed environment (Hadoop, SGE, MPI) and can solve problems beyond billions of examples.

For an introduction to the concept of gradient boosting, see the tutorial Introduction to Boosted Trees in XGBoost's online docs.

For more details about XGBoost's features and usage, see the online documentation which contains more tutorials, examples, and details.

This short vignette outlines the basic usage of the R interface for XGBoost, assuming the reader has some familiarity with the underlying concepts behind statistical modeling with gradient-boosted decision trees.

Building a predictive model

At its core, XGBoost consists of a C++ library which offers bindings for different programming languages, including R. The R package for XGBoost provides an idiomatic interface similar to those of other statistical modeling packages using and x/y design, as well as a lower-level interface that interacts more directly with the underlying core library and which is similar to those of other language bindings like Python, plus various helpers to interact with its model objects such as by plotting their feature importances or converting them to other formats.

The main function of interest is `xgboost(x, y, ...)`, which calls the XGBoost model building procedure on observed data of covariates/features/predictors "x", and a response variable "y" - it should feel familiar to users of packages like `glmnet` or `ncvreg`:

```
library(xgboost)
data(ToothGrowth)

y <- ToothGrowth$supp # the response which we want to model/predict
x <- ToothGrowth[, c("len", "dose")] # the features from which we want to predict it
model <- xgboost(x, y, nthreads = 1, nrounds = 2)
model
```

```
## XGBoost model object
## Call:
##   xgboost(x = x, y = y, nrounds = 2, nthreads = 1)
## Objective: binary:logistic
## Number of iterations: 2
## Number of features: 2
## Classes: OJ, VC
```

In this case, the "y" response variable that was supplied is a "factor" type with two classes ("OJ" and "VC") - hence, XGBoost builds a binary classification model for it based on the features "x", by finding a maximum likelihood estimate (similar to the `family="binomial"` model from R's `glm` function) through rule buckets obtained from the sum of two decision trees (from `nrounds=2`), from which we can then predict probabilities, log-odds, class with highest likelihood, among others:

```
predict(model, x[1:6, ], type = "response") # probabilities for y's last level ("VC")
```

```
##         1         2         3         4         5         6
## 0.6596265 0.5402158 0.6596265 0.6596265 0.6596265 0.4953500
```

```
predict(model, x[1:6, ], type = "raw")      # log-odds
```

```
##           1           2           3           4           5           6
##  0.66163027  0.16121151  0.66163027  0.66163027  0.66163027 -0.01860031
```

```
predict(model, x[1:6, ], type = "class")    # class with highest probability
```

```
## [1] VC VC VC VC VC OJ
## Levels: OJ VC
```

Compared to R's `glm` function which follows the concepts of "families" and "links" from GLM theory to fit models for different kinds of response distributions, XGBoost follows the simpler concept of "objectives" which mix both of them into one, and which just like `glm`, allow modeling very different kinds of response distributions (e.g. discrete choices, real-valued numbers, counts, censored measurements, etc.) through a common framework.

XGBoost will automatically determine a suitable objective for the response given its object class (can pass factors for classification, numeric vectors for regression, `Surv` objects from the `survival` package for survival, etc. - see `?xgboost` for more details), but this can be controlled manually through an `objective` parameter based the kind of model that is desired:

```
data(mtcars)

y <- mtcars$mpg
x <- mtcars[, -1]
model_gaussian <- xgboost(x, y, nthreads = 1, nrounds = 2) # default is squared loss␣
↪(Gaussian)
model_poisson <- xgboost(x, y, objective = "count:poisson", nthreads = 1, nrounds = 2)
model_abserr <- xgboost(x, y, objective = "reg:absoluteerror", nthreads = 1, nrounds = 2)
```

Note: the objective must match with the type of the "y" response variable - for example, classification objectives for discrete choices require "factor" types, while regression models for real-valued data require "numeric" types.

Model parameters

XGBoost models allow a large degree of control over how they are built. By their nature, gradient-boosted decision tree ensembles are able to capture very complex patterns between features in the data and a response variable, which also means they can suffer from overfitting if not controlled appropirately.

For best results, one needs to find suitable parameters for the data being modeled. Note that XGBoost does not adjust its default hyperparameters based on the data, and different datasets will require vastly different hyperparameters for optimal predictive performance.

For example, for a small dataset like "TootGrowth" which has only two features and 60 observations, the defaults from XGBoost are an overkill which lead to severe overfitting - for such data, one might want to have smaller trees (i.e. more convervative decision rules, capturing simpler patterns) and fewer of them, for example.

Parameters can be controlled by passing additional arguments to `xgboost()`. See `?xgb.params` for details about what parameters are available to control.

```
y <- ToothGrowth$supp
x <- ToothGrowth[, c("len", "dose")]
model_conservative <- xgboost(
    x, y, nthreads = 1,
    nrounds = 5,
    max_depth = 2,
    reg_lambda = 0.5,
    learning_rate = 0.15
)
pred_conservative <- predict(
    model_conservative,
```

(continues on next page)

(continued from previous page)

```
    x
)
pred_conservative[1:6] # probabilities are all closer to 0.5 now
```

```
##         1         2         3         4         5         6
## 0.6509258 0.4822042 0.6509258 0.6509258 0.6509258 0.4477925
```

XGBoost also allows the possibility of calculating evaluation metrics for model quality over boosting rounds, with a wide variety of built-in metrics available to use. It's possible to automatically set aside a fraction of the data to use as evaluation set, from which one can then visually monitor progress and overfitting:

```
xgboost(
    x, y, nthreads = 1,
    eval_set = 0.2,
    monitor_training = TRUE,
    verbosity = 1,
    eval_metric = c("auc", "logloss"),
    nrounds = 5,
    max_depth = 2,
    reg_lambda = 0.5,
    learning_rate = 0.15
)
```

```
## [1]       train-auc:0.812063       train-logloss:0.645314       eval-auc:0.234375
        eval-logloss:0.760331
## [2]       train-auc:0.913462       train-logloss:0.618020       eval-auc:0.203125
        eval-logloss:0.775389
## [3]       train-auc:0.913462       train-logloss:0.584422       eval-auc:0.296875
        eval-logloss:0.809631
## [4]       train-auc:0.913462       train-logloss:0.563299       eval-auc:0.203125
        eval-logloss:0.825385
## [5]       train-auc:0.918706       train-logloss:0.536936       eval-auc:0.281250
        eval-logloss:0.857389
```

```
## XGBoost model object
## Call:
##   xgboost(x = x, y = y, nrounds = 5, max_depth = 2, learning_rate = 0.15,
##     reg_lambda = 0.5, verbosity = 1, monitor_training = TRUE,
##     eval_set = 0.2, eval_metric = c("auc", "logloss"), nthreads = 1)
## Objective: binary:logistic
## Number of iterations: 5
## Number of features: 2
## Classes: OJ, VC
```

Examining model objects

XGBoost model objects for the most part consist of a pointer to a C++ object where most of the information is held and which is interfaced through the utility functions and methods in the package, but also contains some R attributes that can be retrieved (and new ones added) through `attributes()`:

```
attributes(model)
```

```
## $call
## xgboost(x = x, y = y, nrounds = 2, nthreads = 1)
##
## $params
## $params$objective
## [1] "binary:logistic"
##
## $params$nthread
## [1] 1
##
## $params$verbosity
## [1] 0
##
## $params$seed
## [1] 0
##
## $params$validate_parameters
## [1] TRUE
##
##
## $names
## [1] "ptr"
##
## $class
## [1] "xgboost"     "xgb.Booster"
##
## $metadata
## $metadata$y_levels
## [1] "OJ" "VC"
##
## $metadata$n_targets
## [1] 1
```

In addition to R attributes (which can be arbitrary R objects), it may also keep some standardized C-level attributes that one can access and modify (but which can only be JSON-format):

```
xgb.attributes(model)
```

```
## list()
```

(they are empty for this model)

… but usually, when it comes to getting something out of a model object, one would typically want to do this through the built-in utility functions. Some examples:

```
xgb.importance(model)
```

```
##    Feature      Gain     Cover Frequency
##     <char>     <num>     <num>     <num>
## 1:     len 0.7444265 0.6830449 0.7333333
## 2:    dose 0.2555735 0.3169551 0.2666667
```

```
xgb.model.dt.tree(model)
```

```
##      Tree  Node     ID Feature Split    Yes     No Missing        Gain
##     <int> <int> <char>  <char> <num> <char> <char>  <char>       <num>
##  1:     0     0    0-0     len  19.7    0-1    0-2     0-2  5.88235283
##  2:     0     1    0-1    dose   1.0    0-3    0-4     0-4  2.50230217
##  3:     0     2    0-2    dose   2.0    0-5    0-6     0-6  2.50230217
##  4:     0     3    0-3     len   8.2    0-7    0-8     0-8  5.02710962
##  5:     0     4    0-4    Leaf    NA   <NA>   <NA>    <NA>  0.36000001
##  6:     0     5    0-5    Leaf    NA   <NA>   <NA>    <NA> -0.36000001
##  7:     0     6    0-6     len  29.5    0-9   0-10    0-10  0.93020594
##  8:     0     7    0-7    Leaf    NA   <NA>   <NA>    <NA>  0.36000001
##  9:     0     8    0-8     len  10.0   0-11   0-12    0-12  0.60633492
## 10:     0     9    0-9     len  24.5   0-13   0-14    0-14  0.78028417
## 11:     0    10   0-10    Leaf    NA   <NA>   <NA>    <NA>  0.15000001
## 12:     0    11   0-11    Leaf    NA   <NA>   <NA>    <NA> -0.30000001
## 13:     0    12   0-12     len  13.6   0-15   0-16    0-16  2.92307687
## 14:     0    13   0-13    Leaf    NA   <NA>   <NA>    <NA>  0.06666667
## 15:     0    14   0-14    Leaf    NA   <NA>   <NA>    <NA> -0.17142859
## 16:     0    15   0-15    Leaf    NA   <NA>   <NA>    <NA>  0.20000002
## 17:     0    16   0-16    Leaf    NA   <NA>   <NA>    <NA> -0.30000001
## 18:     1     0    1-0     len  19.7    1-1    1-2     1-2  3.51329851
## 19:     1     1    1-1    dose   1.0    1-3    1-4     1-4  1.63309026
## 20:     1     2    1-2    dose   2.0    1-5    1-6     1-6  1.65485406
## 21:     1     3    1-3     len   8.2    1-7    1-8     1-8  3.56799269
## 22:     1     4    1-4    Leaf    NA   <NA>   <NA>    <NA>  0.28835031
## 23:     1     5    1-5    Leaf    NA   <NA>   <NA>    <NA> -0.28835031
## 24:     1     6    1-6     len  26.7    1-9   1-10    1-10  0.22153124
## 25:     1     7    1-7    Leaf    NA   <NA>   <NA>    <NA>  0.30163023
## 26:     1     8    1-8     len  11.2   1-11   1-12    1-12  0.25236940
## 27:     1     9    1-9     len  24.5   1-13   1-14    1-14  0.44972166
## 28:     1    10   1-10    Leaf    NA   <NA>   <NA>    <NA>  0.05241550
## 29:     1    11   1-11    Leaf    NA   <NA>   <NA>    <NA> -0.21860033
## 30:     1    12   1-12    Leaf    NA   <NA>   <NA>    <NA> -0.03878851
## 31:     1    13   1-13    Leaf    NA   <NA>   <NA>    <NA>  0.05559399
## 32:     1    14   1-14    Leaf    NA   <NA>   <NA>    <NA> -0.13160129
##      Tree  Node     ID Feature Split    Yes     No Missing        Gain
##     <int> <int> <char>  <char> <num> <char> <char>  <char>       <num>
##         Cover
##         <num>
##  1: 15.000000
##  2:  7.500000
##  3:  7.500000
##  4:  4.750000
##  5:  2.750000
##  6:  2.750000
##  7:  4.750000
##  8:  1.500000
##  9:  3.250000
## 10:  3.750000
## 11:  1.000000
## 12:  1.000000
```

(continues on next page)

(continued from previous page)

```
## 13:  2.250000
## 14:  1.250000
## 15:  2.500000
## 16:  1.250000
## 17:  1.000000
## 18: 14.695991
## 19:  7.308470
## 20:  7.387520
## 21:  4.645680
## 22:  2.662790
## 23:  2.662790
## 24:  4.724730
## 25:  1.452431
## 26:  3.193249
## 27:  2.985818
## 28:  1.738913
## 29:  1.472866
## 30:  1.720383
## 31:  1.248612
## 32:  1.737206
##          Cover
##          <num>
```

Other features

XGBoost supports many additional features on top of its traditional gradient-boosting framework, including, among others:

- Building decision tree models with characteristics such as per-feature monotonicity constraints or interaction constraints.
- Calculating feature contributions in individual predictions.
- Using custom objectives and custom evaluation metrics.
- Fitting linear models.
- Fitting models on GPUs and/or on data that doesn't fit in RAM ("external memory").

See the online documentation - particularly the tutorials section - for a glimpse over further functionalities that XGBoost offers.

The low-level interface

In addition to the `xgboost(x, y, ...)` function, XGBoost also provides a lower-level interface for creating model objects through the function `xgb.train()`, which resembles the same `xgb.train` functions in other language bindings of XGBoost.

This `xgb.train()` interface exposes additional functionalities (such as user-supplied callbacks or external-memory data support) and performs fewer data validations and castings compared to the `xgboost()` function interface.

Some key differences between the two interfaces:

- Unlike `xgboost()` which takes R objects such as `matrix` or `data.frame` as inputs, the function `xgb.train()` uses XGBoost's own data container called "DMatrix", which can be created from R objects through the function `xgb.DMatrix()`. Note that there are other "DMatrix" constructors too, such as "xgb.QuantileDMatrix()", which might be more beneficial for some use-cases.

- A "DMatrix" object may contain a mixture of features/covariates, the response variable, observation weights, base margins, among others; and unlike `xgboost()`, requires its inputs to have already been encoded into the representation that XGBoost uses behind the scenes - for example, while `xgboost()` may take a `factor` object as "y", `xgb.DMatrix()` requires instead a binary response variable to be passed as a vector of zeros and ones.
- Hyperparameters are passed as function arguments in `xgboost()`, while they are passed as a named list to `xgb.train()`.
- The `xgb.train()` interface keeps less metadata about its inputs - for example, it will not add levels of factors as column names to estimated probabilities when calling `predict`.

Example usage of `xgb.train()`:

```
data("agaricus.train")
dmatrix <- xgb.DMatrix(
    data = agaricus.train$data,  # a sparse CSC matrix ('dgCMatrix')
    label = agaricus.train$label, # zeros and ones
    nthread = 1
)
booster <- xgb.train(
    data = dmatrix,
    nrounds = 10,
    params = xgb.params(
        objective = "binary:logistic",
        nthread = 1,
        max_depth = 3
    )
)

data("agaricus.test")
dmatrix_test <- xgb.DMatrix(agaricus.test$data, nthread = 1)
pred_prob <- predict(booster, dmatrix_test)
pred_raw <- predict(booster, dmatrix_test, outputmargin = TRUE)
```

Model objects produced by `xgb.train()` have class `xgb.Booster`, while model objects produced by `xgboost()` have class `xgboost`, which is a subclass of `xgb.Booster`. Their `predict` methods also take different arguments - for example, `predict.xgboost` has a `type` parameter, while `predict.xgb.Booster` controls this through binary arguments - but as `xgboost` is a subclass of `xgb.Booster`, methods for `xgb.Booster` can be called on `xgboost` objects if needed.

Utility functions in the XGBoost R package will work with both model classes - for example:

```
xgb.importance(model)
```

```
##    Feature      Gain     Cover Frequency
##     <char>     <num>     <num>     <num>
## 1:     len 0.7444265 0.6830449 0.7333333
## 2:    dose 0.2555735 0.3169551 0.2666667
```

```
xgb.importance(booster)
```

```
##                             Feature         Gain        Cover  Frequency
##                              <char>        <num>        <num>      <num>
##  1:                       odor=none 0.6083687503 0.3459792871 0.16949153
##  2:                 stalk-root=club 0.0959684807 0.0695742744 0.03389831
```

(continues on next page)

(continued from previous page)

```
##  3:                      odor=anise 0.0645662853 0.0777761744 0.10169492
##  4:                     odor=almond 0.0542574659 0.0865120182 0.10169492
##  5:                bruises?=bruises 0.0532525762 0.0535293301 0.06779661
##  6:               stalk-root=rooted 0.0471992509 0.0610565707 0.03389831
##  7:         spore-print-color=green 0.0326096192 0.1418126308 0.16949153
##  8:                       odor=foul 0.0153302980 0.0103517575 0.01694915
##  9: stalk-surface-below-ring=scaly 0.0126892940 0.0914230316 0.08474576
## 10:                 gill-size=broad 0.0066973198 0.0345993858 0.10169492
## 11:                    odor=pungent 0.0027091458 0.0032193586 0.01694915
## 12:            population=clustered 0.0025750464 0.0015616374 0.03389831
## 13:  stalk-color-below-ring=yellow 0.0016913567 0.0173903519 0.01694915
## 14:         spore-print-color=white 0.0012798160 0.0008031107 0.01694915
## 15:              gill-spacing=close 0.0008052948 0.0044110809 0.03389831
```

While `xgboost()` aims to provide a user-friendly interface, there are still many situations where one should prefer the `xgb.train()` interface - for example:

- For latency-sensitive applications (e.g. when serving models in real time), `xgb.train()` will have a speed advantage, as it performs fewer validations, conversions, and post-processings with metadata.
- If you are developing an R package that depends on XGBoost, `xgb.train()` will provide a more stable interface (less subject to changes) and will have lower time/memory overhead.
- If you need functionalities that are not exposed by the `xgboost()` interface - for example, if your dataset does not fit into the computer's RAM, it's still possible to construct a DMatrix from it if the data is loaded in batches through `xgb.ExtMemDMatrix()`.

XGBoost from JSON

Introduction

The purpose of this Vignette is to show you how to correctly load and work with an **XGBoost** model that has been dumped to JSON. **XGBoost** internally converts all data to 32-bit floats, and the values dumped to JSON are decimal representations of these values. When working with a model that has been parsed from a JSON file, care must be taken to correctly treat:

- the input data, which should be converted to 32-bit floats
- any 32-bit floats that were stored in JSON as decimal representations
- any calculations must be done with 32-bit mathematical operators

Setup

For the purpose of this tutorial we will load the xgboost, jsonlite, and float packages. We'll also set `digits=22` in our options in case we want to inspect many digits of our results.

```
require(xgboost)
```

```
## Loading required package: xgboost
```

```
require(jsonlite)
```

```
## Loading required package: jsonlite
```

```
require(float)
```

```
## Loading required package: float
```

```
options(digits = 22)
```

We will create a toy binary logistic model based on the example first provided here, so that we can easily understand the structure of the dumped JSON model object. This will allow us to understand where discrepancies can occur and how they should be handled.

```
dates <- c(20180130, 20180130, 20180130,
           20180130, 20180130, 20180130,
           20180131, 20180131, 20180131,
           20180131, 20180131, 20180131,
           20180131, 20180131, 20180131,
           20180134, 20180134, 20180134)

labels <- c(1, 1, 1,
            1, 1, 1,
            0, 0, 0,
            0, 0, 0,
            0, 0, 0,
            0, 0, 0)

data <- data.frame(dates = dates, labels = labels)

bst <- xgb.train(
  data = xgb.DMatrix(as.matrix(data$dates), label = labels, missing = NA, nthread = 1),
  nrounds = 1,
  params = xgb.params(
    objective = "binary:logistic",
    nthread = 2,
    max_depth = 1
  )
)
```

Comparing results

We will now dump the model to JSON and attempt to illustrate a variety of issues that can arise, and how to properly deal with them.

First let's dump the model to JSON:

```
bst_json <- xgb.dump(bst, with_stats = FALSE, dump_format = 'json')
bst_from_json <- fromJSON(bst_json, simplifyDataFrame = FALSE)
node <- bst_from_json[[1]]
cat(bst_json)
```

```
## [
##   { "nodeid": 0, "depth": 0, "split": "f0", "split_condition": 20180132, "yes": 1, "no
↪": 2, "missing": 2 , "children": [
##     { "nodeid": 1, "leaf": 0.514285684 },
```

(continues on next page)

(continued from previous page)

```
##     { "nodeid": 2, "leaf": -0.327272743 }
##   ]}
## ]
```

The tree JSON shown by the above code-chunk tells us that if the data is less than 20180132, the tree will output the value in the first leaf. Otherwise it will output the value in the second leaf. Let's try to reproduce this manually with the data we have and confirm that it matches the model predictions we've already calculated.

```
bst_preds_logodds <- predict(bst, as.matrix(data$dates), outputmargin = TRUE)

# calculate the logodds values using the JSON representation
bst_from_json_logodds <- ifelse(data$dates < node$split_condition,
                                node$children[[1]]$leaf,
                                node$children[[2]]$leaf)

bst_preds_logodds
```

```
##  [1] -0.1788614988327026367188 -0.1788614988327026367188
##  [3] -0.1788614988327026367188 -0.1788614988327026367188
##  [5] -0.1788614988327026367188 -0.1788614988327026367188
##  [7] -1.0204199552536010742188 -1.0204199552536010742188
##  [9] -1.0204199552536010742188 -1.0204199552536010742188
## [11] -1.0204199552536010742188 -1.0204199552536010742188
## [13] -1.0204199552536010742188 -1.0204199552536010742188
## [15] -1.0204199552536010742188 -1.0204199552536010742188
## [17] -1.0204199552536010742188 -1.0204199552536010742188
```

```
bst_from_json_logodds
```

```
##  [1]  0.5142856839999999651880  0.5142856839999999651880
##  [3]  0.5142856839999999651880  0.5142856839999999651880
##  [5]  0.5142856839999999651880  0.5142856839999999651880
##  [7]  0.5142856839999999651880  0.5142856839999999651880
##  [9]  0.5142856839999999651880  0.5142856839999999651880
## [11]  0.5142856839999999651880  0.5142856839999999651880
## [13]  0.5142856839999999651880  0.5142856839999999651880
## [15]  0.5142856839999999651880 -0.3272727429999999770871
## [17] -0.3272727429999999770871 -0.3272727429999999770871
```

```
# test that values are equal
bst_preds_logodds == bst_from_json_logodds
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [13] FALSE FALSE FALSE FALSE FALSE FALSE
```

None are equal. What happened?

At this stage two things happened:

- input data was not converted to 32-bit floats
- the JSON variables were not converted to 32-bit floats

Lesson 1: All data is 32-bit floats

When working with imported JSON, all data must be converted to 32-bit floats

To explain this, let's repeat the comparison and round to two decimals:

```
round(bst_preds_logodds, 2) == round(bst_from_json_logodds, 2)
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [13] FALSE FALSE FALSE FALSE FALSE FALSE
```

If we round to two decimals, we see that only the elements related to data values of `20180131` don't agree. If we convert the data to floats, they agree:

```
# now convert the dates to floats first
bst_from_json_logodds <- ifelse(fl(data$dates) < node$split_condition,
                                node$children[[1]]$leaf,
                                node$children[[2]]$leaf)

# test that values are equal
round(bst_preds_logodds, 2) == round(bst_from_json_logodds, 2)
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [13] FALSE FALSE FALSE FALSE FALSE FALSE
```

What's the lesson? If we are going to work with an imported JSON model, any data must be converted to floats first. In this case, since '20180131' cannot be represented as a 32-bit float, it is rounded up to 20180132, as shown here:

```
fl(20180131)
```

```
## # A float32 vector: 1
## [1] 20180132
```

Lesson 2: JSON parameters are 32-bit floats

All JSON parameters stored as floats must be converted to floats.

Let's now say we do care about numbers past the first two decimals.

```
# test that values are equal
bst_preds_logodds == bst_from_json_logodds
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [13] FALSE FALSE FALSE FALSE FALSE FALSE
```

None are exactly equal. What happened? Although we've converted the data to 32-bit floats, we also need to convert the JSON parameters to 32-bit floats. Let's do this:

```
# now convert the dates to floats first
bst_from_json_logodds <- ifelse(fl(data$dates) < fl(node$split_condition),
                                as.numeric(fl(node$children[[1]]$leaf)),
                                as.numeric(fl(node$children[[2]]$leaf)))
```

(continues on next page)

(continued from previous page)

```
# test that values are equal
bst_preds_logodds == bst_from_json_logodds
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [13] FALSE FALSE FALSE FALSE FALSE FALSE
```

All equal. What's the lesson? If we are going to work with an imported JSON model, any JSON parameters that were stored as floats must also be converted to floats first.

Lesson 3: Use 32-bit math

> Always use 32-bit numbers and operators

We were able to get the log-odds to agree, so now let's manually calculate the sigmoid of the log-odds. This should agree with the xgboost predictions.

```
bst_preds <- predict(bst, as.matrix(data$dates))

# calculate the predictions casting doubles to floats
bst_from_json_preds <- ifelse(
  fl(data$dates) < fl(node$split_condition)
  , as.numeric(1 / (1 + exp(-1 * fl(node$children[[1]]$leaf))))
  , as.numeric(1 / (1 + exp(-1 * fl(node$children[[2]]$leaf))))
)

# test that values are equal
bst_preds == bst_from_json_preds
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [13] FALSE FALSE FALSE FALSE FALSE FALSE
```

None are exactly equal again. What is going on here? Well, since we are using the value 1 in the calculations, we have introduced a double into the calculation. Because of this, all float values are promoted to 64-bit doubles and the 64-bit version of the exponentiation operator `exp` is also used. On the other hand, xgboost uses the 32-bit version of the exponentiation operator in its sigmoid function.

How do we fix this? We have to ensure we use the correct data types everywhere and the correct operators. If we use only floats, the float library that we have loaded will ensure the 32-bit float exponentiation operator is applied.

```
# calculate the predictions casting doubles to floats
bst_from_json_preds <- ifelse(
  fl(data$dates) < fl(node$split_condition)
  , as.numeric(fl(1) / (fl(1) + exp(fl(-1) * fl(node$children[[1]]$leaf))))
  , as.numeric(fl(1) / (fl(1) + exp(fl(-1) * fl(node$children[[2]]$leaf))))
)

# test that values are equal
bst_preds == bst_from_json_preds
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [13] FALSE FALSE FALSE FALSE FALSE FALSE
```

All equal. What's the lesson? We have to ensure that all calculations are done with 32-bit floating point operators if we want to reproduce the results that we see with xgboost.

1.11.3 Other topics

Migrating code from previous XGBoost versions

XGBoost's R language bindings had large breaking changes between versions 1.x and 2.x. R code that was working with past XGBoost versions might require modifications to work with the newer versions. This guide outlines the main differences:

- **Function `xgboost()`:**
  - Previously, this function accepted arguments 'data' and 'label', which have now been renamed to 'x' and 'y', in line with other popular R packages.
  - Previously, the 'data' argument which is now 'x' had to be passed as either an XGBoost 'DMatrix' or as an R matrix. Now the argument allows R data.frames, matrices, and sparse matrices from the 'Matrix' package, but not XGBoost's own DMatrices. Categorical columns will be deduced from the types of the columns when passing a data.frame.
  - Previously, the 'label' data which is now 'y' had to be passed to `xgboost()` encoded in the format used by the XGBoost core library - meaning: binary variables had to be encoded to 0/1, bounds for survival objectives had to be passed as different arguments, among others. In the newest versions, 'y' now doesn't need to be manually encoded beforehand: it should be passed as an R object of the corresponding class as regression functions from base R and core R packages for the corresponding XGBoost objective - e.g. classification problems should be passed a `factor`, survival problems a `Surv`, regression problems a numeric vector, and so on. Learning-to-rank is not supported by `xgboost()`, but is supported by `xgb.train`.
  - Previously, `xgboost()` accepted both a `params` argument and named arguments under `...`. Now all training parameters should be passed as named arguments, and all accepted parameters are explicit function arguments with in-package documentation. Some parameters are not allowed as they are determined automatically from the rest of the data, such as the number of classes for multi-classes classification which is determined automatically from 'y'. As well, parameters that have synonyms or which are accepted under different possible arguments (e.g. "eta" and "learning_rate") now accept only their more descriptive form (so "eta" is not accepted, but "learning_rate" is).
  - Models produced by this function `xgboost()` are now returned with a different class "xgboost", which is a subclass of "xgb.Booster" but with more metadata and a `predict` method with different defaults.
  - This function `xgboost()` is now meant for interactive usage only. For package developers who wish to incorporate the XGBoost package, it is highly recommended to use `xgb.train` instead, which is a lower-level function that closely mimics the same function from the Python package and is meant to be less subject to breaking changes.
- **Function `xgb.train()`:**
  - Previously, `xgb.train()` allowed arguments under both a "params" list and as named arguments under `...`. Now, all training arguments should be passed under `params`.
  - In order to make it easier to discover and pass parameters, there is now a function `xgb.params` which can generate a list to pass to the `params` argument. `xgb.params` is simply a function with named arguments that lists everything accepted by `xgb.train` and offers in-package documentation for all of the arguments, returning a simple named list.
  - Arguments that are meant to be consumed by the DMatrix constructor must be passed directly to `xgb.DMatrix` instead (e.g. argument for categorical features or for feature names).
  - Some arguments have been renamed (e.g. previous 'watchlist' is now 'evals', in line with the Python package).
  - The format of the callbacks to pass to `xgb.train` has largely been re-written. See the documentation of `xgb.Callback` for details.

- **Function `xgb.DMatrix()`:**
    - This function now accepts 'data.frame' inputs and determines which features are categorical from their types - anything with type 'factor' or 'character' will be considered as categorical. Note that when passing data to the 'predict' method, the 'factor' variables must have the same encoding (i.e. same levels) as XGBoost will not re-encode them for you.
    - Whereas previously some arguments such as the type of the features had to be passed as a list under argument 'info', they are all now direct function arguments to 'xgb.DMatrix' instead.
    - There are now other varieties of DMatrix constructors that might better fit some uses cases -for example, there is 'xgb.QuantileDMatrix' which will quantize the features straight away (therefore avoiding redundant copies and reducing memory consumption) for the histogram method in XGBoost (but note that quantized DMatrices are not usable with the 'exact' sorted-indices method).
    - Note that data for 'label' still needs to be encoded in the format consumed by the core XGBoost library - e.g. classification objectives should receive 'label' data encoded as zeros and ones.
    - Creation of DMatrices from text files has been deprecated.
- **Function `xgb.cv()`:**
    - While previously this function accepted 'data' and 'label' similarly to the old `xgboost()`, now it accepts only `xgb.DMatrix` objects.
    - The function's scope has been expanded to support more functionalities offered by XGBoost, such as survival and learning-to-rank objectives.
- **Method `predict`:**
    - There are now two predict methods with different default arguments according to whether the model was produced through `xgboost()` or through `xgb.train()`. Function `xgboost()` is more geared towards interactive usage, and thus the defaults for the 'predict' method on such objects (class "xgboost") by default will perform more data validations such as checking that column names match and reordering them otherwise. The 'predict' method for models created through `xgb.train()` (class "xgb.Booster") has the same defaults as before, so for example it will not reorder columns to match names under the default behavior.
    - The 'predict' method for objects of class "xgboost" (produced by `xgboost()`, not by `xgb.train()`) now can control the types of predictions to make through an argument `type`, similarly as the 'predict' methods in the 'stats' module of base R - e.g. one can now do `predict(model, type="class")`; while the 'predict' method for "xgb.Booster" objects (produced by `xgb.train()`), just like before, controls those through separate arguments such as `outputmargin`.
    - Previously, predictions using a subset of the trees were using base-0 indexing and range syntax mimicing Python's ranges, whereas now they use base-1 indexing as is common in R, and their behavior for ranges matches that of R's `seq` function. Note that the syntax for "use all trees" and "use trees up to early-stopped criteria" have changed (see documentation for details).
- **Booster objects:**
    - The structure of these objects has been modified - now they are represented as a simple R "ALTLIST" (a special kind of 'list' object) with additional attributes.
    - These objects now cannot be modified by adding more fields to them, but metadata for them can be added as attributes.
    - The objects distinguish between two types of attributes:
        * R-side attributes (which can be accessed and modified through R function `attributes(model)` and `attributes(model)$field <- val`), which allow arbitrary objects. Many attributes are

automatically added by the model building functions, such as evaluation logs (a `data.table` with metrics calculated per iteration), which previously were model fields.

  * C-level attributes, which allow only JSON-compliant data and which can be accessed and set through function `xgb.attributes(model)`. These C-level attributes are shareable through serialized models in different XGBoost interfaces, while the R-level ones are specific to the R interface. Some attributes that are standard among language bindings of XGBoost, such as the best interation, are kept as C attributes.

- Previously, models that were just de-serialized from an on-disk format required calling method 'xgb.Booster.complete' on them to finish the full de-serialization process before being usable, or would otherwise call this method on their own automatically automatically at the first call to 'predict'. Serialization is now handled more gracefully, and there are no additional functions/methods involved - i.e. if one saves a model to disk with `saveRDS()` and then reads it back with `readRDS()`, the model will be fully loaded straight away, without needing to call additional methods on it.

Other recommendations

By default, XGBoost might recognize that some parameter has been removed or renamed from how it was in a previous version, and still accept the same function call as it used to do before with the renamed or removed arugments, but issuing a deprecation warning along the way that highlights the changes.

These behaviors will be removed in future versions, and function calls which currently return deprecation warnings will stop working in the future, so in order to make sure that code calling XGBoost will still keep working, it should be ensured that it doesn't issue deprecation warnings.

Optionally, these deprecation warnings can be turned into errors (while still keeping other types of warnings as warnings) through an option "xgboost.strict_mode" - example:

```
options("xgboost.strict_mode" = TRUE)
```

It can also be controlled through an environment variable *XGB_STRICT_MODE=1*, which takes precende over the R option - e.g.:

```
Sys.setenv("XGB_STRICT_MODE" = "1")
```

It is highly recommended for package developers to enable this option during their package checks to ensure better compatibility with XGBoost.

Handling of indexable elements

There are many functionalities in XGBoost which refer to indexable elements in a countable set, such as boosting rounds / iterations / trees in a model (which can be referred to by number), classes, categories / levels in categorical features, among others.

XGBoost, being written in C++, uses base-0 indexing and considers ranges / sequences to be inclusive of the left end but not the right one - for example, a range (0, 3) would include the first three elements, numbered 0, 1, and 2.

The Python interface uses this same logic, since this is also the way that indexing in Python works, but other languages like R have different logic. In R, indexing is base-1 and ranges / sequences are inclusive of both ends - for example, to refer to the first three elements in a sequence, the interval would be written as (1, 3), and the elements numbered 1, 2, and 3.

In order to provide a more idiomatic R interface, XGBoost adjusts its user-facing R interface to follow this and similar R conventions, but internally, it needs to convert all these numbers to the format that the C interface uses. This is made more problematic by the fact that models are meant to be serializable and loadable in other interfaces, which will have different indexing logic.

The following adjustments are made in the R interface:

- Slicing method for DMatrix, which takes an array of integers, is converted to base-0 indexing by subtracting 1 from each element. Note that this is done in the C-level wrapper function for R, unlike all other conversions which are done in R before being passed to C.
- Slicing method for Booster takes a sequence defined by start, end, and step. The R interface is made to work the same way as R's `seq` from the user's POV, so it always adjusts the left end by subtracting one, and depending on whether the step size ends exactly or not at the right end, will also adjust the right end to be non-inclusive in C indexing.
- Parameter `iterationrange` in `predict` is also made to behave the same way as R's `seq`. Since it doesn't have a step size, just adjusting the left end by subtracting 1 suffices here.
- `best_iteration`, depending on the context, might be stored as both a C-level booster attribute, and as an R attribute. Since the C-level attributes are shared across interfaces and used in prediction methods, in order to improve compatibility, it leaves this C-level attribute in base-0 indexing, but the R attribute, if present, will be adjusted to base-1 indexing. Note that the `predict` method in R and other interfaces will look at the C-level attribute only.
- Other references to iteration numbers or boosting rounds, such as when printing metrics or saving model snapshots, also follow base-1 indexing. These other references are coded entirely in R, as the C-level functions do not handle such functionalities.
- Terminal leaf / node numbers are returned in base-0 indexing, just like they come from the C interface.
- Tree numbers in plots follow base-1 indexing. Note that these are only displayed when producing these plots through the R interface's own handling of DiagrammeR objects, but not when using the C-level GraphViz 'dot' format generator for plots.
- Feature numbers when producing feature importances, JSONs, trees-to-tables, and SHAP; are all following base-0 indexing.
- Categorical features are defined in R as a `factor` type which encodes with base-1 indexing. When categorical features are passed as R `factor` types, the conversion is done automatically to base-0 indexing, but if the user whishes to manually supply categorical features as already-encoded integers, then those integers need to already be in base-0 encoding.
- Categorical levels (categories) in outputs such as plots, JSONs, and trees-to-tables; are also referred to using base-0 indexing, regardless of whether they went into the model as integers or as `factor`-typed columns.
- Categorical labels for DMatrices do not undergo any extra processing - the user must supply base-0 encoded labels.
- A function to retrieve class-specific coefficients when using the linear coefficients history callback takes a class index parameter, which also does not undergo any conversion (i.e. user must pass a base-0 index), in order to match with the label logic - that is, the same class index will refer to the class encoded with that number in the DMatrix `label` field.

New additions to the R interface that take on indexable elements should be mindful of these conventions and try to mimic R's behavior as much as possible.

