xgboost

Release 2.0.0-dev

xgboost developers

Feb 03, 2023

CONTENTS

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.

CONTENTS

1.1 Installation Guide

XGBoost provides binary packages for some language bindings. The binary packages support the GPU algorithm (`gpu_hist`) 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*
      - *Conda*
    - *R*
    - *JVM*
  - *Nightly Build*
    - *Python*
    - *R*
    - *JVM*

1.1.1 Stable Release

Python

Pre-built binary 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 install xgboost
```

You might need to run the command with `--user` flag or use `virtualenv` if you run into permission errors. Python pre-built binary capability for each platform:

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

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 only
conda install -c conda-forge py-xgboost-cpu
# Use NVIDIA GPU
conda install -c conda-forge py-xgboost-gpu
```

Visit the Miniconda website to obtain Conda.

Note: `py-xgboost-gpu` not available on Windows.

The `py-xgboost-gpu` is currently not available on Windows. If you are using Windows, please use `pip` to install XGBoost with GPU support.

R

- From CRAN:

```
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.

- 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
```

JVM

- XGBoost4j/XGBoost4j-Spark

Listing 1: Maven

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

<dependencies>
  ...
  <dependency>
      <groupId>ml.dmlc</groupId>
      <artifactId>xgboost4j_${scala.binary.version}</artifactId>
      <version>latest_version_num</version>
  </dependency>
  <dependency>
      <groupId>ml.dmlc</groupId>
      <artifactId>xgboost4j-spark_${scala.binary.version}</artifactId>
      <version>latest_version_num</version>
  </dependency>
</dependencies>
```

Listing 2: sbt

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

- XGBoost4j-GPU/XGBoost4j-Spark-GPU

Listing 3: Maven

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

<dependencies>
  ...
  <dependency>
      <groupId>ml.dmlc</groupId>
```

(continues on next page)

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

Listing 4: sbt

```
libraryDependencies ++= Seq(
  "ml.dmlc" %% "xgboost4j-gpu" % "latest_version_num",
  "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 (`tree_method='gpu_hist'`), use artifacts `xgboost4j-gpu_2.12` and `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

```xml
<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 as a dependency:

Listing 7: maven

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

<dependencies>
  ...
  <dependency>
      <groupId>ml.dmlc</groupId>
      <artifactId>xgboost4j_${scala.binary.version}</artifactId>
      <version>latest_version_num-SNAPSHOT</version>
  </dependency>
  <dependency>
      <groupId>ml.dmlc</groupId>
      <artifactId>xgboost4j-spark_${scala.binary.version}</artifactId>
      <version>latest_version_num-SNAPSHOT</version>
  </dependency>
</dependencies>
```

Listing 8: sbt

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

- XGBoost4j-GPU/XGBoost4j-Spark-GPU

Listing 9: maven

```xml
<properties>
  ...
```

(continues on next page)

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

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

Listing 10: sbt

```scala
libraryDependencies ++= Seq(
  "ml.dmlc" %% "xgboost4j-gpu" % "latest_version_num-SNAPSHOT",
  "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 (`tree_method='gpu_hist'`), 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 the user forum.

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

- *Building From Source*

1.2.1 Obtaining the Source Code

To obtain the development repository of XGBoost, one needs to use `git`.

Note: Use of Git submodules

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
```

For windows users who use github tools, you can open the git shell and type the following command:

```
git submodule init
git submodule update
```

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++11 (g++-5.0 or higher)
- CMake 3.14 or higher.

For a list of CMake options like GPU support, see `#-- Options` in CMakeLists.txt on top level of source tree.

Building on Linux and other UNIX-like systems

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

```
cd xgboost
mkdir build
cd build
cmake ..
make -j$(nproc)
```

Building on MacOS

Obtain `libomp` from Homebrew:

```
brew install libomp
```

Rest is the same as building on Linux.

Building on Windows

XGBoost support compilation with Microsoft Visual Studio and MinGW. To build with Visual Studio, we will need CMake. Make sure to install a recent version of CMake. Then run the following from the root of the XGBoost directory:

```
mkdir build
cd build
cmake .. -G"Visual Studio 14 2015 Win64"
# for VS15: cmake .. -G"Visual Studio 15 2017" -A x64
# for VS16: cmake .. -G"Visual Studio 16 2019" -A x64
cmake --build . --config Release
```

This specifies an out of source build using the Visual Studio 64 bit generator. (Change the `-G` option appropriately if you have a different version of Visual Studio installed.)

After the build process successfully ends, you will find a `xgboost.dll` library file inside `./lib/` folder. Some notes on using MinGW is added in *Building Python Package for Windows with MinGW-w64 (Advanced)*.

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 latests 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/`.

From the command line on Linux starting from the XGBoost directory:

```
mkdir build
cd build
# For CUDA toolkit >= 11.4, `BUILD_WITH_CUDA_CUB` is required.
cmake .. -DUSE_CUDA=ON -DBUILD_WITH_CUDA_CUB=ON
make -j4
```

Note: Specifying compute capability

To speed up compilation, the compute version specific to your GPU could be passed to cmake as, e.g., `-DGPU_COMPUTE_VER=50`. A quick explanation and numbers for some architectures can be found in this page.

Note: Faster distributed GPU training with NCCL

By default, distributed GPU training is enabled and uses Rabit for communication. For faster training, set the option `USE_NCCL=ON`. Faster distributed GPU training depends on NCCL2, available at this link. Since NCCL2 is only available for Linux machines, **faster distributed GPU training is available only for Linux**.

```
mkdir build
cd build
cmake .. -DUSE_CUDA=ON -DUSE_NCCL=ON -DNCCL_ROOT=/path/to/nccl2
make -j4
```

On Windows, run CMake as follows:

```
mkdir build
cd build
cmake .. -G"Visual Studio 14 2015 Win64" -DUSE_CUDA=ON
```

(Change the `-G` option appropriately if you have a different version of Visual Studio installed.)

The above cmake configuration run will create an `xgboost.sln` solution file in the build directory. Build this solution in release mode as a x64 build, either from Visual studio or from command line:

```
cmake --build . --target xgboost --config Release
```

To speed up compilation, run multiple jobs in parallel by appending option `-- /MP`.

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. Use Python setuptools directly

   The XGBoost Python package supports most of the setuptools commands, here is a list of tested commands:

   ```
   python setup.py install  # Install the XGBoost to your current Python
   ↪environment.
   python setup.py build    # Build the Python package.
   python setup.py build_ext # Build only the C++ core.
   python setup.py sdist     # Create a source distribution
   python setup.py bdist     # Create a binary distribution
   python setup.py bdist_wheel # Create a binary distribution with wheel format
   ```

   Running `python setup.py install` will compile XGBoost using default CMake flags. For passing additional compilation options, append the flags to the command. For example, to enable CUDA acceleration and NCCL (distributed GPU) support:

   ```
   python setup.py install --use-cuda --use-nccl
   ```

   Please refer to `setup.py` for a complete list of available options. Some other options used for development are only available for using CMake directly. See next section on how to use CMake with setuptools manually.

   You can install the created distribution packages using pip. For example, after running `sdist` setuptools command, a tar ball similar to `xgboost-1.0.0.tar.gz` will be created under the `dist` directory. Then you can install it by invoking the following command under `dist` directory:

   ```
   # under python-package directory
   cd dist
   pip install ./xgboost-1.0.0.tar.gz
   ```

   For details about these commands, please refer to the official document of setuptools, or just Google "how to install Python package from source". XGBoost Python package follows the general convention. Setuptools is usually available with your Python distribution, if not you can install it via system command. For example on Debian or Ubuntu:

   ```
   sudo apt-get install python-setuptools
   ```

   For cleaning up the directory after running above commands, `python setup.py clean` is not sufficient. After copying out the build result, simply running `git clean -xdf` under `python-package` is an efficient way to remove generated cache files. If you find weird behaviors in Python build or running linter, it might be caused by those cached files.

   For using develop command (editable installation), see next section.