Developer guide: parameters from core library

The XGBoost core library accepts a long list of input parameters (e.g. `max_depth` for decision trees, regularization, `device` where compute happens, etc.). New parameters are constantly being added as XGBoost is developed further, and their language bindings should allow passing to the core library everything that it accepts.

In the case of R, these parameters are passed as an R `list` object to function `xgb.train`, but the R interface aims at providing a better, more idiomatic user experience by offering a parameters constructor with full in-package documentation. This requires keeping the list of parameters and their documentation up to date **in the R package** too, in addition to the general online documentation for XGBoost.

In more detail, there is a function `xgb.params` which allows the user to construct such a `list` object to pass to `xgb.train` while getting full IDE autocompletion on it. This function should accept all possible XGBoost parameters as arguments, listing them in the same order as they appear in the online documentation.

In order to add a new parameter from the core library to `xgb.params`:

- Add the parameter at the right location, according to the order in which it appears in the .rst file listing the parameters for the core library. If the parameter appears more than once (e.g. because it applies to more than one type of booster), then add it in a position according to to the first occurrence.
- Copy-paste the docs from the .rst file as another `@param` entry for `xgb.train`. Some easy substitutions might be needed, such as changing double-backticks to single-backticks, enquoting variables that need to be passed as strings, and replacing `:math:` calls with their roxygen equivalent `\eqn{}`, among others.
- If needed, make minimal modifications for the R interface - for example, since parameters are only listed once, should add at the beginning a note about which type of booster they apply to if they are only applicable for one type, or list default values by booster type if they are different.

After adding the parameter to `xgb.params`, it will also need to be added to the function `xgboost` if that function can use it. The function `xgboost` is not meant to support everything that the core library offers - currently parameters related to learning-to-rank are not listed there for example as they are unusable for it (but can be used for `xgb.train`).

In order to add the parameter to `xgboost`:

- Add it to the function signature. The position here differs though: there are a few selected parameters whose positions have been moved closer to the top of the signature. New parameters should not be placed within those "top" positions - instead, place it after parameter `tree_method`, in the most similar place among the remaining parameters according to how it was inserted in `xgb.params`. Note that the rest of the parameters that come after `tree_method` are still meant to follow the same relative order as in `xgb.params`.
- If the parameter applies exactly in the same way as in `xgb.train`, then no additional documentation is needed for `xgboost`, because it inherits parameters from `xgb.params` by default. However, some parameters might need slight modifications - for example, not all objectives are supported by `xgboost`, so modifications are needed for that parameter.
- If the parameter allows aliases, use only one alias, and prefer the most descriptive nomenclature (e.g. "learning_rate" instead of "eta"). These also need a doc entry `@param` in `xgboost`, as the one in `xgb.params` will have the unsupported alias.

As new objectives and evaluation metrics are added, be mindful that they need to be added to the docs of both `xgb.params` and `xgboost`. Documentation for objectives in both functions was originally copied from the same .rst file for the core library, but for `xgboost` it undergoes additional modifications in order to list what is and isn't supported, and to refer only to the parameter aliases that are accepted by `xgboost`.

Keep in mind also that objectives that are a variant of one another but with a different prediction mode, are not meant to be allowed in `xgboost` as they'd break its intended interface - therefore, such objectives are not described in the docs for `xgboost` (but there is a list at the end of what isn't supported by it) and are checked against in function `prescreen.objective`.

1.12 XGBoost JVM Package

You have found the XGBoost JVM Package!

1.12.1 Installation

Checkout the *Installation Guide* for how to install the jvm package, or *Building from Source* on how to build it from the sources.

1.12.2 Contents

Getting Started with XGBoost4J

This tutorial introduces Java API for XGBoost.

Data Interface

Like the XGBoost python module, XGBoost4J uses DMatrix to handle data. LIBSVM txt format file, sparse matrix in CSR/CSC format, and dense matrix are supported.

- The first step is to import DMatrix:

```
import ml.dmlc.xgboost4j.java.DMatrix;
```

- Use DMatrix constructor to load data from a libsvm text format file:

```
DMatrix dmat = new DMatrix("train.svm.txt");
```

- Pass arrays to DMatrix constructor to load from sparse matrix.

  Suppose we have a sparse matrix

```
1 0 2 0
4 0 0 3
3 1 2 0
```

  We can express the sparse matrix in Compressed Sparse Row (CSR) format:

```
long[] rowHeaders = new long[] {0,2,4,7};
float[] data = new float[] {1f,2f,4f,3f,3f,1f,2f};
int[] colIndex = new int[] {0,2,0,3,0,1,2};
int numColumn = 4;
DMatrix dmat = new DMatrix(rowHeaders, colIndex, data, DMatrix.SparseType.CSR,␣
↪numColumn);
```

  … or in Compressed Sparse Column (CSC) format:

```
long[] colHeaders = new long[] {0,3,4,6,7};
float[] data = new float[] {1f,4f,3f,1f,2f,2f,3f};
int[] rowIndex = new int[] {0,1,2,2,0,2,1};
int numRow = 3;
DMatrix dmat = new DMatrix(colHeaders, rowIndex, data, DMatrix.SparseType.CSC,␣
↪numRow);
```

- You may also load your data from a dense matrix. Let’s assume we have a matrix of form

```
1    2
3    4
5    6
```

  Using row-major layout, we specify the dense matrix as follows:

```
float[] data = new float[] {1f,2f,3f,4f,5f,6f};
int nrow = 3;
int ncol = 2;
```

(continues on next page)

(continued from previous page)

```
float missing = 0.0f;
DMatrix dmat = new DMatrix(data, nrow, ncol, missing);
```

- To set weight:

```
float[] weights = new float[] {1f,2f,1f};
dmat.setWeight(weights);
```

Setting Parameters

To set parameters, parameters are specified as a Map:

```
Map<String, Object> params = new HashMap<String, Object>() {
  {
    put("eta", 1.0);
    put("max_depth", 2);
    put("objective", "binary:logistic");
    put("eval_metric", "logloss");
  }
};
```

Training Model

With parameters and data, you are able to train a booster model.

- Import Booster and XGBoost:

```
import ml.dmlc.xgboost4j.java.Booster;
import ml.dmlc.xgboost4j.java.XGBoost;
```

- Training

```
DMatrix trainMat = new DMatrix("train.svm.txt");
DMatrix validMat = new DMatrix("valid.svm.txt");
// Specify a watch list to see model accuracy on data sets
Map<String, DMatrix> watches = new HashMap<String, DMatrix>() {
  {
    put("train", trainMat);
    put("test", testMat);
  }
};
int nround = 2;
Booster booster = XGBoost.train(trainMat, params, nround, watches, null, null);
```

- Saving model

  After training, you can save model and dump it out.

```
booster.saveModel("model.json");
```

- Generating model dump with feature map

```java
// dump without feature map
String[] model_dump = booster.getModelDump(null, false);
// dump with feature map
String[] model_dump_with_feature_map = booster.getModelDump("featureMap.txt",␣
↪false);
```

- Load a model

```java
Booster booster = XGBoost.loadModel("model.json");
```

Prediction

After training and loading a model, you can use it to make prediction for other data. The result will be a two-dimension float array `(nsample, nclass)`; for `predictLeaf()`, the result would be of shape `(nsample, nclass*ntrees)`.

```java
DMatrix dtest = new DMatrix("test.svm.txt");
// predict
float[][] predicts = booster.predict(dtest);
// predict leaf
float[][] leafPredicts = booster.predictLeaf(dtest, 0);
```

XGBoost4J-Spark Tutorial

XGBoost4J-Spark is a project aiming to seamlessly integrate XGBoost and Apache Spark by fitting XGBoost to Apache Spark's MLLIB framework. With the integration, user can not only uses the high-performant algorithm implementation of XGBoost, but also leverages the powerful data processing engine of Spark for:

- Feature Engineering: feature extraction, transformation, dimensionality reduction, and selection, etc.
- Pipelines: constructing, evaluating, and tuning ML Pipelines
- Persistence: persist and load machine learning models and even whole Pipelines

This tutorial is to cover the end-to-end process to build a machine learning pipeline with XGBoost4J-Spark. We will discuss

- Using Spark to preprocess data to fit to XGBoost4J-Spark's data interface
- Training a XGBoost model with XGBoost4J-Spark
- Serving XGBoost model (prediction) with Spark
- Building a Machine Learning Pipeline with XGBoost4J-Spark
- Running XGBoost4J-Spark in Production

- *Build an ML Application with XGBoost4J-Spark*
  - *Refer to XGBoost4J-Spark Dependency*
  - *Data Preparation*
    - *Read Dataset with Spark's Built-In Reader*
    - *Transform Raw Iris Dataset*
      - *Dealing with missing values*

  - *Training*
    - *Early Stopping*
    - *Training with Evaluation Dataset*
  - *Prediction*
    - *Batch Prediction*
    - *Single instance prediction*
  - *Model Persistence*
    - *Model and pipeline persistence*
    - *Interact with Other Bindings of XGBoost*
- *Building a ML Pipeline with XGBoost4J-Spark*
  - *Basic ML Pipeline*
  - *Pipeline with Hyper-parameter Tunning*
- *Run XGBoost4J-Spark in Production*
  - *Parallel/Distributed Training*
  - *Gang Scheduling*
  - *Checkpoint During Training*
- *External Memory*

Build an ML Application with XGBoost4J-Spark

Refer to XGBoost4J-Spark Dependency

Before we go into the tour of how to use XGBoost4J-Spark, you should first consult *Installation from Maven repository* in order to add XGBoost4J-Spark as a dependency for your project. We provide both stable releases and snapshots.

Note

XGBoost4J-Spark requires Apache Spark 3.0+

XGBoost4J-Spark now requires **Apache Spark 3.0+**. Latest versions of XGBoost4J-Spark uses facilities of *org.apache.spark.ml.param.shared* extensively to provide for a tight integration with Spark MLLIB framework, and these facilities are not fully available on earlier versions of Spark.

Also, make sure to install Spark directly from Apache website. **Upstream XGBoost is not guaranteed to work with third-party distributions of Spark, such as Cloudera Spark.** Consult appropriate third parties to obtain their distribution of XGBoost.

Warning

Spark 4.0.0 is not compatible with XGBoost4J-Spark

Apache Spark 4.0.0 introduced a breaking change to the `org.apache.spark.ml.param.Param` class constructor (SPARK-52259), which causes a `NoSuchMethodError` when instantiating any XGBoost estimator:

```
java.lang.NoSuchMethodError: 'void org.apache.spark.ml.param.Param.<init>(
    org.apache.spark.ml.util.Identifiable, java.lang.String, java.lang.String, scala.
↪Function1)'
```

This issue affects all third-party ML libraries that use `Param`. It was fixed in **Spark 4.0.1** and **Spark 4.1.0**. If you are using Spark 4.x, please upgrade to **Spark 4.0.1 or later**.

Data Preparation

As aforementioned, XGBoost4J-Spark seamlessly integrates Spark and XGBoost. The integration enables users to apply various types of transformation over the training/test datasets with the convenient and powerful data processing framework: Spark.

In this section, we use Iris dataset as an example to showcase how we use Spark to transform raw dataset and make it fit to the data interface of XGBoost.

Iris dataset is shipped in CSV format. Each instance contains 4 features, "sepal length", "sepal width", "petal length" and "petal width". In addition, it contains the "class" column, which is essentially the label with three possible values: "Iris Setosa", "Iris Versicolour" and "Iris Virginica".

Read Dataset with Spark's Built-In Reader

The first thing in data transformation is to load the dataset as Spark's structured data abstraction, DataFrame.

```scala
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.types.{DoubleType, StringType, StructField, StructType}

val spark = SparkSession.builder().getOrCreate()
val schema = new StructType(Array(
  StructField("sepal length", DoubleType, true),
  StructField("sepal width", DoubleType, true),
  StructField("petal length", DoubleType, true),
  StructField("petal width", DoubleType, true),
  StructField("class", StringType, true)))
val rawInput = spark.read.schema(schema).csv("input_path")
```

At the first line, we create a instance of SparkSession which is the entry of any Spark program working with DataFrame. The `schema` variable defines the schema of DataFrame wrapping Iris data. With this explicitly set schema, we can define the columns' name as well as their types; otherwise the column name would be the default ones derived by Spark, such as `_col0`, etc. Finally, we can use Spark's built-in csv reader to load Iris csv file as a DataFrame named `rawInput`.

Spark also contains many built-in readers for other format. The latest version of Spark supports CSV, JSON, Parquet, and LIBSVM.

Transform Raw Iris Dataset

To make Iris dataset be recognizable to XGBoost, we need to

1. Transform String-typed label, i.e. "class", to Double-typed label.
2. Assemble the feature columns as a vector to fit to the data interface of Spark ML framework.

To convert String-typed label to Double, we can use Spark's built-in feature transformer StringIndexer.

```
import org.apache.spark.ml.feature.StringIndexer
val stringIndexer = new StringIndexer().
  setInputCol("class").
  setOutputCol("classIndex").
  fit(rawInput)
val labelTransformed = stringIndexer.transform(rawInput).drop("class")
```

With a newly created StringIndexer instance:

1. we set input column, i.e. the column containing String-typed label.
2. we set output column, i.e. the column containing the Double-typed label.
3. Then we `fit` StringIndex with our input DataFrame `rawInput`, so that Spark internals can get information like total number of distinct values, etc.

Now we have a StringIndexer which is ready to be applied to our input DataFrame. To execute the transformation logic of StringIndexer, we `transform` the input DataFrame `rawInput` and to keep a concise DataFrame, we drop the column "class" and only keeps the feature columns and the transformed Double-typed label column (in the last line of the above code snippet).

The `fit` and `transform` are two key operations in MLLIB. Basically, `fit` produces a "transformer", e.g. StringIndexer, and each transformer applies `transform` method on DataFrame to add new column(s) containing transformed features/labels or prediction results, etc. To understand more about `fit` and `transform`, You can find more details in here.

Similarly, we can use another transformer, VectorAssembler, to assemble feature columns "sepal length", "sepal width", "petal length" and "petal width" as a vector.

```
import org.apache.spark.ml.feature.VectorAssembler
val vectorAssembler = new VectorAssembler().
  setInputCols(Array("sepal length", "sepal width", "petal length", "petal width")).
  setOutputCol("features")
val xgbInput = vectorAssembler.transform(labelTransformed).select("features", "classIndex
↪")
```

Now, we have a DataFrame containing only two columns, "features" which contains vector-represented "sepal length", "sepal width", "petal length" and "petal width" and "classIndex" which has Double-typed labels. A DataFrame like this (containing vector-represented features and numeric labels) can be fed to XGBoost4J-Spark's training engine directly.

Dealing with missing values

XGBoost supports missing values by default (as desribed here). If given a SparseVector, XGBoost will treat any values absent from the SparseVector as missing. You are also able to specify to XGBoost to treat a specific value in your Dataset as if it was a missing value. By default XGBoost will treat NaN as the value representing missing.

Example of setting a missing value (e.g. -999) to the "missing" parameter in XGBoostClassifier:

```
import ml.dmlc.xgboost4j.scala.spark.XGBoostClassifier
val xgbParam = Map("eta" -> 0.1f,
      "missing" -> -999,
      "objective" -> "multi:softprob",
      "num_class" -> 3,
      "num_round" -> 100,
      "num_workers" -> 2)
val xgbClassifier = new XGBoostClassifier(xgbParam).
```

(continues on next page)

(continued from previous page)

```
      setFeaturesCol("features").
      setLabelCol("classIndex")
```

Note

Missing values

If the feature is vector type, the single feature instance could be a SparseVector, where "0" will be treated as the missing value. In order to get the correct model, XGBoost4j-Spark will convert the SparseVector to array by restoring the "0". However, we can't assume 0 for missing values as it may be meaningful. So in this case, users need to specify the missing value explicitly even the missing value has been set to *Float.NaN* by default in the XGBoost4j-Spark.

Training

XGBoost supports regression, classification and ranking. While we use Iris dataset in this tutorial to show how we use XGBoost4J-Spark to resolve a multi-classes classification problem, the usage in Regression and Ranking is very similar to classification.

To train a XGBoost model for classification, we need to create a XGBoostClassifier first:

```
import ml.dmlc.xgboost4j.scala.spark.XGBoostClassifier
val xgbParam = Map("eta" -> 0.1f,
      "max_depth" -> 2,
      "objective" -> "multi:softprob",
      "num_class" -> 3)
val xgbClassifier = new XGBoostClassifier(xgbParam).
      setNumRound(100).
      setNumWorkers(2).
      setFeaturesCol("features").
      setLabelCol("classIndex")
```

The available parameters for training a XGBoost model can be found in *here*. In XGBoost4J-Spark, we support not only the default set of parameters but also the camel-case variant of these parameters to keep consistent with Spark's MLLIB parameters.

Specifically, each parameter in *this page* has its equivalent form in XGBoost4J-Spark with camel case. For example, to set `max_depth` for each tree, you can pass parameter just like what we did in the above code snippet (as `max_depth` wrapped in a Map), or you can do it through setters in XGBoostClassifer:

```
val xgbClassifier = new XGBoostClassifier().
  setFeaturesCol("features").
  setLabelCol("classIndex")
xgbClassifier.setMaxDepth(2)
```

After we set XGBoostClassifier parameters and feature/label column, we can build a transformer, XGBoostClassificationModel by fitting XGBoostClassifier with the input DataFrame. This `fit` operation is essentially the training process and the generated model can then be used in prediction.

```
val xgbClassificationModel = xgbClassifier.fit(xgbInput)
```

Early Stopping

Early stopping is a feature to prevent the unnecessary training iterations. By specifying num_early_stopping_rounds or directly call setNumEarlyStoppingRounds over a XGBoostClassifier or XGBoostRegressor, we can define number of rounds if the evaluation metric going away from the best iteration and early stop training iterations.

When it comes to custom eval metrics, in additional to num_early_stopping_rounds, you also need to define maximize_evaluation_metrics or call setMaximizeEvaluationMetrics to specify whether you want to maximize or minimize the metrics in training. For built-in eval metrics, XGBoost4J-Spark will automatically select the direction.

For example, we need to maximize the evaluation metrics (set maximize_evaluation_metrics with true), and set num_early_stopping_rounds with 5. The evaluation metric of 10th iteration is the maximum one until now. In the following iterations, if there is no evaluation metric greater than the 10th iteration's (best one), the training would be early stopped at 15th iteration.

Training with Evaluation Dataset

You can also monitor the performance of the model during training with evaluation dataset. By calling setEvalDataset over a XGBoostClassifier, XGBoostRegressor or XGBoostRanker.

Prediction

XGBoost4j-Spark supports two ways for model serving: batch prediction and single instance prediction.

Batch Prediction

When we get a model, either XGBoostClassificationModel, XGBoostRegressionModel or XGBoostRankerModel, it takes a DataFrame, read the column containing feature vectors, predict for each feature vector, and output a new DataFrame with the following columns by default:

- XGBoostClassificationModel will output margins (rawPredictionCol), probabilities(probabilityCol) and the eventual prediction labels (predictionCol) for each possible label.
- XGBoostRegressionModel will output prediction label(predictionCol).
- XGBoostRankerModel will output prediction label(predictionCol).

Batch prediction expects the user to pass the testset in the form of a DataFrame. XGBoost4J-Spark starts a XGBoost worker for each partition of DataFrame for parallel prediction and generates prediction results for the whole DataFrame in a batch.

```
val xgbClassificationModel = xgbClassifier.fit(xgbInput)
val results = xgbClassificationModel.transform(testSet)
```

With the above code snippet, we get a result DataFrame, result containing margin, probability for each class and the prediction for each instance

```
+-----------------+----------+--------------------+--------------------+----------+
|         features|classIndex|       rawPrediction|         probability|prediction|
+-----------------+----------+--------------------+--------------------+----------+
|[5.1,3.5,1.4,0.2]|       0.0|[3.45569849014282...|[0.99579632282257...|       0.0|
|[4.9,3.0,1.4,0.2]|       0.0|[3.45569849014282...|[0.99618089199066...|       0.0|
|[4.7,3.2,1.3,0.2]|       0.0|[3.45569849014282...|[0.99643349647521...|       0.0|
|[4.6,3.1,1.5,0.2]|       0.0|[3.45569849014282...|[0.99636095762252...|       0.0|
|[5.0,3.6,1.4,0.2]|       0.0|[3.45569849014282...|[0.99579632282257...|       0.0|
```

(continues on next page)

(continued from previous page)

```
|[5.4,3.9,1.7,0.4]|       0.0|[3.45569849014282...|[0.99428516626358...|       0.0|
|[4.6,3.4,1.4,0.3]|       0.0|[3.45569849014282...|[0.99643349647521...|       0.0|
|[5.0,3.4,1.5,0.2]|       0.0|[3.45569849014282...|[0.99579632282257...|       0.0|
|[4.4,2.9,1.4,0.2]|       0.0|[3.45569849014282...|[0.99618089199066...|       0.0|
|[4.9,3.1,1.5,0.1]|       0.0|[3.45569849014282...|[0.99636095762252...|       0.0|
|[5.4,3.7,1.5,0.2]|       0.0|[3.45569849014282...|[0.99428516626358...|       0.0|
|[4.8,3.4,1.6,0.2]|       0.0|[3.45569849014282...|[0.99643349647521...|       0.0|
|[4.8,3.0,1.4,0.1]|       0.0|[3.45569849014282...|[0.99618089199066...|       0.0|
|[4.3,3.0,1.1,0.1]|       0.0|[3.45569849014282...|[0.99618089199066...|       0.0|
|[5.8,4.0,1.2,0.2]|       0.0|[3.45569849014282...|[0.97809928655624...|       0.0|
|[5.7,4.4,1.5,0.4]|       0.0|[3.45569849014282...|[0.97809928655624...|       0.0|
|[5.4,3.9,1.3,0.4]|       0.0|[3.45569849014282...|[0.99428516626358...|       0.0|
|[5.1,3.5,1.4,0.3]|       0.0|[3.45569849014282...|[0.99579632282257...|       0.0|
|[5.7,3.8,1.7,0.3]|       0.0|[3.45569849014282...|[0.97809928655624...|       0.0|
|[5.1,3.8,1.5,0.3]|       0.0|[3.45569849014282...|[0.99579632282257...|       0.0|
+-----------------+----------+--------------------+--------------------+----------+
```

Single instance prediction

XGBoostClassificationModel, XGBoostRegressionModel or XGBoostRankerModel supports making prediction on single instance as well. It accepts a single Vector as feature, and output the prediction label.

However, the overhead of single-instance prediction is high due to the internal overhead of XGBoost, use it carefully!

```
val features = xgbInput.head().getAs[Vector]("features")
val result = xgbClassificationModel.predict(features)
```

Model Persistence

Model and pipeline persistence

A data scientist produces an ML model and hands it over to an engineering team for deployment in a production environment. Reversely, a trained model may be used by data scientists, for example as a baseline, across the process of data exploration. So it's important to support model persistence to make the models available across usage scenarios and programming languages.

XGBoost4j-Spark supports saving and loading XGBoostClassifier/XGBoostClassificationModel and XGBoostRegressor/XGBoostRegressionModel and XGBoostRanker/XGBoostRankerModel to/from file system. It also supports saving and loading a ML pipeline which includes these estimators and models.

We can save the XGBoostClassificationModel to file system:

```
val xgbClassificationModelPath = "/tmp/xgbClassificationModel"
xgbClassificationModel.write.overwrite().save(xgbClassificationModelPath)
```

and then loading the model in another session:

```
import ml.dmlc.xgboost4j.scala.spark.XGBoostClassificationModel

val xgbClassificationModel2 = XGBoostClassificationModel.load(xgbClassificationModelPath)
xgbClassificationModel2.transform(xgbInput)
```

Note

Besides dumping the model to raw format, users are able to dump the model to be json or ubj format.

```
val xgbClassificationModelPath = "/tmp/xgbClassificationModel"
xgbClassificationModel.write.overwrite().option("format", "json").
→save(xgbClassificationModelPath)
```

With regards to ML pipeline save and load, please refer the next section.

Interact with Other Bindings of XGBoost

After we train a model with XGBoost4j-Spark on massive dataset, sometimes we want to do model serving in single machine or integrate it with other single node libraries for further processing.

After saving the model, we can load this model with single node Python XGBoost directly.

```
val xgbClassificationModelPath = "/tmp/xgbClassificationModel"
xgbClassificationModel.write.overwrite().save(xgbClassificationModelPath)
```

```
import xgboost as xgb
bst = xgb.Booster({'nthread': 4})
bst.load_model("/tmp/xgbClassificationModel/data/model")
```

Note

Consistency issue between XGBoost4J-Spark and other bindings

There is a consistency issue between XGBoost4J-Spark and other language bindings of XGBoost.

When users use Spark to load training/test data in LIBSVM format with the following code snippet:

```
spark.read.format("libsvm").load("trainingset_libsvm")
```

Spark assumes that the dataset is using 1-based indexing (feature indices staring with 1). However, when you do prediction with other bindings of XGBoost (e.g. Python API of XGBoost), XGBoost assumes that the dataset is using 0-based indexing (feature indices starting with 0) by default. It creates a pitfall for the users who train model with Spark but predict with the dataset in the same format in other bindings of XGBoost. The solution is to transform the dataset to 0-based indexing before you predict with, for example, Python API, or you append `?indexing_mode=1` to your file path when loading with DMatirx. For example in Python:

```
xgb.DMatrix('test.libsvm?indexing_mode=1')
```

Building a ML Pipeline with XGBoost4J-Spark

Basic ML Pipeline

Spark ML pipeline can combine multiple algorithms or functions into a single pipeline. It covers from feature extraction, transformation, selection to model training and prediction. XGBoost4j-Spark makes it feasible to embed XGBoost into such a pipeline seamlessly. The following example shows how to build such a pipeline consisting of Spark MLlib feature transformer and XGBoostClassifier estimator.

We still use Iris dataset and the `rawInput` DataFrame. First we need to split the dataset into training and test dataset.

```
val Array(training, test) = rawInput.randomSplit(Array(0.8, 0.2), 123)
```

The we build the ML pipeline which includes 4 stages:

- Assemble all features into a single vector column.
- From string label to indexed double label.
- Use XGBoostClassifier to train classification model.
- Convert indexed double label back to original string label.

We have shown the first three steps in the earlier sections, and the last step is finished with a new transformer IndexToString:

```
val labelConverter = new IndexToString()
.setInputCol("prediction")
.setOutputCol("realLabel")
.setLabels(stringIndexer.labels)
```

We need to organize these steps as a Pipeline in Spark ML framework and evaluate the whole pipeline to get a PipelineModel:

```
import org.apache.spark.ml.feature._
import org.apache.spark.ml.Pipeline

val pipeline = new Pipeline()
    .setStages(Array(assembler, stringIndexer, booster, labelConverter))
val model = pipeline.fit(training)
```

After we get the PipelineModel, we can make prediction on the test dataset and evaluate the model accuracy.

```
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator

val prediction = model.transform(test)
val evaluator = new MulticlassClassificationEvaluator()
val accuracy = evaluator.evaluate(prediction)
```

Pipeline with Hyper-parameter Tunning

The most critical operation to maximize the power of XGBoost is to select the optimal parameters for the model. Tuning parameters manually is a tedious and labor-consuming process. With the latest version of XGBoost4J-Spark, we can utilize the Spark model selecting tool to automate this process.

The following example shows the code snippet utilizing CrossValidation and MulticlassClassificationEvaluator to search the optimal combination of two XGBoost parameters, `max_depth` and `eta`. (See *XGBoost Parameters*.) The model producing the maximum accuracy defined by MulticlassClassificationEvaluator is selected and used to generate the prediction for the test set.

```
import org.apache.spark.ml.tuning._
import org.apache.spark.ml.PipelineModel
import ml.dmlc.xgboost4j.scala.spark.XGBoostClassificationModel

val paramGrid = new ParamGridBuilder()
    .addGrid(booster.maxDepth, Array(3, 8))
    .addGrid(booster.eta, Array(0.2, 0.6))
```

(continues on next page)

(continued from previous page)

```
    .build()
val cv = new CrossValidator()
    .setEstimator(pipeline)
    .setEvaluator(evaluator)
    .setEstimatorParamMaps(paramGrid)
    .setNumFolds(3)

val cvModel = cv.fit(training)

val bestModel = cvModel.bestModel.asInstanceOf[PipelineModel].stages(2)
    .asInstanceOf[XGBoostClassificationModel]
bestModel.extractParamMap()
```

Run XGBoost4J-Spark in Production

XGBoost4J-Spark is one of the most important steps to bring XGBoost to production environment easier. In this section, we introduce three key features to run XGBoost4J-Spark in production.

Parallel/Distributed Training

The massive size of training dataset is one of the most significant characteristics in production environment. To ensure that training in XGBoost scales with the data size, XGBoost4J-Spark bridges the distributed/parallel processing framework of Spark and the parallel/distributed training mechanism of XGBoost.

In XGBoost4J-Spark, each XGBoost worker is wrapped by a Spark task and the training dataset in Spark's memory space is fed to XGBoost workers in a transparent approach to the user.

In the code snippet where we build XGBoostClassifier, we set parameter `num_workers` (or `numWorkers`). This parameter controls how many parallel workers we want to have when training a XGBoostClassificationModel.

Note

Regarding OpenMP optimization

By default, we allocate a core per each XGBoost worker. Therefore, the OpenMP optimization within each XGBoost worker does not take effect and the parallelization of training is achieved by running multiple workers (i.e. Spark tasks) at the same time.

If you do want OpenMP optimization, you have to

1. set `nthread` to a value larger than 1 when creating XGBoostClassifier/XGBoostRegressor
2. set `spark.task.cpus` in Spark to the same value as `nthread`

Gang Scheduling

XGBoost uses AllReduce. algorithm to synchronize the stats, e.g. histogram values, of each worker during training. Therefore XGBoost4J-Spark requires that all of `nthread * numWorkers` cores should be available before the training runs.

In the production environment where many users share the same cluster, it's hard to guarantee that your XGBoost4J-Spark application can get all requested resources for every run. By default, the communication layer in XGBoost will block the whole application when it requires more resources to be available. This process usually brings unnecessary resource waste as it keeps the ready resources and try to claim more. Additionally, this usually happens silently and does not bring the attention of users.

XGBoost4J-Spark allows the user to setup a timeout threshold for claiming resources from the cluster. If the application cannot get enough resources within this time period, the application would fail instead of wasting resources for hanging long. To enable this feature, you can set with XGBoostClassifier/XGBoostRegressor/XGBoostRanker:

```
xgbClassifier.setRabitTrackerTimeout(60000L)
```

or pass in `rabit_tracker_timeout` in `xgbParamMap` when building XGBoostClassifier:

```
val xgbParam = Map("eta" -> 0.1f,
   "max_depth" -> 2,
   "objective" -> "multi:softprob",
   "num_class" -> 3,
   "num_round" -> 100,
   "num_workers" -> 2,
   "rabit_tracker_timeout" -> 60000L)
val xgbClassifier = new XGBoostClassifier(xgbParam).
    setFeaturesCol("features").
    setLabelCol("classIndex")
```

If XGBoost4J-Spark cannot get enough resources for running two XGBoost workers, the application would fail. Users can have external mechanism to monitor the status of application and get notified for such case.

Checkpoint During Training

Transient failures are also commonly seen in production environment. To simplify the design of XGBoost, we stop training if any of the distributed workers fail. However, if the training fails after having been through a long time, it would be a great waste of resources.

We support creating checkpoint during training to facilitate more efficient recovery from failure. To enable this feature, you can set how many iterations we build each checkpoint with `setCheckpointInterval` and the location of checkpoints with `setCheckpointPath`:

```
xgbClassifier.setCheckpointInterval(2)
xgbClassifier.setCheckpointPath("/checkpoint_path")
```

An equivalent way is to pass in parameters in XGBoostClassifier's constructor:

```
val xgbParam = Map("eta" -> 0.1f,
   "max_depth" -> 2,
   "objective" -> "multi:softprob",
   "num_class" -> 3,
   "num_round" -> 100,
   "num_workers" -> 2,
   "checkpoint_path" -> "/checkpoints_path",
   "checkpoint_interval" -> 2)
val xgbClassifier = new XGBoostClassifier(xgbParam).
    setFeaturesCol("features").
    setLabelCol("classIndex")
```

If the training failed during these 100 rounds, the next run of training would start by reading the latest checkpoint file in `/checkpoints_path` and start from the iteration when the checkpoint was built until to next failure or the specified 100 rounds.

External Memory

Added in version 3.0.

Warning

The feature is experimental.

Here we refer to the iterator-based external memory instead of the one that uses special URL parameters. XGBoost-Spark has experimental support for GPU-based external memory training (*XGBoost4J-Spark-GPU Tutorial*) since 3.0. When it's used in combination with GPU-based training, data is first cached on disk and then staged on CPU memory. See *Using XGBoost External Memory Version* for general concept and best practices for the external memory training. In addition, see the doc string of the estimator parameter *useExternalMemory*. With Spark estimators:

```
val xgbClassifier = new XGBoostClassifier(xgbParam)
    .setFeaturesCol(featuresNames)
    .setLabelCol(labelName)
    .setUseExternalMemory(true)
    .setDevice("cuda")  // CPU is not yet supported
```

XGBoost4J-Spark-GPU Tutorial

XGBoost4J-Spark-GPU is an open source library aiming to accelerate distributed XGBoost training on Apache Spark cluster from end to end with GPUs by leveraging the RAPIDS Accelerator for Apache Spark product.

This tutorial will show you how to use **XGBoost4J-Spark-GPU**.

- *Build an ML Application with XGBoost4J-Spark-GPU*
  - *Add XGBoost to Your Project*
  - *Data Preparation*
    - *Read Dataset with Spark's Built-In Reader*
    - *Transform Raw Iris Dataset*
  - *Training*
  - *Prediction*
- *Submit the application*
- *RMM Support*

Build an ML Application with XGBoost4J-Spark-GPU

Add XGBoost to Your Project

Prior to delving into the tutorial on utilizing XGBoost4J-Spark-GPU, it is advisable to refer to *Installation from Maven repository* for instructions on adding XGBoost4J-Spark-GPU as a project dependency. We offer both stable releases and snapshots for your convenience.

Data Preparation

In this section, we use the Iris dataset as an example to showcase how we use Apache Spark to transform a raw dataset and make it fit the data interface of XGBoost.

The Iris dataset is shipped in CSV format. Each instance contains 4 features, “sepal length”, “sepal width”, “petal length” and “petal width”. In addition, it contains the “class” column, which is essentially the label with three possible values: “Iris Setosa”, “Iris Versicolour” and “Iris Virginica”.

Read Dataset with Spark’s Built-In Reader

```
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.types.{DoubleType, StringType, StructField, StructType}

val spark = SparkSession.builder().getOrCreate()

val labelName = "class"
val schema = new StructType(Array(
    StructField("sepal length", DoubleType, true),
    StructField("sepal width", DoubleType, true),
    StructField("petal length", DoubleType, true),
    StructField("petal width", DoubleType, true),
    StructField(labelName, StringType, true)))

val xgbInput = spark.read.option("header", "false")
    .schema(schema)
    .csv(dataPath)
```

At first, we create an instance of a SparkSession which is the entry point of any Spark application working with DataFrames. The `schema` variable defines the schema of the DataFrame wrapping Iris data. With this explicitly set schema, we can define the column names as well as their types; otherwise the column names would be the default ones derived by Spark, such as `_col0`, etc. Finally, we can use Spark’s built-in CSV reader to load the Iris CSV file as a DataFrame named `xgbInput`.

Apache Spark also contains many built-in readers for other formats such as ORC, Parquet, Avro, JSON.

Transform Raw Iris Dataset

To make the Iris dataset recognizable to XGBoost, we need to encode the String-typed label, i.e. “class”, to the Double-typed label.

One way to convert the String-typed label to Double is to use Spark’s built-in feature transformer StringIndexer. But this feature is not accelerated in RAPIDS Accelerator, which means it will fall back to CPU. Instead, we use an alternative way to achieve the same goal with the following code:

```
import org.apache.spark.sql.expressions.Window
import org.apache.spark.sql.functions._

val spec = Window.orderBy(labelName)
val Array(train, test) = xgbInput
    .withColumn("tmpClassName", dense_rank().over(spec) - 1)
    .drop(labelName)
    .withColumnRenamed("tmpClassName", labelName)
    .randomSplit(Array(0.7, 0.3), seed = 1)
```

(continues on next page)

(continued from previous page)

```
train.show(5)
```

```
+------------+-----------+------------+-----------+-----+
|sepal length|sepal width|petal length|petal width|class|
+------------+-----------+------------+-----------+-----+
|         4.3|        3.0|         1.1|        0.1|    0|
|         4.4|        2.9|         1.4|        0.2|    0|
|         4.4|        3.0|         1.3|        0.2|    0|
|         4.4|        3.2|         1.3|        0.2|    0|
|         4.6|        3.2|         1.4|        0.2|    0|
+------------+-----------+------------+-----------+-----+
```

With window operations, we have mapped the string column of labels to label indices.

Training

XGBoost4j-Spark-Gpu supports regression, classification and ranking models. Although we use the Iris dataset in this tutorial to show how we use `XGBoost4J-Spark-GPU` to resolve a multi-classes classification problem, the usage in Regression and Ranking is very similar to classification.

To train a XGBoost model for classification, we need to define a XGBoostClassifier first:

```
import ml.dmlc.xgboost4j.scala.spark.XGBoostClassifier
val xgbParam = Map(
    "objective" -> "multi:softprob",
    "num_class" -> 3,
    "num_round" -> 100,
    "device" -> "cuda",
    "num_workers" -> 1)

val featuresNames = schema.fieldNames.filter(name => name != labelName)

val xgbClassifier = new XGBoostClassifier(xgbParam)
    .setFeaturesCol(featuresNames)
    .setLabelCol(labelName)
```

The `device` parameter is for informing XGBoost that CUDA devices should be used instead of CPU. Unlike the single-node mode, GPUs are managed by spark instead of by XGBoost. Therefore, explicitly specified device ordinal like `cuda:1` is not support.

The available parameters for training a XGBoost model can be found in *here*. Similar to the XGBoost4J-Spark package, in addition to the default set of parameters, XGBoost4J-Spark-GPU also supports the camel-case variant of these parameters to be consistent with Spark's MLlib naming convention.

Specifically, each parameter in *this page* has its equivalent form in XGBoost4J-Spark-GPU with camel case. For example, to set `max_depth` for each tree, you can pass parameter just like what we did in the above code snippet (as `max_depth` wrapped in a Map), or you can do it through setters in XGBoostClassifer:

```
val xgbClassifier = new XGBoostClassifier(xgbParam)
    .setFeaturesCol(featuresNames)
    .setLabelCol(labelName)
xgbClassifier.setMaxDepth(2)
```

Note

In contrast with XGBoost4j-Spark which accepts both a feature column with VectorUDT type and an array of feature column names, XGBoost4j-Spark-GPU only accepts an array of feature column names by `setFeaturesCol(value: Array[String])`.

After setting XGBoostClassifier parameters and feature/label columns, we can build a transformer, XGBoostClassificationModel by fitting XGBoostClassifier with the input DataFrame. This `fit` operation is essentially the training process and the generated model can then be used in other tasks like prediction.

```
val xgbClassificationModel = xgbClassifier.fit(train)
```

Prediction

When we get a model, a XGBoostClassificationModel or a XGBoostRegressionModel or a XGBoostRankerModel, it takes a DataFrame as an input, reads the column containing feature vectors, predicts for each feature vector, and outputs a new DataFrame with the following columns by default:

- XGBoostClassificationModel will output margins (`rawPredictionCol`), probabilities(`probabilityCol`) and the eventual prediction labels (`predictionCol`) for each possible label.
- XGBoostRegressionModel will output prediction a label(`predictionCol`).
- XGBoostRankerModel will output prediction a label(`predictionCol`).

```
val xgbClassificationModel = xgbClassifier.fit(train)
val results = xgbClassificationModel.transform(test)
results.show()
```

With the above code snippet, we get a DataFrame as result, which contains the margin, probability for each class, and the prediction for each instance.

```
+------------+-----------+------------------+-------------------+-----+------------------
↪--+--------------------+----------+
|sepal length|sepal width|      petal length|        petal width|class|      ␣
↪rawPrediction|         probability|prediction|
+------------+-----------+------------------+-------------------+-----+------------------
↪--+--------------------+----------+
|         4.5|        2.3|               1.3|0.30000000000000004|    0|[3.16666603088378.
↪..|[0.98853939771652...|       0.0|
|         4.6|        3.1|               1.5|                0.2|    0|[3.25857257843017.
↪..|[0.98969423770904...|       0.0|
|         4.8|        3.1|               1.6|                0.2|    0|[3.25857257843017.
↪..|[0.98969423770904...|       0.0|
|         4.8|        3.4|               1.6|                0.2|    0|[3.25857257843017.
↪..|[0.98969423770904...|       0.0|
|         4.8|        3.4|1.9000000000000001|                0.2|    0|[3.25857257843017.
↪..|[0.98969423770904...|       0.0|
|         4.9|        2.4|               3.3|                1.0|    1|[-2.1498908996582.
↪..|[0.00596602633595...|       1.0|
|         4.9|        2.5|               4.5|                1.7|    2|[-2.1498908996582.
↪..|[0.00596602633595...|       1.0|
|         5.0|        3.5|               1.3|0.30000000000000004|    0|[3.25857257843017.
↪..|[0.98969423770904...|       0.0|
```

(continues on next page)

(continued from previous page)

```
|         5.1|        2.5|               3.0|               1.1|    1|[3.16666603088378.
↪..|[0.98853939771652...|       0.0|
|         5.1|        3.3|               1.7|               0.5|    0|[3.25857257843017.
↪..|[0.98969423770904...|       0.0|
|         5.1|        3.5|               1.4|               0.2|    0|[3.25857257843017.
↪..|[0.98969423770904...|       0.0|
|         5.1|        3.8|               1.6|               0.2|    0|[3.25857257843017.
↪..|[0.98969423770904...|       0.0|
|         5.2|        3.4|               1.4|               0.2|    0|[3.25857257843017.
↪..|[0.98969423770904...|       0.0|
|         5.2|        3.5|               1.5|               0.2|    0|[3.25857257843017.
↪..|[0.98969423770904...|       0.0|
|         5.2|        4.1|               1.5|               0.1|    0|[3.25857257843017.
↪..|[0.98969423770904...|       0.0|
|         5.4|        3.9|               1.7|               0.4|    0|[3.25857257843017.
↪..|[0.98969423770904...|       0.0|
|         5.5|        2.4|               3.8|               1.1|    1|[-2.1498908996582.
↪..|[0.00596602633595...|       1.0|
|         5.5|        4.2|               1.4|               0.2|    0|[3.25857257843017.
↪..|[0.98969423770904...|       0.0|
|         5.7|        2.5|               5.0|               2.0|    2|[-2.1498908996582.
↪..|[0.00280966912396...|       2.0|
|         5.7|        3.0|               4.2|               1.2|    1|[-2.1498908996582.
↪..|[0.00643939292058...|       1.0|
+------------+-----------+------------------+------------------+-----+------------------
↪--+-------------------+----------+
```

Submit the application

Assuming you have configured the Spark standalone cluster with GPU support. Otherwise, please refer to spark standalone configuration with GPU support.

Starting from XGBoost 2.1.0, stage-level scheduling is automatically enabled. Therefore, if you are using Spark standalone cluster version 3.4.0 or higher, we strongly recommend configuring the `"spark.task.resource.gpu.amount"` as a fractional value. This will enable running multiple tasks in parallel during the ETL phase. An example configuration would be `"spark.task.resource.gpu.amount=1/spark.executor.cores"`. However, if you are using a XGBoost version earlier than 2.1.0 or a Spark standalone cluster version below 3.4.0, you still need to set `"spark.task.resource.gpu.amount"` equal to `"spark.executor.resource.gpu.amount"`.

Assuming that the application main class is "Iris" and the application jar is "iris-1.0.0.jar", provided below is an instance demonstrating how to submit the xgboost application to an Apache Spark Standalone cluster.

```
rapids_version=24.08.0
xgboost_version=$LATEST_VERSION
main_class=Iris
app_jar=iris-1.0.0.jar

spark-submit \
  --master $master \
  --packages com.nvidia:rapids-4-spark_2.12:${rapids_version},ml.dmlc:xgboost4j-spark-
↪gpu_2.12:${xgboost_version} \
  --conf spark.executor.cores=12 \
  --conf spark.task.cpus=1 \
```

(continues on next page)

(continued from previous page)

```
  --conf spark.executor.resource.gpu.amount=1 \
  --conf spark.task.resource.gpu.amount=0.08 \
  --conf spark.rapids.sql.csv.read.double.enabled=true \
  --conf spark.rapids.sql.hasNans=false \
  --conf spark.plugins=com.nvidia.spark.SQLPlugin \
  --class ${main_class} \
   ${app_jar}
```

- First, we need to specify the `RAPIDS Accelerator, xgboost4j-spark-gpu` packages by `--packages`
- Second, `RAPIDS Accelerator` is a Spark plugin, so we need to configure it by specifying `spark.plugins=com.nvidia.spark.SQLPlugin`

For details about other `RAPIDS Accelerator` other configurations, please refer to the configuration.

For `RAPIDS Accelerator Frequently Asked Questions`, please refer to the frequently-asked-questions.

RMM Support

Added in version 3.0.

When compiled with the RMM plugin (see *Building From Source*), the XGBoost spark package can reuse the RMM memory pool automatically based on *spark.rapids.memory.gpu.pooling.enabled* and *spark.rapids.memory.gpu.pool*. Please note that both submit options need to be set accordingly. In addition, XGBoost employs NCCL for GPU communication, which requires some GPU memory for communication buffers and one should not let RMM take all the available memory. Example configuration related to memory pool:

```
spark-submit \
  --master $master \
  --conf spark.rapids.memory.gpu.allocFraction=0.5 \
  --conf spark.rapids.memory.gpu.maxAllocFraction=0.8 \
  --conf spark.rapids.memory.gpu.pool=ARENA \
  --conf spark.rapids.memory.gpu.pooling.enabled=true \
  ...
```

API Docs for the JVM packages

- XGBoost4J Java API
- XGBoost4J Scala API
- XGBoost4J-Spark Scala API
- XGBoost4J-Spark-GPU Scala API
- XGBoost4J-Flink Scala API

Migration Guide: How to migrate to XGBoost4j-Spark jvm 3.x

XGBoost4j-Spark jvm packages underwent significant modifications in version 3.0, which may cause compatibility issues with existing user code.

This guide will walk you through the process of updating your code to ensure it's compatible with XGBoost4j-Spark 3.0 and later versions.

XGBoost4j Spark Packages

XGBoost4j-Spark 3.0 has assembled xgboost4j package into xgboost4j-spark_2.12-3.0.0.jar, which means you can now simply use *xgboost4j-spark* for your application.

- For CPU

```
<dependency>
    <groupId>ml.dmlc</groupId>
    <artifactId>xgboost4j-spark_${scala.binary.version}</artifactId>
    <version>3.0.0</version>
</dependency>
```

- For GPU

```
<dependency>
    <groupId>ml.dmlc</groupId>
    <artifactId>xgboost4j-spark-gpu_${scala.binary.version}</artifactId>
    <version>3.0.0</version>
</dependency>
```

When submitting the XGBoost application to the Spark cluster, you only need to specify the single *xgboost4j-spark* package.

- For CPU

```
spark-submit \
  --jars xgboost4j-spark_2.12-3.0.0.jar \
  ... \
```

- For GPU

```
spark-submit \
  --jars xgboost4j-spark-gpu_2.12-3.0.0.jar \
  ... \
```

XGBoost Ranking

Learning to rank using XGBoostRegressor has been replaced by a dedicated *XGBoostRanker*, which is specifically designed to support ranking algorithms.

```
// before xgboost4j-spark 3.0
val regressor = new XGBoostRegressor().setObjective("rank:ndcg")

// after xgboost4j-spark 3.0
val ranker = new XGBoostRanker()
```

Removed Parameters

Starting from xgboost4j-spark 3.0, below parameters are removed.

- cacheTrainingSet

  If you wish to cache the training dataset, you have the option to implement caching in your code prior to fitting the data to an estimator.

```scala
val df = input.cache()
val model = new XGBoostClassifier().fit(df)
```

- trainTestRatio

  The following method can be employed to do the evaluation.

```scala
val Array(train, eval) = trainDf.randomSplit(Array(0.7, 0.3))
val classifier = new XGBoostClassifer().setEvalDataset(eval)
val model = classifier.fit(train)
```

- tracker_conf

  The following method can be used to configure RabitTracker.

```scala
val classifier = new XGBoostClassifer()
  .setRabitTrackerTimeout(100)
  .setRabitTrackerHostIp("192.168.0.2")
  .setRabitTrackerPort(19203)
```

- rabitRingReduceThreshold
- rabitTimeout
- rabitConnectRetry
- singlePrecisionHistogram
- lambdaBias
- objectiveType

Spark 4.0 Compatibility

XGBoost4J-Spark JARs built against Spark 3.5 are binary compatible with **Spark 4.0.1+** and **Spark 4.1.0+**. No code changes are required.

Warning

Spark 4.0.0 is **not** compatible due to an upstream bug (SPARK-52259) that changed the constructor signature of `org.apache.spark.ml.param.Param`. This causes a `NoSuchMethodError` at runtime when instantiating any XGBoost estimator. The fix is included in Spark 4.0.1 and later. Please avoid Spark 4.0.0 and upgrade to at least **Spark 4.0.1**.

Note

Please note that the flink interface is still under construction.

1.13 XGBoost.jl

See XGBoost.jl Project page.

1.14 XGBoost C Package

XGBoost implements a set of C API designed for various bindings, we maintain its stability and the CMake/make build interface. See *C API Tutorial* for an introduction and `demo/c-api/` for related examples. Also one can generate doxygen document by providing `-DBUILD_C_DOC=ON` as parameter to `CMake` during build, or simply look at function comments in `include/xgboost/c_api.h`. The reference is exported to sphinx with the help of breathe, which doesn't contain links to examples but might be easier to read. For the original doxygen pages please visit:

- C API documentation (latest master branch)
- C API documentation (last stable release)

1.14.1 C API Reference

- *Library*
- *DMatrix*
  - *Streaming*
- *Booster*
  - *Prediction*
  - *Serialization*
- *Collective*

Library

group **`Library`**

These functions are used to obtain general information about XGBoost including version, build info and current global configuration.

Typedefs

typedef void ***`DMatrixHandle`**

Handle to the DMatrix.

typedef void ***`BoosterHandle`**

Handle to the Booster.

typedef void ***`CategoriesHandle`**

Handle to the categories container.

Since

3.2.0

Functions

void **XGBoostVersion**(int *major, int *minor, int *patch)

Return the version of the XGBoost library.

The output variable is only written if it's not NULL.

Parameters

- **major** – Store the major version number.
- **minor** – Store the minor version number.
- **patch** – Store the patch (revision) number.

int **XGBuildInfo**(char const **out)

Get compile information of the shared XGBoost library.

Parameters

out – string encoded JSON object containing build flags and dependency versions.

Returns

0 when success, -1 when failure happens

const char ***XGBGetLastError**()

Get the string message of the last error.

Most functions in XGBoost returns 0 when success and non-zero when an error occurred. In the case of error, *XGBGetLastError* can be used to retrieve the error message

This function is thread safe.

Returns

The error message from the last error.

int **XGBRegisterLogCallback**(void (*callback)(const char*))

register callback function for LOG(INFO) messages — helpful messages that are not errors.

Note

This function can be called by multiple threads. The callback function will run on the thread that registered it.

Returns

0 when success, -1 when failure happens

int **XGBSetGlobalConfig**(char const *config)

Set global configuration (collection of parameters that apply globally). This function accepts the list of key-value pairs representing the global-scope parameters to be configured. The list of key-value pairs are passed in as a JSON string.

Parameters

config – a JSON string representing the list of key-value pairs. The JSON object shall be flat: no value can be a JSON object or an array.

Returns

0 when success, -1 when failure happens

int **XGBGetGlobalConfig**(char const **out_config)

Get current global configuration (collection of parameters that apply globally).

Parameters

out_config – pointer to received returned global configuration, represented as a JSON string.

Returns

0 when success, -1 when failure happens

DMatrix

group **DMatrix**

DMatrix is the basic data storage for XGBoost used by all XGBoost algorithms including both training, prediction and explanation. There are a few variants of `DMatrix` including normal `DMatrix`, which is a CSR matrix, `QuantileDMatrix`, which is used by histogram-based tree methods for saving memory, and lastly the experimental external-memory-based DMatrix, which reads data in batches during training. For the last two variants, see the *Streaming* group.

Functions

int **XGDMatrixCreateFromFile**(const char *fname, int silent, *DMatrixHandle* *out)

load a data matrix

Deprecated:

since 2.0.0

See also

XGDMatrixCreateFromURI()

Parameters

- **fname** – the name of the file
- **silent** – whether print messages during loading
- **out** – a loaded data matrix

Returns

0 when success, -1 when failure happens

int **XGDMatrixCreateFromURI**(char const *config, *DMatrixHandle* *out)

load a data matrix

Parameters

- **config** – JSON encoded parameters for DMatrix construction. Accepted fields are:
  - uri: The URI of the input file. The URI parameter `format` is required when loading text data.

    See *Text Input Format of DMatrix* for more info.
  - silent (optional): Whether to print message during loading. Default to true.

- data_split_mode (optional): Whether the file was split by row or column beforehand for distributed computing. Default to row.

- **out** – a loaded data matrix

Returns
0 when success, -1 when failure happens

int **XGDMatrixCreateFromColumnar**(char const *data, char const *config, *DMatrixHandle* *out)

Create a DMatrix from columnar data. (table)

A special type of input to the `DMatrix` is the columnar format, which refers to column-based dataframes. XGBoost can accept both numeric data types like integers and floats, along with the categorical type, called dictionary in arrow's term. The addition of categorical type is introduced in 3.1.0. The dataframe is represented by a list array interfaces with one object for each column.

A categorical type is represented by 3 buffers, the validity mask, the names of the categories (called index for most of the dataframe implementation), and the codes used to represent the categories in the rows. XGBoost consumes a categorical column by accepting two JSON-encoded arrow arrays in a list. The first item in the list is a JSON object with `{"offsets": IntegerArray, "values": StringArray }` representing the string names defined by the arrow columnar format. The second buffer is an masked integer array that stores the categorical codes along with the validity mask:

```
[
  // categorical column, represented as an array (list)
  [
    {
      'offsets':
      {
        'data': (129412626415808, True),
        'typestr': '<i4', 'version': 3, 'strides': None, 'shape': (3,), 'mask':␣
↪None
      },
      'values':
      {
        'data': (129412626416000, True),
        'typestr': '<i1', 'version': 3, 'strides': None, 'shape': (7,), 'mask':␣
↪None
      }
    },
    {
      'data': (106200854378448, True),
      'typestr': '<i1', 'version': 3, 'strides': None, 'shape': (2,), 'mask':␣
↪None
    }
  ],
  // numeric column, represented as an object, same number of rows as the␣
↪previous column (2)
  {
    'data': (106200854378448, True),
    'typestr': '<f4', 'version': 3, 'strides': None, 'shape': (2,), 'mask': None
  }
]
```

As for numeric inputs, it's the same as dense array.

Parameters

- **data** – A list of JSON-encoded array interfaces.
- **config** – See *XGDMatrixCreateFromDense* for details.
- **out** – The created DMatrix.

Returns
0 when success, -1 when failure happens

int **XGDMatrixCreateFromCSR**(char const *indptr, char const *indices, char const *data, bst_ulong ncol, char const *config, *DMatrixHandle* *out)

Create a DMatrix from CSR matrix.

Parameters

- **indptr** – JSON encoded **array_interface** to row pointers in CSR.
- **indices** – JSON encoded **array_interface** to column indices in CSR.
- **data** – JSON encoded **array_interface** to values in CSR.
- **ncol** – The number of columns.
- **config** – See *XGDMatrixCreateFromDense* for details.
- **out** – The created dmatrix

Returns
0 when success, -1 when failure happens

int **XGDMatrixCreateFromDense**(char const *data, char const *config, *DMatrixHandle* *out)

Create a DMatrix from dense array.

The array interface is defined in https://numpy.org/doc/2.1/reference/arrays.interface.html We encode the interface as a JSON object.

Parameters

- **data** – JSON encoded **array_interface** to array values.
- **config** – JSON encoded configuration. Required values are:
  - missing: Which value to represent missing value.
  - nthread (optional): Number of threads used for initializing DMatrix.
  - data_split_mode (optional): Whether the data was split by row or column beforehand. Default to row.
- **out** – The created DMatrix

Returns
0 when success, -1 when failure happens

int **XGDMatrixCreateFromCSC**(char const *indptr, char const *indices, char const *data, bst_ulong nrow, char const *config, *DMatrixHandle* *out)

Create a DMatrix from a CSC matrix.

Parameters

- **indptr** – JSON encoded **array_interface** to column pointers in CSC.
- **indices** – JSON encoded **array_interface** to row indices in CSC.
- **data** – JSON encoded **array_interface** to values in CSC.
- **nrow** – The number of rows in the matrix.

- **config** – See *XGDMatrixCreateFromDense* for details.
- **out** – The created dmatrix.

Returns
0 when success, -1 when failure happens

int **XGDMatrixCreateFromMat**(const float *data, bst_ulong nrow, bst_ulong ncol, float missing, *DMatrixHandle* *out)

create matrix content from dense matrix

Parameters

- **data** – pointer to the data space
- **nrow** – number of rows
- **ncol** – number columns
- **missing** – which value to represent missing value
- **out** – created dmatrix

Returns
0 when success, -1 when failure happens

int **XGDMatrixCreateFromMat_omp**(const float *data, bst_ulong nrow, bst_ulong ncol, float missing, *DMatrixHandle* *out, int nthread)

create matrix content from dense matrix

Parameters

- **data** – pointer to the data space
- **nrow** – number of rows
- **ncol** – number columns
- **missing** – which value to represent missing value
- **out** – created dmatrix
- **nthread** – number of threads (up to maximum cores available, if <=0 use all cores)

Returns
0 when success, -1 when failure happens

int **XGDMatrixCreateFromCudaColumnar**(char const *data, char const *config, *DMatrixHandle* *out)

Create DMatrix from CUDA columnar format. (cuDF)

See *XGDMatrixCreateFromColumnar* for a brief description of the columnar format.

Parameters

- **data** – A list of JSON-encoded array interfaces.
- **config** – See *XGDMatrixCreateFromDense* for details.
- **out** – Created dmatrix

Returns
0 when success, -1 when failure happens

int **XGDMatrixCreateFromCudaArrayInterface**(char const *data, char const *config, *DMatrixHandle* *out)

Create DMatrix from CUDA array.

Parameters

- **data** – JSON encoded **cuda_array_interface** for array data.
- **config** – JSON encoded configuration. Required values are:
  - missing: Which value to represent missing value.
  - nthread (optional): Number of threads used for initializing DMatrix.
  - data_split_mode (optional): Whether the data was split by row or column beforehand. Default to row.
- **out** – created dmatrix

Returns
0 when success, -1 when failure happens

int **XGDMatrixSliceDMatrix**(*DMatrixHandle* handle, const int *idxset, bst_ulong len, *DMatrixHandle* *out)

create a new dmatrix from sliced content of existing matrix

Parameters

- **handle** – instance of data matrix to be sliced
- **idxset** – index set
- **len** – length of index set
- **out** – a sliced new matrix

Returns
0 when success, -1 when failure happens

int **XGDMatrixSliceDMatrixEx**(*DMatrixHandle* handle, const int *idxset, bst_ulong len, *DMatrixHandle* *out, int allow_groups)

create a new dmatrix from sliced content of existing matrix

Parameters

- **handle** – instance of data matrix to be sliced
- **idxset** – index set
- **len** – length of index set
- **out** – a sliced new matrix
- **allow_groups** – allow slicing of an array with groups

Returns
0 when success, -1 when failure happens

int **XGDMatrixFree**(*DMatrixHandle* handle)

Free a DMatrix object.

Returns
0 when success, -1 when failure happens

int **XGDMatrixSaveBinary**(*DMatrixHandle* handle, const char *fname, int silent)

Save the DMatrix object into a file. `QuantileDMatrix` and external memory DMatrix are not supported.

Parameters

- **handle** – a instance of data matrix
- **fname** – File name

- **silent** – print statistics when saving

Returns
0 when success, -1 when failure happens

int **XGDMatrixSetInfoFromInterface**(*DMatrixHandle* handle, char const *field, char const *data)

Set content in array interface to a content in info.

Parameters

- **handle** – An instance of data matrix
- **field** – Field name.
- **data** – JSON encoded **array_interface** to values in the dense matrix/vector.

Returns
0 when success, -1 when failure happens

int **XGDMatrixSetFloatInfo**(*DMatrixHandle* handle, const char *field, const float *array, bst_ulong len)

set float vector to a content in info

Parameters

- **handle** – a instance of data matrix
- **field** – field name, can be label, weight
- **array** – pointer to float vector
- **len** – length of array

Returns
0 when success, -1 when failure happens

int **XGDMatrixSetUIntInfo**(*DMatrixHandle* handle, const char *field, const unsigned *array, bst_ulong len)

Deprecated:
since 2.1.0

Use *XGDMatrixSetInfoFromInterface* instead.

int **XGDMatrixSetStrFeatureInfo**(*DMatrixHandle* handle, const char *field, const char **features, const bst_ulong size)

Set string encoded information of all features.

Accepted fields are:

- feature_name
- feature_type

```
char const* feat_names [] {"feat_0", "feat_1"};
XGDMatrixSetStrFeatureInfo(handle, "feature_name", feat_names, 2);

// i for integer, q for quantitive, c for categorical.  Similarly "int" and
↪"float"
// are also recognized.
char const* feat_types [] {"i", "q"};
XGDMatrixSetStrFeatureInfo(handle, "feature_type", feat_types, 2);
```

Parameters

- **handle** – An instance of data matrix
- **field** – Field name
- **features** – Pointer to array of strings.
- **size** – Size of `features` pointer (number of strings passed in).

Returns

0 when success, -1 when failure happens

int **XGDMatrixGetStrFeatureInfo**(*DMatrixHandle* handle, const char *field, bst_ulong *size, const char ***out_features)

Get string encoded information of all features.

Accepted fields are:

- feature_name
- feature_type

Caller is responsible for copying out the data, before next call to any API function of XGBoost.

```
char const **c_out_features = NULL;
bst_ulong out_size = 0;

// Asumming the feature names are already set by `XGDMatrixSetStrFeatureInfo`.
XGDMatrixGetStrFeatureInfo(handle, "feature_name", &out_size,
                           &c_out_features)

for (bst_ulong i = 0; i < out_size; ++i) {
  // Here we are simply printing the string.  Copy it out if the feature name is
  // useful after printing.
  printf("feature %lu: %s\n", i, c_out_features[i]);
}
```

Parameters

- **handle** – An instance of data matrix
- **field** – Field name
- **size** – Size of output pointer `features` (number of strings returned).
- **out_features** – Address of a pointer to array of strings. Result is stored in thread local memory.

Returns

0 when success, -1 when failure happens

int **XGDMatrixGetCategories**(*DMatrixHandle* handle, char const *config, *CategoriesHandle* *out)

Create an opaque handle to the internal category container.

The container should be freed by *XGBCategoriesFree*

Since
3.2.0

```
DMatrixHandle fmat;
// Create a DMatrix from categorical data
// ...
CategoriesHandle cats;
int err = XGBoosterGetCategories(fmat, NULL, &cats)
if (err != 0) {
    exit(-1);
}
err = XGBCategoriesFree(cats);
if (err != 0) {
    exit(-1);
}
```

> **Note**
>
> Experimental API, subject to change in the future.

Parameters
- **handle** – An instance of the data matrix.
- **config** – Unused, reserved for the future.
- **out** – Created handle to the category container. Set to NULL if there's no category.

Returns
0 when success, -1 when failure happens.

int **XGDMatrixGetCategoriesExportToArrow**(*DMatrixHandle* handle, char const *config, *CategoriesHandle* *out, char const **export_out)

Create an opaque handle to the internal container and export it to arrow.

The container should be freed by *XGBCategoriesFree*

Since
3.2.0

> **Note**
>
> Experimental API, subject to change in the future.

Parameters
- **handle** – An instance of the data matrix.
- **config** – Unused, reserved for the future.
- **out** – Created handle to the category container

- **export_out** – JSON encoded array of categories, with length equal to the number of features.

Returns
0 when success, -1 when failure happens.

int **XGBCategoriesFree**(*CategoriesHandle* handle)

Free the opaque handle.

Since
3.2.0

Note

Experimental API, subject to change in the future.

Parameters
handle – An instance of the category container.

Returns
0 when success, -1 when failure happens.

int **XGDMatrixSetDenseInfo**(*DMatrixHandle* handle, const char *field, void const *data, bst_ulong size, int type)

Deprecated:

since 2.1.0

Use *XGDMatrixSetInfoFromInterface* instead.

int **XGDMatrixGetInfoRef**(*DMatrixHandle* handle, char const *field, char const **out_array)

Get a reference to data like label or weight.

This method replaces the existing *XGDMatrixGetFloatInfo* and *XGDMatrixGetUIntInfo* to support non-vector (like a matrix) output. The output data directly references the internal storage, as a result, it's read-only and user should copy data before the next XGBoost call.

Since
3.2.0

Parameters

- **handle** – An instance of data matrix
- **field** – Field name
- **out_array** – JSON encoded __(cuda)_array_interface__ to the output.

Returns
0 when success, -1 when failure happens

int **XGDMatrixGetFloatInfo**(const *DMatrixHandle* handle, const char *field, bst_ulong *out_len, const float **out_dptr)

get float info vector from matrix.

Parameters

- **handle** – a instance of data matrix
- **field** – field name
- **out_len** – used to set result length
- **out_dptr** – pointer to the result

Returns
0 when success, -1 when failure happens

int **XGDMatrixGetUIntInfo**(const *DMatrixHandle* handle, const char *field, bst_ulong *out_len, const unsigned **out_dptr)

get uint32 info vector from matrix

Parameters

- **handle** – a instance of data matrix
- **field** – field name
- **out_len** – The length of the field.
- **out_dptr** – pointer to the result

Returns
0 when success, -1 when failure happens

int **XGDMatrixNumRow**(*DMatrixHandle* handle, bst_ulong *out)

Get the number of rows from a DMatrix.

Parameters

- **handle** – the handle to the DMatrix
- **out** – The address to hold number of rows.

Returns
0 when success, -1 when failure happens

int **XGDMatrixNumCol**(*DMatrixHandle* handle, bst_ulong *out)

Get the number of columns from a DMatrix.

Parameters

- **handle** – the handle to the DMatrix
- **out** – The output of number of columns

Returns
0 when success, -1 when failure happens

int **XGDMatrixNumNonMissing**(*DMatrixHandle* handle, bst_ulong *out)

Get number of valid values from a DMatrix.

Parameters

- **handle** – the handle to the DMatrix
- **out** – The output of number of non-missing values

Returns
0 when success, -1 when failure happens

int **XGDMatrixDataSplitMode**(*DMatrixHandle* handle, bst_ulong *out)

Get the data split mode from DMatrix.

Parameters

- **handle** – the handle to the DMatrix
- **out** – The output of the data split mode

Returns
0 when success, -1 when failure happens

int **XGDMatrixGetDataAsCSR**(*DMatrixHandle* const handle, char const *config, bst_ulong *out_indptr, unsigned *out_indices, float *out_data)

Get the predictors from DMatrix as CSR matrix for testing. If this is a quantized DMatrix, quantized values are returned instead.

Unlike most of XGBoost C functions, caller of `XGDMatrixGetDataAsCSR` is required to allocate the memory for return buffer instead of using thread local memory from XGBoost. This is to avoid allocating a huge memory buffer that can not be freed until exiting the thread.

Since
1.7.0

Parameters

- **handle** – the handle to the DMatrix
- **config** – JSON configuration string. At the moment it should be an empty document, preserved for future use.
- **out_indptr** – indptr of output CSR matrix.
- **out_indices** – Column index of output CSR matrix.
- **out_data** – Data value of CSR matrix.

Returns
0 when success, -1 when failure happens

int **XGDMatrixGetQuantileCut**(*DMatrixHandle* const handle, char const *config, char const **out_indptr, char const **out_data)

Export the quantile cuts used for training histogram-based models like `hist` and `approx`. Useful for model compression.

Since
2.0.0

Parameters

- **handle** – the handle to the DMatrix
- **config** – JSON configuration string. At the moment it should be an empty document, preserved for future use.
- **out_indptr** – indptr of output CSC matrix represented by a JSON encoded __(cuda_)array_interface__.

- **out_data** – Data value of CSC matrix represented by a JSON encoded __(cuda_)array_interface__.

Streaming

group **Streaming**

Quantile DMatrix and external memory DMatrix can be created from batches of data.

There are 2 sets of data callbacks for DMatrix. The first one is currently exclusively used by JVM packages. It uses *XGBoostBatchCSR* to accept batches for CSR formated input, and concatenate them into 1 final big CSR. The related functions are:

- *XGBCallbackSetData*
- *XGBCallbackDataIterNext*
- *XGDMatrixCreateFromDataIter*

Another set is used by external data iterator. It accepts foreign data iterators as callbacks. There are 2 different senarios where users might want to pass in callbacks instead of raw data. First it's the Quantile DMatrix used by the hist and GPU-based hist tree method. For this case, the data is first compressed by quantile sketching then merged. This is particular useful for distributed setting as it eliminates 2 copies of data. First one by a `concat` from external library to make the data into a blob for normal DMatrix initialization, another one by the internal CSR copy of DMatrix.

The second use case is external memory support where users can pass a custom data iterator into XGBoost for loading data in batches. For both cases, the iterator is only used during the construction of the DMatrix and can be safely freed after construction finishes. There are short notes on each of the use cases in respected DMatrix factory function.

Related functions are:

Factory functions

- *XGDMatrixCreateFromCallback* for external memory
- *XGQuantileDMatrixCreateFromCallback* for quantile DMatrix
- *XGExtMemQuantileDMatrixCreateFromCallback* for External memory Quantile DMatrix

Proxy that callers can use to pass data to XGBoost

- *XGProxyDMatrixCreate*
- *XGDMatrixCallbackNext*
- *DataIterResetCallback*
- *XGProxyDMatrixSetDataCudaArrayInterface*
- *XGProxyDMatrixSetDataColumnar*
- *XGProxyDMatrixSetDataCudaColumnar*
- *XGProxyDMatrixSetDataDense*

- *XGProxyDMatrixSetDataCSR*
- … (data setters)

Typedefs

typedef void ***DataIterHandle**

handle to a external data iterator

typedef void ***DataHolderHandle**

handle to an internal data holder.

typedef int **XGBCallbackSetData**(*DataHolderHandle* handle, *XGBoostBatchCSR* batch)

Callback to set the data to handle,.

Param handle
The handle to the callback.

Param batch
The data content to be set.

typedef int **XGBCallbackDataIterNext**(*DataIterHandle* data_handle, *XGBCallbackSetData* *set_function, *DataHolderHandle* set_function_handle)

The data reading callback function. The iterator will be able to give subset of batch in the data.

If there is data, the function will call set_function to set the data.

Param data_handle
The handle to the callback.

Param set_function
The batch returned by the iterator

Param set_function_handle
The handle to be passed to set function.

Return
0 if we are reaching the end and batch is not returned.

typedef int **XGDMatrixCallbackNext**(*DataIterHandle* iter)

Callback function prototype for getting next batch of data.

Param iter
A handler to the user defined iterator.

Return
0 when success, -1 when failure happens.

typedef void **DataIterResetCallback**(*DataIterHandle* handle)

Callback function prototype for resetting the external iterator.

Functions

int **XGDMatrixCreateFromDataIter**(*DataIterHandle* data_handle, *XGBCallbackDataIterNext* *callback, const char *cache_info, float missing, *DMatrixHandle* *out)

Create a DMatrix from a data iterator.

Parameters

- **data_handle** – The handle to the data.

- **callback** – The callback to get the data.
- **cache_info** – Additional information about cache file, can be null.
- **missing** – Which value to represent missing value.
- **out** – The created DMatrix

Returns
0 when success, -1 when failure happens.

int **XGProxyDMatrixCreate**(*DMatrixHandle* *out)

Create a DMatrix proxy for setting data, can be freed by *XGDMatrixFree*.

Second set of callback functions, used by constructing Quantile DMatrix or external memory DMatrix using a custom iterator.

The DMatrix proxy is only a temporary reference (wrapper) to the actual user data. For instance, if a dense matrix (like a numpy array) is passed into the proxy DMatrix via the *XGProxyDMatrixSetDataDense* method, then the proxy DMatrix holds only a reference and the input array cannot be freed until the next iteration starts, signaled by a call to the *XGDMatrixCallbackNext* by XGBoost. It's called `ProxyDMatrix` because it reuses the interface of the DMatrix class in XGBoost, but it's just a mid interface for the *XGDMatrixCreateFromCallback* and related constructors to consume various user input types.

```
User inputs -> Proxy DMatrix (wrapper) -> Actual DMatrix
```

Parameters
out – The created Proxy DMatrix.

Returns
0 when success, -1 when failure happens.

int **XGDMatrixCreateFromCallback**(*DataIterHandle* iter, *DMatrixHandle* proxy, *DataIterResetCallback* *reset, *XGDMatrixCallbackNext* *next, char const *config, *DMatrixHandle* *out)

Create an external memory DMatrix with data iterator.

Short note for how to use second set of callback for external memory data support:

- Step 0: Define a data iterator with 2 methods `reset`, and `next`.
- Step 1: Create a DMatrix proxy by *XGProxyDMatrixCreate* and hold the handle.
- Step 2: Pass the iterator handle, proxy handle and 2 methods into *XGDMatrixCreateFromCallback*, along with other parameters encoded as a JSON object.
- Step 3: Call appropriate data setters in `next` functions.

Parameters

- **iter** – A handle to external data iterator.
- **proxy** – A DMatrix proxy handle created by *XGProxyDMatrixCreate*.
- **reset** – Callback function resetting the iterator state.
- **next** – Callback function yielding the next batch of data.
- **config** – JSON encoded parameters for DMatrix construction. Accepted fields are:
  - missing: Which value to represent missing value

- cache_prefix: The path of cache file, caller must initialize all the directories in this path.
- nthread (optional): Number of threads used for initializing DMatrix.