```
python setup.py develop    # Create a editable installation.
pip install -e .           # Same as above, but carried out by pip.
```

2. Build C++ core with CMake first

   This is mostly for C++ developers who don't want to go through the hooks in Python setuptools. You can build C++ library directly using CMake as described in above sections. After compilation, a shared object (or called dynamic linked library, jargon depending on your platform) will appear in XGBoost's source tree under `lib/` directory. On Linux distributions it's `lib/libxgboost.so`. From there all Python setuptools commands will reuse that shared object instead of compiling it again. This is especially convenient if you are using the editable installation, where the installed package is simply a link to the source tree. We can perform rapid testing during development. Here is a simple bash script does that:

```
# Under xgboost source tree.
mkdir build
cd build
cmake ..
make -j$(nproc)
cd ../python-package
pip install -e .  # or equivalently python setup.py develop
```

3. Use `libxgboost.so` on system path.

   This is for distributing xgboost in a language independent manner, where `libxgboost.so` is separately packaged with Python package. Assuming *libxgboost.so* is already presented in system library path, which can be queried via:

```
import sys
import os
os.path.join(sys.prefix, 'lib')
```

   Then one only needs to provide an user option when installing Python package to reuse the shared object in system path:

```
cd xgboost/python-package
python setup.py install --use-system-libxgboost
```

Building Python Package for Windows with MinGW-w64 (Advanced)

Windows versions of Python are built with Microsoft Visual Studio. Usually Python binary modules are built with the same compiler the interpreter is built with. However, you may not be able to use Visual Studio, for following reasons:

1. VS is proprietary and commercial software. Microsoft provides a freeware "Community" edition, but its licensing terms impose restrictions as to where and how it can be used.

2. Visual Studio contains telemetry, as documented in Microsoft Visual Studio Licensing Terms. Running software with telemetry may be against the policy of your organization.

So you may want to build XGBoost with GCC own your own risk. This presents some difficulties because MSVC uses Microsoft runtime and MinGW-w64 uses own runtime, and the runtimes have different incompatible memory allocators. But in fact this setup is usable if you know how to deal with it. Here is some experience.

1. The Python interpreter will crash on exit if XGBoost was used. This is usually not a big issue.

2. `-O3` is OK.

3. `-mtune=native` is also OK.

4. Don't use `-march=native` gcc flag. Using it causes the Python interpreter to crash if the DLL was actually used.

5. You may need to provide the lib with the runtime libs. If `mingw32/bin` is not in `PATH`, build a wheel (`python setup.py bdist_wheel`), open it with an archiver and put the needed dlls to the directory where `xgboost.dll` is situated. Then you can install the wheel with `pip`.

1.2.4 Building R Package From Source

By default, the package installed by running `install.packages` is built from source. 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 .
```

But if you want to use CMake build for better performance (which has the logic for detecting available CPU instructions) or greater flexibility around compile flags, the above snippet can be replaced by:

```
mkdir build
cd build
cmake .. -DR_LIB=ON
make -j$(nproc)
make install
```

Installing the development version with Visual Studio (Windows)

On Windows, CMake with Visual C++ Build Tools (or Visual Studio) can be used to build the R package.

While not required, this build can be faster if you install the R package `processx` with `install.packages("processx")`.

Note: Setting correct PATH environment variable on Windows

If you are using Windows, make sure to include the right directories in the PATH environment variable.

- If you are using R 4.x with RTools 4.0: - `C:\rtools40\usr\bin` - `C:\rtools40\mingw64\bin`

- If you are using R 3.x with RTools 3.x:

    - `C:\Rtools\bin`

    - `C:\Rtools\mingw_64\bin`

Open the Command Prompt and navigate to the XGBoost directory, and then run the following commands. Make sure to specify the correct R version.

```
cd C:\path\to\xgboost
mkdir build
cd build
cmake .. -G"Visual Studio 16 2019" -A x64 -DR_LIB=ON -DR_VERSION=4.0.0
cmake --build . --target install --config Release
```

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:

```
mkdir build
cd build
cmake .. -DUSE_CUDA=ON -DR_LIB=ON
make 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`.

On Windows, CMake with Visual Studio has to be used to build an R package with GPU support. Rtools must also be installed.

Note: Setting correct PATH environment variable on Windows

If you are using Windows, make sure to include the right directories in the PATH environment variable.

- If you are using R 4.x with RTools 4.0:

    - `C:\rtools40\usr\bin`

    - `C:\rtools40\mingw64\bin`

- If you are using R 3.x with RTools 3.x:

    - `C:\Rtools\bin`

    - `C:\Rtools\mingw_64\bin`

Open the Command Prompt and navigate to the XGBoost directory, and then run the following commands. Make sure to specify the correct R version.

```
cd C:\path\to\xgboost
mkdir build
cd build
cmake .. -G"Visual Studio 16 2019" -A x64 -DUSE_CUDA=ON -DR_LIB=ON -DR_VERSION=4.0.0
cmake --build . --target install --config Release
```

If CMake can't find your R during the configuration step, you might provide the location of R to CMake like this: `-DLIBR_HOME="C:\Program Files\R\R-4.0.0"`.

If on Windows you get a "permission denied" error when trying to write to . . . Program Files/R/. . . during the package installation, create a `.Rprofile` file in your personal home directory (if you don't already have one in there), and add a line to it which specifies the location of your R packages user library, like the following:

```
.libPaths( unique(c("C:/Users/USERNAME/Documents/R/win-library/3.4", .libPaths())))
```

You might find the exact location by running `.libPaths()` in R GUI or RStudio.

1.2.5 Building JVM Packages

Building XGBoost4J using Maven requires Maven 3 or newer, Java 7+ and CMake 3.13+ for compiling Java code as well as the Java Native Interface (JNI) bindings.

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 2.3+**. 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.

Enabling OpenMP for Mac OS

If you are on Mac OS and using a compiler that supports OpenMP, you need to go to the file `xgboost/jvm-packages/create_jni.py` and comment out the line

```
CONFIG["USE_OPENMP"] = "OFF"
```

in order to get the benefit of multi-threading.

Building with GPU support

If you want to build XGBoost4J that supports distributed GPU training, run

```
mvn -Duse.cuda=ON install
```

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.

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

```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

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

1.3.4 Julia

```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

```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.