- **out** – **[out]** The created external memory DMatrix

Returns
0 when success, -1 when failure happens

int **XGQuantileDMatrixCreateFromCallback**(*DataIterHandle* iter, *DMatrixHandle* proxy, *DataIterHandle* ref, *DataIterResetCallback* *reset, *XGDMatrixCallbackNext* *next, char const *config, *DMatrixHandle* *out)

Create a Quantile DMatrix with a data iterator.

Short note for how to use the second set of callback for (GPU)Hist tree method:

- Step 0: Define a data iterator with 2 methods `reset`, and `next`.
- Step 1: Create a DMatrix proxy by *XGProxyDMatrixCreate* and hold the handle.
- Step 2: Pass the iterator handle, proxy handle and 2 methods into *XGQuantileDMatrixCreateFromCallback*.
- Step 3: Call appropriate data setters in `next` functions.

See test_iterative_dmatrix.cu or Python interface for examples.

Parameters

- **iter** – A handle to external data iterator.
- **proxy** – A DMatrix proxy handle created by *XGProxyDMatrixCreate*.
- **ref** – Reference DMatrix for providing quantile information.
- **reset** – Callback function resetting the iterator state.
- **next** – Callback function yielding the next batch of data.
- **config** – JSON encoded parameters for DMatrix construction. Accepted fields are:
  - missing: Which value to represent missing value
  - nthread (optional): Number of threads used for initializing DMatrix.
  - max_bin (optional): Maximum number of bins for building histogram. Must be consistent with the corresponding booster training parameter.
  - max_quantile_blocks (optional, deprecated): This parameter no longer has any effect and will be removed in a future release.
- **out** – The created Quantile DMatrix.

Returns
0 when success, -1 when failure happens

int **XGExtMemQuantileDMatrixCreateFromCallback**(*DataIterHandle* iter, *DMatrixHandle* proxy, *DataIterHandle* ref, *DataIterResetCallback* *reset, *XGDMatrixCallbackNext* *next, char const *config, *DMatrixHandle* *out)

Create a Quantile DMatrix backed by external memory.

See *Using XGBoost External Memory Version* for more info.

Since
3.0.0

- cache_host_ratio (optioinal): For GPU-based inputs, XGBoost can split the cache into host and device portitions to reduce the data transfer overhead. This parameter specifies the size of host cache compared to the size of the entire cache: `host / (host + device)`.

Note

This is experimental and subject to change.

Parameters

- **out** – The created Quantile DMatrix.
- **iter** – A handle to external data iterator.
- **proxy** – A DMatrix proxy handle created by *XGProxyDMatrixCreate*.
- **ref** – Reference DMatrix for providing quantile information.
- **reset** – Callback function resetting the iterator state.
- **next** – Callback function yielding the next batch of data.
- **config** – JSON encoded parameters for DMatrix construction. Accepted fields are:
  - missing: Which value to represent missing value
  - cache_prefix: The path of cache file, caller must initialize all the directories in this path.
  - nthread (optional): Number of threads used for initializing DMatrix.
  - max_bin (optional): Maximum number of bins for building histogram. Must be consistent with the corresponding booster training parameter.
  - on_host (optional): Whether the data should be placed on host memory. Used by GPU inputs.
  - min_cache_page_bytes (optional): The minimum number of bytes for each internal GPU page. Set to 0 to disable page concatenation. Automatic configuration if the parameter is not provided or set to None.
  - max_quantile_blocks (optional, deprecated): This parameter no longer has any effect and will be removed in a future release.

Returns
0 when success, -1 when failure happens

int **XGProxyDMatrixSetDataCudaArrayInterface**(*DMatrixHandle* handle, const char *data)

Set data on a DMatrix proxy.

Parameters

- **handle** – A DMatrix proxy created by *XGProxyDMatrixCreate*
- **data** – Null terminated JSON document string representation of CUDA array interface.

Returns
0 when success, -1 when failure happens

int **XGProxyDMatrixSetDataColumnar**(*DMatrixHandle* handle, char const *data)

Set columnar (table) data on a DMatrix proxy.

Parameters

- **handle** – A DMatrix proxy created by *XGProxyDMatrixCreate*
- **data** – See *XGDMatrixCreateFromColumnar* for details.

Returns

0 when success, -1 when failure happens

int **XGProxyDMatrixSetDataCudaColumnar**(*DMatrixHandle* handle, const char *data)

Set CUDA-based columnar (table) data on a DMatrix proxy.

Parameters

- **handle** – A DMatrix proxy created by *XGProxyDMatrixCreate*
- **data** – See *XGDMatrixCreateFromColumnar* for details.

Returns

0 when success, -1 when failure happens

int **XGProxyDMatrixSetDataDense**(*DMatrixHandle* handle, char const *data)

Set data on a DMatrix proxy.

Parameters

- **handle** – A DMatrix proxy created by *XGProxyDMatrixCreate*
- **data** – Null terminated JSON document string representation of array interface.

Returns

0 when success, -1 when failure happens

int **XGProxyDMatrixSetDataCSR**(*DMatrixHandle* handle, char const *indptr, char const *indices, char const *data, bst_ulong ncol)

Set data on a DMatrix proxy.

Parameters

- **handle** – A DMatrix proxy created by *XGProxyDMatrixCreate*
- **indptr** – JSON encoded **array_interface** to row pointer in CSR.
- **indices** – JSON encoded **array_interface** to column indices in CSR.
- **data** – JSON encoded **array_interface** to values in CSR..
- **ncol** – The number of columns of input CSR matrix.

Returns

0 when success, -1 when failure happens

struct **XGBoostBatchCSR**

#include <c_api.h> Mini batch used in XGBoost Data Iteration.

Booster

group `Booster`

The `Booster` class is the gradient-boosted model for XGBoost.

During training, the booster object has many caches for improved performance. In addition to gradient and prediction, it also includes runtime buffers like leaf partitions. These buffers persist with the Booster object until either *XGBoosterReset()* is called or the booster is deleted by the *XGBoosterFree()*.

Functions

int **XGBoosterCreate**(const *DMatrixHandle* dmats[], bst_ulong len, *BoosterHandle* *out)

Create a XGBoost learner (booster)

Parameters

- **dmats** – matrices that are set to be cached by the booster.
- **len** – length of dmats
- **out** – handle to the result booster

Returns

0 when success, -1 when failure happens

int **XGBoosterFree**(*BoosterHandle* handle)

Delete the booster.

Parameters

handle – The handle to be freed.

Returns

0 when success, -1 when failure happens

int **XGBoosterReset**(*BoosterHandle* handle)

Reset the booster object to release data caches used for training.

Since

3.0.0

Returns

0 when success, -1 when failure happens

int **XGBoosterSlice**(*BoosterHandle* handle, int begin_layer, int end_layer, int step, *BoosterHandle* *out)

Slice a model using boosting index. The slice m:n indicates taking all trees that were fit during the boosting rounds m, (m+1), (m+2), …, (n-1).

Parameters

- **handle** – Booster to be sliced.
- **begin_layer** – start of the slice
- **end_layer** – end of the slice; end_layer=0 is equivalent to end_layer=num_boost_round
- **step** – step size of the slice
- **out** – Sliced booster.

Returns

0 when success, -1 when failure happens, -2 when index is out of bound.

int **XGBoosterBoostedRounds**(*BoosterHandle* handle, int *out)

Get number of boosted rounds from gradient booster. When process_type is update, this number might drop due to removed tree.

Parameters

- **handle** – Handle to booster.
- **out** – Pointer to output integer.

Returns
0 when success, -1 when failure happens

int **XGBoosterSetParam**(*BoosterHandle* handle, const char *name, const char *value)

set parameters

Parameters

- **handle** – handle
- **name** – parameter name
- **value** – value of parameter

Returns
0 when success, -1 when failure happens

int **XGBoosterGetNumFeature**(*BoosterHandle* handle, bst_ulong *out)

get number of features

Parameters

- **handle** – Handle to booster.
- **out** – number of features

Returns
0 when success, -1 when failure happens

int **XGBoosterUpdateOneIter**(*BoosterHandle* handle, int iter, *DMatrixHandle* dtrain)

update the model in one round using dtrain

Parameters

- **handle** – handle
- **iter** – current iteration rounds
- **dtrain** – training data

Returns
0 when success, -1 when failure happens

int **XGBoosterBoostOneIter**(*BoosterHandle* handle, *DMatrixHandle* dtrain, float *grad, float *hess, bst_ulong len)

Deprecated:

since 2.1.0

int **XGBoosterTrainOneIter**(*BoosterHandle* handle, *DMatrixHandle* dtrain, int iter, char const *grad, char const *hess)

Update a model with gradient and Hessian. This is used for training with a custom objective function.

Since
2.0.0

Parameters

- **handle** – handle
- **dtrain** – The training data.
- **iter** – The current iteration round. When training continuation is used, the count should restart.
- **grad** – Json encoded __(cuda)_array_interface__ for gradient.
- **hess** – Json encoded __(cuda)_array_interface__ for Hessian.

Returns
0 when success, -1 when failure happens

int **XGBoosterEvalOneIter**(*BoosterHandle* handle, int iter, *DMatrixHandle* dmats[], const char *evnames[], bst_ulong len, const char **out_result)

get evaluation statistics for xgboost

Parameters

- **handle** – handle
- **iter** – current iteration rounds
- **dmats** – pointers to data to be evaluated
- **evnames** – pointers to names of each data
- **len** – length of dmats
- **out_result** – the string containing evaluation statistics

Returns
0 when success, -1 when failure happens

int **XGBoosterDumpModel**(*BoosterHandle* handle, const char *fmap, int with_stats, bst_ulong *out_len, const char ***out_dump_array)

dump model, return array of strings representing model dump

Parameters

- **handle** – handle
- **fmap** – name to fmap can be empty string
- **with_stats** – whether to dump with statistics
- **out_len** – length of output array
- **out_dump_array** – pointer to hold representing dump of each model

Returns
0 when success, -1 when failure happens

int **XGBoosterDumpModelEx**(*BoosterHandle* handle, const char *fmap, int with_stats, const char *format, bst_ulong *out_len, const char ***out_dump_array)

dump model, return array of strings representing model dump

Parameters

- **handle** – handle
- **fmap** – name to fmap can be empty string
- **with_stats** – whether to dump with statistics
- **format** – the format to dump the model in
- **out_len** – length of output array
- **out_dump_array** – pointer to hold representing dump of each model

Returns
0 when success, -1 when failure happens

int **XGBoosterDumpModelWithFeatures**(*BoosterHandle* handle, int fnum, const char **fname, const char **ftype, int with_stats, bst_ulong *out_len, const char ***out_models)

dump model, return array of strings representing model dump

Parameters

- **handle** – handle
- **fnum** – number of features
- **fname** – names of features
- **ftype** – types of features
- **with_stats** – whether to dump with statistics
- **out_len** – length of output array
- **out_models** – pointer to hold representing dump of each model

Returns
0 when success, -1 when failure happens

int **XGBoosterDumpModelExWithFeatures**(*BoosterHandle* handle, int fnum, const char **fname, const char **ftype, int with_stats, const char *format, bst_ulong *out_len, const char ***out_models)

dump model, return array of strings representing model dump

Parameters

- **handle** – handle
- **fnum** – number of features
- **fname** – names of features
- **ftype** – types of features
- **with_stats** – whether to dump with statistics
- **format** – the format to dump the model in
- **out_len** – length of output array
- **out_models** – pointer to hold representing dump of each model

Returns
0 when success, -1 when failure happens

int **XGBoosterGetCategories**(*BoosterHandle* handle, char const *config, *CategoriesHandle* *out)

See *XGDMatrixGetCategories*

Since

3.2.0

Note

Experimental API, subject to change in the future.

int **XGBoosterGetCategoriesExportToArrow**(*BoosterHandle* handle, char const *config, *CategoriesHandle* *out, char const **export_out)

See *XGDMatrixGetCategoriesExportToArrow*

Since

3.2.0

Note

Experimental API, subject to change in the future.

int **XGBoosterGetAttr**(*BoosterHandle* handle, const char *key, const char **out, int *success)

Get string attribute from Booster.

Parameters

- **handle** – handle
- **key** – The key of the attribute.
- **out** – The result attribute, can be NULL if the attribute do not exist.
- **success** – Whether the result is contained in out.

Returns

0 when success, -1 when failure happens

int **XGBoosterSetAttr**(*BoosterHandle* handle, const char *key, const char *value)

Set or delete string attribute.

Parameters

- **handle** – handle
- **key** – The key of the attribute.
- **value** – The value to be saved. If nullptr, the attribute would be deleted.

Returns

0 when success, -1 when failure happens

int **XGBoosterGetAttrNames**(*BoosterHandle* handle, bst_ulong *out_len, const char ***out)

Get the names of all attribute from Booster.

Parameters

- **handle** – handle
- **out_len** – the argument to hold the output length
- **out** – pointer to hold the output attribute stings

Returns
0 when success, -1 when failure happens

int **XGBoosterSetStrFeatureInfo**(*BoosterHandle* handle, const char *field, const char **features, const bst_ulong size)

Set string encoded feature info in Booster, similar to the feature info in DMatrix.

Accepted fields are:

- feature_name
- feature_type

Parameters

- **handle** – An instance of Booster
- **field** – Field name
- **features** – Pointer to array of strings.
- **size** – Size of `features` pointer (number of strings passed in).

Returns
0 when success, -1 when failure happens

int **XGBoosterGetStrFeatureInfo**(*BoosterHandle* handle, const char *field, bst_ulong *len, const char ***out_features)

Get string encoded feature info from Booster, similar to the feature info in DMatrix.

Accepted field names are:

- feature_name
- feature_type

Caller is responsible for copying out the data, before the next call to any API function of XGBoost.

Parameters

- **handle** – An instance of Booster
- **field** – Field name
- **len** – Size of output pointer `features` (number of strings returned).
- **out_features** – Address of a pointer to array of strings. Result is stored in thread local memory.

Returns
0 when success, -1 when failure happens

int **XGBoosterFeatureScore**(*BoosterHandle* handle, const char *config, bst_ulong *out_n_features, char const ***out_features, bst_ulong *out_dim, bst_ulong const **out_shape, float const **out_scores)

Calculate feature scores for tree models. When used on linear model, only the `weight` importance type is defined, and output scores is a row major matrix with shape [n_features, n_classes] for multi-class model. For tree model, out_n_feature is always equal to out_n_scores and has multiple definitions of importance type.

Parameters

- **handle** – An instance of Booster
- **config** – Parameters for computing scores encoded as JSON. Accepted JSON keys are:
  - importance_type: A JSON string with following possible values:
    - 'weight': the number of times a feature is used to split the data across all trees.
    - 'gain': the average gain across all splits the feature is used in.
    - 'cover': the average coverage across all splits the feature is used in.
    - 'total_gain': the total gain across all splits the feature is used in.
    - 'total_cover': the total coverage across all splits the feature is used in.
  - feature_map: An optional JSON string with URI or path to the feature map file.
  - feature_names: An optional JSON array with string names for each feature.
- **out_n_features** – Length of output feature names.
- **out_features** – An array of string as feature names, ordered the same as output scores.
- **out_dim** – Dimension of output feature scores.
- **out_shape** – Shape of output feature scores with length of `out_dim`.
- **out_scores** – An array of floating point as feature scores with shape of `out_shape`.

Returns

0 when success, -1 when failure happens

Prediction

group **Prediction**

These functions are used for running prediction and explanation algorithms.

Functions

int **XGBoosterPredict**(*BoosterHandle* handle, *DMatrixHandle* dmat, int option_mask, unsigned ntree_limit, int training, bst_ulong *out_len, const float **out_result)

make prediction based on dmat (deprecated, use *XGBoosterPredictFromDMatrix* instead)

Deprecated:

See also

XGBoosterPredictFromDMatrix()

Parameters

- **handle** – handle
- **dmat** – data matrix

- **option_mask** – bit-mask of options taken in prediction, possible values 0:normal prediction 1:output margin instead of transformed value 2:output leaf index of trees instead of leaf value, note leaf index is unique per tree 4:output feature contributions to individual predictions
- **ntree_limit** – limit number of trees used for prediction, this is only valid for boosted trees when the parameter is set to 0, we will use all the trees
- **training** – Whether the prediction function is used as part of a training loop. Prediction can be run in 2 scenarios:

  a. Given data matrix X, obtain prediction y_pred from the model.

  b. Obtain the prediction for computing gradients. For example, DART booster performs dropout during training, and the prediction result will be different from the one obtained by normal inference step due to dropped trees. Set training=false for the first scenario. Set training=true for the second scenario. The second scenario applies when you are defining a custom objective function.

- **out_len** – used to store length of returning result
- **out_result** – used to set a pointer to array

Returns

0 when success, -1 when failure happens

int **XGBoosterPredictFromDMatrix**(*BoosterHandle* handle, *DMatrixHandle* dmat, char const *config, bst_ulong const **out_shape, bst_ulong *out_dim, float const **out_result)

Make prediction from DMatrix, replacing *XGBoosterPredict*.

"type": [0, 6]

- 0: normal prediction
- 1: output margin
- 2: predict contribution
- 3: predict approximated contribution
- 4: predict feature interaction
- 5: predict approximated feature interaction
- 6: predict leaf "training": bool Whether the prediction function is used as part of a training loop. **Not used for inplace prediction**.

Prediction can be run in 2 scenarios:

a. Given data matrix X, obtain prediction y_pred from the model.

b. Obtain the prediction for computing gradients. For example, DART booster performs dropout during training, and the prediction result will be different from the one obtained by normal inference step due to dropped trees. Set training=false for the first scenario. Set training=true for the second scenario. The second scenario applies when you are defining a custom objective function. "iteration_begin": int Beginning iteration of prediction. "iteration_end": int End iteration of prediction. Set to 0 this will become the size of tree model (all the trees). "strict_shape": bool Whether should we reshape the output with stricter rules. If set to true, normal/margin/contrib/interaction predict will output consistent shape disregarding the use of multi-class model, and leaf prediction will output 4-dim array representing: (n_samples, n_iterations, n_classes, n_trees_in_forest)

Example JSON input for running a normal prediction with strict output shape, 2 dim for softprob , 1 dim for others.

```
{
   "type": 0,
   "training": false,
   "iteration_begin": 0,
   "iteration_end": 0,
   "strict_shape": true
}
```

See also

XGBoosterPredictFromDense *XGBoosterPredictFromCSR* *XGBoosterPredictFromCudaArray* *XGBoosterPredictFromCudaColumnar*

Parameters

- **handle** – Booster handle
- **dmat** – DMatrix handle
- **config** – String encoded predict configuration in JSON format, with following available fields in the JSON object:
- **out_shape** – Shape of output prediction (copy before use).
- **out_dim** – Dimension of output prediction.
- **out_result** – Buffer storing prediction value (copy before use).

Returns

0 when success, -1 when failure happens

int **XGBoosterPredictFromDense**(*BoosterHandle* handle, char const *values, char const *config, *DMatrixHandle* m, bst_ulong const **out_shape, bst_ulong *out_dim, const float **out_result)

Inplace prediction from CPU dense matrix.

Note

If the booster is configured to run on a CUDA device, XGBoost falls back to run prediction with DMatrix with a performance warning.

Parameters

- **handle** – Booster handle.
- **values** – JSON encoded **array_interface** to values.
- **config** – See *XGBoosterPredictFromDMatrix* for more info. Additional fields for inplace prediction are:
  - "missing": float
- **m** – An optional (NULL if not available) proxy DMatrix instance storing meta info.

- **out_shape** – See *XGBoosterPredictFromDMatrix* for more info.
- **out_dim** – See *XGBoosterPredictFromDMatrix* for more info.
- **out_result** – See *XGBoosterPredictFromDMatrix* for more info.

Returns
0 when success, -1 when failure happens

int **XGBoosterPredictFromColumnar**(*BoosterHandle* handle, char const *values, char const *config, *DMatrixHandle* m, bst_ulong const **out_shape, bst_ulong *out_dim, const float **out_result)

Inplace prediction from CPU columnar data. (Table)

Note

If the booster is configured to run on a CUDA device, XGBoost falls back to run prediction with DMatrix with a performance warning.

Parameters

- **handle** – Booster handle.
- **data** – See *XGDMatrixCreateFromColumnar* for more info.
- **config** – See *XGBoosterPredictFromDMatrix* for more info. Additional fields for inplace prediction are:
  - "missing": float
- **m** – An optional (NULL if not available) proxy DMatrix instance storing meta info.
- **out_shape** – See *XGBoosterPredictFromDMatrix* for more info.
- **out_dim** – See *XGBoosterPredictFromDMatrix* for more info.
- **out_result** – See *XGBoosterPredictFromDMatrix* for more info.

Returns
0 when success, -1 when failure happens

int **XGBoosterPredictFromCSR**(*BoosterHandle* handle, char const *indptr, char const *indices, char const *values, bst_ulong ncol, char const *config, *DMatrixHandle* m, bst_ulong const **out_shape, bst_ulong *out_dim, const float **out_result)

Inplace prediction from CPU CSR matrix.

Note

If the booster is configured to run on a CUDA device, XGBoost falls back to run prediction with DMatrix with a performance warning.

Parameters

- **handle** – Booster handle.
- **indptr** – JSON encoded **array_interface** to row pointer in CSR.
- **indices** – JSON encoded **array_interface** to column indices in CSR.

- **values** – JSON encoded **array_interface** to values in CSR..
- **ncol** – Number of features in data.
- **config** – See *XGBoosterPredictFromDMatrix* for more info. Additional fields for inplace prediction are:
  - "missing": float
- **m** – An optional (NULL if not available) proxy DMatrix instance storing meta info.
- **out_shape** – See *XGBoosterPredictFromDMatrix* for more info.
- **out_dim** – See *XGBoosterPredictFromDMatrix* for more info.
- **out_result** – See *XGBoosterPredictFromDMatrix* for more info.

Returns
0 when success, -1 when failure happens

int **XGBoosterPredictFromCudaArray**(*BoosterHandle* handle, char const *values, char const *config, *DMatrixHandle* proxy, bst_ulong const **out_shape, bst_ulong *out_dim, const float **out_result)

Inplace prediction from CUDA Dense matrix (cupy in Python).

Note

If the booster is configured to run on a CPU, XGBoost falls back to run prediction with DMatrix with a performance warning.

Parameters

- **handle** – Booster handle
- **values** – JSON encoded **cuda_array_interface** to values.
- **config** – See *XGBoosterPredictFromDMatrix* for more info. Additional fields for inplace prediction are:
  - "missing": float
- **proxy** – An optional (NULL if not available) proxy DMatrix instance storing meta info.
- **out_shape** – See *XGBoosterPredictFromDMatrix* for more info.
- **out_dim** – See *XGBoosterPredictFromDMatrix* for more info.
- **out_result** – See *XGBoosterPredictFromDMatrix* for more info.

Returns
0 when success, -1 when failure happens

int **XGBoosterPredictFromCudaColumnar**(*BoosterHandle* handle, char const *data, char const *config, *DMatrixHandle* proxy, bst_ulong const **out_shape, bst_ulong *out_dim, const float **out_result)

Inplace prediction from CUDA dense dataframe (cuDF in Python).

> **Note**
>
> If the booster is configured to run on a CPU, XGBoost falls back to run prediction with DMatrix with a performance warning.

Parameters

- **handle** – Booster handle
- **data** – See *XGDMatrixCreateFromColumnar* for more info.
- **config** – See *XGBoosterPredictFromDMatrix* for more info. Additional fields for inplace prediction are:
  - ”missing”: float
- **proxy** – An optional (NULL if not available) proxy DMatrix instance storing meta info.
- **out_shape** – See *XGBoosterPredictFromDMatrix* for more info.
- **out_dim** – See *XGBoosterPredictFromDMatrix* for more info.
- **out_result** – See *XGBoosterPredictFromDMatrix* for more info.

Returns

0 when success, -1 when failure happens

Serialization

group **Serialization**

There are multiple ways to serialize a Booster object depending on the use case.

Short note for serialization APIs. There are 3 different sets of serialization API.

- Functions with the term “Model” handles saving/loading XGBoost model like trees or linear weights. Striping out parameters configuration like training algorithms or CUDA device ID. These functions are designed to let users reuse the trained model for different tasks, examples are prediction, training continuation or model interpretation.
- Functions with the term “Config” handles save/loading configuration. It helps user to study the internal of XGBoost. Also user can use the load method for specifying parameters in a structured way. These functions were introduced in 1.0.0.
- Functions with the term “Serialization” are combination of above two. They are used in situations like check-pointing, or continuing training task in a distributed environment. In these cases the task must be carried out without any user intervention.

Functions

int **XGBoosterLoadModel**(*BoosterHandle* handle, const char *fname)

Load the model from an existing file.

Parameters

- **handle** – handle
- **fname** – File name. The string must be UTF-8 encoded.

Returns
0 when success, -1 when failure happens

int **XGBoosterSaveModel**(*BoosterHandle* handle, const char *fname)

Save the model into an existing file.

Parameters

- **handle** – handle
- **fname** – File name. The string must be UTF-8 encoded.

Returns
0 when success, -1 when failure happens

int **XGBoosterLoadModelFromBuffer**(*BoosterHandle* handle, const void *buf, bst_ulong len)

load model from in memory buffer

Parameters

- **handle** – handle
- **buf** – pointer to the buffer
- **len** – the length of the buffer

Returns
0 when success, -1 when failure happens

int **XGBoosterSaveModelToBuffer**(*BoosterHandle* handle, char const *config, bst_ulong *out_len, char const **out_dptr)

Save model into raw bytes, return header of the array. User must copy the result out, before next xgboost call.

Parameters

- **handle** – handle
- **config** – JSON encoded string storing parameters for the function. Following keys are expected in the JSON document:
  - ”format”: str
    - json: Output booster will be encoded as JSON.
    - ubj: Output booster will be encoded as Universal binary JSON. this format except for compatibility reasons.
- **out_len** – The argument to hold the output length
- **out_dptr** – The argument to hold the output data pointer

Returns
0 when success, -1 when failure happens

int **XGBoosterSerializeToBuffer**(*BoosterHandle* handle, bst_ulong *out_len, const char **out_dptr)

Memory snapshot based serialization method. Saves everything states into buffer.

Parameters

- **handle** – handle
- **out_len** – the argument to hold the output length
- **out_dptr** – the argument to hold the output data pointer

Returns
0 when success, -1 when failure happens

int **XGBoosterUnserializeFromBuffer**(*BoosterHandle* handle, const void *buf, bst_ulong len)

Memory snapshot based serialization method. Loads the buffer returned from *XGBoosterSerializeToBuffer*.

Parameters

- **handle** – handle
- **buf** – pointer to the buffer
- **len** – the length of the buffer

Returns
0 when success, -1 when failure happens

int **XGBoosterSaveJsonConfig**(*BoosterHandle* handle, bst_ulong *out_len, char const **out_str)

Save XGBoost's internal configuration into a JSON document. Currently the support is experimental, function signature may change in the future without notice.

Parameters

- **handle** – handle to Booster object.
- **out_len** – length of output string
- **out_str** – A valid pointer to array of characters. The characters array is allocated and managed by XGBoost, while pointer to that array needs to be managed by caller.

Returns
0 when success, -1 when failure happens

int **XGBoosterLoadJsonConfig**(*BoosterHandle* handle, char const *config)

Load XGBoost's internal configuration from a JSON document. Currently the support is experimental, function signature may change in the future without notice.

Parameters

- **handle** – handle to Booster object.
- **config** – string representation of a JSON document.

Returns
0 when success, -1 when failure happens

Collective

group **Collective**

Experimental support for exposing internal communicator in XGBoost.

The collective communicator in XGBoost evolved from the `rabit` project of dmlc but has changed significantly since its adoption. It consists of a tracker and a set of workers. The tracker is responsible for bootstrapping the communication group and handling centralized tasks like logging. The workers are actual communicators performing collective tasks like allreduce.

To use the collective implementation, one needs to first create a tracker with corresponding parameters, then get the arguments for workers using *XGTrackerWorkerArgs()*. The obtained arguments can then be passed to the *XGCommunicatorInit()* function. Call to *XGCommunicatorInit()* must be accompanied with a *XGCommunicatorFinalize()* call for cleanups. Please note that the communicator uses `std::thread` in C++, which has

undefined behavior in a C++ destructor due to the runtime shutdown sequence. It's preferable to call *XGCommunicatorFinalize()* before the runtime is shutting down. This requirement is similar to a Python thread or socket, which should not be relied upon in a `__del__` function.

Since it's used as a part of XGBoost, errors will be returned when a XGBoost function is called, for instance, training a booster might return a connection error.

Note

This is still under development.

Typedefs

typedef void ***`TrackerHandle`**

Handle to the tracker.

There are currently two types of tracker in XGBoost, first one is `rabit`, while the other one is `federated`. `rabit` is used for normal collective communication, while `federated` is used for federated learning.

Functions

int **`XGTrackerCreate`**(char const *config, *TrackerHandle* *handle)

Create a new tracker.

- dmlc_communicator: String, the type of tracker to create. Available options are `rabit` and `federated`. See *TrackerHandle* for more info.
- n_workers: Integer, the number of workers.
- port: (Optional) Integer, the port this tracker should listen to.
- timeout: (Optional) Integer, timeout in seconds for various networking operations. Default is 300 seconds.

Some configurations are `rabit` specific:

- host: (Optional) String, Used by the the `rabit` tracker to specify the address of the host. This can be useful when the communicator cannot reliably obtain the host address.
- sortby: (Optional) Integer.
  - 0: Sort workers by their host name.
  - 1: Sort workers by task IDs.

Some `federated` specific configurations:

- federated_secure: Boolean, whether this is a secure server. False for testing.
- server_key_path: Path to the server key. Used only if this is a secure server.
- server_cert_path: Path to the server certificate. Used only if this is a secure server.
- client_cert_path: Path to the client certificate. Used only if this is a secure server.

Parameters

- **`config`** – JSON encoded parameters.

- **handle** – The handle to the created tracker.

Returns
0 when success, -1 when failure happens

int **XGTrackerWorkerArgs**(*TrackerHandle* handle, char const **args)

Get the arguments needed for running workers. This should be called after *XGTrackerRun()*.

Parameters

- **handle** – The handle to the tracker.
- **args** – The arguments returned as a JSON document.

Returns
0 when success, -1 when failure happens

int **XGTrackerRun**(*TrackerHandle* handle, char const *config)

Start the tracker. The tracker runs in the background and this function returns once the tracker is started.

Parameters

- **handle** – The handle to the tracker.
- **config** – Unused at the moment, preserved for the future.

Returns
0 when success, -1 when failure happens

int **XGTrackerWaitFor**(*TrackerHandle* handle, char const *config)

Wait for the tracker to finish, should be called after *XGTrackerRun()*. This function will block until the tracker task is finished or timeout is reached.

Parameters

- **handle** – The handle to the tracker.
- **config** – JSON encoded configuration. No argument is required yet, preserved for the future.

Returns
0 when success, -1 when failure happens

int **XGTrackerFree**(*TrackerHandle* handle)

Free a tracker instance. This should be called after *XGTrackerWaitFor()*. If the tracker is not properly waited, this function will shutdown all connections with the tracker, potentially leading to undefined behavior.

Parameters
handle – The handle to the tracker.

Returns
0 when success, -1 when failure happens

int **XGCommunicatorInit**(char const *config)

Initialize the collective communicator.

Currently the communicator API is experimental, function signatures may change in the future without notice.

Call this once in the worker process before using anything. Please make sure *XGCommunicatorFinalize()* is called after use. The initialized commuicator is a global thread-local variable.

Only applicable to the `rabit` communicator:

- dmlc_tracker_uri: Hostname or IP address of the tracker.
- dmlc_tracker_port: Port number of the tracker.
- dmlc_task_id: ID of the current task, can be used to obtain deterministic rank assignment.
- dmlc_retry: The number of retries for connection failure.
- dmlc_timeout: Timeout in seconds.
- dmlc_nccl_path: Path to the nccl shared library `libnccl.so`.

Only applicable to the `federated` communicator (use upper case for environment variables, use lower case for runtime configuration):

- federated_server_address: Address of the federated server.
- federated_world_size: Number of federated workers.
- federated_rank: Rank of the current worker.
- federated_server_cert_path: Server certificate file path. Only needed for the SSL mode.
- federated_client_key_path: Client key file path. Only needed for the SSL mode.
- federated_client_cert_path: Client certificate file path. Only needed for the SSL mode.

Parameters

`config` – JSON encoded configuration. Accepted JSON keys are:

- dmlc_communicator: The type of the communicator, this should match the tracker type.
  - rabit: Use Rabit. This is the default if the type is unspecified.
  - federated: Use the gRPC interface for Federated Learning.

Returns

0 when success, -1 when failure happens

int **`XGCommunicatorFinalize`**(void)

Finalize the collective communicator.

Call this function after you have finished all jobs.

Returns

0 when success, -1 when failure happens

int **`XGCommunicatorGetRank`**(void)

Get rank of the current process.

Returns

Rank of the worker.

int **`XGCommunicatorGetWorldSize`**(void)

Get the total number of processes.

Returns

Total world size.

int **XGCommunicatorIsDistributed**(void)

Get if the communicator is distributed.

Returns
True if the communicator is distributed.

int **XGCommunicatorPrint**(char const *message)

Print the message to the tracker.

This function can be used to communicate the information of the progress to the user who monitors the tracker.

Parameters
message – The message to be printed.

Returns
0 when success, -1 when failure happens

int **XGCommunicatorGetProcessorName**(const char **name_str)

Get the name of the processor.

Parameters
name_str – Pointer to received returned processor name.

Returns
0 when success, -1 when failure happens

int **XGCommunicatorBroadcast**(void *send_receive_buffer, size_t size, int root)

Broadcast a memory region to all others from root. This function is NOT thread-safe.

Example:

```
int a = 1;
Broadcast(&a, sizeof(a), root);
```

Parameters

- **send_receive_buffer** – Pointer to the send or receive buffer.
- **size** – Size of the data in bytes.
- **root** – The process rank to broadcast from.

Returns
0 when success, -1 when failure happens

int **XGCommunicatorAllreduce**(void *send_receive_buffer, size_t count, int data_type, int op)

Perform in-place allreduce. This function is NOT thread-safe.

Example Usage: the following code gives sum of the result

```
enum class Op {
    kMax = 0, kMin = 1, kSum = 2, kBitwiseAND = 3, kBitwiseOR = 4, kBitwiseXOR␣
↪= 5
};
std::vector<int> data(10);
...
Allreduce(data.data(), data.size(), DataType:kInt32, Op::kSum);
...
```

Parameters

- `send_receive_buffer` – Buffer for both sending and receiving data.
- `count` – Number of elements to be reduced.
- `data_type` – Enumeration of data type, see xgboost::collective::DataType in communicator.h.
- `op` – Enumeration of operation type, see xgboost::collective::Operation in communicator.h.

Returns

0 when success, -1 when failure happens

1.15 XGBoost C++ API

Starting from 1.0 release, CMake will generate installation rules to export all C++ headers. But the c++ interface is much closer to the internal of XGBoost than other language bindings. As a result it's changing quite often and we don't maintain its stability. Along with the plugin system (see `plugin/example` in XGBoost's source tree), users can utilize some existing c++ headers for gaining more access to the internal of XGBoost.

- C++ interface documentation (latest master branch)
- C++ interface documentation (last stable release)

1.16 Security disclosure

1.16.1 Use of Python pickle

We use `pickle` and `cloudpickle` in several places, including a convenient helper function for the `broadcast` collective operation to share a Python object. The `broadcast` method is not used internally during training but is here to assist with implementing custom metrics. Also, a distributed interface like PySpark might use pickle to transfer Python objects, like the callback functions. Many security scanners will point out the use of pickle as unsafe.

XGBoost as a machine learning library is not designed to protect against pickle data from an untrusted source. Please use appropriate protection mechanisms to ensure that no one can control your network environment and tamper with the pickle data sent between XGBoost workers or the Spark executors. For example, cloud vendors provide managed solutions for running XGBoost in isolated network environments. As for all Python pickles in general, read the warning in the pickle document.

Suggestion:

- Do not load pickle files from an unknown source.
- Use secured network for distributed training.

1.16.2 The lack of authentication in the collective implementation

XGBoost uses TCP sockets for communication between workers during distributed model training. XGBoost is a numeric computation library; the collective module in intended for high-performance numeric operations (allreduce, allgather, etc.). For performance reasons, we decided that the collective module will NOT support TLS authentication or encryption.

Suggestion:

- Use secured network for distributed training.

1.16.3 The lack of sanitizing for inputs, including models

If someone can manipulate XGBoost inputs, whether with an incorrect model or an altered numpy array, XGBoost will crash due to a memory read error (out-of-bounds access). The reports we received describe manipulating the JSON files to mislead XGBoost into reading out-of-bounds values or using conflicting tree indices. We acknowledge that we can add stronger sanitization to the JSON parser when loading from a file. However, it is currently impractical for us to comprehensively validate all potential issues in a supplied model file. Instead, deployments are expected to rely on standard operating-system–level protections. Examples of non-sanitized inputs:

- Manipulated leaf index in a tree model.
- Manipulated length in a UBJSON model.

Suggestions:

- For most users, this should not cause a security issue. Your Python program might crash when loading a manipulated JSON model file.
- Test the model in an isolated environment before loading it in a critical environment.

1.16.4 Security Policy

Supported Versions

Only the latest XGBoost release is supported.

Reporting a Vulnerability

To report a security issue, please email security@xgboost-ci.net with a description of the issue, the steps you took to create the issue, affected versions, and, if known, mitigations for the issue.

All support will be made on the best effort base, so please indicate the "urgency level" of the vulnerability as Critical, High, Medium or Low.

1.17 Contribute to XGBoost

XGBoost has been developed by community members. Everyone is welcome to contribute. We value all forms of contributions, including, but not limited to:

- Code reviews for pull requests
- Documentation and usage examples
- Community participation in forums and issues
- Code readability and developer guide
  - We welcome contributions that add code comments to improve readability.
  - We also welcome contributions to docs to explain the design choices of the XGBoost internals.
- Test cases to make the codebase more robust.
- Tutorials, blog posts, talks that promote the project.

Here are guidelines for contributing to various aspect of the XGBoost project:

1.17.1 XGBoost Community Guideline

XGBoost adopts the Apache style model and governs by merit. We believe that it is important to create an inclusive community where everyone can use, contribute to, and influence the direction of the project. See CONTRIBUTORS.md for the current list of contributors.

General Development Process

Everyone in the community is welcomed to send patches, documents, and propose new directions to the project. The key guideline here is to enable everyone in the community to get involved and participate the decision and development. When major changes are proposed, an RFC should be sent to allow discussion by the community. We encourage public discussion, archivable channels such as issues and discuss forum, so that everyone in the community can participate and review the process later.

Code reviews are one of the key ways to ensure the quality of the code. High-quality code reviews prevent technical debt for long-term and are crucial to the success of the project. A pull request needs to be reviewed before it gets merged. A committer who has the expertise of the corresponding area would moderate the pull request and then merge the code when it is ready. The corresponding committer could request multiple reviewers who are familiar with the area of the code. We encourage contributors to request code reviews themselves and help review each other's code – remember everyone is volunteering their time to the community, high-quality code review itself costs as much as the actual code contribution, you could get your code quickly reviewed if you do others the same favor.

The community should strive to reach a consensus on technical decisions through discussion. We expect committers and PMCs to moderate technical discussions in a diplomatic way, and provide suggestions with clear technical reasoning when necessary.

Committers

Committers are individuals who are granted the write access to the project. A committer is usually responsible for a certain area or several areas of the code where they oversee the code review process. The area of contribution can take all forms, including code contributions and code reviews, documents, education, and outreach. Committers are essential for a high quality and healthy project. The community actively look for new committers from contributors. Here is a list of useful traits that help the community to recognize potential committers:

- Sustained contribution to the project, demonstrated by discussion over RFCs, code reviews and proposals of new features, and other development activities. Being familiar with, and being able to take ownership on one or several areas of the project.
- Quality of contributions: High-quality, readable code contributions indicated by pull requests that can be merged without a substantial code review. History of creating clean, maintainable code and including good test cases. Informative code reviews to help other contributors that adhere to a good standard.
- Community involvement: active participation in the discussion forum, promote the projects via tutorials, talks and outreach. We encourage committers to collaborate broadly, e.g. do code reviews and discuss designs with community members that they do not interact physically.

The Project Management Committee(PMC) consists of a group of active committers that moderate the discussion, manage the project release, and proposes new committer/PMC members. Potential candidates are usually proposed via an internal discussion among PMCs, followed by a consensus approval, i.e. least 3 +1 votes, and no vetoes. Any veto must be accompanied by reasoning. PMCs should serve the community by upholding the community practices and guidelines in order to make XGBoost a better community for everyone. PMCs should strive to only nominate new candidates outside of their own organization.

The PMC is in charge of the project's continuous integration (CI) and testing infrastructure. Currently, we host our own Jenkins server at https://xgboost-ci.net. The PMC shall appoint committer(s) to manage the CI infrastructure. The PMC may accept 3rd-party donations and sponsorships that would defray the cost of the CI infrastructure. See *Donations*.

Reviewers

Reviewers are individuals who actively contributed to the project and are willing to participate in the code review of new contributions. We identify reviewers from active contributors. The committers should explicitly solicit reviews from reviewers. High-quality code reviews prevent technical debt for the long-term and are crucial to the success of the project. A pull request to the project has to be reviewed by at least one reviewer in order to be merged.

1.17.2 Donations

Motivation

DMLC/XGBoost has grown from a research project incubated in academia to one of the most widely used gradient boosting framework in production environment. On one side, with the growth of volume and variety of data in the production environment, users are putting accordingly growing expectation to XGBoost in terms of more functions, scalability and robustness. On the other side, as an open source project which develops in a fast pace, XGBoost has been receiving contributions from many individuals and organizations around the world. Given the high expectation from the users and the increasing channels of contribution to the project, delivering the high quality software presents a challenge to the project maintainers.

A robust and efficient **continuous integration (CI)** infrastructure is one of the most critical solutions to address the above challenge. A CI service will monitor an open-source repository and run a suite of integration tests for every incoming contribution. This way, the CI ensures that every proposed change in the codebase is compatible with existing functionalities. Furthermore, XGBoost can enable more thorough tests with a powerful CI infrastructure to cover cases which are closer to the production environment.

There are several CI services available free to open source projects, such as Travis CI and AppVeyor. The XGBoost project already utilizes GitHub Actions. However, the XGBoost project has needs that these free services do not adequately address. In particular, the limited usage quota of resources such as CPU and memory leaves XGBoost developers unable to bring "too-intensive" tests. In addition, they do not offer test machines with GPUs for testing XGBoost-GPU code base which has been attracting more and more interest across many organizations. Consequently, the XGBoost project uses a cloud-hosted test farm. We host Amazon Web Services (AWS) to host the test machines, along with GitHub Actions and RunsOn (SaaS app) to organize the CI pipelines.

The cloud-hosted test farm has recurring operating expenses. RunsOn launches worker machines on AWS on demand to run the test suite on incoming contributions. To save cost, the worker machines are terminated when they are no longer needed.

To help defray the hosting cost, the XGBoost project seeks donations from third parties.

Donations and Sponsorships

Donors may choose to make one-time donations or recurring donations on monthly or yearly basis. Donors who commit to the Sponsor tier will have their logo displayed on the front page of the XGBoost project.

Fiscal host: Open Source Collective 501(c)(6)

The Project Management Committee (PMC) of the XGBoost project appointed Open Source Collective as their **fiscal host**. The platform is a 501(c)(6) registered entity and will manage the funds on the behalf of the PMC so that PMC members will not have to manage the funds directly. The platform currently hosts several well-known JavaScript frameworks such as Babel, Vue, and Webpack.

All expenses incurred for hosting CI will be submitted to the fiscal host with receipts. Only the expenses in the following categories will be approved for reimbursement:

- Cloud expenses for the cloud test farm
- Cost of domain https://xgboost-ci.net
- Annual subscription for RunsOn

Administration of cloud CI infrastructure

The PMC shall appoint committer(s) to administer the cloud CI infrastructure on their behalf. The current administrators are as follows:

- Primary administrator: Hyunsu Cho
- Secondary administrator: Jiaming Yuan

The administrators shall make good-faith effort to keep the CI expenses under control. The expenses shall not exceed the available funds. The administrators should post regular updates on CI expenses.

1.17.3 Coding Guideline

Contents

- *C++ Coding Guideline*
- *Python Coding Guideline*
- *R Coding Guideline*
  - *Code Style*
  - *Rmarkdown Vignettes*
  - *R package versioning*
  - *Testing R package with different compilers*
  - *Registering native routines in R*
  - *Generating the Package and Running Tests*
- *Running Formatting Checks Locally*
  - *Pre-commit*
  - *Linter*
  - *Clang-tidy*
- *Guide for handling user input data*

C++ Coding Guideline

- Follow Google style for C++, with two exceptions:
  - Each line of text may contain up to 100 characters.
  - The use of C++ exceptions is allowed.
- Use C++17 features such as smart pointers, braced initializers, lambda functions, and `std::thread`.
- Use Doxygen to document all the interface code.
- We have some comments around symbols imported by headers, some of those are hinted by include-what-you-use. It's not required.
- We use clang-tidy. Its configuration lives in the root directory of the XGBoost source tree.
- We have a series of automatic checks to ensure that all of our codebase complies with the Google style. Before submitting your pull request, you are encouraged to run the style checks on your machine. See *R Coding Guideline*.

Python Coding Guideline

- Follow PEP 8: Style Guide for Python Code. We use Pylint to automatically enforce PEP 8 style across our Python codebase. Before submitting your pull request, you are encouraged to run Pylint on your machine. See *R Coding Guideline*.
- Docstrings should be in NumPy docstring format.

R Coding Guideline

Code Style

- We follow Google's C++ Style guide for C++ code.
  - This is mainly to be consistent with the rest of the project.
  - Another reason is we will be able to check style automatically with a linter.
- When needed, you can disable the linter warning of certain line with `// NOLINT(*)` comments.
- We use roxygen for documenting the R package.

Rmarkdown Vignettes

Rmarkdown vignettes are placed in R-package/vignettes. These Rmarkdown files are not compiled. We host the compiled version on doc/R-package.

The following steps are followed to add a new Rmarkdown vignettes:

- Add the original rmarkdown to `R-package/vignettes`.
- Modify `doc/R-package/Makefile` to add the markdown files to be build.
- Clone the dmlc/web-data repo to folder `doc`.
- Now type the following command on `doc/R-package`:

```
make the-markdown-to-make.md
```

- This will generate the markdown, as well as the figures in `doc/web-data/xgboost/knitr`.
- Modify the `doc/R-package/index.md` to point to the generated markdown.
- Add the generated figure to the `dmlc/web-data` repo.
  - If you already cloned the repo to doc, this means `git add`
- Create PR for both the markdown and `dmlc/web-data`.
- You can also build the document locally by typing the following command at the `doc` directory:

```
make html
```

The reason we do this is to avoid exploded repo size due to generated images.

R package versioning

See *XGBoost Release Policy*.

Testing R package with different compilers

You can change the default compiler of R by changing the configuration file in home directory. For instance, if you want to test XGBoost built with clang++ instead of g++ on Linux, put the following in your `~/.R/Makevars` file:

```
CC=clang-15
CXX17=clang++-15
```

Be aware that the variable name should match with the name used by `R CMD`:

```
R CMD config CXX17
```

Registering native routines in R

According to R extension manual, it is good practice to register native routines and to disable symbol search. When any changes or additions are made to the C++ interface of the R package, please make corresponding changes in `src/init.c` as well.

Generating the Package and Running Tests

The source layout of XGBoost is a bit unusual to normal R packages as XGBoost is primarily written in C++ with multiple language bindings in mind. As a result, some special cares need to be taken to generate a standard R tarball. Most of the tests are being run on CI, and as a result, the best way to see how things work is by looking at the CI configuration files (GitHub action, at the time of writing). There are helper scripts in `ops/script` and `R-package/tests/helper_scripts` for running various checks including linter and making the standard tarball.

Running Formatting Checks Locally

Once you submit a pull request to dmlc/xgboost, we perform two automatic checks to enforce coding style conventions. To expedite the code review process, you are encouraged to run the checks locally on your machine prior to submitting your pull request.

Pre-commit

We provide a pre-commit configuration for basic formatting, linting, and file-sanity checks. By default, pre-commit runs on files that are staged for commit, and the hooks in this repository are configured accordingly. To run on modified or untracked files, you can use `pre-commit run --files <path> [...]` or `pre-commit run --all-files`.

To enable it locally:

```
python -m pip install pre-commit
pre-commit install
```

To run it on the files you have staged for commit:

```
pre-commit run
```

To run it on a specific range of commits (e.g. in CI or for a local comparison):

```
pre-commit run --from-ref <base> --to-ref <head>
```

Linter

We use a combination of linters to enforce style convention and find potential errors. Linting is especially useful for scripting languages like Python, as we can catch many errors that would have otherwise occurred at run-time.

For Python scripts, pylint, black and isort are used for providing guidance on coding style, and mypy is required for type checking. The Python formatting and pylint checks are provided via the corresponding pre-commit hooks, which operate on changed files. For C++, cpplint is used along with `clang-tidy`. For R, `lintr` is used.

To run Python checks locally, install the checkers mentioned previously and run the pre-commit hooks for the files you changed:

```
cd /path/to/xgboost/
pre-commit run
```

To run checks for R:

```
cd /path/to/xgboost/
R CMD INSTALL R-package/
Rscript ops/script/lint_r.R $(pwd)
```

To run checks for cpplint locally:

```
cd /path/to/xgboost/
python ./ops/script/lint_cpp.py
```

Lastly, the linter for jvm-packages is integrated into the maven build process.

Clang-tidy

Clang-tidy is an advance linter for C++ code, made by the LLVM team. We use it to conform our C++ codebase to modern C++ practices and conventions.

To run this check locally, use the clang-CUDA helper from the top level source tree:

```
cd /path/to/xgboost/
bash ops/pipeline/run-clang-tidy-clang-cuda.sh
```

This helper configures a clang-generated CUDA compilation database and then runs `run-clang-tidy -p` against it. The same path is used by CI.

By default it lints files under `src/`. Use environment variables to narrow the scope or adjust the checks:

```
cd /path/to/xgboost/
XGBOOST_TIDY_FILES='src/common/timer.cc,src/predictor/interpretability/shap.cu' \
XGBOOST_TIDY_CHECKS='-*,google-runtime-int' \
bash ops/pipeline/run-clang-tidy-clang-cuda.sh
```

The helper accepts command-line overrides for the build directory, job count, source filter, checks, extra arguments, and `warnings-as-errors` filter. For example:

```
cd /path/to/xgboost/
bash ops/pipeline/run-clang-tidy-clang-cuda.sh \
  --build-dir build-clang-tidy-cuda \
  --jobs 16 \
  --source-filter '.*/(src|include)/.*' \
  --checks '-*,google-runtime-int'
```

The helper expects a clang toolchain with `clang++`, `clang-linker-wrapper`, and `run-clang-tidy` available either from the active conda base environment or from `XGBOOST_CLANG_PREFIX`. When using conda, install both `clangxx` and `clang-tools` so the compiler resource directory and `run-clang-tidy` are both present.

Guide for handling user input data

This is an in-comprehensive guide for handling user input data. XGBoost has wide verity of native supported data structures, mostly come from higher level language bindings. The inputs ranges from basic contiguous 1 dimension memory buffer to more sophisticated data structures like columnar data with validity mask. Raw input data can be used in 2 places, firstly it's the construction of various `DMatrix`, secondly it's the in-place prediction. For plain memory buffer, there's not much to discuss since it's just a pointer with a size. But for general n-dimension array and columnar data, there are many subtleties. XGBoost has 3 different data structures for handling optionally masked arrays (tensors), for consuming user inputs `ArrayInterface` should be chosen. There are many existing functions that accept only plain pointer due to legacy reasons (XGBoost started as a much simpler library and didn't care about memory usage that much back then). The `ArrayInterface` is a in memory representation of `__array_interface__` protocol defined by numpy or the `__cuda_array_interface__` defined by numba. Following is a check list of things to have in mind when accepting related user inputs:

- [ ] Is it strided? (identified by the `strides` field)
- [ ] If it's a vector, is it row vector or column vector? (Identified by both `shape` and `strides`).
- [ ] Is the data type supported? Half type and 128 integer types should be converted before going into XGBoost.
- [ ] Does it have higher than 1 dimension? (identified by `shape` field)
- [ ] Are some of dimensions trivial? (shape[dim] <= 1)
- [ ] Does it have mask? (identified by `mask` field)
- [ ] Can the mask be broadcasted? (unsupported at the moment)
- [ ] Is it on CUDA memory? (identified by `data` field, and optionally `stream`)

Most of the checks are handled by the `ArrayInterface` during construction, except for the data type issue since it doesn't know how to cast such pointers with C builtin types. But for safety reason one should still try to write related tests for the all items. The data type issue should be taken care of in language binding for each of the specific data input. For single-chunk columnar format, it's just a masked array for each column so it should be treated uniformly as normal array. For input predictor `X`, we have adapters for each type of input. Some are composition of the others. For instance, CSR matrix has 3 potentially strided arrays for `indptr`, `indices` and `values`. No assumption should be made to these components (all the check boxes should be considered). Slicing row of CSR matrix should calculate the offset of each field based on respective strides.

For meta info like labels, which is growing both in size and complexity, we accept only masked array at the moment (no specialized adapter). One should be careful about the input data shape. For base margin it can be 2 dim or higher if we have multiple targets in the future. The getters in `DMatrix` returns only 1 dimension flatten vectors at the moment, which can be improved in the future when it's needed.

1.17.4 Consistency for Language Bindings

XGBoost has many different language bindings developed over the years, some are in the main repository while others live independently. Many features and interfaces are inconsistent with each others, this document aims to provide some guidelines and actionable items for language binding designers.

Model Serialization

XGBoost C API exposes a couple functions for serializing a model for persistence storage. These saved files are backward compatible, meaning one can load an older XGBoost model with a newer XGBoost version. If there's change in the model format, we have deprecation notice inside the C++ implementation and public issue for tracking the status. See *Introduction to Model IO* for details.

As a result, these are considered to be stable and should work across language bindings. For instance, a model trained in R should be fully functioning in C or Python. Please don't pad anything to the output file or buffer.

If there are extra fields that must be saved:

- First review whether the attribute can be retrieved from known properties of the model. For instance, there's a `classes_` attribute in the scikit-learn interface `XGBClassifier`, which can be obtained through *numpy.arange(n_classes)* and doesn't need to be saved into the model. Preserving version compatibility is not a trivial task and we are still spending a significant amount of time to maintain it. Please don't make complication if it's not necessary.
- Then please consider whether it's universal. For instance, we have added *feature_types* to the model serialization for categorical features (which is a new feature after 1.6), the attribute is useful or will be useful in the future regardless of the language binding.
- If the field is small, we can save it as model attribute (which is a key-value structure). These attributes are ignored by all other language bindings and mostly an ad-hoc storage.
- Lastly, we should use the UBJSON as the default output format when given a chance (not to be burdened by the old binary format).

Training Continuation

There are cases where we want to train a model based on the previous model, for boosting trees, it's either adding new trees or modifying the existing trees. This can be normal model update, error recovery, or other special cases we don't know of yet. When it happens, the training iteration should start from 0, not from the last boosted rounds of the model. 0 is a special iteration number, we perform some extra checks like whether the label is valid during that iteration. These checks can be expensive but necessary for eliminating silent errors. Keeping the iteration starts from zero allows us to perform these checks only once for each input data.

Inference

The inference function is quite inconsistent among language bindings at the time of writing due to historical reasons, but this makes more important for us to have consistency in mind in the future development.

- Firstly, it's the output shape. There's a relatively new parameter called `strict_shape` in XGBoost and is rarely used. We want to make it as the default behavior but couldn't due to compatibility concerns. See *Prediction* for details. In short, if specified, XGBoost C++ implementation can output prediction with the correct shape, instead of letting the language binding to handle it.
- Policy around early stopping is at the moment inconsistent between various interfaces. Some considers the `best_iteration` attribute while others don't. We should formalize that all interfaces in the future should use the `best_iteration` during inference unless user has explicitly specified the `iteration_range` parameter.

Parameter naming

There are many parameter naming conventions out there, Some XGBoost interfaces try to align with the larger communities. For example, the R package might support parameters naming like `max.depth=3`, while the Spark package might support `MaxDepth=3`. These are fine, it's better for the users to keep their pipeline consistent. However, while supporting naming variants, the normal, XGBoost way of naming should also be supported, meaning `max_depth=3` should be a valid parameter no-matter what language one is using. If someone were to write duplicated parameter `max.depth=3, max_depth=3`, a clear error should be preferred instead of prioritizing one over the other.

Default Parameters

Like many other machine learning libraries, all parameters from XGBoost can either be inferred from the data or have default values. Bindings should not make copies of these default values and let the XGBoost core decide. When the parameter key is not passed into the C++ core, XGBoost will pick the default accordingly. These defaults are not necessarily optimal, but they are there for consistency. If there's a new choice of default parameter, we can change

it inside the core and it will be automatically propagated to all bindings. Given the same set of parameters and data, various bindings should strive to produce the same model. One exception is the *num_boost_rounds*, which exists only in high-level bindings and has various alias like `n_estimators`. Its default value is close to arbitrary at the moment, we haven't been able to get a good default yet.

Logging

XGBoost has a default logger builtin that can be a wrapper over binding-specific logging facility. For instance, the Python binding registers a callback to use Python `warnings` and `print()` function to output logging. We want to keep logging native to the larger communities instead of using the `std::cerr` from C++.

Minimum Amount of Data Manipulation

XGBoost is mostly a machine learning library providing boosting algorithm implementation. Some other implementations might perform some sort of data manipulation implicitly like deciding the coding of the data, and transforming the data according to some heuristic before training. We prefer to keep these operations based on necessities instead of convenience to keep the scope of the project well-defined. Whenever possible, we should leave these features to 3-party libraries and consider how a user can compose their pipeline. For instance, XGBoost itself should not perform ordinal encoding for categorical data, users will pick an encoder that fits their use cases (like out-of-core implementation, distributed implementation, known mapping, etc). If some transformations are decided to be part of the algorithm, we can have it inside the core instead of the language binding. Examples would be target-encoding or sketching the response variables. If we were to support them, we could have it inside the core implementation as part of the ML algorithm. This aligns with the same principles of default parameters, various bindings should provide similar (if not the same) results given the same set of parameters and data.

Feature Info

XGBoost accepts data structures that contain meta info about predictors, including the names and types of features. Example inputs are `pandas.DataFrame`, R *data.frame*. We have the following heuristics: - When the input data structure contains such information, we set the *feature_names* and *feature_types* for *DMatrix* accordingly. - When a user provides this information as explicit parameters, the user-provided version should override the one provided by the data structure. - When both sources are missing, the *DMatrix* class contain empty info.

1.17.5 Notes on packaging XGBoost's Python package

Contents

- *How to build binary wheels and source distributions*
  - *Building sdists*
  - *Building binary wheels*

How to build binary wheels and source distributions

Wheels and source distributions (sdist for short) are the two main mechanisms for packaging and distributing Python packages.

- A **source distribution** (sdist) is a tarball (`.tar.gz` extension) that contains the source code.
- A **wheel** is a ZIP-compressed archive (with `.whl` extension) representing a *built* distribution. Unlike an sdist, a wheel can contain compiled components. The compiled components are compiled prior to distribution, making it more convenient for end-users to install a wheel. Wheels containing compiled components are referred to as **binary wheels**.

See Python Packaging User Guide to learn more about how Python packages in general are packaged and distributed.

For the remainder of this document, we will focus on packaging and distributing XGBoost.

Building sdists

In the case of XGBoost, an sdist contains both the Python code as well as the C++ code, so that the core part of XGBoost can be compiled into the shared library `libxgboost.so`[1].

You can obtain an sdist as follows:

```
$ python -m build --sdist .
```

(You'll need to install the `build` package first: `pip install build` or `conda install python-build`.)

Running `pip install` with an sdist will launch CMake and a C++ compiler to compile the bundled C++ code into `libxgboost.so`:

```
$ pip install -v xgboost-2.0.0.tar.gz  # Add -v to show build progress
```

Building binary wheels

You can also build a wheel as follows:

```
$ pip wheel --no-deps -v .
```

Notably, the resulting wheel contains a copy of the shared library `libxgboost.so`[1]. The wheel is a **binary wheel**, since it contains a compiled binary.

Running `pip install` with the binary wheel will extract the content of the wheel into the current Python environment. Since the wheel already contains a pre-built copy of `libxgboost.so`, it does not have to be built at the time of install. So `pip install` with the binary wheel completes quickly:

```
$ pip install xgboost-2.0.0-py3-none-linux_x86_64.whl  # Completes quickly
```

1.17.6 Adding and running tests

A high-quality suite of tests is crucial in ensuring correctness and robustness of the codebase. Here, we provide instructions how to run unit tests, and also how to add a new one.

Contents

- *Adding a new unit test*
  - *Python package: pytest*
  - *C++: Google Test*
  - *JVM packages: JUnit / scalatest*
  - *R package: testthat*
- *Running Unit Tests Locally*
  - *R package*

[1] The name of the shared library file will differ depending on the operating system in use. See *Building the Shared Library*.

  - *JVM packages*
  - *Python package: pytest*
  - *C++: Google Test*
- *Sanitizers: Detect memory errors and data races*
  - *How to build XGBoost with sanitizers*
  - *How to use sanitizers with CUDA support*
  - *Other sanitizer runtime options*

Adding a new unit test

Python package: pytest

Add your test under the directories

- tests/python/
- tests/python-gpu/ (if you are testing GPU code)
- tests/test_distributed. (if a distributed framework is used)

Refer to the PyTest tutorial to learn how to write tests for Python code.

You may try running your test by following instructions in *this section*.

C++: Google Test

Add your test under the directory tests/cpp/. Refer to this excellent tutorial on using Google Test.

You may try running your test by following instructions in *this section*. Note. Google Test version 1.8.1 or later is required.

JVM packages: JUnit / scalatest

The JVM packages for XGBoost (XGBoost4J / XGBoost4J-Spark) use the Maven Standard Directory Layout. Specifically, the tests for the JVM packages are located in the following locations:

- jvm-packages/xgboost4j/src/test/
- jvm-packages/xgboost4j-spark/src/test/

To write a test for Java code, see JUnit 5 tutorial. To write a test for Scala, see Scalatest tutorial.

You may try running your test by following instructions in *this section*.

R package: testthat

Add your test under the directory R-package/tests/testthat. Refer to this excellent tutorial on testthat.

You may try running your test by following instructions in *this section*.

Running Unit Tests Locally

R package

Run

```
python ./ops/script/test_r_package.py --task=check
```

at the root of the project directory. The command builds and checks the XGBoost r-package. Alternatively, if you want to just run the tests, you can use the following commands after installing XGBoost:

```
cd R-package/tests/
Rscript testthat.R
```

JVM packages

Maven is used

```
mvn test
```

Python package: pytest

To run Python unit tests, first install pytest package:

```
pip3 install pytest
```

Then compile XGBoost according to instructions in *Building the Shared Library*. Finally, invoke pytest at the project root directory:

```
# Tell Python where to find XGBoost module
export PYTHONPATH=./python-package
pytest -v -s --fulltrace tests/python
```

In addition, to test CUDA code, run:

```
# Tell Python where to find XGBoost module
export PYTHONPATH=./python-package
pytest -v -s --fulltrace tests/python-gpu
```

(For this step, you should have compiled XGBoost with CUDA enabled.)

For testing with distributed frameworks like `Dask` and `PySpark`:

```
# Tell Python where to find XGBoost module
export PYTHONPATH=./python-package
pytest -v -s --fulltrace tests/test_distributed
```

C++: Google Test

To build and run C++ unit tests enable tests while running CMake:

```
cmake -B build -S . -GNinja -DGOOGLE_TEST=ON -DUSE_DMLC_GTEST=ON -DUSE_CUDA=ON -DUSE_
↪NCCL=ON
cmake --build build
```

(continues on next page)

(continued from previous page)

```
cd ./build
./testxgboost
```

Flags like USE_CUDA, USE_DMLC_GTEST are optional. For more info about how to build XGBoost from source, see *Building From Source*. One can also run all unit tests using ctest tool which provides higher flexibility. For example:

```
ctest --verbose
```

If you need to debug errors on Windows using the debugger from VS, you can append the gtest flags in *test_main.cc*:

```
::testing::GTEST_FLAG(filter) = "Suite.Test";
::testing::GTEST_FLAG(repeat) = 10;
```

Sanitizers: Detect memory errors and data races

By default, sanitizers are bundled in GCC and Clang/LLVM. One can enable sanitizers with GCC >= 4.8 or LLVM >= 3.1, But some distributions might package sanitizers separately. Here is a list of supported sanitizers with corresponding library names:

- Address sanitizer: libasan
- Undefined sanitizer: libubsan
- Leak sanitizer: liblsan
- Thread sanitizer: libtsan

Memory sanitizer is exclusive to LLVM, hence not supported in XGBoost. With latest compilers like gcc-9, when sanitizer flags are specified, the compiler driver should be able to link the runtime libraries automatically.

How to build XGBoost with sanitizers

One can build XGBoost with sanitizer support by specifying -DUSE_SANITIZER=ON. By default, address sanitizer and leak sanitizer are used when you turn the USE_SANITIZER flag on. You can always change the default by providing a semicolon separated list of sanitizers to ENABLED_SANITIZERS. Note that thread sanitizer is not compatible with the other two sanitizers.

```
cmake -DUSE_SANITIZER=ON -DENABLED_SANITIZERS="address;undefined" /path/to/xgboost
```

By default, CMake will search regular system paths for sanitizers, you can also supply a specified SANITIZER_PATH.

```
cmake -DUSE_SANITIZER=ON -DENABLED_SANITIZERS="address;undefined" \
-DSANITIZER_PATH=/path/to/sanitizers /path/to/xgboost
```

How to use sanitizers with CUDA support

Running XGBoost on CUDA with address sanitizer (asan) will raise memory error. To use asan with CUDA correctly, you need to configure asan via ASAN_OPTIONS environment variable:

```
ASAN_OPTIONS=protect_shadow_gap=0 ${BUILD_DIR}/testxgboost
```

Other sanitizer runtime options

By default undefined sanitizer doesn't print out the backtrace. You can enable it by exporting environment variable:

```
UBSAN_OPTIONS=print_stacktrace=1 ${BUILD_DIR}/testxgboost
```

For details, please consult official documentation for sanitizers.

1.17.7 Documentation and Examples

Contents

- *Documentation*
  - *Build the Python Docs using pip and Conda*
- *Read The Docs*
- *Examples*
- *Doc Tests*

Documentation

- Python and C documentation is built using Sphinx.
- Each document is written in reStructuredText.
- The documentation is the `doc/` directory.
- You can build it locally using `make html` command.

  ```
  make html
  ```

  Run `make help` to learn about the other commands.

The online document is hosted by Read the Docs where the imported project is managed by Hyunsu Cho and Jiaming Yuan.

Build the Python Docs using pip and Conda

1. Create a conda environment.

   ```
   conda create -n xgboost-docs --yes python=3.12
   ```

   Note

   Python is required to match XGBoost's minimum supported Python version.

2. Activate the environment

   ```
   conda activate xgboost-docs
   ```

3. Install required packages (in the current environment) using `pip` command.

```
pip install -r requirements.txt
```

Note

It is currently not possible to install the required packages using conda due to xgboost_ray being unavailable in conda channels.

```
conda install --file requirements.txt --yes -c conda-forge
```

4. (optional) Install graphviz

```
conda install graphviz --yes
```

5. Eventually, build the docs.

```
make html
```

You should see the following messages in the console:

```
$ make html
sphinx-build -b html -d _build/doctrees   . _build/html
Running Sphinx v6.2.1
...
The HTML pages are in _build/html.

Build finished. The HTML pages are in _build/html.
```

Read The Docs

Read the Docs (RTD for short) is an online document hosting service and hosts the XGBoost document site. The document builder used by RTD is relatively lightweight. However some of the packages like the R binding require a compiled XGBoost along with all the optional dependencies to render the document. As a result, both jvm-based packages and the R package's document is built with an independent CI pipeline and fetched during online document build.

The sphinx configuration file `xgboost/doc/conf.py` acts as the fetcher. During build, the fetched artifacts are stored in `xgboost/doc/tmp/jvm_docs` and `xgboost/doc/tmp/r_docs` respectively. For the R package, there's a dummy index file in `xgboost/doc/R-package/r_docs` . Jvm doc is similar. As for the C doc, it's generated using doxygen and processed by breathe during build as it's relatively cheap. The generated xml files are stored in `xgboost/doc/tmp/dev` .

The `xgboost/doc/tmp` is part of the `html_extra_path` sphinx configuration specified in the `conf.py` file, which informs sphinx to copy the extracted html files to the build directory. Following is a list of environment variables used by the fetchers in `conf.py`:

- `READTHEDOCS`: Read the docs flag. Build the full documentation site including R, JVM and C doc when set to `True` (case sensitive).
- `XGBOOST_R_DOCS`: Local path for pre-built R document, used for development. If it points to a file that doesn't exist, the configuration script will download the packaged document to that path for future reuse.
- `XGBOOST_JVM_DOCS`: Local path for pre-built JVM document, used for development. Similar to the R docs environment variable when it points to a non-existent file.

As of writing, RTD doesn't provide any facility to be embedded as a GitHub action but we need a way to specify the dependency between the CI pipelines and the document build in order to fetch the correct artifact. The workaround is to use an extra GA step to notify RTD using its REST API.

Examples

- Use cases and examples are in demo directory.
- We are super excited to hear about your story. If you have blog posts, tutorials, or code solutions using XGBoost, please tell us, and we will add a link in the example pages.

Doc Tests

We use Sphinx doctest to test selected snippets in the documentation. At the moment, this only covers Python and R snippets written with `.. code-tab:: python` or `.. code-tab:: r`. Regular code blocks are rendered as examples but are not executed by the doctest job.

The doctest job runs snippets from each `.rst` file as an independent group. Snippets in the same document share state, while snippets from different documents do not. To skip a tabbed snippet, add the `no-doctest` class:

```
.. code-tab:: python
   :class: no-doctest

   # This snippet is rendered but not tested.
```

1.17.8 XGBoost Internal Feature Map

The following is a reference to the features supported by XGBoost. It is not a beginner's guide, but rather a list meant to help those looking to add new features to XGBoost understand what needs to be covered.

Core Features

Core features are not dependent on language binding and any language binding can choose to support them.

Data Storage

The primary data structure in XGBoost for storing user inputs is `DMatrix`; it's a container for all data that XGBoost can use. `QuantileDMatrix` is a variant specifically designed for the `hist` tree method. Both can take GPU-based inputs. They take an optional parameter `missing` to specify which input value should be ignored. For external memory support, please refer to *Using XGBoost External Memory Version*.

Single Node Training

There are two different model types in XGBoost: the tree model, which we primarily focus on, and the linear model. For the tree model, we have various methods to build decision trees; please see the *Tree Methods* for a complete reference. In addition to the tree method, we have many hyper-parameters for tuning the model and injecting prior knowledge into the training process. Two noteworthy examples are *monotonic constraints* and *feature interaction constraints*. These two constraints require special treatment during tree construction. Both the `hist` and the `approx` tree methods support GPU acceleration. Also, XGBoost GPU supports gradient-based sampling, which supports external-memory data as well.

The objective function plays an important role in training. It not only provides the gradient, but also responsible for estimating a good starting point for Newton optimization. Please note that users can define custom objective functions for the task at hand. In addition to numerical features, XGBoost also supports categorical features with two different algorithms, including one-hot encoding and optimal partitioning. For more information, refer to the *categorical feature tutorial*. The `hist` and the `approx` tree methods support categorical features for CPU and GPU.

There's working-in-progress support for vector leaves, which are decision tree leaves that contain multiple values. This type of tree is used to support efficient multi-class and multi-target models.

Inference

By inference, we specifically mean getting model prediction for the response variable. XGBoost supports two inference methods. The first one is the prediction on the `DMatrix` object (or `QuantileDMatrix`, which is a subclass). Using a `DMatrix` object allows XGBoost to cache the prediction, hence getting faster performance when running prediction on the same data with new trees. The second method is `inplace_predict`, which bypasses the construction of `DMatrix`. It's more efficient but doesn't support cached prediction. In addtion to returning the estimated response, we also support returning the leaf index, which can be used to analyse the model and as a feature to another model.

Model IO

We have a set of methods for different model serialization methods, including complete serialization, saving to a file, and saving to a buffer. For more, refer to the *Introduction to Model IO*.

Model Explanation

XGBoost includes features designed to improve understanding of the model. Here's a list:

- Global feature importance.
- SHAP value, including contribution and intervention.
- Tree dump.
- Tree visualization.
- Tree as dataframe.

SHAP values use XGBoost's in-tree `QuadratureTreeSHAP` implementation on both CPU and GPU. It supports categorical features, while vector-leaf is still in progress.

Evaluation

XGBoost has built-in support for a wide range of metrics, from basic regression to learning to rank and survival modeling. They can handle distributed training and GPU-based acceleration. Custom metrics are supported as well, please see *Custom Objective and Evaluation Metric*.

Distributed Training

XGBoost has built-in support for three distributed frameworks, including `Dask`, `PySpark`, and `Spark (Scala)`. In addition, there's `flink` support for the Java binding and the `ray-xgboost` project. Please see the respective tutorial on how to use them. By default, XGBoost uses sample-based parallelism for distributed training. The column-based split is still working in progress and needs to be supported in these high-level framework integrations. On top of distributed training, we are also working on federated learning for both sample-based and column-based splits.

Distributed training works with custom objective functions and metrics as well. XGBoost aggregates the evaluation result automatically during training.

The distributed training is enabled by a built-in implementation of a collective library. It's based on the RABIT project and has evolved significantly since its early adoption. The collective implementation supports GPU via NCCL, and has variants for handling federated learning and federated learning on GPU.

Inference normally doesn't require any special treatment since we are using sample-based split. However, with column-based data split, we need to initialize the communicator context as well.

Language Bindings

We have a list of bindings for various languages. Inside the XGBoost repository, there's Python, R, Java, Scala, and C. All language bindings are built on top of the C version. Some others, like Julia and Rust, have their own repository. For guidance on adding a new binding, please see *Consistency for Language Bindings*.