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 **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**:

$$\text{obj}(\theta) = L(\theta) + \Omega(\theta)$$
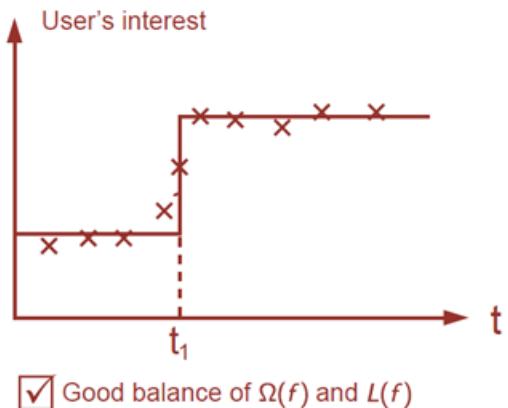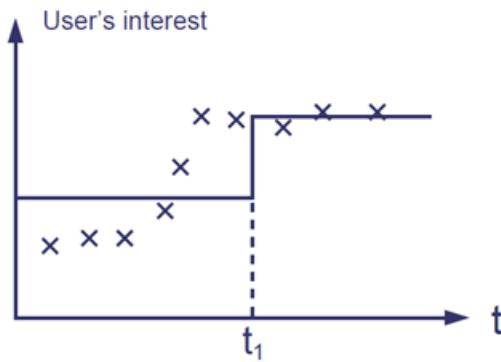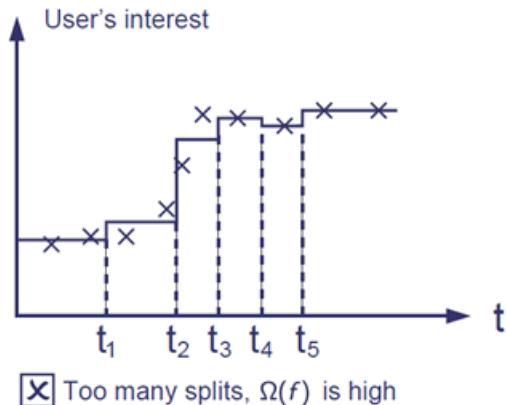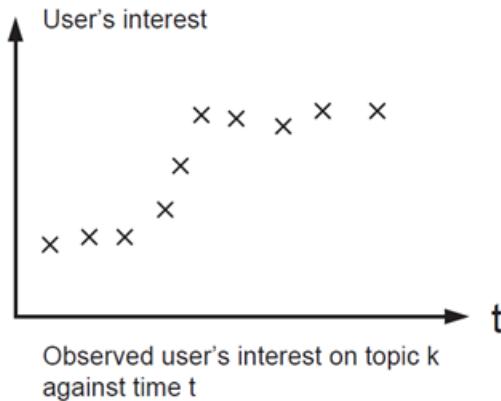
where L is the training loss function, and Ω is the regularization term. The training loss measures how *predictive* our model is with respect to the training data. A common choice of L is the *mean squared error*, which is given by

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

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?

Observed user's interest on topic k against time t

☒ Too many splits, $\Omega(f)$ is high

☒ Wrong split point, $L(f)$ is high

☑ Good balance of $\Omega(f)$ and $L(f)$
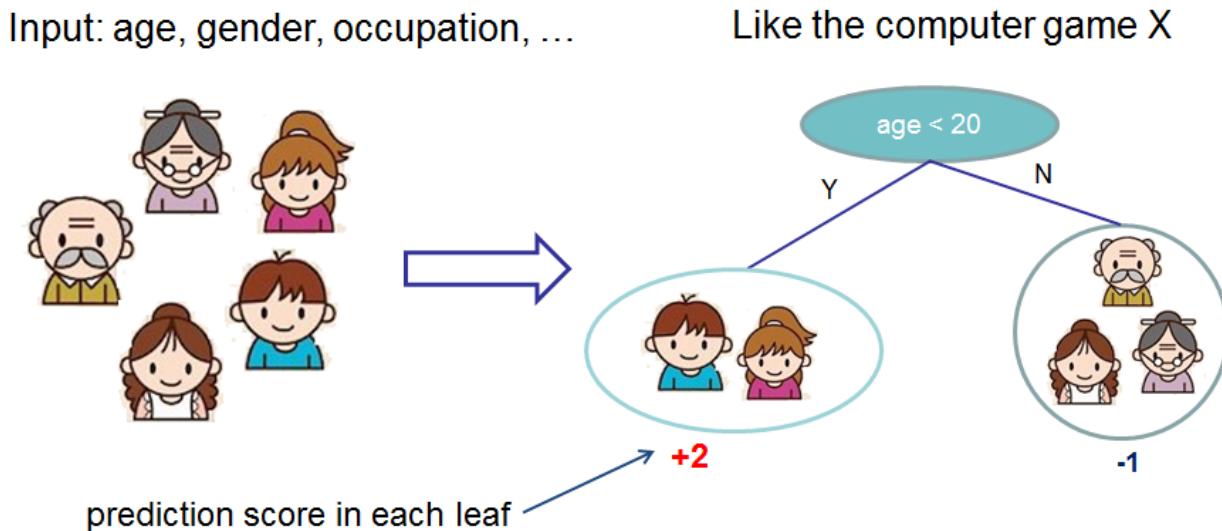
The correct answer is marked in red. Please consider if this visually seems a reasonable fit to you. The general principle is we want both a *simple* and *predictive* model. The tradeoff between the two is also referred as **bias-variance tradeoff** in machine learning.

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.
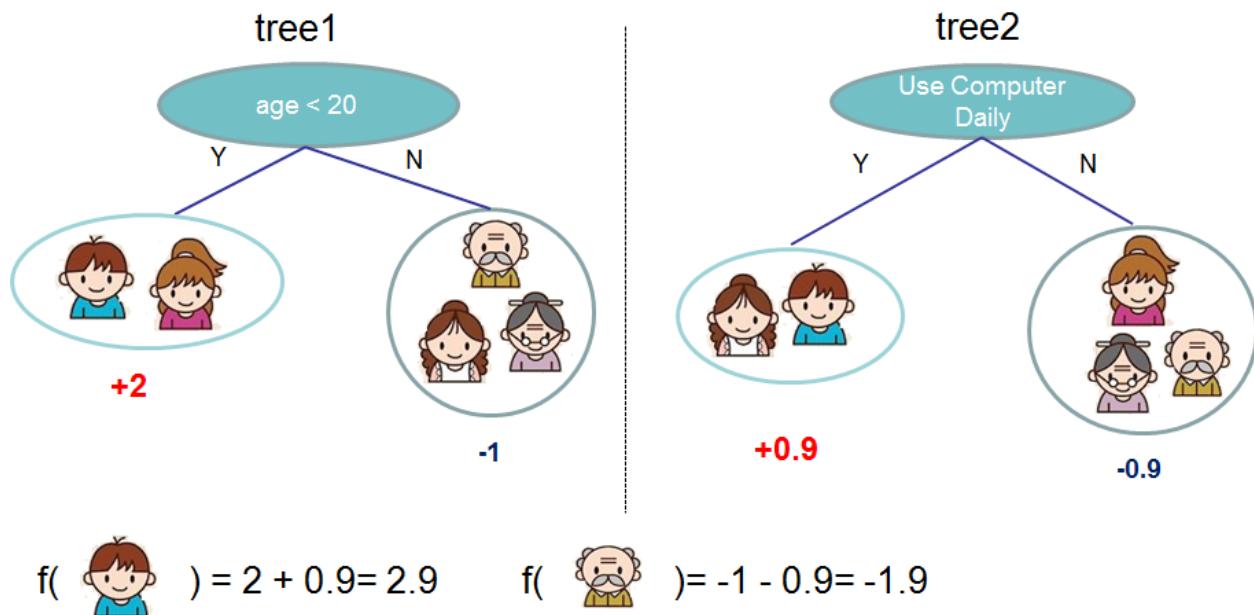
Decision Tree Ensembles