1.17.9 Git Workflow Howtos

Contents

- *How to resolve conflict with master*
- *How to combine multiple commits into one*
- *What is the consequence of force push*

How to resolve conflict with master

- First rebase to most recent master

```
# The first two steps can be skipped after you do it once.
git remote add upstream https://github.com/dmlc/xgboost
git fetch upstream
git rebase upstream/master
```

- The git may show some conflicts it cannot merge, say `conflicted.py`.
  - Manually modify the file to resolve the conflict.
  - After you resolved the conflict, mark it as resolved by

```
git add conflicted.py
```

- Then you can continue rebase by

```
git rebase --continue
```

- Finally push to your fork, you may need to force push here.

```
git push --force
```

How to combine multiple commits into one

Sometimes we want to combine multiple commits, especially when later commits are only fixes to previous ones, to create a PR with set of meaningful commits. You can do it by following steps.

- Before doing so, configure the default editor of git if you haven't done so before.

```
git config core.editor the-editor-you-like
```

- Assume we want to merge last 3 commits, type the following commands

```
git rebase -i HEAD~3
```

- It will pop up an text editor. Set the first commit as `pick`, and change later ones to `squash`.

- After you saved the file, it will pop up another text editor to ask you modify the combined commit message.
- Push the changes to your fork, you need to force push.

```
git push --force
```

What is the consequence of force push

The previous two tips requires force push, this is because we altered the path of the commits. It is fine to force push to your own fork, as long as the commits changed are only yours.

1.17.10 XGBoost Release Policy

Versioning Policy

Starting from XGBoost 1.0.0, each XGBoost release will be versioned as [MAJOR].[FEATURE].[MAINTENANCE]

- MAJOR: We guarantee the API compatibility across releases with the same major version number. We expect to have a 1+ years development period for a new MAJOR release version.
- FEATURE: We ship new features, improvements and bug fixes through feature releases. The cycle length of a feature is decided by the size of feature roadmap. The roadmap is decided right after the previous release.
- MAINTENANCE: Maintenance version only contains bug fixes. This type of release only occurs when we found significant correctness and/or performance bugs and barrier for users to upgrade to a new version of XGBoost smoothly.

Making a Release

1. Create an issue for the release, noting the estimated date and expected features or major fixes, pin that issue.
2. Create a release branch if this is a major release. Bump release version. There's a helper script `ops/script/change_version.py`.
3. Commit the change, create a PR on GitHub on release branch. Port the bumped version to default branch, optionally with the postfix `SNAPSHOT`.
4. Create a tag on release branch, either on GitHub or locally.
5. Make a release on GitHub tag page, which might be done with previous step if the tag is created on GitHub.
6. Submit pip, R-universe, CRAN, and Maven packages.

   There are helper scripts for automating the process in `xgboost/dev/`.

   - The pip package is maintained by Hyunsu Cho and Jiaming Yuan.
   - The CRAN package and the R-universe packages are maintained by Jiaming Yuan.
   - The Maven package is maintained by Nan Zhu and Hyunsu Cho.

R Universe Packages

Since XGBoost 3.0.0, we host the R package on R-Universe. To make a new release, change the `packages.json` in dmlc.r-universe.dev with a new release branch.

R CRAN Package

Before submitting a release, one should test the package on R-hub and win-builder first. Please note that the R-hub Windows instance doesn't have the exact same environment as the one hosted on win-builder.

According to the CRAN policy:

> If running a package uses multiple threads/cores it must never use more than two simultaneously: the check farm is a shared resource and will typically be running many checks simultaneously.

We need to check the number of CPUs used in examples. Export `_R_CHECK_EXAMPLE_TIMING_CPU_TO_ELAPSED_THRESHOLD_=2.5` before running `R CMD check --as-cran` *[1]* and make sure the machine you are using has enough CPU cores to reveal any potential policy violation.

Read The Docs

We might need to manually activate the new release branch for read the docs and set it as the default branch in the console *[2]*. Please check the document build and make sure the correct branch is activated and selected after making a new release.

References

[1] https://stat.ethz.ch/pipermail/r-package-devel/2022q4/008610.html

[2] https://github.com/readthedocs/readthedocs.org/issues/12073

1.17.11 Automated testing in XGBoost project

This document collects tips for using the Continuous Integration (CI) service of the XGBoost project.

Contents

- *Tips for testing*
  - *R tests*
    - *Running R tests with `noLD` option*
    - *Using container images from r-hub*
  - *Making changes to CI containers*
  - *Reproducing CI testing environments locally*
    - *To build a Docker container*
    - *To run commands within a Docker container*
    - *Examples: useful tasks for local development*
- *Tour of the CI infrastructure*
  - *GitHub Actions*
  - *Self-Hosted Runners with RunsOn*
  - *The Lay of the Land: how CI pipelines are organized in the codebase*
  - *Artifact sharing between jobs via Amazon S3*
  - *Custom actions for GitHub Actions*
  - *Infra for building and publishing CI containers and VM images*
    - *Notes on Docker containers*
    - *Notes on VM images*

Tips for testing

R tests

Running R tests with `noLD` option

You can run R tests using a custom-built R with compilation flag `--disable-long-double`. See this page for more details about noLD. This is a requirement for keeping XGBoost on CRAN (the R package index). Unlike other tests, this test must be invoked manually. Simply add a review comment `/gha run r-nold-test` to a pull request to kick off the test. (Ordinary comment won't work. It needs to be a review comment.)

Using container images from r-hub

The r-hub project provides a list of container images for reproducing CRAN environments.

Making changes to CI containers

Many of the CI pipelines use Docker containers to ensure consistent testing environment with a variety of software packages. We have a separate repo, dmlc/xgboost-devops, to host the logic for building and publishing CI containers.

To make changes to the CI container, carry out the following steps:

1. Identify which container needs updating. Example: `492475357299.dkr.ecr.us-west-2.amazonaws.com/xgb-ci.gpu:main`
2. Clone dmlc/xgboost-devops and make changes to the corresponding Dockerfile. Example: `containers/dockerfile/Dockerfile.gpu`.
3. Locally build the container, to ensure that the container successfully builds. Consult *Reproducing CI testing environments locally* for this step.
4. Submit a pull request to dmlc/xgboost-devops with the proposed changes to the Dockerfile. Make note of the pull request number. Example: `#204`
5. Clone dmlc/xgboost. Locate the file `ops/pipeline/get-image-tag.sh`, which should have a single line

   ```
   IMAGE_TAG=main
   ```

   To use the new container, revise the file as follows:

   ```
   IMAGE_TAG=PR-XX
   ```

   where `XX` is the pull request number. E.g. `PR-204`.
6. Now submit a pull request to dmlc/xgboost. The CI will run tests using the new container. Verify that all tests pass.
7. Merge the pull request in `dmlc/xgboost-devops`. Wait until the CI completes on the `main` branch.
8. Go back to the the pull request for `dmlc/xgboost` and change `ops/pipeline/get-image-tag.sh` back to `IMAGE_TAG=main`.
9. Merge the pull request in `dmlc/xgboost`.

Reproducing CI testing environments locally

You can reproduce the same testing environment as the CI pipelines by building and running Docker containers locally.

Prerequisites

1. Install Docker: https://docs.docker.com/engine/install/ubuntu/

2. Install NVIDIA Docker runtime: https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/latest/install-guide.html. The runtime lets you access NVIDIA GPUs inside a Docker container.

To build a Docker container

Clone the repository dmlc/xgboost-devops and invoke `containers/docker_build.sh` as follows:

```
# The following env vars are only relevant for CI
# For local testing, set them to "main"
export GITHUB_SHA="main"
export BRANCH_NAME="main"
bash containers/docker_build.sh IMAGE_REPO
```

where `IMAGE_REPO` is the name of the container image. The wrapper script will look up the YAML file `containers/ci_container.yml`. For example, when `IMAGE_REPO` is set to `xgb-ci.gpu`, the script will use the corresponding entry from `containers/ci_container.yml`:

```
xgb-ci.gpu:
  container_def: gpu
  build_args:
    CUDA_VERSION_ARG: "12.4.1"
    NCCL_VERSION_ARG: "2.23.4-1"
    RAPIDS_VERSION_ARG: "24.10"
```

The `container_def` entry indicates where the Dockerfile is located. The container definition will be fetched from `containers/dockerfile/Dockerfile.CONTAINER_DEF` where `CONTAINER_DEF` is the value of `container_def` entry. In this example, the Dockerfile is `containers/dockerfile/Dockerfile.gpu`.

The `build_args` entry lists all the build arguments for the Docker build. In this example, the build arguments are:

```
--build-arg CUDA_VERSION_ARG=12.4.1 --build-arg NCCL_VERSION_ARG=2.23.4-1 \
  --build-arg RAPIDS_VERSION_ARG=24.10
```

The build arguments provide inputs to the `ARG` instructions in the Dockerfile.

When `containers/docker_build.sh` completes, you will have access to the container with the (fully qualified) URI `492475357299.dkr.ecr.us-west-2.amazonaws.com/[image_repo]:main`. The prefix `492475357299.dkr.ecr.us-west-2.amazonaws.com/` was added so that the container could later be uploaded to AWS Elastic Container Registry (ECR), a private Docker registry.

To run commands within a Docker container

Invoke `ops/docker_run.py` from the main `dmlc/xgboost` repo as follows:

```
python3 ops/docker_run.py \
  --image-uri 492475357299.dkr.ecr.us-west-2.amazonaws.com/[image_repo]:[image_tag] \
  [--use-gpus] \
  -- "command to run inside the container"
```

where `--use-gpus` should be specified to expose NVIDIA GPUs to the Docker container.

For example:

```
# Run without GPU
python3 ops/docker_run.py \
  --image-uri 492475357299.dkr.ecr.us-west-2.amazonaws.com/xgb-ci.cpu:main \
```

(continues on next page)

(continued from previous page)

```
  -- bash ops/pipeline/build-cpu-impl.sh cpu

# Run with NVIDIA GPU
python3 ops/docker_run.py \
  --image-uri 492475357299.dkr.ecr.us-west-2.amazonaws.com/xgb-ci.gpu:main \
  --use-gpus \
  -- bash ops/pipeline/test-python-wheel.sh gpu
```

Optionally, you can specify `--run-args` to pass extra arguments to `docker run`:

```
# Allocate extra space in /dev/shm to enable NCCL
# Also run the container with elevated privileges
python3 ops/docker_run.py \
  --image-uri 492475357299.dkr.ecr.us-west-2.amazonaws.com/xgb-ci.gpu:main \
  --use-gpus \
  --run-args='--shm-size=4g --privileged' \
  -- bash ops/pipeline/test-python-wheel.sh gpu
```

See *Infra for building and publishing CI containers and VM images* to read about how containers are built and managed in the CI pipelines.

Examples: useful tasks for local development

- Build XGBoost with GPU support + package it as a Python wheel

```
export DOCKER_REGISTRY=492475357299.dkr.ecr.us-west-2.amazonaws.com
python3 ops/docker_run.py \
  --image-uri ${DOCKER_REGISTRY}/xgb-ci.gpu_build_rockylinux8:main \
  -- ops/pipeline/build-cuda-impl.sh
```

- Build XGBoost with GPU support on Linux ARM64

```
export DOCKER_REGISTRY=492475357299.dkr.ecr.us-west-2.amazonaws.com
python3 ops/docker_run.py \
  --image-uri ${DOCKER_REGISTRY}/xgb-ci.gpu_build_rockylinux8_aarch64:main \
  -- ops/pipeline/build-cuda-impl.sh
```

- Run Python tests

```
export DOCKER_REGISTRY=492475357299.dkr.ecr.us-west-2.amazonaws.com
python3 ops/docker_run.py \
  --image-uri ${DOCKER_REGISTRY}/xgb-ci.cpu:main \
  -- ops/pipeline/test-python-wheel.sh cpu
```

- Run Python tests with GPU algorithm

```
export DOCKER_REGISTRY=492475357299.dkr.ecr.us-west-2.amazonaws.com
python3 ops/docker_run.py \
  --image-uri ${DOCKER_REGISTRY}/xgb-ci.gpu:main \
  --use-gpus \
  -- ops/pipeline/test-python-wheel.sh gpu
```

- Run Python tests with GPU algorithm on Linux ARM64

```
export DOCKER_REGISTRY=492475357299.dkr.ecr.us-west-2.amazonaws.com
python3 ops/docker_run.py \
  --image-uri ${DOCKER_REGISTRY}/xgb-ci.gpu_aarch64:main \
  --use-gpus \
  -- ops/pipeline/test-python-wheel.sh gpu-arm64
```

- Run Python tests with GPU algorithm, with multiple GPUs

```
export DOCKER_REGISTRY=492475357299.dkr.ecr.us-west-2.amazonaws.com
python3 ops/docker_run.py \
  --image-uri ${DOCKER_REGISTRY}/xgb-ci.gpu:main \
  --use-gpus \
  --run-args='--shm-size=4g' \
  -- ops/pipeline/test-python-wheel.sh mgpu
  # --shm-size=4g is needed for multi-GPU algorithms to function
```

- Build and test JVM packages

```
export DOCKER_REGISTRY=492475357299.dkr.ecr.us-west-2.amazonaws.com
export SCALA_VERSION=2.12  # Specify Scala version (2.12 or 2.13)
python3 ops/docker_run.py \
  --image-uri ${DOCKER_REGISTRY}/xgb-ci.jvm:main \
  --run-args "-e SCALA_VERSION" \
  -- ops/pipeline/build-test-jvm-packages-impl.sh
```

- Build and test JVM packages, with GPU support

```
export DOCKER_REGISTRY=492475357299.dkr.ecr.us-west-2.amazonaws.com
export SCALA_VERSION=2.12  # Specify Scala version (2.12 or 2.13)
export USE_CUDA=1
python3 ops/docker_run.py \
  --image-uri ${DOCKER_REGISTRY}/xgb-ci.jvm_gpu_build:main \
  --use-gpus \
  --run-args "-e SCALA_VERSION -e USE_CUDA --shm-size=4g" \
  -- ops/pipeline/build-test-jvm-packages-impl.sh
  # --shm-size=4g is needed for multi-GPU algorithms to function
```

Tour of the CI infrastructure

GitHub Actions

We make the extensive use of GitHub Actions to host our CI pipelines. Most of the tests listed in the configuration files run automatically for every incoming pull requests and every update to branches.

Self-Hosted Runners with RunsOn

RunsOn is a SaaS (Software as a Service) app that lets us to easily create self-hosted runners to use with GitHub Actions pipelines. RunsOn uses Amazon Web Services (AWS) under the hood to provision runners with access to various amount of CPUs, memory, and NVIDIA GPUs. Thanks to this app, we are able to test GPU-accelerated and distributed algorithms of XGBoost while using the familar interface of GitHub Actions.

In GitHub Actions, jobs run on Microsoft-hosted runners by default. To opt into self-hosted runners (enabled by RunsOn), we use the following special syntax:

```
runs-on:
  - runs-on
  - runner=runner-name
  - run-id=${{ github.run_id }}
  - tag=[unique tag that uniquely identifies the job in the GH Action workflow]
```

where the runner is defined in `.github/runs-on.yml`. For CUDA-enabled ARM64 builds and tests we rely on the `linux-arm64-gpu` runner, which provisions a Graviton + NVIDIA GPU instance.

The Lay of the Land: how CI pipelines are organized in the codebase

The XGBoost project stores the configuration for its CI pipelines as part of the codebase. The git repository therefore stores not only the change history for its source code but also the change history for the CI pipelines.

The CI pipelines are organized into the following directories and files:

- `.github/workflows/`: Definition of CI pipelines, using the GitHub Actions syntax
- `.github/runs-on.yml`: Configuration for the RunsOn service. Specifies the spec for the self-hosted CI runners.
- `ops/conda_env/`: Definitions for Conda environments
- `ops/patch/`: Patch files
- `ops/pipeline/`: Shell scripts defining CI/CD pipelines. Most of these scripts can be run locally (to assist with development and debugging); a few must run in the CI.
- `ops/script/`: Various utility scripts useful for testing
- `ops/docker_run.py`: Wrapper script to run commands inside a container

To inspect a given CI pipeline, inspect files in the following order:

Many of the CI pipelines use Docker containers to ensure consistent testing environment with a variety of software packages. We have a separate repo, dmlc/xgboost-devops, that hosts the code for building the CI containers. The repository is organized as follows:

- `actions/`: Custom actions to be used with GitHub Actions. See *Custom actions for GitHub Actions* for more details.
- `containers/dockerfile/`: Dockerfiles to define containers
- `containers/ci_container.yml`: Defines the mapping between Dockerfiles and containers. Also specifies the build arguments to be used with each container.
- `containers/docker_build.{py,sh}`: Wrapper scripts to build and test CI containers.
- `vm_images/`: Defines bootstrap scripts to build VM images for Amazon EC2. See *Notes on VM images* to learn about how VM images relate to container images.

See *Reproducing CI testing environments locally* to learn about the utility scripts for building and using containers.

Artifact sharing between jobs via Amazon S3

We make artifacts from one workflow job available to another job, by uploading the artifacts to Amazon S3. In the CI, we utilize the script `ops/pipeline/manage-artifacts.py` to coordinate artifact sharing.

To upload files to S3: In the workflow YAML, add the following lines:

```
- name: Upload files to S3
  run: |
    REMOTE_PREFIX="remote directory to place the artifact(s)"
    python3 ops/pipeline/manage-artifacts.py upload \
      --s3-bucket ${{ env.RUNS_ON_S3_BUCKET_CACHE }} \
      --prefix cache/${{ github.run_id }}/${REMOTE_PREFIX} \
      path/to/file
```

The `--prefix` argument specifies the remote directory in which the artifact(s) should be placed. The artifact(s) will be placed in `s3://{RUNS_ON_S3_BUCKET_CACHE}/cache/{GITHUB_RUN_ID}/{REMOTE_PREFIX}/` where `RUNS_ON_S3_BUCKET_CACHE` and `GITHUB_RUN_ID` are set by the CI.

You can upload multiple files, possibly with wildcard globbing:

```
- name: Upload files to S3
  run: |
    python3 ops/pipeline/manage-artifacts.py upload \
      --s3-bucket ${{ env.RUNS_ON_S3_BUCKET_CACHE }} \
      --prefix cache/${{ github.run_id }}/build-cuda \
      build/testxgboost python-package/dist/*.whl
```

To download files from S3: In the workflow YAML, add the following lines:

```
- name: Download files from S3
  run: |
    REMOTE_PREFIX="remote directory where the artifact(s) were placed"
    python3 ops/pipeline/manage-artifacts.py download \
      --s3-bucket ${{ env.RUNS_ON_S3_BUCKET_CACHE }} \
      --prefix cache/${{ github.run_id }}/${REMOTE_PREFIX} \
      --dest-dir path/to/destination_directory \
      artifacts
```

You can also use the wildcard globbing. The script will locate all artifacts under the given prefix that matches the wildcard pattern.

```
- name: Download files from S3
  run: |
    # Locate all artifacts with name *.whl under prefix
    # cache/${GITHUB_RUN_ID}/${REMOTE_PREFIX} and
    # download them to wheelhouse/.
    python3 ops/pipeline/manage-artifacts.py download \
      --s3-bucket ${{ env.RUNS_ON_S3_BUCKET_CACHE }} \
      --prefix cache/${{ github.run_id }}/${REMOTE_PREFIX} \
      --dest-dir wheelhouse/ \
      *.whl
```

Custom actions for GitHub Actions

XGBoost implements a few custom composite actions to reduce duplicated code within workflow YAML files. The custom actions are hosted in a separate repository, dmlc/xgboost-devops, to make it easy to test changes to the custom actions in a pull request or a fork.

In a workflow file, we'd refer to `dmlc/xgboost-devops/actions/{custom-action}@main`. For example:

```
- uses: dmlc/xgboost-devops/actions/miniforge-setup@main
  with:
    environment-name: cpp_test
    environment-file: ops/conda_env/cpp_test.yml
```

Each custom action consists of two components:

- Main script (`dmlc/xgboost-devops/actions/{custom-action}/action.yml`): dispatches to a specific version of the implementation script (see the next item). The main script clones `xgboost-devops` from a specified fork at a particular ref, allowing us to easily test changes to the custom action.
- Implementation script (`dmlc/xgboost-devops/actions/impls/{custom-action}/action.yml`): Implements the custom script.

This design was inspired by Mike Sarahan's work in rapidsai/shared-actions.

Infra for building and publishing CI containers and VM images

Notes on Docker containers

CI pipeline for containers

The dmlc/xgboost-devops repo hosts a CI pipeline to build new Docker containers at a regular schedule. New containers are built in the following occasions:

- New commits are added to the `main` branch of `dmlc/xgboost-devops`.
- New pull requests are submitted to `dmlc/xgboost-devops`.
- Every week, at a set day and hour.

This setup ensures that the CI containers remain up-to-date.

How wrapper scripts work

The wrapper scripts `docker_build.sh`, `docker_build.py` (in `dmlc/xgboost-devops`) and `docker_run.py` (in `dmlc/xgboost`) are designed to transparently log what commands are being carried out under the hood. For example, when you run `bash containers/docker_build.sh xgb-ci.gpu`, the logs will show the following:

```
# docker_build.sh calls docker_build.py...
python3 containers/docker_build.py --container-def gpu \
  --image-uri 492475357299.dkr.ecr.us-west-2.amazonaws.com/xgb-ci.gpu:main \
  --build-arg CUDA_VERSION_ARG=12.4.1 --build-arg NCCL_VERSION_ARG=2.23.4-1 \
  --build-arg RAPIDS_VERSION_ARG=24.10

...

# .. and docker_build.py in turn calls "docker build"...
docker build --build-arg CUDA_VERSION_ARG=12.4.1 \
  --build-arg NCCL_VERSION_ARG=2.23.4-1 \
  --build-arg RAPIDS_VERSION_ARG=24.10 \
  --load --progress=plain \
  --ulimit nofile=1024000:1024000 \
  -t 492475357299.dkr.ecr.us-west-2.amazonaws.com/xgb-ci.gpu:main \
  -f containers/dockerfile/Dockerfile.gpu \
  containers/
```

The logs come in handy when debugging the container builds.

Here is an example with docker_run.py:

```
# Run without GPU
python3 ops/docker_run.py \
  --image-uri 492475357299.dkr.ecr.us-west-2.amazonaws.com/xgb-ci.cpu:main \
  -- bash ops/pipeline/build-cpu-impl.sh cpu

# Run with NVIDIA GPU
# Allocate extra space in /dev/shm to enable NCCL
# Also run the container with elevated privileges
python3 ops/docker_run.py \
  --image-uri 492475357299.dkr.ecr.us-west-2.amazonaws.com/xgb-ci.gpu:main \
  --use-gpus \
  --run-args='--shm-size=4g --privileged' \
  -- bash ops/pipeline/test-python-wheel.sh gpu
```

which are translated to the following docker run invocations:

```
docker run --rm --pid=host \
  -w /workspace -v /path/to/xgboost:/workspace \
  -e CI_BUILD_UID=<uid> -e CI_BUILD_USER=<user_name> \
  -e CI_BUILD_GID=<gid> -e CI_BUILD_GROUP=<group_name> \
  492475357299.dkr.ecr.us-west-2.amazonaws.com/xgb-ci.cpu:main \
  bash ops/pipeline/build-cpu-impl.sh cpu

docker run --rm --pid=host --gpus all \
  -w /workspace -v /path/to/xgboost:/workspace \
  -e CI_BUILD_UID=<uid> -e CI_BUILD_USER=<user_name> \
  -e CI_BUILD_GID=<gid> -e CI_BUILD_GROUP=<group_name> \
  --shm-size=4g --privileged \
  492475357299.dkr.ecr.us-west-2.amazonaws.com/xgb-ci.gpu:main \
  bash ops/pipeline/test-python-wheel.sh gpu
```

Notes on VM images

In the vm_images/ directory of dmlc/xgboost-devops, we define Packer scripts to build images for Virtual Machines (VM) on Amazon EC2. The VM image contains the minimal set of drivers and system software that are needed to run the containers.

We update container images much more often than VM images. Whereas it takes only 10 minutes to build a new container image, it takes 1-2 hours to build a new VM image.

To enable quick development iteration cycle, we place the most of the development environment in containers and keep VM images small. Packages need for testing should be baked into containers, not VM images. Developers can make changes to containers and see the results of the changes quickly.

Note

Special note for the Windows platform

We do not use containers when testing XGBoost on Windows. All software must be baked into the VM image. Containers are not used because NVIDIA Container Toolkit does not yet support Windows natively.

The dmlc/xgboost-devops repo hosts a CI pipeline to build new VM images at a regular schedule (currently monthly).

1.18 Release Notes

For release notes prior to the 2.1 release, please see news .

1.18.1 3.3.0 (2026 Jun 17)

XGBoost 3.3 adds expectile regression, enables categorical feature support by default, expands SHAP support for vector-leaf models, and includes optimizations for histogram building, quantile sketching, and distributed GPU training.

SHAP Support

- Change the TreeSHAP implementation for improved numerical stability and faster execution with `QuadratureTreeSHAP` (#12179, #12192, #12207)
- Add exact SHAP contribution and interaction prediction for vector-leaf multi-output trees on both CPU and GPU. (#12209, #12210, #12247, #11985, #12208)

Quantile Sketching and Distributed Training

The quantile sketching went through some major refactoring and optimizations. XGBoost 3.3.0 simplifies quantile sketch internals and improves the weighted quantile sketch implementation (#12033, #12046, #12048, #12049, #12054, #12067, #12074, #12146, #12148, #12150, #12151, #12155, #12167), with significantly reduced memory use in the GPU quantile sketch (#12047, #12079, #12090, #12099, #12104, #12105, #12118, #12147, #12159, #12160). Also, we have a more efficient distributed quantile construction using tree reductions. (#12061, #12128, #12171)

Features

- Add expectile regression with the `reg:expectileerror` objective, the `expectile` metric, and the `expectile_alpha` parameter. Multiple expectiles are supported. (#11988, #12228, #12243)
- Enable categorical feature support by default while keeping `enable_categorical` available for users who need to disable it. CPU `hist` also gained one-hot categorical split support for the working-in-progress vector leaf. (#12015, #12072, #12244)
- Deprecate the `gblinear` booster. Support will be removed in a future release. (#12030)
- Use a local RNG and serialize RNG state. Training multiple models with sampling within the same session is now reproducible. (#12043, #12083)

Optimizations

- Optimize CPU histogram building for wide datasets with column block tiling and detect CPU cache sizes via Linux sysfs on aarch64. (#12158, #12233)
- Use Philox for faster GPU sampling. (#12223)
- Support customizing worker port for distributed training. In addition, XGBoost now doesn't need all-to-all collective connections for improved scalability. (#12010, #12075, #12171, #12082)

Python Package

- Bump the minimum supported Python version to 3.12. (#12195)
- Add PySpark support for Spark Connect ML. (#11970)
- Require `Enum` support from Polars and validate unique pandas column names. (#12240, #12199)
- Fix the default verbose behavior mismatch between `XGBClassifier.fit` and `XGBRegressor.fit`. (#12184)
- Fix `python -OO` crashes caused by assigning to missing docstrings (#12094)

- Handle boolean indicator features in `trees_to_dataframe`. (#12089)
- Improve validation and error messages for feature information and deprecated functions. (#12142, #12200)
- Clean up imports, type checking comments, and legacy compatibility guards. (#12027, #12110, #12163)

JVM Packages

- Document Spark 4.0 compatibility for JVM packages. (#12136)
- Support regressor and ranker pipelines with columnar input. (#12058)
- Add automatic module names for Java packages and support xgboost4j on FreeBSD. (#12114, #12222)

Build and Platform

- Support Visual Studio 2026. (#12245)
- Update CUDA Toolkit support, including CUDA Toolkit 13.3 type updates and default architecture alignment for recent CUDA versions. (#12230, #12204)
- Support latest RAPIDS. (#12140, #12013, #12212, #12144)
- Fix CMake export targets and builds with system-installed `dmlc-core`. (#12238, #12123)
- Fix macOS build and packaging issues. (#12187, #12108)
- Improve Linux packaging with versioned shared object. (#12055)

Fixes

- Fix out-of-vocabulary categorical encoding and categorical splits with vector-leaf models. (#12193, #12244)
- Fix SYCL multiclass objective calculation. (#12041)
- Fix in-place prune aliasing and assorted prediction/documentation typos. (#12202, #12076, #12070, #12003)

Documents

- Add TreeSHAP references, distributed XGBoost on Kubernetes documentation, and updates to competition-winning solution examples. (#12207, #12080, #12087, #12109)
- Add notes for pickling, expectile margin output, Spark 4.0 compatibility. (#12042, #12243, #12136, #12247)
- Update LightGBM links, the security disclosure, the `xgboost-cpu` package note, and the Read the Docs canonical URL. (#12084, #12113, #12169, #12246)

CI and Maintenance

- Unify CI configure workflows, document the clang-tidy flow, and keep CI images and dependency pins up to date. (#12170, #12175, #12165)
- Keep GitHub Actions dependency groups and release bookkeeping up to date. (#12005, #12029, #12034, #12038, #12057, #12095, #12124, #12134, #12145, #12161, #12178, #12188, #12206, #12231, #12249)
- Update CI jobs for `cibuildwheel`, CRAN, R CI, and documentation tests. (#12066, #12125, #12173, #12216)
- Continue test cleanup and refactoring across quantile sketching, PySpark, global configuration. (#12002, #12007, #12009, #12019, #12141, #12164, #12180, #12190)
- Continue cleanup of prediction and DART internals, including moving DART state into `GBTree` and unifying DART SHAP forwarding. (#12068, #12071, #12073, #12078, #12081, #12022)
- Always save DART configuration. (#12098)

- Improved support for using clang-tidy with CUDA code. (#12165, #12174)
- Various small cleanups. (#12053, #12092)
- Improve CUDA resource handling and diagnostics, small cleanups for external memory. (#12092, #12127, #12185, #12191, #12137, #12069)

1.18.2 3.2.0 (2026 Feb 09)

We are excited to announce the XGBoost 3.2 release. This release features significant progress on multi-target tree support with vector leaf, enhanced GPU external memory training, various optimizations, and the removal of the deprecated CLI.

External Memory

The latest XGBoost release features enhanced support for external memory training with GPUs. XGBoost has experimental support for using the CUDA async memory pool, which users can opt in to enable asynchronous memory management for efficient external memory training. Prior to 3.2, the RMM plugin was required. The feature is Linux-only at the moment. (#11706, #11715, #11718, #11931, #11865, #11959, #11962)

The adaptive cache is now used for all device types, including devices with full C2C bandwidth, like GH200 and DGX station. Users can continue to specify the `cache_host_ratio` parameter in case of memory fragmentation. XGBoost now supports devices with mixed GPU models for configuring the host cache (#11998). As part of the work for improved NUMA system support, we co-developed the `pyhwloc` project (#11992).

Lastly, the old page-concat option for GPU external memory has been removed. XGBoost will use the full dataset for training. (#11882, #11897)

Multi-Target/Class

This release brings substantial progress on the vector-leaf-based multi-target tree model, building on the multi-target intercept work from 3.1. The vector leaf tree stores a vector of weights in each leaf node, enabling the model to capture correlations across targets during tree construction. In 3.2, we expanded the feature set to cover most of the commonly used training configurations.

> **Warning**
>
> The vector leaf is still a work in progress. Feedback is welcome.

New features for the multi-target tree include:

- Reduced gradient (sketch boost) for the hist tree method, which avoids using the full gradient matrix to find tree structures for improving scalability with the number of targets. Users can use a custom objective to define the tree split gradient in addition to the full leaf gradient. Built-in objectives are not yet supported.
- Support for all regression objectives, including MAE and the quantile loss.
- GPU `hist` tree method implementation has features on par with the CPU one.
- Regularization parameters including L1/L2, `min_split_loss`, and `max_delta_step`.
- Row subsampling with both uniform sampling and gradient-based sampling.
- Column sampling (feature selection), including feature weights.
- Feature importance variants (gain and coverage).
- Model dump support for all formats (JSON, text, graphviz).
- External memory.

In addition, intercept initialization for the multinomial logistic objective now adheres to GLM semantics.

Related PRs: #11950, #11914, #11913, #11965, #11941, #11967, #11940, #11896, #11894, #11889, #11917, #11883, #11786, #11881, #11862, #11855, #11829, #11825, #11820, #11814, #11729, #11724, #11747, #11798, #11791, #11789, #11781, #11778, #11777, #11744, #11922, #11920

Currently missing features for the `hist` tree method with vector leaf:

- Distributed training
- Categorical features
- Feature interaction constraints
- Monotone constraints, which are not defined when the output is a vector.
- Shapley values

Features

- As part of the vector leaf work, CPU `hist` now supports gradient-based sampling.
- The deprecated CLI (command line interface) has been removed. It was deprecated in 2.1. (#11720)
- Expose the categories container to the C API, allowing C users to access category information from the trained model. (#11794)
- Upgrade to CUDA 12.9. (#11972, #11968)
- Support oneapi 2026 release. (#11994)
- Compatibility fixes for the latest versions of nvcomp, RMM, and CCCL. (#11930, #11834, #11871, #11995, #11861, #11785, #11997). A nightly CI pipeline was added to test XGBoost with the latest versions of CCCL and RMM. (#11863)

Optimizations

- Various optimizations for the GPU hist tree method, some of which were done as part of the vector leaf work. (#11895)
- Enable multi-threaded data initialization for CPU. (#11974)
- Make the `block_size` of the CPU histogram building kernel adaptive based on model parameters and CPU cache size, demonstrating up to 2x speedup for certain workloads. (#11808)
- Small optimizations for some GPU kernels to use TMA. (#11841, #11802)
- We now use device memory for storing the tree model, which eliminates data copies between host and device during training and inference. (#11759, #11735, #11750, #11741, #11752)

Fixes

- Fix logistic regression with constant labels. (#11973)
- Fix OpenMP configuration for macOS. (#11976)
- Fix SYCL build. (#11844)

Python Package

- Fix memory leak with Python DataFrame inputs where temporary buffers were stored as class variables instead of instance variables. (#11961)
- Pandas 3.0 support. (#11975)

- Add Python type hints for tests and demos, various type hint fixes. (#11795, #11797)
- Add Python 3.14 classifier. (#11793)
- Maintenance (#11717, #11783)

R Package

- Fix RCHK warnings and memory safety issues. (#11938, #11935, #11847)
- Error out on factors passed to `DMatrix` with an informative message. (#11810)
- Remove calls to R's global RNG that are no longer needed. (#11848, #11887)
- Various documentation fixes and updates. (#11773, #11890, #11732, #11846, #11981, #11842)

JVM Packages

- Remove `synchronized` from predict, as internal prediction is already thread-safe, with a concurrency test added to verify. (#11746)
- Set GPU device ID explicitly at the beginning of training and avoid CUDA API guard for the tracker process, allowing Spark executors to run in exclusive mode. (#11939, #11929)
- Use `inferBatchSizeParameter` instead of a hardcoded value. (#11745)
- Documentation updates, maintenance. (#11691, #11915, #11743)

Documents

- Update references from XGBoost Operator to Kubeflow Trainer. (#11710)
- Document the categories container and add notes for handling unseen categories. (#11788, #11868, #11774)
- Add Intel as sponsor. (#11850)

CI and Maintenance

- Support `pre-commit` for various linting and formatting tasks. `clang-format` is now required by the CI. (#11984, #11978, #11980, #11958, #11953, #11946, #11993)
- We added sccache integration to XGBoost's CI workflows, which brings significant speedup since a majority of the time is spent on compiling variants of XGBoost. In addition, most of the workflows now use GHA container support. (#11956, #11952, #11949, #11937, #11934, #11927, #11932, #11924, #11979)
- Plenty of optimizations for tests. (#11990, #11975, #11964)
- Various dependency updates, fixes, test refactoring, and cleanups. (#11955, #11957, #11963, #11945, #11912, #11909, #11888, #11898, #11925, #11877, #11824, #11748, #11721, #11705, #11699, #11832, #11796, #11828, #11852, #11800, #11999, #11991)

1.18.3 3.1.3 Patch Release (Jan 08 2026)

- Scikit-learn 1.8 compatibility fix (#11858)
- Add ARM CUDA wheels for PyPI. (#11827) Add nccl as dep for aarch64. (#11753)
- [R] Fix off-by-one bug: nrounds=0 resulted in 2 iterations #11856
- [R] Fix mingw warnings, winbuilder check warnings, memory safety issues. (#11859, #11847, #11830, #11906)
- Avoid overflow in rounding estimation. (#11910)
- Workaround compiler issue on Windows, affects the use of max_delta_step with CUDA. (#11916)

1.18.4 3.1.2 Patch Release (Nov 20 2025)

- Fix loading nccl 2.28. (#11806)
- Fix ordering of Python callbacks. (#11812)
- Infer the `enable_categorical` during model load. (#11816)

1.18.5 3.1.1 Patch Release (Oct 22 2025)

- Emit correct error when performing inplace-predict using a CPU-only version of XGBoost, but with a GPU input. (#11761)
- Enhance the error message for loading the removed binary model format. (#11760)
- Use the correct group ID for SHAP when the intercept is a vector. (#11764)

1.18.6 3.1.0 (2025 Sep 22)

We are delighted to share the latest 3.1.0 update for XGBoost.

Categorical Re-coder

This release features a major update to categorical data support by introducing a re-coder. This re-coder saves categories in the trained model and re-codes the data during inference, to keep the categorical encoding consistent. Aside from primitive types like integers, it also supports string-based categories. The implementation works with all supported Python DataFrame implementations. (#11609, #11665, #11605, #11628, #11598, #11591, #11568, #11561, #11650, #11621, #11611, #11313, #11311, #11310, #11315, #11303, #11612, #11098, #11347) See *Auto-recoding (Data Consistency)* for more information. (#11297)

In addition, categorical support for Polars data frames is now available (#11565).

Lastly, we removed the experimental tag for categorical feature support in this release. (#11690)

External Memory

We continue the work on external memory support on 3.1. In this release, XGBoost features an adaptive cache for CUDA external memory. The improved cache can split the data between CPU memory and GPU memory according to the underlying hardware and data size. (#11556, #11465, #11664, #11594, #11469, #11547, #11339, #11477, #11453, #11446, #11458, #11426, #11566, #11497)

Also, there's an optional support (opt-in) for using `nvcomp` and the GB200 decompression engine to handle sparse data (requires nvcomp as a plugin) (#11451, #11464, #11460, #11512, #11520). We improved the memory usage of quantile sketching with external memory (#11641) and optimized the predictor for training (#11548). To help ensure the training performance, the latest XGBoost features detection for NUMA (Non-Uniform Memory Access) node (#11538, #11576) for checking cross-socket data access. We are working on additional tooling to enhance NUMA node performance. Aside from features, we have also added various documentation improvements. (#11412, #11631)

Lastly, external memory support with text file input has been removed (#11562). Moving forward, we will focus on iterator inputs.

Multi-Target/Class Intercept

Starting with 3.1, the base-score (intercept) is estimated and stored as a vector when the model has multiple outputs, be it multi-target regression or multi-class classification. This change enhances the initial estimation for multi-output models and will be the starting point for future work on vector-leaf. (#11277, #11651, #11625, #11649, #11630, #11647, #11656, #11663)

Features

- Support leaf prediction with QDM on CPU. (#11620)
- Improve seed with mean sampling for the first iteration. (#11639)
- Optionally include git hash in CMake build. (#11587)

Removing Deprecated Features

This version removes some deprecated features, notably, the binary IO format, along with features deprecated in 2.0.

- Binary serialization format has been removed in 3.1. The format has been formally deprecated in 1.6. (#11307, #11553, #11552, #11602)
- Removed old GPU-related parameters including `use_gpu` (pyspark), `gpu_id`, `gpu_hist`, and `gpu_coord_descent`. These parameters have been deprecated in 2.0. Use the `device` parameter instead. (#11395, #11554, #11549, #11543, #11539, #11402)
- Remove deprecated C functions: `XGDMatrixCreateFromCSREx`, `XGDMatrixCreateFromCSCEx`. (#11514, #11513)
- XGBoost starts emit warning for text inputs. (#11590)

Optimizations

- Optimize CPU inference with Array-Based Tree Traversal (#11519)
- Specialize for GPU dense histogram. (#11443)
- [sycl] Improve L1 cache locality for histogram building. (#11555)
- [sycl] Reduce predictor memory consumption and improve L2 locality (#11603)

Fixes

- Fix static linking C++ libraries on macOS (#11522)
- Rename param.hh/cc to hist_param.hh/cc to fix xcode build (#11378)
- [sycl] Fix build with updated compiler (#11618)
- [sycl] Various fixes for fp32-only devices. (#11527, #11524)
- Fix compilation on android older than API 26 (#11366)
- Fix loading Gamma model from 1.3. (#11377)

Python Package

- Support mixing Python metrics and built-in metrics for the skl interface. (#11536)
- CUDA 13 Support for PyPI with the new `xgboost-cu13` package. (#11677, #11662)
- Remove wheels for manylinux2014. (#11673)
- Initial support for building variant wheels (#11531, #11645, #11294)
- Minimum PySpark version is now set to 3.4 (#11364). In addition, the PySpark interface now checks the validation indicator column type and has a fix for None column input. (#11535, #11523)
- [dask] Small cleanup for the predict function. (#11423)

R Package

Now that most of the deprecated features have been removed in this release, we will try to bring the latest R package back to CRAN.

- Implement Booster reset. (#11357)
- Improvements for documentation, including having code examples in XGBoost's sphinx documentation side, and notes for R-universe release. (#11369, #11410, #11685, #11316)

JVM Packages

- Support columnar inputs for cpu pipeline (#11352)
- Rewrite the *LabeledPoint* as a Java class (#11545)
- Various fixes and document updates. (#11525, #11508, #11489, #11682)

Documents

Changes for general documentation:

- Update notes about GPU memory usage. (#11375)
- Various fixes and updates. (#11503, #11532, #11328, #11344, #11626)

CI and Maintenance

- Code cleanups. (#11367, #11342, #11658, #11528, #11585, #11672, #11642, #11667, #11495, #11567)
- Various cleanup and fixes for tests. (#11405, #11389, #11396, #11456)
- Support CMake 4.0 (#11382)
- Various CI updates and fixes (#11318, #11349, #11653, #11637, #11683, #11638, #11644, #11306, #11560, #11323, #11617, #11341, #11693)

1.18.7 3.0.3 Patch Release (Jul 30 2025)

- Fix NDCG metric with non-exp gain. (#11534)
- Avoid using mean intercept for `rmsle`. (#11588)
- [jvm-packages] add `setNumEarlyStoppingRounds` API (#11571)
- Avoid implicit synchronization in GPU evaluation. (#11542)
- Remove CUDA check in the array interface handler (#11386)
- Fix check in GPU histogram. (#11574)
- Support Rapids 25.06 (#11504)
- Adding `enable_categorical` to the sklearn `.apply` method (#11550)
- Make xgboost.testing compatible with scikit-learn 1.7 (#11502)
- Add support for building xgboost wheels on Win-ARM64 (#11572, #11597, #11559)

1.18.8 3.0.2 Patch Release (May 25 2025)

- Dask 2025.4.0 scheduler info compatibility fix (#11462)
- Fix CUDA virtual memory fallback logic on WSL2 (#11471)

1.18.9 3.0.1 Patch Release (May 13 2025)

- Use `nvidia-smi` to detect the driver version and handle old drivers that don't support virtual memory. (#11391)
- Optimize deep trees for GPU external memory. (#11387)
- Small fix for page concatenation with external memory (#11338)
- Build xgboost-cpu for `manylinux_2_28_x86_64` (#11406)
- Workaround for different Dask versions (#11436)
- Output models now use denormal floating-point instead of `nan`. (#11428)
- Fix aarch64 CI. (#11454)

1.18.10 3.0.0 (2025 Feb 27)

3.0.0 is a milestone for XGBoost. This note will summarize some general changes and then list package-specific updates. The bump in the major version is for a reworked R package along with a significant update to the JVM packages.

- *External Memory Support*
- *Networking*
- *SYCL*
- *Features*
- *Optimization*
- *Breaking Changes*
- *Bug Fixes*
- *Documentation*
- *Python Package*
- *R Package*
- *JVM Packages*
- *Maintenance*
- *CI*

External Memory Support

This release features a major update to the external memory implementation with improved performance, a new *ExtMemQuantileDMatrix* for more efficient data initialization, new feature coverage including categorical data support and quantile regression support. Additionally, GPU-based external memory is reworked to support using CPU memory as a data cache. Last but not least, we worked on distributed training using external memory along with the spark package's initial support.

- A new *ExtMemQuantileDMatrix* class for fast data initialization with the `hist` tree method. The new class supports both CPU and GPU training. (#10689, #10682, #10886, #10860, #10762, #10694, #10876)
- External memory now supports distributed training (#10492, #10861). In addition, the Spark package can use external memory (the host memory) when the device is GPU. The default package on maven doesn't support RMM yet. For better performance, one needs to compile XGBoost from the source for now. (#11186, #11238, #11219)

- Improved performance with new optimizations for both the `hist`-specific training and the `approx` (*DMatrix*) method. (#10529, #10980, #10342)
- New demos and documents for external memory, including distributed training. (#11234, #10929, #10916, #10426, #11113)
- Reduced binary cache size and memory allocation overhead by not writing the cut matrix. (#10444)
- More feature coverage, including categorical data and all objective functions, including quantile regression. In addition, various prediction types like SHAP values are supported. (#10918, #10820, #10751, #10724)

Significant updates for the GPU-based external memory training implementation. (#10924, #10895, #10766, #10544, #10677, #10615, #10927, #10608, #10711)

- GPU-based external memory supports both batch-based and sampling-based training. Before the 3.0 release, XGBoost concatenates the data during training and stores the cache on disk. In 3.0, XGBoost can now stage the data on the host and fetch them by batch. (#10602, #10595, #10606, #10549, #10488, #10766, #10765, #10764, #10760, #10753, #10734, #10691, #10713, #10826, #10811, #10810, #10736, #10538, #11333)
- XGBoost can now utilize *NVLink-C2C* for GPU-based external memory training and can handle up to terabytes of data.
- Support prediction cache (#10707).
- Automatic page concatenation for improved GPU utilization (#10887).
- Improved quantile sketching algorithm for batch-based inputs. See the section for *new features* for more info.
- Optimization for nearly-dense input, see the section for *optimization* for more info.

See our latest document for details *Using XGBoost External Memory Version*. The PyPI package (`pip install`) doesn't have `RMM` support, which is required by the GPU external memory implementation. To experiment, you can compile XGBoost from source or wait for the RAPIDS conda package to be available.

Networking

Continuing the work from the previous release, we updated the network module to improve reliability. (#10453, #10756, #11111, #10914, #10828, #10735, #10693, #10676, #10349, #10397, #10566, #10526, #10349)

The timeout option is now supported for NCCL using the NCCL asynchronous mode (#10850, #10934, #10945, #10930).

In addition, a new *Config* class is added for users to specify various options including timeout, tracker port, etc for distributed training. Both the Dask interface and the PySpark interface support the new configuration. (#11003, #10281, #10983, #10973)

SYCL

Continuing the work on the SYCL integration, there are significant improvements in the feature coverage for this release from more training parameters and more objectives to distributed training, along with various optimization (#10884, #10883).

Starting with 3.0, the SYCL-plugin is close to feature-complete, users can start working on SYCL devices for in-core training and inference. Newly introduced features include:

- Dask support for distributed training (#10812)
- Various training procedures, including split evaluation (#10605, #10636), grow policy (#10690, #10681), cached prediction (#10701).
- Updates for objective functions. (#11029, #10931, #11016, #10993, #11064, #10325)
- On-going work for float32-only devices. (#10702)

Other related PRs (#10842, #10543, #10806, #10943, #10987, #10548, #10922, #10898, #10576)

Features

This section describes new features in the XGBoost core. For language-specific features, please visit corresponding sections.

- A new initialization method for objectives that are derived from GLM. The new method is based on the mean value of the input labels. The new method changes the result of the estimated `base_score`. (#10298, #11331)
- The *xgboost.QuantileDMatrix* can be used with all prediction types for both CPU and GPU.
- In prior releases, XGBoost makes a copy for the booster to release memory held by internal tree methods. We formalize the procedure into a new booster method *reset()* / *XGBoosterReset()*. (#11042)
- OpenMP thread setting is exposed to the XGBoost global configuration. Users can use it to workaround hard-coded OpenMP environment variables. (#11175)
- We improved learning to rank tasks for better hyper-parameter configuration and for distributed training.
  - In 3.0, all three distributed interfaces, including Dask, Spark, and PySpark, support sorting the data based on query ID. The option for the `DaskXGBRanker` is true by default and can be opted out. (#11146, #11007, #11047, #11012, #10823, #11023)
  - Also for learning to rank, a new parameter `lambdarank_score_normalization` is introduced to make one of the normalizations optional. (#11272)
  - The `lambdarank_normalization` now uses the number of pairs when normalizing the `mean` pair strategy. Previously, the gradient was used for both `topk` and `mean`. #11322
- We have improved GPU quantile sketching to reduce memory usage. The improvement helps the construction of the *QuantileDMatrix* and the new *ExtMemQuantileDMatrix*.
  - A new multi-level sketching algorithm is employed to reduce the overall memory usage with batched inputs.
  - In addition to algorithmic changes, internal memory usage estimation and the quantile container is also updated. (#10761, #10843)
  - The change introduces two more parameters for the *QuantileDMatrix* and *DataIter*, namely, `max_quantile_batches` and `min_cache_page_bytes`.
- More work is needed to improve the support of categorical features. This release supports plotting trees with stat for categorical nodes (#11053). In addition, some preparation work is ongoing for auto re-coding categories. (#11094, #11114, #11089) These are feature enhancements instead of blocking issues.
- Implement weight-based feature importance for vector-leaf. (#10700)
- Reduced logging in the DMatrix construction. (#11080)

Optimization

In addition to the external memory and quantile sketching improvements, we have a number of optimizations and performance fixes.

- GPU tree methods now use significantly less memory for both dense inputs and near-dense inputs. (#10821, #10870)
- For near-dense inputs, GPU training is much faster for both `hist` (about 2x) and `approx`.
- Quantile regression on CPU now can handle imbalance trees much more efficiently. (#11275)
- Small optimization for DMatrix construction to reduce latency. Also, C users can now reuse the *ProxyDMatrix* for multiple inference calls. (#11273)

- CPU prediction performance for *QuantileDMatrix* has been improved (#11139) and now is on par with normal DMatrix.
- Fixed a performance issue for running inference using CPU with extremely sparse *QuantileDMatrix* (#11250).
- Optimize CPU training memory allocation for improved performance. (#11112)
- Improved RMM (rapids memory manager) integration. Now, with the help of *config_context()*, all memory allocated by XGBoost should be routed to RMM. As a bonus, all thrust algorithms now use async policy. (#10873, #11173, #10712, #10712, #10562)
- When used without RMM, XGBoost is more careful with its use of caching allocator to avoid holding too much device memory. (#10582)

Breaking Changes

This section lists breaking changes that affect all packages.

- Remove the deprecated DeviceQuantileDMatrix. (#10974, #10491)
- Support for saving the model in the deprecated has been removed. Users can still load old models in 3.0. (#10490)
- Support for the legacy (blocking) CUDA stream is removed (#10607)
- XGBoost now requires CUDA 12.0 or later.

Bug Fixes

- Fix the quantile error metric (pinball loss) with multiple quantiles. (#11279)
- Fix potential access error when running prediction in multi-thread environment. (#11167)
- Check the correct dump format for the gblinear. (#10831)

Documentation

- A new tutorial for advanced usage with custom objective functions. (#10283, #10725)
- The new online document site now shows documents for all packages including Python, R, and JVM-based packages. (#11240, #11216, #11166)
- Lots of enhancements. (#10822, 11137, #11138, #11246, #11266, #11253, #10731, #11222, #10551, #10533)
- Consistent use of cmake in documents. (#10717)
- Add a brief description for using the offset from the GLM setting (like Poisson). (#10996)
- Cleanup document for building from source. (#11145)
- Various fixes. (#10412, #10405, #10353, #10464, #10587, #10350, #11131, #10815)
- Maintenance. (#11052, #10380)

Python Package

- The feature_weights parameter in the sklearn interface is now defined as a scikit-learn parameter. (#9506)
- Initial support for polars, categorical feature is not yet supported. (#11126, #11172, #11116)
- Reduce pandas dataframe overhead and overhead for various imports. (#11058, #11068)
- Better xlabel in *plot_importance()* (#11009)
- Validate reference dataset for training. The *train()* function now throws an error if a *QuantileDMatrix* is used as a validation dataset without a reference. (#11105)

- Fix misleading errors when feature names are missing during inference (#10814)
- Add Stacklevel to Python warning callback. The change helps improve the error message for the Python package. (#10977)
- Remove circular reference in DataIter. It helps reduce memory usage. (#11177)
- Add checks for invalid inputs for *cv*. (#11255)
- Update Python project classifiers. (#10381, #11028)
- Support doc link for the sklearn module. Users can now find links to documents in a jupyter notebook. (#10287)
- Dask
  - Prevent the training from hanging due to aborted workers. (#10985) This helps Dask XGBoost be robust against error. When a worker is killed, the training will fail with an exception instead of hang.
  - Optional support for client-side logging. (#10942)
  - Fix LTR with empty partition and NCCL error. (#11152)
  - Update to work with the latest Dask. (#11291)
  - See the *Features* section for changes to ranking models.
  - See the *Networking* section for changes with the communication module.
- PySpark
  - Expose Training and Validation Metrics. (#11133)
  - Add barrier before initializing the communicator. (#10938)
  - Extend support for columnar input to CPU (GPU-only previously). (#11299)
  - See the *Features* section for changes to ranking models.
  - See the *Networking* section for changes with the communication module.
- Document updates (#11265).
- Maintenance. (#11071, #11211, #10837, #10754, #10347, #10678, #11002, #10692, #11006, #10972, #10907, #10659, #10358, #11149, #11178, #11248)
- Breaking changes
  - Remove deprecated *feval*. (#11051)
  - Remove dask from the default import. (#10935) Users are now required to import the XGBoost Dask through:

    ```
    from xgboost import dask as dxgb
    ```

    instead of:

    ```
    import xgboost as xgb
    xgb.dask
    ```

    The change helps avoid introducing dask into the default import set.
  - Bump Python requirement to 3.10. (#10434)
  - Drop support for datatable. (#11070)

R Package

We have been reworking the R package for a few releases now. In 3.0, we will start publishing a new R package on R-universe, before moving toward a CRAN update. The new package features a much more ergonomic interface, which is also more idiomatic to R speakers. In addition, a range of new features are introduced to the package. To name a few, the new package includes categorical feature support, `QuantileDMatrix`, and an initial implementation of the external memory training. To test the new package:

```
install.packages('xgboost', repos = c('https://dmlc.r-universe.dev', 'https://cloud.r-
↪project.org'))
```

Also, we finally have an online documentation site for the R package featuring both vignettes and API references (#11166, #11257). A good starting point for the new interface is the new `xgboost()` function. We won't list all the feature gains here, as there are too many! Please visit the *XGBoost R Package* for more info. There's a migration guide (#11197) there if you use a previous XGBoost R package version.

- Support for the MSVC build was dropped due to incompatibility with R headers. (#10355, #11150)
- Maintenance (#11259)
- Related PRs. (#11171, #11231, #11223, #11073, #11224, #11076, #11084, #11081, #11072, #11170, #11123, #11168, #11264, #11140, #11117, #11104, #11095, #11125, #11124, #11122, #11108, #11102, #11101, #11100, #11077, #11099, #11074, #11065, #11092, #11090, #11096, #11148, #11151, #11159, #11204, #11254, #11109, #11141, #10798, #10743, #10849, #10747, #11022, #10989, #11026, #11060, #11059, #11041, #11043, #11025, #10674, #10727, #10745, #10733, #10750, #10749, #10744, #10794, #10330, #10698, #10687, #10688, #10654, #10456, #10556, #10465, #10337)

JVM Packages

The XGBoost 3.0 release features a significant update to the JVM packages, and in particular, the Spark package. There are breaking changes in packaging and some parameters. Please visit the *migration guide* for related changes. The work brings new features and a more unified feature set between CPU and GPU implementation. (#10639, #10833, #10845, #10847, #10635, #10630, #11179, #11184)

- Automatic partitioning for distributed learning to rank. See the *features* section above (#11023).
- Resolve spark compatibility issue (#10917)
- Support missing value when constructing dmatrix with iterator (#10628)
- Fix transform performance issue (#10925)
- Honor skip.native.build option in xgboost4j-gpu (#10496)
- Support array features type for CPU (#10937)
- Change default missing value to `NaN` for better alignment (#11225)
- Don't cast to float if it's already float (#10386)
- Maintenance. (#10982, #10979, #10978, #10673, #10660, #10835, #10836, #10857, #10618, #10627)

Maintenance

Code maintenance includes both refactoring (#10531, #10573, #11069), cleanups (#11129, #10878, #11244, #10401, #10502, #11107, #11097, #11130, #10758, #10923, #10541, #10990), and improvements for tests (#10611, #10658, #10583, #11245, #10708), along with fixing various warnings in compilers and test dependencies (#10757, #10641, #11062, #11226). Also, miscellaneous updates, including some dev scripts and profiling annotations (#10485, #10657, #10854, #10718, #11158, #10697, #11276).

Lastly, dependency updates (#10362, #10363, #10360, #10373, #10377, #10368, #10369, #10366, #11032, #11037, #11036, #11035, #11034, #10518, #10536, #10586, #10585, #10458, #10547, #10429, #10517, #10497, #10588,

#10975, #10971, #10970, #10949, #10947, #10863, #10953, #10954, #10951, #10590, #10600, #10599, #10535, #10516, #10786, #10859, #10785, #10779, #10790, #10777, #10855, #10848, #10778, #10772, #10771, #10862, #10952, #10768, #10770, #10769, #10664, #10663, #10892, #10979, #10978).

CI

- The CI is reworked to use *RunsOn* to integrate custom CI pipelines with GitHub action. The migration helps us reduce the maintenance burden and make the CI configuration more accessible to others. (#11001, #11079, #10649, #11196, #11055, #10483, #11078, #11157)
- Other maintenance work includes various small fixes, enhancements, and tooling updates. (#10877, #10494, #10351, #10609, #11192, #11188, #11142, #10730, #11066, #11063, #10800, #10995, #10858, #10685, #10593, #11061)

1.18.11 2.1.4 Patch Release (2025 Feb 6)

The 2.1.4 patch release incorporates the following fixes on top of the 2.1.3 release:

- XGBoost is now compatible with scikit-learn 1.6 (#11021, #11162)
- Build wheels with CUDA 12.8 and enable Blackwell support (#11187, #11202)
- Adapt to RMM 25.02 logger changes (#11153)

1.18.12 2.1.3 Patch Release (2024 Nov 26)

The 2.1.3 patch release makes the following bug fixes:

- [pyspark] Support large model size (#10984).
- Fix rng for the column sampler (#10998).
- Handle *cudf.pandas* proxy objects properly (#11014).

1.18.13 2.1.2 Patch Release (2024 Oct 23)

The 2.1.2 patch release makes the following bug fixes:

- Clean up and modernize release-artifacts.py (#10818)
- Fix ellpack categorical feature with missing values. (#10906)
- Fix unbiased ltr with training continuation. (#10908)
- Fix potential race in feature constraint. (#10719)
- Fix boolean array for arrow-backed DF. (#10527)
- Ensure that pip check does not fail due to a bad platform tag (#10755)
- Check cub errors (#10721)
- Limit the maximum number of threads. (#10872)
- Fixes for large size clusters. (#10880)
- POSIX compliant poll.h and mmap (#10767)

1.18.14 2.1.1 Patch Release (2024 Jul 31)

The 2.1.1 patch release makes the following bug fixes:

- [Dask] Disable broadcast in the scatter call so that predict function won't hang (#10632)

- [Dask] Handle empty partitions correctly (#10559)
- Fix federated learning for the encrypted GRPC backend (#10503)
- Fix a race condition in column splitter (#10572)
- Gracefully handle cases where system files like /sys/fs/cgroup/cpu.max are not readable by the user (#10623)
- Fix build and C++ tests for FreeBSD (#10480)
- Clarify the requirement Pandas 1.2+ (#10476)
- More robust endianness detection in R package build (#10642)

In addition, it contains several enhancements:

- Publish JVM packages targeting Linux ARM64 (#10487)
- Publish a CPU-only wheel under name xgboost-cpu (#10603)
- Support building with CUDA Toolkit 12.5 and latest CCCL (#10624, #10633, #10574)

1.18.15 2.1.0 (2024 Jun 20)

We are thrilled to announce the XGBoost 2.1 release. This note will start by summarizing some general changes and then highlighting specific package updates. As we are working on a new R interface, this release will not include the R package. We'll update the R package as soon as it's ready. Stay tuned!

- *Networking Improvements*
- *NCCL is now fetched from PyPI*
- *Parts of the Python package now require glibc 2.28+*
- *Multi-output*
- *Federated Learning*
- *Ongoing work for SYCL support*
- *Optimizations*
- *Deprecation and breaking changes*
- *Features*
- *Bug fixes*
- *Document*
- *Python package*
  - *Dask*
  - *PySpark*
  - *Breaking changes*
  - *Features*
  - *Fixes*
  - *Document*
  - *Maintenance*

- *JVM packages*
  - *Features and related documents*
  - *Bug Fixes*
- *Maintenance*
- *CI*

Networking Improvements

An important ongoing work for XGBoost, which we've been collaborating on, is to support resilience for improved scaling and federated learning on various platforms. The existing networking library in XGBoost, adopted from the RABIT project, can no longer meet the feature demand. We've revamped the RABIT module in this release to pave the way for future development. The choice of using an in-house version instead of an existing library is due to the active development status with frequent new feature requests like loading extra plugins for federated learning. The new implementation features:

- Both CPU and GPU communication (based on NCCL).
- A reusable tracker for both the Python package and JVM packages. With the new release, the JVM packages no longer require Python as a runtime dependency.
- Supports federated communication patterns for both CPU and GPU.
- Supports timeout. The high-level interface parameter is currently hard-coded to 30 minutes, which we plan to improve.
- Supports significantly more data types.
- Supports thread-based workers.
- Improved handling for worker errors, including better error messages when one of the peers dies during training.
- Work with IPv6. Currently, this is only supported by the dask interface.
- Built-in support for various operations like broadcast, allgatherV, allreduce, etc.

Related PRs (#9597, #9576, #9523, #9524, #9593, #9596, #9661, #10319, #10152, #10125, #10332, #10306, #10208, #10203, #10199, #9784, #9777, #9773, #9772, #9759, #9745, #9695, #9738, #9732, #9726, #9688, #9681, #9679, #9659, #9650, #9644, #9649, #9917, #9990, #10313, #10315, #10112, #9531, #10075, #9805, #10198, #10414).

The existing option of using `MPI` in RABIT is removed in the release. (#9525)

NCCL is now fetched from PyPI

In the previous version, XGBoost statically linked NCCL, which significantly increased the binary size and led to hitting the PyPI repository limit. With the new release, we have made a significant improvement. The new release can now dynamically load NCCL from an external source, reducing the binary size. For the PyPI package, the `nvidia-nccl-cu12` package will be fetched during installation. With more downstream packages reusing NCCL, we expect the user environments to be slimmer in the future as well. (#9796, #9804, #10447)

Parts of the Python package now require glibc 2.28+

Starting from 2.1.0, XGBoost Python package will be distributed in two variants:

- `manylinux_2_28`: for recent Linux distros with glibc 2.28 or newer. This variant comes with all features enabled.

- `manylinux2014`: for old Linux distros with glibc older than 2.28. This variant does not support GPU algorithms or federated learning.

The `pip` package manager will automatically choose the correct variant depending on your system.

Starting from **May 31, 2025**, we will stop distributing the `manylinux2014` variant and exclusively distribute the `manylinux_2_28` variant. We made this decision so that our CI/CD pipeline won't have depend on software components that reached end-of-life (such as CentOS 7). We strongly encourage everyone to migrate to recent Linux distros in order to use future versions of XGBoost.

Note. If you want to use GPU algorithms or federated learning on an older Linux distro, you have two alternatives:

1. Upgrade to a recent Linux distro with glibc 2.28+. OR
2. Build XGBoost from the source.

Multi-output

We continue the work on multi-target and vector leaf in this release:

- Revise the support for custom objectives with a new API, `XGBoosterTrainOneIter`. This new function supports strided matrices and CUDA inputs. In addition, custom objectives now return the correct shape for prediction. (#9508)
- The `hinge` objective now supports multi-target regression (#9850)
- Fix the gain calculation with vector leaf (#9978)
- Support graphviz plot for multi-target tree. (#10093)
- Fix multi-output with alternating strategies. (#9933)

Please note that the feature is still in progress and not suitable for production use.

Federated Learning

Progress has been made on federated learning with improved support for column-split, including the following updates:

- Column split work for both CPU and GPU. In addition, categorical data is now compatible with column split. (#9562, #9609, #9611, #9628, #9539, #9578, #9685, #9623, #9613, #9511, #9384, #9595)
- The use of UBJson to serialize split entries for column split has been implemented, aiding vector-leaf with column-based data split. (#10059, #10055, #9702)
- Documentation and small fixes. (#9610, #9552, #9614, #9867)

Ongoing work for SYCL support

XGBoost is developing a SYCL plugin for SYCL devices, starting with the `hist` tree method. (#10216, #9800, #10311, #9691, #10269, #10251, #10222, #10174, #10080, #10057, #10011, #10138, #10119, #10045, #9876, #9846, #9682) XGBoost now supports launchable inference on SYCL devices, and that work on adding SYCL support for training is ongoing.

Looking ahead, we plan to complete the training in coming releases and then focus on improving test coverage for SYCL, particularly for Python tests.

Optimizations

- Implement column sampler in CUDA for GPU-based tree methods. This helps us get faster training time when column sampling is employed (#9785)
- CMake LTO and CUDA arch (#9677)

- Small optimization to external memory with a thread pool. This reduces the number of threads launched during iteration. (#9605, #10288, #10374)

Deprecation and breaking changes

Package-specific breaking changes are outlined in respective sections. Here we list general breaking changes in this release:

- The command line interface is deprecated due to the increasing complexity of the machine learning ecosystem. Building a machine learning model using a command shell is no longer feasible and could mislead newcomers. (#9485)
- `Universal binary JSON` is now the default format for saving models (#9947, #9958, #9954, #9955). See https://github.com/dmlc/xgboost/issues/7547 for more info.
- The `XGBoosterGetModelRaw` is now removed after deprecation in 1.6. (#9617)
- Drop support for loading remote files. This feature lacks any test. Users are encouraged to use dedicated libraries to fetch remote content. (#9504)
- Remove the dense libsvm parser plugin. This plugin is never tested or documented (#9799)
- `XGDMatrixSetDenseInfo` and `XGDMatrixSetUIntInfo` are now deprecated. Use the array interface based alternatives instead.

Features

This section lists some new features that are general to all language bindings. For package-specific changes, please visit respective sections.

- Adopt a new XGBoost logo (#10270)
- Now supports dataframe data format in native XGBoost. This improvement enhances performance and reduces memory usage when working with dataframe-based structures such as pandas, arrow, and R dataframe. (#9828, #9616, #9905)
- Change default metric for gamma regression to `deviance`. (#9757)
- Normalization for learning to rank is now optional with the introduction of the new `lambdarank_normalization` parameter. (#10094)
- Contribution prediction with `QuantileDMatrix` on CPU. (#10043)
- XGBoost on macos no longer bundles OpenMP runtime. Users can install the latest runtime from their dependency manager of choice. (#10440). Along with which, JVM packages on MacoOS are now built with OpenMP support (#10449).

Bug fixes

- Fix training with categorical data from external memory. (#10433)
- Fix compilation with CTK-12. (#10123)
- Fix inconsistent runtime library on Windows. (#10404)
- Fix default metric configuration. (#9575)
- Fix feature names with special characters. (#9923)
- Fix global configuration for external memory training. (#10173)
- Disable column sample by node for the exact tree method. (#10083)
- Fix the `FieldEntry` constructor specialization syntax error (#9980)

- Fix pairwise objective with NDCG metric along with custom gain. (#10100)
- Fix the default value for `lambdarank_pair_method`. (#10098)
- Fix UBJSON with boolean values. No existing code is affected by this fix. (#10054)
- Be more lenient on floating point errors for AUC. This prevents the AUC > 1.0 error. (#10264)
- Check support status for categorical features. This prevents `gblinear` from treating categorical features as numerical. (#9946)

Document

Here is a list of documentation changes not specific to any XGBoost package.

- A new coarse map for XGBoost features to assist development. (#10310)
- New language binding consistency guideline. (#9755, #9866)
- Fixes, cleanups, small updates (#9501, #9988, #10023, #10013, #10143, #9904, #10179, #9781, #10340, #9658, #10182, #9822)
- Update document for parameters (#9900)
- Brief introduction to `base_score`. (#9882)
- Mention data consistency for categorical features. (#9678)

Python package

Dask

Other than the changes in networking, we have some optimizations and document updates in dask:

- Filter models on workers instead of clients; this prevents an OOM error on the client machine. (#9518)
- Users are now encouraged to use *from xgboost import dask* instead of *import xgboost.dask* to avoid drawing in unnecessary dependencies for non-dask users. (#9742)
- Add seed to demos. (#10009)
- New document for using dask XGBoost with k8s. (#10271)
- Workaround potentially unaligned pointer from an empty partition. (#10418)
- Workaround a race condition in the latest dask. (#10419)
- [doc] Add typing to dask demos. (#10207)

PySpark

PySpark has several new features along with some small fixes:

- Support stage-level scheduling for training on various platforms, including yarn/k8s. (#9519, #10209, #9786, #9727)
- Support GPU-based transform methods (#9542)
- Avoid expensive repartition when appropriate. (#10408)
- Refactor the logging and the GPU code path (#10077, 9724)
- Sort workers by task ID. This helps the PySpark interface obtain deterministic results. (#10220)
- Fix PySpark with `verbosity=3`. (#10172)
- Fix spark estimator doc. (#10066)

- Rework transform for improved code reusing. (#9292)

Breaking changes

For the Python package, `eval_metric`, `early_stopping_rounds`, and `callbacks` from now removed from the `fit` method in the sklearn interface. They were deprecated in 1.6. Use the parameters with the same name in constructors instead. (#9986)

Features

Following is a list of new features in the Python package:

- Support sample weight in sklearn custom objective. (#10050)
- New supported data types, including `cudf.pandas` (#9602), `torch.Tensor` (#9971), and more scipy types (#9881).
- Support pandas 2.2 and numpy 2.0. (#10266, #9557, #10252, #10175)
- Support the latest rapids including rmm. (#10435)
- Improved data cache option in data iterator. (#10286)
- Accept numpy generators as `random_state` (#9743)
- Support returning base score as intercept in the sklearn interface. (#9486)
- Support arrow through pandas ext types. This is built on top of the new DataFrame API in XGBoost. See general features for more info. (#9612)
- Handle np integer in model slice and prediction. (#10007)
- Improved sklearn tags support. (#10230)
- The base image for building Linux binary wheels is updated to rockylinux8. (#10399)
- Improved handling for float128. (#10322)

Fixes

- Fix `DMatrix` with `None` input. (#10052)
- Fix native library discovery logic. (#9712, #9860)
- Fix using categorical data with the score function for the ranker. (#9753)

Document

- Clarify the effect of `enable_categorical` (#9877, #9884)
- Update the Python introduction. (#10033)
- Fixes. (#10058, #9991, #9573)

Maintenance

- Use array interface in Python prediction return. (#9855)
- Synthesize the AMES housing dataset for tests. (#9963)
- linter, formatting, etc. (#10296, #10014)
- Tests. (#9962, #10285, #9997, #9943, #9934)

JVM packages

Here is a list of JVM-specific changes. Like the PySpark package, the JVM package also gains stage-level scheduling.