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):

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

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_i, 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

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

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

$$\text{obj}^{(t)} = \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t)}) + \sum_{i=1}^{t} \omega(f_i)$$
$$= \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \omega(f_t) + \text{constant}$$

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

$$\text{obj}^{(t)} = \sum_{i=1}^{n}(y_i - (\hat{y}_i^{(t-1)} + f_t(x_i)))^2 + \sum_{i=1}^{t}\omega(f_i)$$

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

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

$$g_i = \partial_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)})$$

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

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

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

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:

$$\text{obj}^{(t)} \approx \sum_{i=1}^{n} [g_i w_{q(x_i)} + \frac{1}{2} h_i w_{q(x_i)}^2] + \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$$

$$= \sum_{j=1}^{T} [(\sum_{i \in I_j} g_i) w_j + \frac{1}{2}(\sum_{i \in I_j} h_i + \lambda) w_j^2] + \gamma T$$
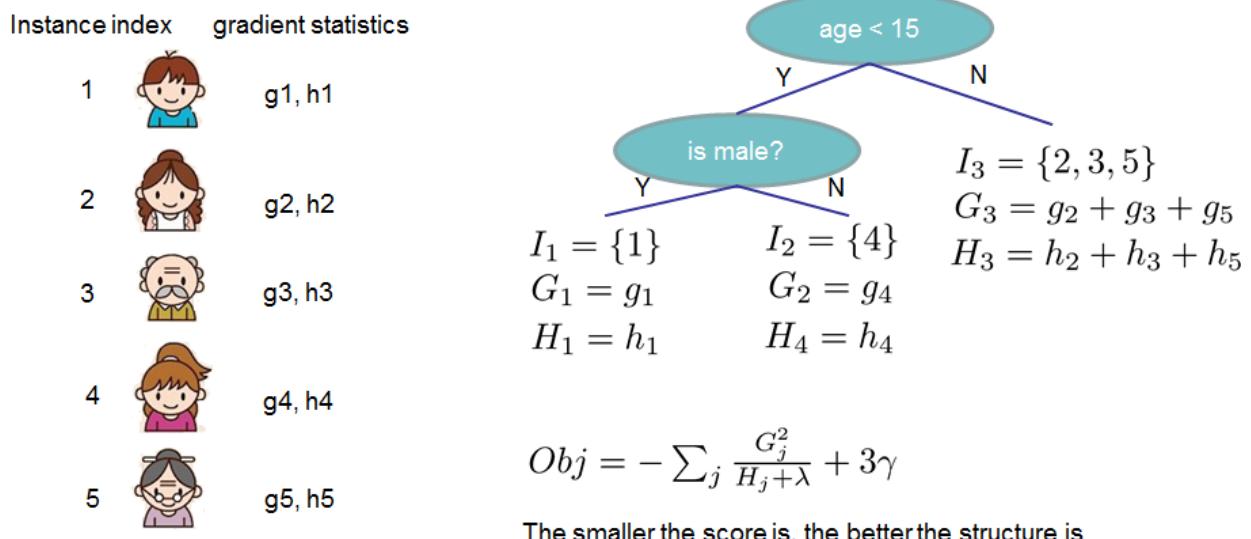
where $I_j = \{i | q(x_i) = j\}$ is the set of indices of data points assigned to the j-th leaf. Notice that in the second line we have changed the index of the summation because all the data points on the same leaf get the same score. We could further compress the expression by defining $G_j = \sum_{i \in I_j} g_i$ 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:

$$w_j^* = -\frac{G_j}{H_j + \lambda}$$

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

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

Instance index   gradient statistics

$I_1 = \{1\}$
$G_1 = g_1$
$H_1 = h_1$

$I_2 = \{4\}$
$G_2 = g_4$
$H_4 = h_4$

$I_3 = \{2, 3, 5\}$
$G_3 = g_2 + g_3 + g_5$
$H_3 = h_2 + h_3 + h_5$

$$Obj = -\sum_j \frac{G_j^2}{H_j + \lambda} + 3\gamma$$

The smaller the score is, the better the structure 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.
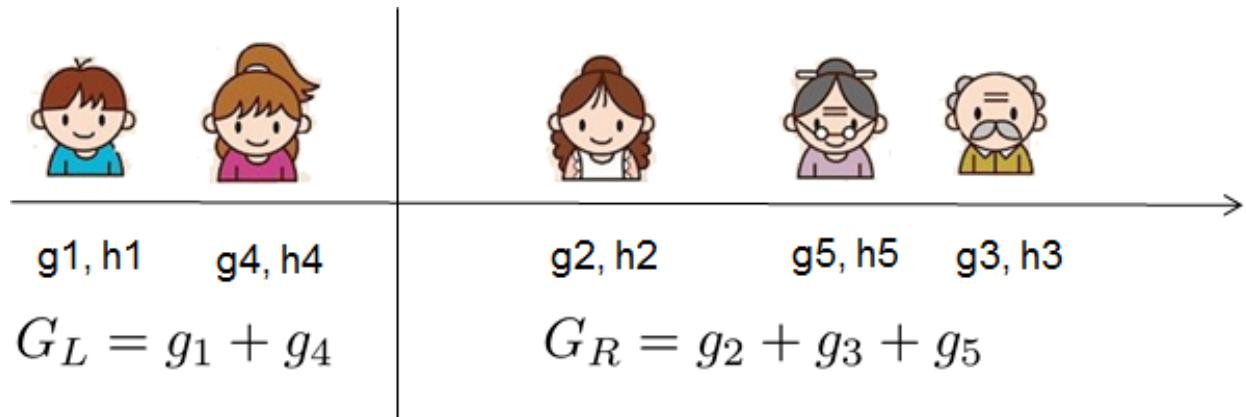
Learn the tree structure

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.

$g1, h1 \qquad g4, h4 \qquad\qquad g2, h2 \qquad\qquad g5, h5 \qquad g3, h3$

$$G_L = g_1 + g_4 \qquad\qquad G_R = g_2 + g_3 + g_5$$

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

In XGBoost 1.0.0, we introduced support of using JSON for saving/loading XGBoost models and related hyper-parameters for training, aiming to replace the 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 is added as an optimization for more efficient model IO. They 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 model, 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 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` in XGBoost). Users can share this model with others for prediction, 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`.

Other language bindings are still working in progress.

Note: The old binary format doesn't distinguish difference between model and raw memory serialisation format, it's a mix of everything, which is part of the reason why we want to replace it with a more robust serialisation method. JVM Package has its own memory based serialisation methods.

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

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

Listing 12: R

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

Listing 13: Scala

```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.

While for memory snapshot, UBJSON is the default starting with xgboost 1.6.

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 and metric functions as an extension. 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, the output from pickle is not a stable serialization format and doesn't work on different Python version nor XGBoost version, not to mention different language environments. Another way to workaround this limitation is to provide these functions again after the model is loaded. 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.

Loading pickled file from different version of XGBoost

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, export the model by calling *save_model*.

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:

```r
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:

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

or in R:

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

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

```json
{
  "Learner": {
    "generic_parameter": {
      "gpu_id": "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",
        "predictor": "gpu_predictor",
        "process_type": "default",
        "tree_method": "gpu_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)

```
            ...

            "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 Booster object, 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 or visualization, and is not supposed to be loaded back to XGBoost. The JSON version has a schema. See next section for more info.

JSON Schema

Another important feature of JSON format is a documented schema, based on which one can easily reuse the output model from XGBoost. Here is the JSON schema for the output model (not serialization, which will not be stable as noted above). For an example of parsing XGBoost tree model, see `/demo/json-model`. Please notice the "weight_drop" field used in "dart" booster. XGBoost does not scale tree leaf directly, instead it saves the weights as a separated array.

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "definitions": {
    "gbtree": {
      "type": "object",
      "properties": {
        "name": {
          "const": "gbtree"
```

```
      },
      "model": {
        "type": "object",
        "properties": {
          "gbtree_model_param": {
            "$ref": "#/definitions/gbtree_model_param"
          },
          "trees": {
            "type": "array",
            "items": {
              "type": "object",
              "properties": {
                "tree_param": {
                  "type": "object",
                  "properties": {
                    "num_nodes": {
                      "type": "string"
                    },
                    "size_leaf_vector": {
                      "type": "string"
                    },
                    "num_feature": {
                      "type": "string"
                    }
                  },
                  "required": [
                    "num_nodes",
                    "num_feature",
                    "size_leaf_vector"
                  ]
                },
                "id": {
                  "type": "integer"
                },
                "loss_changes": {
                  "type": "array",
                  "items": {
                    "type": "number"
                  }
                },
                "sum_hessian": {
                  "type": "array",
                  "items": {
                    "type": "number"
                  }
                },
                "base_weights": {
                  "type": "array",
                  "items": {
                    "type": "number"
                  }
                },
```

```json
        "left_children": {
          "type": "array",
          "items": {
            "type": "integer"
          }
        },
        "right_children": {
          "type": "array",
          "items": {
            "type": "integer"
          }
        },
        "parents": {
          "type": "array",
          "items": {
            "type": "integer"
          }
        },
        "split_indices": {
          "type": "array",
          "items": {
            "type": "integer"
          }
        },
        "split_conditions": {
          "type": "array",
          "items": {
            "type": "number"
          }
        },
        "split_type": {
          "type": "array",
          "items": {
            "type": "integer"
          }
        },
        "default_left": {
          "type": "array",
          "items": {
            "type": "integer"
          }
        },
        "categories": {
          "type": "array",
          "items": {
            "type": "integer"
          }
        },
        "categories_nodes": {
          "type": "array",
          "items": {
            "type": "integer"
```

```
              }
            },
            "categories_segments": {
              "type": "array",
              "items": {
                "type": "integer"
              }
            },
            "categories_sizes": {
              "type": "array",
              "items": {
                "type": "integer"
              }
            }
          },
          "required": [
            "tree_param",
            "loss_changes",
            "sum_hessian",
            "base_weights",
            "left_children",
            "right_children",
            "parents",
            "split_indices",
            "split_conditions",
            "default_left",
            "categories",
            "categories_nodes",
            "categories_segments",
            "categories_sizes"
          ]
        }
      },
      "tree_info": {
        "type": "array",
        "items": {
          "type": "integer"
        }
      }
    },
    "required": [
      "gbtree_model_param",
      "trees",
      "tree_info"
    ]
  }
},
"required": [
  "name",
  "model"
]
},
```

```
    "gbtree_model_param": {
      "type": "object",
      "properties": {
        "num_trees": {
          "type": "string"
        },
        "num_parallel_tree": {
          "type": "string"
        },
        "size_leaf_vector": {
          "type": "string"
        }
      },
      "required": [
        "num_trees",
        "size_leaf_vector"
      ]
    },
    "tree_param": {
      "type": "object",
      "properties": {
        "num_nodes": {
          "type": "string"
        },
        "size_leaf_vector": {
          "type": "string"
        },
        "num_feature": {
          "type": "string"
        }
      },
      "required": [
        "num_nodes",
        "num_feature",
        "size_leaf_vector"
      ]
    },
    "reg_loss_param": {
      "type": "object",
      "properties": {
        "scale_pos_weight": {
          "type": "string"
        }
      }
    },
    "pseudo_huber_param": {
      "type": "object",
      "properties": {
        "huber_slope": {
          "type": "string"
        }
      }
    }
```

```
    },
    "aft_loss_param": {
      "type": "object",
      "properties": {
        "aft_loss_distribution": {
          "type": "string"
        },
        "aft_loss_distribution_scale": {
          "type": "string"
        }
      }
    },
    "softmax_multiclass_param": {
      "type": "object",
      "properties": {
        "num_class": { "type": "string" }
      }
    },
    "lambda_rank_param": {
      "type": "object",
      "properties": {
        "num_pairsample": { "type": "string" },
        "fix_list_weight": { "type": "string" }
      }
    }
  }
},
"type": "object",
"properties": {
  "version": {
    "type": "array",
    "items": [
      {
        "type": "number",
        "minimum": 1
      },
      {
        "type": "number",
        "minimum": 0
      },
      {
        "type": "number",
        "minimum": 0
      }
    ],
    "minItems": 3,
    "maxItems": 3
  },
  "learner": {
    "type": "object",
    "properties": {
      "feature_names": {
        "type": "array",
```

```
        "items": {
            "type": "string"
        }
    },
    "feature_types": {
      "type": "array",
      "items": {
          "type": "string"
      }
    },
    "gradient_booster": {
      "oneOf": [
        {
          "$ref": "#/definitions/gbtree"
        },
        {
          "type": "object",
          "properties": {
            "name": { "const": "gblinear" },
            "model": {
              "type": "object",
              "properties": {
                "weights": {
                  "type": "array",
                  "items": {
                      "type": "number"
                  }
                }
              }
            }
          }
        },
        {
          "type": "object",
          "properties": {
            "name": { "const": "dart" },
            "gbtree": {
              "$ref": "#/definitions/gbtree"
            },
            "weight_drop": {
              "type": "array",
              "items": {
                  "type": "number"
              }
            }
          },
          "required": [
            "name",
            "gbtree",
            "weight_drop"
          ]
        }
```

```
      ]
    },

    "objective": {
      "oneOf": [
        {
          "type": "object",
          "properties": {
            "name": { "const": "reg:squarederror" },
            "reg_loss_param": { "$ref": "#/definitions/reg_loss_param"}
          },
          "required": [
            "name",
            "reg_loss_param"
          ]
        },
        {
          "type": "object",
          "properties": {
            "name": { "const": "reg:pseudohubererror" },
            "reg_loss_param": { "$ref": "#/definitions/reg_loss_param"}
          },
          "required": [
            "name",
            "reg_loss_param"
          ]
        },
        {
          "type": "object",
          "properties": {
            "name": { "const": "reg:squaredlogerror" },
            "reg_loss_param": { "$ref": "#/definitions/reg_loss_param"}
          },
          "required": [
            "name",
            "reg_loss_param"
          ]
        },
        {
          "type": "object",
          "properties": {
            "name": { "const": "reg:linear" },
            "reg_loss_param": { "$ref": "#/definitions/reg_loss_param"}
          },
          "required": [
            "name",
            "reg_loss_param"
          ]
        },
        {
          "type": "object",
          "properties": {
```