Features and related documents

- Support stage-level scheduling (#9775)
- Allow JVM-Package to access inplace predict method (#9167)
- Support JDK 17 for test (#9959)
- Various dependency updates.(#10211, #10210, #10217, #10156, #10070, #9809, #9517, #10235, #10276, #9331, #10335, #10309, #10240, #10244, #10260, #9489, #9326, #10294, #10197, #10196, #10193, #10202, #10191, #10188, #9328, #9311, #9951, #10151, #9827, #9820, #10253)
- Update and fixes for document. (#9752, #10385)
- Remove rabit checkpoint. (#9599)

Bug Fixes

- Fixes memory leak in error handling. (#10307)
- Fixes group col for GPU packages (#10254)

Maintenance

- Add formatting and linting requirements to the CMake script. (#9653, #9641, #9637, #9728, #9674)
- Refactors and cleanups (#10085, #10120, #10074, #9645, #9992, #9568, #9731, #9527).
- Update nvtx. (#10227)
- Tests. (#9499, #9553, #9737)
- Throw error for 32-bit architectures (#10005)
- Helpers. (#9505, #9572, #9750, #9541, #9983, #9714)
- Fix mingw hanging on regex in context (#9729)
- Linters. (#10010, #9634)

CI

- Meta info about the Python package is uploaded for easier parsing (#10295)
- Various dependency updates (#10274, #10280, #10278, #10275, #10320, #10305, #10267, #9544, #10228, #10133, #10187, #9857, #10042, #10268, #9654, #9835)
- GitHub Action fixes (#10067, #10134, #10064)
- Improved support for Apple devices. (#10225, #9886, #9699, #9748, #9704, #9749)
- Stop Windows pipeline upon a failing pytest (#10003)
- Cancel GH Action job if a newer commit is published (#10088)
- CI images. (#9666, #10201, #9932)
- Test R package with CMake (#10087)
- Test building for the 32-bit arch (#10021)
- Test federated plugin using GitHub action. (#10336)

PYTHON MODULE INDEX

X

xgboost.callback, 260
xgboost.collective, 296
xgboost.core, 163
xgboost.plotting, 258
xgboost.sklearn, 195
xgboost.spark, 264
xgboost.tracker, 298
xgboost.training, 192

INDEX

Symbols

__getitem__() (*xgboost.Booster method*), 181

A

after_iteration() (*xgboost.callback.EarlyStopping method*), 262
after_iteration() (*xgboost.callback.EvaluationMonitor method*), 261
after_iteration() (*xgboost.callback.LearningRateScheduler method*), 263
after_iteration() (*xgboost.callback.TrainingCallback method*), 260
after_iteration() (*xgboost.callback.TrainingCheckPoint method*), 264
after_training() (*xgboost.callback.EarlyStopping method*), 263
after_training() (*xgboost.callback.EvaluationMonitor method*), 261
after_training() (*xgboost.callback.TrainingCallback method*), 260
apply() (*xgboost.XGBClassifier method*), 211
apply() (*xgboost.XGBRanker method*), 224
apply() (*xgboost.XGBRegressor method*), 199
apply() (*xgboost.XGBRFClassifier method*), 249
apply() (*xgboost.XGBRFRegressor method*), 237
attr() (*xgboost.Booster method*), 181
attributes() (*xgboost.Booster method*), 181

B

before_iteration() (*xgboost.callback.TrainingCallback method*), 260
before_training() (*xgboost.callback.EarlyStopping method*), 263
before_training() (*xgboost.callback.TrainingCallback method*), 261
before_training() (*xgboost.callback.TrainingCheckPoint method*), 264
best_iteration (*xgboost.Booster property*), 181
best_iteration (*xgboost.XGBClassifier property*), 211
best_iteration (*xgboost.XGBRanker property*), 225
best_iteration (*xgboost.XGBRegressor property*), 199
best_iteration (*xgboost.XGBRFClassifier property*), 249
best_iteration (*xgboost.XGBRFRegressor property*), 237
best_score (*xgboost.Booster property*), 181
best_score (*xgboost.XGBClassifier property*), 211
best_score (*xgboost.XGBRanker property*), 225
best_score (*xgboost.XGBRegressor property*), 199
best_score (*xgboost.XGBRFClassifier property*), 249
best_score (*xgboost.XGBRFRegressor property*), 237
boost() (*xgboost.Booster method*), 181
Booster (*class in xgboost*), 181
BoosterHandle (*C++ type*), 434
build_info() (*in module xgboost*), 163

C

Categories (*class in xgboost.core*), 191
CategoriesHandle (*C++ type*), 434
clear() (*xgboost.spark.SparkXGBClassifier method*), 266
clear() (*xgboost.spark.SparkXGBClassifierModel method*), 271
clear() (*xgboost.spark.SparkXGBRanker method*), 287
clear() (*xgboost.spark.SparkXGBRankerModel method*), 292
clear() (*xgboost.spark.SparkXGBRegressor method*), 277
clear() (*xgboost.spark.SparkXGBRegressorModel method*), 281
coef_ (*xgboost.XGBClassifier property*), 212
coef_ (*xgboost.XGBRanker property*), 225
coef_ (*xgboost.XGBRegressor property*), 200
coef_ (*xgboost.XGBRFClassifier property*), 249
coef_ (*xgboost.XGBRFRegressor property*), 237

CommunicatorContext (*class in xgboost.collective*), 298
Config (*class in xgboost.collective*), 296
config_context() (*in module xgboost*), 160
copy() (*xgboost.Booster method*), 182
copy() (*xgboost.spark.SparkXGBClassifier method*), 266
copy() (*xgboost.spark.SparkXGBClassifierModel method*), 271
copy() (*xgboost.spark.SparkXGBRanker method*), 288
copy() (*xgboost.spark.SparkXGBRankerModel method*), 292
copy() (*xgboost.spark.SparkXGBRegressor method*), 277
copy() (*xgboost.spark.SparkXGBRegressorModel method*), 281
cv() (*in module xgboost*), 193

D

data_split_mode() (*xgboost.DMatrix method*), 165
data_split_mode() (*xgboost.ExtMemQuantileDMatrix method*), 176
data_split_mode() (*xgboost.QuantileDMatrix method*), 171
DataHolderHandle (*C++ type*), 448
DataIter (*class in xgboost*), 189
DataIterHandle (*C++ type*), 448
DataIterResetCallback (*C++ type*), 448
DMatrix (*class in xgboost*), 163
DMatrixHandle (*C++ type*), 434
dump_model() (*xgboost.Booster method*), 182

E

EarlyStopping (*class in xgboost.callback*), 262
eval() (*xgboost.Booster method*), 182
eval_set() (*xgboost.Booster method*), 182
evals_result() (*xgboost.XGBClassifier method*), 212
evals_result() (*xgboost.XGBRanker method*), 225
evals_result() (*xgboost.XGBRegressor method*), 200
evals_result() (*xgboost.XGBRFClassifier method*), 250
evals_result() (*xgboost.XGBRFRegressor method*), 238
EvaluationMonitor (*class in xgboost.callback*), 261
explainParam() (*xgboost.spark.SparkXGBClassifier method*), 267
explainParam() (*xgboost.spark.SparkXGBClassifierModel method*), 271
explainParam() (*xgboost.spark.SparkXGBRanker method*), 288
explainParam() (*xgboost.spark.SparkXGBRankerModel method*), 292
explainParam() (*xgboost.spark.SparkXGBRegressor method*), 277
explainParam() (*xgboost.spark.SparkXGBRegressorModel method*), 282
explainParams() (*xgboost.spark.SparkXGBClassifier method*), 267
explainParams() (*xgboost.spark.SparkXGBClassifierModel method*), 271
explainParams() (*xgboost.spark.SparkXGBRanker method*), 288
explainParams() (*xgboost.spark.SparkXGBRankerModel method*), 292
explainParams() (*xgboost.spark.SparkXGBRegressor method*), 278
explainParams() (*xgboost.spark.SparkXGBRegressorModel method*), 282
ExtMemQuantileDMatrix (*class in xgboost*), 176
extractParamMap() (*xgboost.spark.SparkXGBClassifier method*), 267
extractParamMap() (*xgboost.spark.SparkXGBClassifierModel method*), 271
extractParamMap() (*xgboost.spark.SparkXGBRanker method*), 288
extractParamMap() (*xgboost.spark.SparkXGBRankerModel method*), 292
extractParamMap() (*xgboost.spark.SparkXGBRegressor method*), 278
extractParamMap() (*xgboost.spark.SparkXGBRegressorModel method*), 282

F

feature_importances_ (*xgboost.XGBClassifier property*), 212
feature_importances_ (*xgboost.XGBRanker property*), 225
feature_importances_ (*xgboost.XGBRegressor property*), 200
feature_importances_ (*xgboost.XGBRFClassifier property*), 250
feature_importances_ (*xgboost.XGBRFRegressor property*), 238
feature_names (*xgboost.Booster property*), 183
feature_names (*xgboost.DMatrix property*), 165
feature_names (*xgboost.ExtMemQuantileDMatrix property*), 176
feature_names (*xgboost.QuantileDMatrix property*), 171

feature_names_in_ (*xgboost.XGBClassifier property*), 212
feature_names_in_ (*xgboost.XGBRanker property*), 226
feature_names_in_ (*xgboost.XGBRegressor property*), 200
feature_names_in_ (*xgboost.XGBRFClassifier property*), 250
feature_names_in_ (*xgboost.XGBRFRegressor property*), 238
feature_types (*xgboost.Booster property*), 183
feature_types (*xgboost.DMatrix property*), 165
feature_types (*xgboost.ExtMemQuantileDMatrix property*), 176
feature_types (*xgboost.QuantileDMatrix property*), 172
finalize() (*in module xgboost.collective*), 298
fit() (*xgboost.spark.SparkXGBClassifier method*), 267
fit() (*xgboost.spark.SparkXGBRanker method*), 288
fit() (*xgboost.spark.SparkXGBRegressor method*), 278
fit() (*xgboost.XGBClassifier method*), 212
fit() (*xgboost.XGBRanker method*), 226
fit() (*xgboost.XGBRegressor method*), 200
fit() (*xgboost.XGBRFClassifier method*), 250
fit() (*xgboost.XGBRFRegressor method*), 238
fitMultiple() (*xgboost.spark.SparkXGBClassifier method*), 267
fitMultiple() (*xgboost.spark.SparkXGBRanker method*), 289
fitMultiple() (*xgboost.spark.SparkXGBRegressor method*), 278

G

get_base_margin() (*xgboost.DMatrix method*), 165
get_base_margin() (*xgboost.ExtMemQuantileDMatrix method*), 177
get_base_margin() (*xgboost.QuantileDMatrix method*), 172
get_booster() (*xgboost.spark.SparkXGBClassifierModel method*), 273
get_booster() (*xgboost.spark.SparkXGBRankerModel method*), 293
get_booster() (*xgboost.spark.SparkXGBRegressorModel method*), 283
get_booster() (*xgboost.XGBClassifier method*), 213
get_booster() (*xgboost.XGBRanker method*), 227
get_booster() (*xgboost.XGBRegressor method*), 201
get_booster() (*xgboost.XGBRFClassifier method*), 251
get_booster() (*xgboost.XGBRFRegressor method*), 239
get_callbacks() (*xgboost.DataIter method*), 191
get_categories() (*xgboost.Booster method*), 183
get_categories() (*xgboost.DMatrix method*), 165
get_categories() (*xgboost.ExtMemQuantileDMatrix method*), 177
get_categories() (*xgboost.QuantileDMatrix method*), 172
get_config() (*in module xgboost*), 162
get_data() (*xgboost.DMatrix method*), 165
get_data() (*xgboost.ExtMemQuantileDMatrix method*), 177
get_data() (*xgboost.QuantileDMatrix method*), 172
get_dump() (*xgboost.Booster method*), 183
get_feature_importances() (*xgboost.spark.SparkXGBClassifierModel method*), 273
get_feature_importances() (*xgboost.spark.SparkXGBRankerModel method*), 294
get_feature_importances() (*xgboost.spark.SparkXGBRegressorModel method*), 283
get_float_info() (*xgboost.DMatrix method*), 165
get_float_info() (*xgboost.ExtMemQuantileDMatrix method*), 177
get_float_info() (*xgboost.QuantileDMatrix method*), 172
get_fscore() (*xgboost.Booster method*), 183
get_group() (*xgboost.DMatrix method*), 166
get_group() (*xgboost.ExtMemQuantileDMatrix method*), 177
get_group() (*xgboost.QuantileDMatrix method*), 172
get_label() (*xgboost.DMatrix method*), 166
get_label() (*xgboost.ExtMemQuantileDMatrix method*), 177
get_label() (*xgboost.QuantileDMatrix method*), 172
get_metadata_routing() (*xgboost.XGBClassifier method*), 213
get_metadata_routing() (*xgboost.XGBRanker method*), 228
get_metadata_routing() (*xgboost.XGBRegressor method*), 201
get_metadata_routing() (*xgboost.XGBRFClassifier method*), 251
get_metadata_routing() (*xgboost.XGBRFRegressor method*), 239
get_num_boosting_rounds() (*xgboost.XGBClassifier method*), 214
get_num_boosting_rounds() (*xgboost.XGBRanker method*), 228
get_num_boosting_rounds() (*xgboost.XGBRegressor method*), 202
get_num_boosting_rounds() (*xgboost.XGBRFClassifier method*), 251
get_num_boosting_rounds() (*xgboost.XGBRFRegressor method*), 239

get_params() (*xgboost.XGBClassifier method*), 214
get_params() (*xgboost.XGBRanker method*), 228
get_params() (*xgboost.XGBRegressor method*), 202
get_params() (*xgboost.XGBRFClassifier method*), 252
get_params() (*xgboost.XGBRFRegressor method*), 239
get_quantile_cut() (*xgboost.DMatrix method*), 166
get_quantile_cut() (*xgboost.ExtMemQuantileDMatrix method*), 177
get_quantile_cut() (*xgboost.QuantileDMatrix method*), 173
get_rank() (*in module xgboost.collective*), 298
get_score() (*xgboost.Booster method*), 184
get_split_value_histogram() (*xgboost.Booster method*), 184
get_uint_info() (*xgboost.DMatrix method*), 166
get_uint_info() (*xgboost.ExtMemQuantileDMatrix method*), 178
get_uint_info() (*xgboost.QuantileDMatrix method*), 173
get_weight() (*xgboost.DMatrix method*), 166
get_weight() (*xgboost.ExtMemQuantileDMatrix method*), 178
get_weight() (*xgboost.QuantileDMatrix method*), 173
get_world_size() (*in module xgboost.collective*), 298
get_xgb_params() (*xgboost.XGBClassifier method*), 214
get_xgb_params() (*xgboost.XGBRanker method*), 228
get_xgb_params() (*xgboost.XGBRegressor method*), 202
get_xgb_params() (*xgboost.XGBRFClassifier method*), 252
get_xgb_params() (*xgboost.XGBRFRegressor method*), 240
getFeaturesCol() (*xgboost.spark.SparkXGBClassifier method*), 268
getFeaturesCol() (*xgboost.spark.SparkXGBClassifierModel method*), 272
getFeaturesCol() (*xgboost.spark.SparkXGBRanker method*), 289
getFeaturesCol() (*xgboost.spark.SparkXGBRankerModel method*), 293
getFeaturesCol() (*xgboost.spark.SparkXGBRegressor method*), 278
getFeaturesCol() (*xgboost.spark.SparkXGBRegressorModel method*), 282
getLabelCol() (*xgboost.spark.SparkXGBClassifier method*), 268
getLabelCol() (*xgboost.spark.SparkXGBClassifierModel method*), 272
getLabelCol() (*xgboost.spark.SparkXGBRanker method*), 289
getLabelCol() (*xgboost.spark.SparkXGBRankerModel method*), 293
getLabelCol() (*xgboost.spark.SparkXGBRegressor method*), 279
getLabelCol() (*xgboost.spark.SparkXGBRegressorModel method*), 282
getOrDefault() (*xgboost.spark.SparkXGBClassifier method*), 268
getOrDefault() (*xgboost.spark.SparkXGBClassifierModel method*), 272
getOrDefault() (*xgboost.spark.SparkXGBRanker method*), 289
getOrDefault() (*xgboost.spark.SparkXGBRankerModel method*), 293
getOrDefault() (*xgboost.spark.SparkXGBRegressor method*), 279
getOrDefault() (*xgboost.spark.SparkXGBRegressorModel method*), 282
getParam() (*xgboost.spark.SparkXGBClassifier method*), 268
getParam() (*xgboost.spark.SparkXGBClassifierModel method*), 272
getParam() (*xgboost.spark.SparkXGBRanker method*), 289
getParam() (*xgboost.spark.SparkXGBRankerModel method*), 293
getParam() (*xgboost.spark.SparkXGBRegressor method*), 279
getParam() (*xgboost.spark.SparkXGBRegressorModel method*), 283
getPredictionCol() (*xgboost.spark.SparkXGBClassifier method*), 268
getPredictionCol() (*xgboost.spark.SparkXGBClassifierModel method*), 272
getPredictionCol() (*xgboost.spark.SparkXGBRanker method*), 289
getPredictionCol() (*xgboost.spark.SparkXGBRankerModel method*), 293
getPredictionCol() (*xgboost.spark.SparkXGBRegressor method*), 279
getPredictionCol() (*xgboost.spark.SparkXGBRegressorModel method*), 283
getProbabilityCol() (*xgboost.spark.SparkXGBClassifier method*), 268
getProbabilityCol() (*xgboost.spark.SparkXGBClassifierModel method*), 272

getRawPredictionCol() (xgboost.spark.SparkXGBClassifier method), 268
getRawPredictionCol() (xgboost.spark.SparkXGBClassifierModel method), 272
getValidationIndicatorCol() (xgboost.spark.SparkXGBClassifier method), 268
getValidationIndicatorCol() (xgboost.spark.SparkXGBClassifierModel method), 272
getValidationIndicatorCol() (xgboost.spark.SparkXGBRanker method), 289
getValidationIndicatorCol() (xgboost.spark.SparkXGBRankerModel method), 293
getValidationIndicatorCol() (xgboost.spark.SparkXGBRegressor method), 279
getValidationIndicatorCol() (xgboost.spark.SparkXGBRegressorModel method), 283
getWeightCol() (xgboost.spark.SparkXGBClassifier method), 269
getWeightCol() (xgboost.spark.SparkXGBClassifierModel method), 273
getWeightCol() (xgboost.spark.SparkXGBRanker method), 290
getWeightCol() (xgboost.spark.SparkXGBRankerModel method), 293
getWeightCol() (xgboost.spark.SparkXGBRegressor method), 279
getWeightCol() (xgboost.spark.SparkXGBRegressorModel method), 283

H

hasDefault() (xgboost.spark.SparkXGBClassifier method), 269
hasDefault() (xgboost.spark.SparkXGBClassifierModel method), 273
hasDefault() (xgboost.spark.SparkXGBRanker method), 290
hasDefault() (xgboost.spark.SparkXGBRankerModel method), 294
hasDefault() (xgboost.spark.SparkXGBRegressor method), 279
hasDefault() (xgboost.spark.SparkXGBRegressorModel method), 283
hasParam() (xgboost.spark.SparkXGBClassifier method), 269
hasParam() (xgboost.spark.SparkXGBClassifierModel method), 273
hasParam() (xgboost.spark.SparkXGBRanker method), 290
hasParam() (xgboost.spark.SparkXGBRankerModel method), 294
hasParam() (xgboost.spark.SparkXGBRegressor method), 279
hasParam() (xgboost.spark.SparkXGBRegressorModel method), 284

I

init() (in module xgboost.collective), 297
inplace_predict() (xgboost.Booster method), 185
intercept_ (xgboost.XGBClassifier property), 214
intercept_ (xgboost.XGBRanker property), 228
intercept_ (xgboost.XGBRegressor property), 202
intercept_ (xgboost.XGBRFClassifier property), 252
intercept_ (xgboost.XGBRFRegressor property), 240
isDefined() (xgboost.spark.SparkXGBClassifier method), 269
isDefined() (xgboost.spark.SparkXGBClassifierModel method), 273
isDefined() (xgboost.spark.SparkXGBRanker method), 290
isDefined() (xgboost.spark.SparkXGBRankerModel method), 294
isDefined() (xgboost.spark.SparkXGBRegressor method), 280
isDefined() (xgboost.spark.SparkXGBRegressorModel method), 284
isSet() (xgboost.spark.SparkXGBClassifier method), 269
isSet() (xgboost.spark.SparkXGBClassifierModel method), 274
isSet() (xgboost.spark.SparkXGBRanker method), 290
isSet() (xgboost.spark.SparkXGBRankerModel method), 294
isSet() (xgboost.spark.SparkXGBRegressor method), 280
isSet() (xgboost.spark.SparkXGBRegressorModel method), 284

L

LearningRateScheduler (class in xgboost.callback), 263
load() (xgboost.spark.SparkXGBClassifier class method), 269
load() (xgboost.spark.SparkXGBClassifierModel class method), 274
load() (xgboost.spark.SparkXGBRanker class method), 290
load() (xgboost.spark.SparkXGBRankerModel class method), 294
load() (xgboost.spark.SparkXGBRegressor class method), 280

load() (*xgboost.spark.SparkXGBRegressorModel class method*), 284
load_config() (*xgboost.Booster method*), 186
load_model() (*xgboost.Booster method*), 186
load_model() (*xgboost.XGBClassifier method*), 214
load_model() (*xgboost.XGBRanker method*), 228
load_model() (*xgboost.XGBRegressor method*), 202
load_model() (*xgboost.XGBRFClassifier method*), 252
load_model() (*xgboost.XGBRFRegressor method*), 240

M

module
    xgboost.callback, 260
    xgboost.collective, 296
    xgboost.core, 163
    xgboost.plotting, 258
    xgboost.sklearn, 195
    xgboost.spark, 264
    xgboost.tracker, 298
    xgboost.training, 192

N

n_features_in_ (*xgboost.XGBClassifier property*), 215
n_features_in_ (*xgboost.XGBRanker property*), 229
n_features_in_ (*xgboost.XGBRegressor property*), 203
n_features_in_ (*xgboost.XGBRFClassifier property*), 252
n_features_in_ (*xgboost.XGBRFRegressor property*), 240
next() (*xgboost.DataIter method*), 191
num_boosted_rounds() (*xgboost.Booster method*), 186
num_col() (*xgboost.DMatrix method*), 166
num_col() (*xgboost.ExtMemQuantileDMatrix method*), 178
num_col() (*xgboost.QuantileDMatrix method*), 173
num_features() (*xgboost.Booster method*), 186
num_nonmissing() (*xgboost.DMatrix method*), 166
num_nonmissing() (*xgboost.ExtMemQuantileDMatrix method*), 178
num_nonmissing() (*xgboost.QuantileDMatrix method*), 173
num_row() (*xgboost.DMatrix method*), 166
num_row() (*xgboost.ExtMemQuantileDMatrix method*), 178
num_row() (*xgboost.QuantileDMatrix method*), 173

P

params (*xgboost.spark.SparkXGBClassifier property*), 269
params (*xgboost.spark.SparkXGBClassifierModel property*), 274
params (*xgboost.spark.SparkXGBRanker property*), 290
params (*xgboost.spark.SparkXGBRankerModel property*), 295
params (*xgboost.spark.SparkXGBRegressor property*), 280
params (*xgboost.spark.SparkXGBRegressorModel property*), 284
plot_importance() (*in module xgboost*), 258
plot_tree() (*in module xgboost*), 258
predict() (*xgboost.Booster method*), 186
predict() (*xgboost.XGBClassifier method*), 215
predict() (*xgboost.XGBRanker method*), 229
predict() (*xgboost.XGBRegressor method*), 203
predict() (*xgboost.XGBRFClassifier method*), 252
predict() (*xgboost.XGBRFRegressor method*), 240
predict_proba() (*xgboost.XGBClassifier method*), 215
predict_proba() (*xgboost.XGBRFClassifier method*), 253
proxy (*xgboost.DataIter property*), 191

Q

QuantileDMatrix (*class in xgboost*), 169

R

RabitTracker (*class in xgboost.tracker*), 298
read() (*xgboost.spark.SparkXGBClassifier class method*), 269
read() (*xgboost.spark.SparkXGBClassifierModel class method*), 274
read() (*xgboost.spark.SparkXGBRanker class method*), 290
read() (*xgboost.spark.SparkXGBRankerModel class method*), 295
read() (*xgboost.spark.SparkXGBRegressor class method*), 280
read() (*xgboost.spark.SparkXGBRegressorModel class method*), 284
ref (*xgboost.ExtMemQuantileDMatrix property*), 178
ref (*xgboost.QuantileDMatrix property*), 173
reraise() (*xgboost.DataIter method*), 191
reset() (*xgboost.Booster method*), 187
reset() (*xgboost.DataIter method*), 191
retry (*xgboost.collective.Config attribute*), 296

S

save() (*xgboost.spark.SparkXGBClassifier method*), 270
save() (*xgboost.spark.SparkXGBClassifierModel method*), 274
save() (*xgboost.spark.SparkXGBRanker method*), 291
save() (*xgboost.spark.SparkXGBRankerModel method*), 295
save() (*xgboost.spark.SparkXGBRegressor method*), 280
save() (*xgboost.spark.SparkXGBRegressorModel method*), 284

save_binary() (*xgboost.DMatrix method*), 166
save_binary() (*xgboost.ExtMemQuantileDMatrix method*), 178
save_binary() (*xgboost.QuantileDMatrix method*), 173
save_config() (*xgboost.Booster method*), 188
save_model() (*xgboost.Booster method*), 188
save_model() (*xgboost.XGBClassifier method*), 216
save_model() (*xgboost.XGBRanker method*), 229
save_model() (*xgboost.XGBRegressor method*), 203
save_model() (*xgboost.XGBRFClassifier method*), 254
save_model() (*xgboost.XGBRFRegressor method*), 241
save_raw() (*xgboost.Booster method*), 188
score() (*xgboost.XGBClassifier method*), 216
score() (*xgboost.XGBRanker method*), 230
score() (*xgboost.XGBRegressor method*), 203
score() (*xgboost.XGBRFClassifier method*), 254
score() (*xgboost.XGBRFRegressor method*), 241
set() (*xgboost.spark.SparkXGBClassifier method*), 270
set() (*xgboost.spark.SparkXGBClassifierModel method*), 274
set() (*xgboost.spark.SparkXGBRanker method*), 291
set() (*xgboost.spark.SparkXGBRankerModel method*), 295
set() (*xgboost.spark.SparkXGBRegressor method*), 280
set() (*xgboost.spark.SparkXGBRegressorModel method*), 285
set_attr() (*xgboost.Booster method*), 188
set_base_margin() (*xgboost.DMatrix method*), 167
set_base_margin() (*xgboost.ExtMemQuantileDMatrix method*), 178
set_base_margin() (*xgboost.QuantileDMatrix method*), 174
set_coll_cfg() (*xgboost.spark.SparkXGBClassifier method*), 270
set_coll_cfg() (*xgboost.spark.SparkXGBClassifierModel method*), 274
set_coll_cfg() (*xgboost.spark.SparkXGBRanker method*), 291
set_coll_cfg() (*xgboost.spark.SparkXGBRankerModel method*), 295
set_coll_cfg() (*xgboost.spark.SparkXGBRegressor method*), 281
set_coll_cfg() (*xgboost.spark.SparkXGBRegressorModel method*), 285
set_config() (*in module xgboost*), 161
set_device() (*xgboost.spark.SparkXGBClassifier method*), 270
set_device() (*xgboost.spark.SparkXGBClassifierModel method*), 274
set_device() (*xgboost.spark.SparkXGBRanker method*), 291
set_device() (*xgboost.spark.SparkXGBRankerModel method*), 295
set_device() (*xgboost.spark.SparkXGBRegressor method*), 281
set_device() (*xgboost.spark.SparkXGBRegressorModel method*), 285
set_fit_request() (*xgboost.XGBClassifier method*), 216
set_fit_request() (*xgboost.XGBRanker method*), 230
set_fit_request() (*xgboost.XGBRegressor method*), 204
set_fit_request() (*xgboost.XGBRFClassifier method*), 254
set_fit_request() (*xgboost.XGBRFRegressor method*), 242
set_float_info() (*xgboost.DMatrix method*), 167
set_float_info() (*xgboost.ExtMemQuantileDMatrix method*), 179
set_float_info() (*xgboost.QuantileDMatrix method*), 174
set_float_info_npy2d() (*xgboost.DMatrix method*), 167
set_float_info_npy2d() (*xgboost.ExtMemQuantileDMatrix method*), 179
set_float_info_npy2d() (*xgboost.QuantileDMatrix method*), 174
set_group() (*xgboost.DMatrix method*), 167
set_group() (*xgboost.ExtMemQuantileDMatrix method*), 179
set_group() (*xgboost.QuantileDMatrix method*), 174
set_info() (*xgboost.DMatrix method*), 167
set_info() (*xgboost.ExtMemQuantileDMatrix method*), 179
set_info() (*xgboost.QuantileDMatrix method*), 174
set_label() (*xgboost.DMatrix method*), 168
set_label() (*xgboost.ExtMemQuantileDMatrix method*), 180
set_label() (*xgboost.QuantileDMatrix method*), 175
set_param() (*xgboost.Booster method*), 188
set_params() (*xgboost.XGBClassifier method*), 217
set_params() (*xgboost.XGBRanker method*), 231
set_params() (*xgboost.XGBRegressor method*), 205
set_params() (*xgboost.XGBRFClassifier method*), 255
set_params() (*xgboost.XGBRFRegressor method*), 243
set_predict_proba_request() (*xgboost.XGBClassifier method*), 218
set_predict_proba_request() (*xgboost.XGBRFClassifier method*), 255
set_predict_request() (*xgboost.XGBClassifier method*), 218
set_predict_request() (*xgboost.XGBRanker method*), 231
set_predict_request() (*xgboost.XGBRegressor*

method), 205
set_predict_request() (*xgboost.XGBRFClassifier method*), 256
set_predict_request() (*xgboost.XGBRFRegressor method*), 243
set_score_request() (*xgboost.XGBClassifier method*), 219
set_score_request() (*xgboost.XGBRegressor method*), 206
set_score_request() (*xgboost.XGBRFClassifier method*), 257
set_score_request() (*xgboost.XGBRFRegressor method*), 244
set_uint_info() (*xgboost.DMatrix method*), 168
set_uint_info() (*xgboost.ExtMemQuantileDMatrix method*), 180
set_uint_info() (*xgboost.QuantileDMatrix method*), 175
set_weight() (*xgboost.DMatrix method*), 168
set_weight() (*xgboost.ExtMemQuantileDMatrix method*), 180
set_weight() (*xgboost.QuantileDMatrix method*), 175
setParams() (*xgboost.spark.SparkXGBClassifier method*), 270
setParams() (*xgboost.spark.SparkXGBRanker method*), 291
setParams() (*xgboost.spark.SparkXGBRegressor method*), 280
slice() (*xgboost.DMatrix method*), 169
slice() (*xgboost.ExtMemQuantileDMatrix method*), 180
slice() (*xgboost.QuantileDMatrix method*), 175
SparkXGBClassifier (*class in xgboost.spark*), 264
SparkXGBClassifierModel (*class in xgboost.spark*), 270
SparkXGBRanker (*class in xgboost.spark*), 285
SparkXGBRankerModel (*class in xgboost.spark*), 291
SparkXGBRegressor (*class in xgboost.spark*), 275
SparkXGBRegressorModel (*class in xgboost.spark*), 281

T

timeout (*xgboost.collective.Config attribute*), 296
to_graphviz() (*in module xgboost*), 259
tracker_host_ip (*xgboost.collective.Config attribute*), 296
tracker_port (*xgboost.collective.Config attribute*), 297
tracker_timeout (*xgboost.collective.Config attribute*), 297
TrackerHandle (*C++ type*), 467
train() (*in module xgboost*), 192
TrainingCallback (*class in xgboost.callback*), 260
TrainingCheckPoint (*class in xgboost.callback*), 263
transform() (*xgboost.spark.SparkXGBClassifierModel method*), 275
transform() (*xgboost.spark.SparkXGBRankerModel method*), 295
transform() (*xgboost.spark.SparkXGBRegressorModel method*), 285
trees_to_dataframe() (*xgboost.Booster method*), 189

U

uid (*xgboost.spark.SparkXGBClassifier attribute*), 270
uid (*xgboost.spark.SparkXGBClassifierModel attribute*), 275
uid (*xgboost.spark.SparkXGBRanker attribute*), 291
uid (*xgboost.spark.SparkXGBRankerModel attribute*), 296
uid (*xgboost.spark.SparkXGBRegressor attribute*), 281
uid (*xgboost.spark.SparkXGBRegressorModel attribute*), 285
update() (*xgboost.Booster method*), 189

W

worker_port (*xgboost.collective.Config attribute*), 297
write() (*xgboost.spark.SparkXGBClassifier method*), 270
write() (*xgboost.spark.SparkXGBClassifierModel method*), 275
write() (*xgboost.spark.SparkXGBRanker method*), 291
write() (*xgboost.spark.SparkXGBRankerModel method*), 296
write() (*xgboost.spark.SparkXGBRegressor method*), 281
write() (*xgboost.spark.SparkXGBRegressorModel method*), 285

X

XGBCallbackDataIterNext (*C++ type*), 448
XGBCallbackSetData (*C++ type*), 448
XGBCategoriesFree (*C++ function*), 444
XGBClassifier (*class in xgboost*), 207
XGBGetGlobalConfig (*C++ function*), 435
XGBGetLastError (*C++ function*), 435
xgboost.callback
    module, 260
xgboost.collective
    module, 296
xgboost.core
    module, 163
xgboost.plotting
    module, 258
xgboost.sklearn
    module, 195
xgboost.spark
    module, 264
xgboost.tracker

module, 298
xgboost.training
module, 192
XGBoostBatchCSR (*C++ struct*), 452
XGBoosterBoostedRounds (*C++ function*), 453
XGBoosterBoostOneIter (*C++ function*), 454
XGBoosterCreate (*C++ function*), 453
XGBoosterDumpModel (*C++ function*), 455
XGBoosterDumpModelEx (*C++ function*), 455
XGBoosterDumpModelExWithFeatures (*C++ function*), 456
XGBoosterDumpModelWithFeatures (*C++ function*), 456
XGBoosterEvalOneIter (*C++ function*), 455
XGBoosterFeatureScore (*C++ function*), 458
XGBoosterFree (*C++ function*), 453
XGBoosterGetAttr (*C++ function*), 457
XGBoosterGetAttrNames (*C++ function*), 457
XGBoosterGetCategories (*C++ function*), 456
XGBoosterGetCategoriesExportToArrow (*C++ function*), 457
XGBoosterGetNumFeature (*C++ function*), 454
XGBoosterGetStrFeatureInfo (*C++ function*), 458
XGBoosterLoadJsonConfig (*C++ function*), 466
XGBoosterLoadModel (*C++ function*), 464
XGBoosterLoadModelFromBuffer (*C++ function*), 465
XGBoosterPredict (*C++ function*), 459
XGBoosterPredictFromColumnar (*C++ function*), 462
XGBoosterPredictFromCSR (*C++ function*), 462
XGBoosterPredictFromCudaArray (*C++ function*), 463
XGBoosterPredictFromCudaColumnar (*C++ function*), 463
XGBoosterPredictFromDense (*C++ function*), 461
XGBoosterPredictFromDMatrix (*C++ function*), 460
XGBoosterReset (*C++ function*), 453
XGBoosterSaveJsonConfig (*C++ function*), 466
XGBoosterSaveModel (*C++ function*), 465
XGBoosterSaveModelToBuffer (*C++ function*), 465
XGBoosterSerializeToBuffer (*C++ function*), 465
XGBoosterSetAttr (*C++ function*), 457
XGBoosterSetParam (*C++ function*), 454
XGBoosterSetStrFeatureInfo (*C++ function*), 458
XGBoosterSlice (*C++ function*), 453
XGBoosterTrainOneIter (*C++ function*), 454
XGBoosterUnserializeFromBuffer (*C++ function*), 466
XGBoosterUpdateOneIter (*C++ function*), 454
XGBoostVersion (*C++ function*), 435
XGBRanker (*class in xgboost*), 220
XGBRegisterLogCallback (*C++ function*), 435
XGBRegressor (*class in xgboost*), 195
XGBRFClassifier (*class in xgboost*), 244
XGBRFRegressor (*class in xgboost*), 232
XGBSetGlobalConfig (*C++ function*), 435
XGBuildInfo (*C++ function*), 435
XGCommunicatorAllreduce (*C++ function*), 470
XGCommunicatorBroadcast (*C++ function*), 470
XGCommunicatorFinalize (*C++ function*), 469
XGCommunicatorGetProcessorName (*C++ function*), 470
XGCommunicatorGetRank (*C++ function*), 469
XGCommunicatorGetWorldSize (*C++ function*), 469
XGCommunicatorInit (*C++ function*), 468
XGCommunicatorIsDistributed (*C++ function*), 469
XGCommunicatorPrint (*C++ function*), 470
XGDMatrixCallbackNext (*C++ type*), 448
XGDMatrixCreateFromCallback (*C++ function*), 449
XGDMatrixCreateFromColumnar (*C++ function*), 437
XGDMatrixCreateFromCSC (*C++ function*), 438
XGDMatrixCreateFromCSR (*C++ function*), 438
XGDMatrixCreateFromCudaArrayInterface (*C++ function*), 439
XGDMatrixCreateFromCudaColumnar (*C++ function*), 439
XGDMatrixCreateFromDataIter (*C++ function*), 448
XGDMatrixCreateFromDense (*C++ function*), 438
XGDMatrixCreateFromFile (*C++ function*), 436
XGDMatrixCreateFromMat (*C++ function*), 439
XGDMatrixCreateFromMat_omp (*C++ function*), 439
XGDMatrixCreateFromURI (*C++ function*), 436
XGDMatrixDataSplitMode (*C++ function*), 445
XGDMatrixFree (*C++ function*), 440
XGDMatrixGetCategories (*C++ function*), 442
XGDMatrixGetCategoriesExportToArrow (*C++ function*), 443
XGDMatrixGetDataAsCSR (*C++ function*), 446
XGDMatrixGetFloatInfo (*C++ function*), 444
XGDMatrixGetInfoRef (*C++ function*), 444
XGDMatrixGetQuantileCut (*C++ function*), 446
XGDMatrixGetStrFeatureInfo (*C++ function*), 442
XGDMatrixGetUIntInfo (*C++ function*), 445
XGDMatrixNumCol (*C++ function*), 445
XGDMatrixNumNonMissing (*C++ function*), 445
XGDMatrixNumRow (*C++ function*), 445
XGDMatrixSaveBinary (*C++ function*), 440
XGDMatrixSetDenseInfo (*C++ function*), 444
XGDMatrixSetFloatInfo (*C++ function*), 441
XGDMatrixSetInfoFromInterface (*C++ function*), 441
XGDMatrixSetStrFeatureInfo (*C++ function*), 441
XGDMatrixSetUIntInfo (*C++ function*), 441
XGDMatrixSliceDMatrix (*C++ function*), 440
XGDMatrixSliceDMatrixEx (*C++ function*), 440
XGExtMemQuantileDMatrixCreateFromCallback (*C++ function*), 450
XGProxyDMatrixCreate (*C++ function*), 449

XGProxyDMatrixSetDataColumnar (*C++ function*), 452
XGProxyDMatrixSetDataCSR (*C++ function*), 452
XGProxyDMatrixSetDataCudaArrayInterface (*C++ function*), 451
XGProxyDMatrixSetDataCudaColumnar (*C++ function*), 452
XGProxyDMatrixSetDataDense (*C++ function*), 452
XGQuantileDMatrixCreateFromCallback (*C++ function*), 450
XGTrackerCreate (*C++ function*), 467
XGTrackerFree (*C++ function*), 468
XGTrackerRun (*C++ function*), 468
XGTrackerWaitFor (*C++ function*), 468
XGTrackerWorkerArgs (*C++ function*), 468