```
      "name": { "const": "reg:logistic" },
      "reg_loss_param": { "$ref": "#/definitions/reg_loss_param"}
    },
    "required": [
      "name",
      "reg_loss_param"
    ]
  },
  {
    "type": "object",
    "properties": {
      "name": { "const": "binary:logistic" },
      "reg_loss_param": { "$ref": "#/definitions/reg_loss_param"}
    },
    "required": [
      "name",
      "reg_loss_param"
    ]
  },
  {
    "type": "object",
    "properties": {
      "name": { "const": "binary:logitraw" },
      "reg_loss_param": { "$ref": "#/definitions/reg_loss_param"}
    },
    "required": [
      "name",
      "reg_loss_param"
    ]
  },
  {
    "type": "object",
    "properties": {
      "name": { "const": "count:poisson" },
      "poisson_regression_param": {
        "type": "object",
        "properties": {
          "max_delta_step": { "type": "string" }
        }
      }
    },
    "required": [
      "name",
      "poisson_regression_param"
    ]
  },
  {
    "type": "object",
    "properties": {
      "name": { "const": "reg:tweedie" },
      "tweedie_regression_param": {
        "type": "object",
```

```
            "properties": {
              "tweedie_variance_power": { "type": "string" }
            }
          }
        },
        "required": [
          "name",
          "tweedie_regression_param"
        ]
      },
      {
        "properties": {
          "name": {
            "const": "reg:absoluteerror"
          }
        },
        "type": "object"
      },
      {
        "type": "object",
        "properties": {
          "name": { "const": "survival:cox" }
        },
        "required": [ "name" ]
      },
      {
        "type": "object",
        "properties": {
          "name": { "const": "reg:gamma" }
        },
        "required": [ "name" ]
      },

      {
        "type": "object",
        "properties": {
          "name": { "const": "multi:softprob" },
          "softmax_multiclass_param": { "$ref": "#/definitions/softmax_multiclass_
→param"}
        },
        "required": [
          "name",
          "softmax_multiclass_param"
        ]
      },
      {
        "type": "object",
        "properties": {
          "name": { "const": "multi:softmax" },
          "softmax_multiclass_param": { "$ref": "#/definitions/softmax_multiclass_
→param"}
        },
```

```
        "required": [
          "name",
          "softmax_multiclass_param"
        ]
      },

      {
        "type": "object",
        "properties": {
          "name": { "const": "rank:pairwise" },
          "lambda_rank_param": { "$ref": "#/definitions/lambda_rank_param"}
        },
        "required": [
          "name",
          "lambda_rank_param"
        ]
      },
      {
        "type": "object",
        "properties": {
          "name": { "const": "rank:ndcg" },
          "lambda_rank_param": { "$ref": "#/definitions/lambda_rank_param"}
        },
        "required": [
          "name",
          "lambda_rank_param"
        ]
      },
      {
        "type": "object",
        "properties": {
          "name": { "const": "rank:map" },
          "lambda_rank_param": { "$ref": "#/definitions/lambda_rank_param"}
        },
        "required": [
          "name",
          "lambda_rank_param"
        ]
      },
      {
        "type": "object",
        "properties": {
          "name": {"const": "survival:aft"},
          "aft_loss_param": { "$ref": "#/definitions/aft_loss_param"}
        }
      },
      {
        "type": "object",
        "properties": {
          "name": {"const": "binary:hinge"}
        }
      }
```

```
        ]
      },

      "learner_model_param": {
        "type": "object",
        "properties": {
          "base_score": { "type": "string" },
          "num_class": { "type": "string" },
          "num_feature": { "type": "string" },
          "num_target": { "type": "string" }
        }
      }
    },
    "required": [
      "gradient_booster",
      "objective"
    ]
  }
},
"required": [
  "version",
  "learner"
]
}
```

1.4.3 Distributed XGBoost YARN on AWS

[This page is under construction.]

Note: XGBoost with Spark

If you are preprocessing training data with Spark, consider using *XGBoost4J-Spark*.

1.4.4 Distributed XGBoost on Kubernetes

Distributed XGBoost training and batch prediction on Kubernetes are supported via Kubeflow XGBoost Training Operator.

Instructions

In order to run a XGBoost job in a Kubernetes cluster, perform the following steps:

1. Install XGBoost Operator on the Kubernetes cluster.

    a. XGBoost Operator is designed to manage the scheduling and monitoring of XGBoost jobs. Follow this installation guide to install XGBoost Operator.

2. Write application code that will be executed by the XGBoost Operator.

    a. To use XGBoost Operator, you'll have to write a couple of Python scripts that implement the distributed training logic for XGBoost. Please refer to the Iris classification example.

b. Data reader/writer: you need to implement the data reader and writer based on the specific requirements of your chosen data source. For example, if your dataset is stored in a Hive table, you have to write the code to read from or write to the Hive table based on the index of the worker.

c. Model persistence: in the Iris classification example, the model is stored in Alibaba OSS. If you want to store your model in other storages such as Amazon S3 or Google NFS, you'll need to implement the model persistence logic based on the requirements of the chosen storage system.

3. Configure the XGBoost job using a YAML file.

a. YAML file is used to configure the computational resources and environment for your XGBoost job to run, e.g. the number of workers/masters and the number of CPU/GPUs. Please refer to this YAML template for an example.

4. Submit XGBoost job to a Kubernetes cluster.

a. Use kubectl to submit a distributed XGBoost job as illustrated here.

Support

Please submit an issue on XGBoost Operator repo for any feature requests or problems.

1.4.5 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.

Contents

- *Requirements*
- *Overview*
- *Running prediction*
- *Scikit-Learn interface*
- *Working with other clusters*
- *Threads*
- *Working with asyncio*
- *Evaluation and Early Stopping*
- *Other customization*
- *Troubleshooting*
- *IPv6 Support*
- *Why is the initialization of* `DaskDMatrix` *so slow and throws weird errors*
- *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:

```python
import xgboost as xgb
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
    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 = xgb.dask.DaskDMatrix(client, X, y)

    output = xgb.dask.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 guared 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:

```python
{'booster': Booster,
 'history': dict}
```

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

```
prediction = xgb.dask.predict(client, output, dtrain)
# Or equivalently, pass ``output['booster']``:
prediction = xgb.dask.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 = xgb.dask.predict(client, output, X)
# Use inplace version.
prediction = xgb.dask.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 = xgb.dask.predict(client, booster, dtrain)
```

or equivalently:

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

Also for inplace prediction:

```
booster.set_param({'predictor': 'gpu_predictor'})
# where X is a dask DataFrame or dask Array containing cupy or cuDF backed data.
prediction = xgb.dask.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 = xgb.dask.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 = xgb.dask.DaskXGBClassifier()
cls.fit(X, y)

booster = cls.get_booster()
```

Scikit-Learn 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
import xgboost as xgb


def main(client: Client) -> None:
    X, y = load_data()
    clf = xgb.dask.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)
```

Working with other clusters

LocalCluster is mostly used for testing. In real world applications some other clusters might be preferred. Examples are like LocalCUDACluster for single node multi-GPU instance, manually launched cluster by using command line utilities like dask-worker from distributed for not yet automated environments. Some special clusters like KubeCluster from dask-kubernetes package are also possible. The dask API in xgboost is orthogonal to the cluster type and can be used with any of them. A typical testing workflow with KubeCluster looks like this:

```python
from dask_kubernetes import KubeCluster  # Need to install the ``dask-kubernetes`` package
from dask.distributed import Client
import xgboost as xgb
import dask
import dask.array as da

dask.config.set({"kubernetes.scheduler-service-type": "LoadBalancer",
                 "kubernetes.scheduler-service-wait-timeout": 360,
                 "distributed.comm.timeouts.connect": 360})


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 youself.  See document in dask_kubernetes for
    # its usage.  Here we just want to show that XGBoost works on various clusters.
    cluster = KubeCluster.from_yaml('worker-spec.yaml', deploy_mode='remote')
    cluster.scale(kWorkers)      # scale to use all GPUs

    with Client(cluster) as client:
        X = da.random.random(size=(m, n), chunks=100)
        y = da.random.random(size=(m, ), chunks=100)

        regressor = xgb.dask.DaskXGBRegressor(n_estimators=10, missing=0.0)
        regressor.client = client
        regressor.set_params(tree_method='gpu_hist')
        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()
```

However, these clusters might have their 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.

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 = xgb.dask.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

New 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.

Functional interface:

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

    with_m = await xgb.dask.predict(client, output, m)
    with_X = await xgb.dask.predict(client, output, X)
    inplace = await xgb.dask.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 xgb.dask.DaskXGBRegressor(verbosity=1, n_estimators=2)
    regressor.set_params(tree_method='hist')  # trivial method, synchronous operation
    regressor.client = client  # accessing attribute, synchronous operation
```

(continues on next page)

```
regressor = await regressor.fit(X, y, eval_set=[(X, y)])
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

New 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
import xgboost as xgb


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 = xgb.dask.DaskDMatrix(
    client=client,
    data=data,
    label=labels
)

dvalid = xgb.dask.DaskDMatrix(
    client=client,
    data=X_eval,
```

```
    label=y_eval
)

result = xgb.dask.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 `xgb.dask.train()` in this way, the model object returned by `xgb.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 = xgb.dask.train(
    client,
```

```
    params={
        "objective": "binary:logistic",
        "eval_metric": ["error", "rmse"],
        "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],
)
```

Troubleshooting

New in version 1.6.0.

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 dask config is used:

```python
import dask
from distributed import Client
from xgboost import dask as dxgb
# let xgboost know the scheduler address
dask.config.set({"xgboost.scheduler_address": "192.0.0.100"})

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

# or we can specify the port too
with dask.config.set({"xgboost.scheduler_address": "192.0.0.100:12345"}):
    reg = dxgb.DaskXGBRegressor()
```

IPv6 Support

New 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.

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.

Memory Usage

Here are some pratices 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

1.4.6 Distributed XGBoost with PySpark

Starting from version 1.7.0, xgboost supports pyspark estimator APIs.

Note: The feature is still experimental and not yet ready for production use.

- *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*

XGBoost PySpark Estimator

SparkXGBRegressor

SparkXGBRegressor is a PySpark ML estimator. It implements the XGBoost classification algorithm based on XG-Boost 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:

```python
from xgboost.spark import SparkXGBRegressor
spark_reg_estimator = 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`, `use_gpu`, 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.

```python
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.

```python
transformed_test_spark_dataframe = xgb_regressor.predict(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 PySpark standalone GPU cluster. To get started, first we need to install some additional packages, then we can set the `use_gpu` parameter to `True`.

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://rapids.ai/pip.html#install
pip install cudf-cu11 --extra-index-url=https://pypi.ngc.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 and the additional parameter `use_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)

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]
```

(continues on next page)

```python
# create a xgboost pyspark regressor estimator and set use_gpu=True
regressor = SparkXGBRegressor(
  features_col=feature_names,
  label_col=label_name,
  num_workers=2,
  use_gpu=True,
)


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


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

Submit the PySpark application

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

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

spark-submit \
  --master spark://<master-ip>:7077 \
  --conf spark.executor.resource.gpu.amount=1 \
  --conf spark.task.resource.gpu.amount=1 \
  --archives xgboost_env.tar.gz#environment \
  xgboost_app.py
```

The submit command sends the Python environment created by pip or conda along with the specification of GPU allocation. We will revisit this command later on.

Model Persistence

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

```python
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:

```python
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 any Python code change. 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.resource.gpu.amount=1 \
  --conf spark.task.resource.gpu.amount=1 \
  --packages com.nvidia:rapids-4-spark_2.12:22.08.0 \
  --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.

1.4.7 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 *ray_params* 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.

```python
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 *RayParams* object.

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

Simple prediction example

```python
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)

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.

class xgboost_ray.**RayParams**(*num_actors=0*, *cpus_per_actor=0*, *gpus_per_actor=-1*, *resources_per_actor=None*, *elastic_training=False*, *max_failed_actors=0*, *max_actor_restarts=0*, *checkpoint_frequency=5*, *distributed_callbacks=None*, *verbose=None*, *placement_options=None*)

> Parameters to configure Ray-specific behavior.

> > **Parameters**

> > > - **num_actors** (*int*) – Number of parallel Ray actors.
> > >
> > > - **cpus_per_actor** (*int*) – Number of CPUs to be used per Ray actor.
> > >
> > > - **gpus_per_actor** (*int*) – Number of GPUs to be used per Ray actor.
> > >
> > > - **resources_per_actor** (*Optional[Dict]*) – Dict of additional resources required per Ray actor.
> > >
> > > - **elastic_training** (*bool*) – If True, training will continue with fewer actors if an actor fails. Default False.
> > >
> > > - **max_failed_actors** (*int*) – If *elastic_training* is True, this specifies the maximum number of failed actors with which we still continue training.
> > >
> > > - **max_actor_restarts** (*int*) – Number of retries when Ray actors fail. Defaults to 0 (no retries). Set to -1 for unlimited retries.
> > >
> > > - **checkpoint_frequency** (*int*) – How often to save checkpoints. Defaults to 5 (every 5th iteration).
> > >
> > > - **verbose** (*bool*) – Whether to output Ray-specific info messages during training/prediction.

- **placement_options** (`dict`) – Optional kwargs to pass to `PlacementGroupFactory` in `get_tune_resources()`.

- **distributed_callbacks** (`Optional[List[DistributedCallback]]`) –

PublicAPI (beta): This API is in beta and may change before becoming stable.

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 taining.

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 *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 *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",
    "eval_metric": ["logloss", "error"],
```

(continues on next page)

```
    "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

1.4.8 DART booster

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 new tree booster `dart`.

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:

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

- 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

The booster `dart` inherits `gbtree` booster, so it supports all parameters that `gbtree` does, such as `eta`, `gamma`, `max_depth` etc.

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.

$$a\left(\sum_{i \in \mathbf{K}} F_i + \frac{1}{k}F_m\right) = a\left(\sum_{i \in \mathbf{K}} F_i + \frac{\eta}{k}\tilde{F}_m\right)$$

$$\sim a\left(1 + \frac{\eta}{k}\right)D$$

$$= a\frac{k+\eta}{k}D = D,$$

$$a = \frac{k}{k+\eta}$$

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

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

- `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')
dtest = xgb.DMatrix('demo/data/agaricus.txt.test')
# specify parameters via map
param = {'booster': 'dart',
         '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
bst = xgb.train(param, dtrain, num_round)
preds = bst.predict(dtest)
```

1.4.9 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:

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

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

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

whenever $x \le x'$ is a **decreasing constraint**.
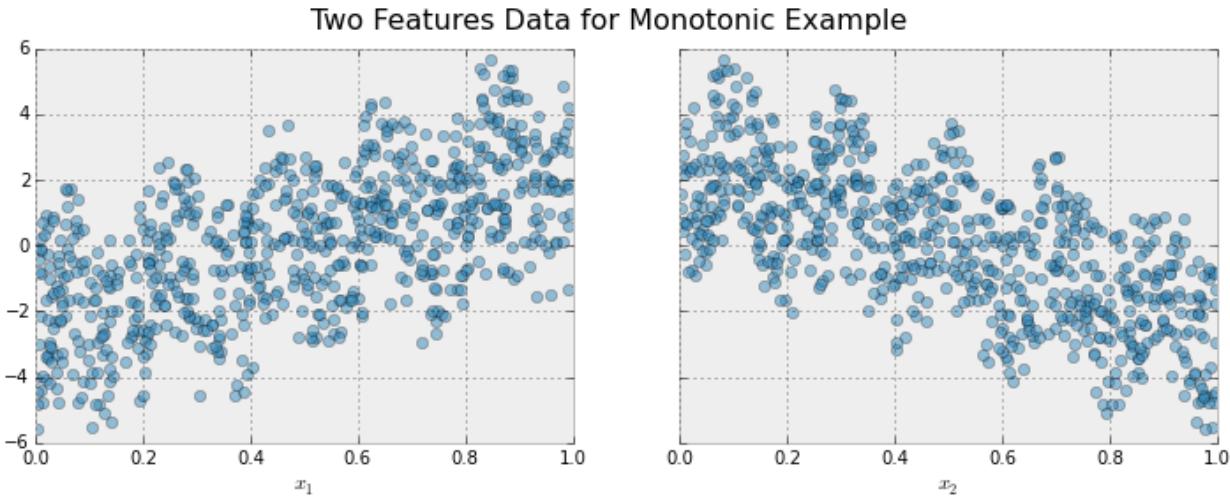
XGBoost has the ability to enforce monotonicity constraints on any features used in a boosted model.

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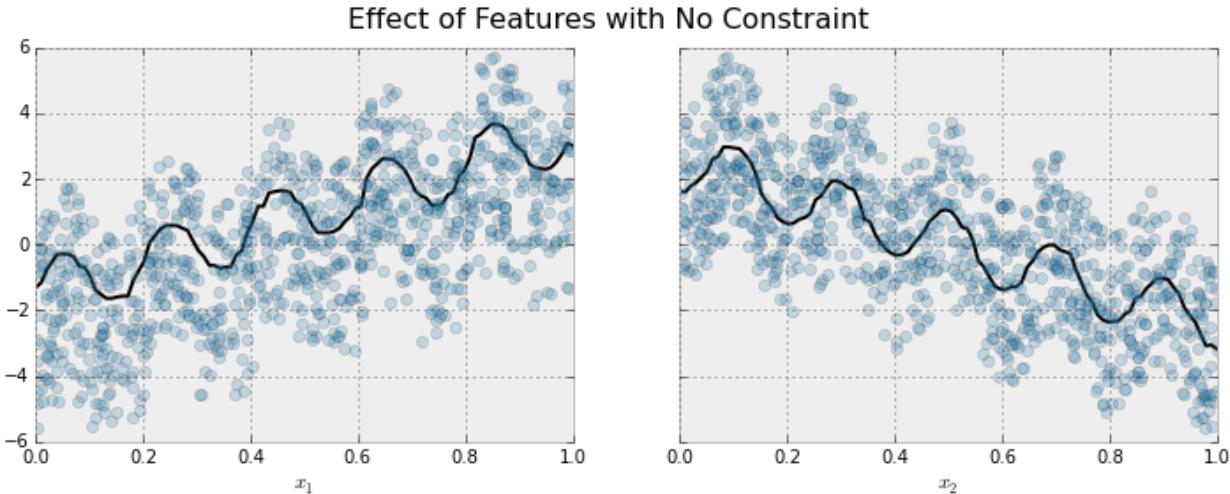
To illustrate, let's create some simulated data with two features and a response according to the following scheme

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

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.
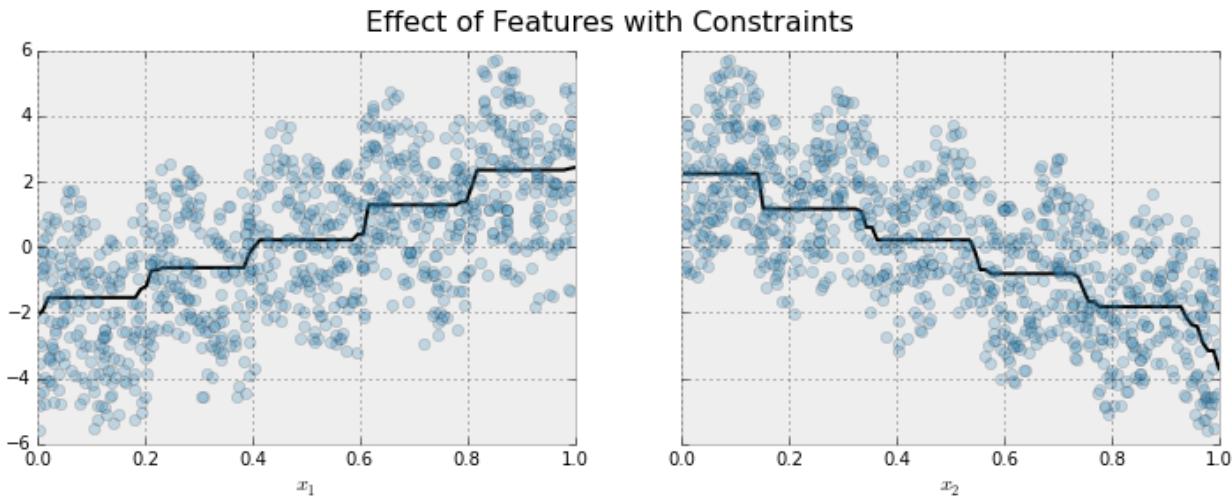

Let's fit a boosted tree model to this data without imposing any monotonic constraints:

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:

Effect of Features with 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 for the 'hist' tree construction algorithm. If `tree_method` is set to either `hist`, `approx` or `gpu_hist`, 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 package supports using feature names instead of feature index for specifying the constraints. Given a data frame with columns `["f0", "f1", "f2"]`, the monotonic constraint can be specified as `{"f0": 1, "f2": -1}`, and `"f1"` will default to `0` (no constraint).

1.4.10 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,
  'num_parallel_tree': 100,
  'objective': 'binary:logistic',
  'subsample': 0.8,
  'tree_method': 'gpu_hist'
}
```

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:

```python
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.

1.4.11 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 package supports 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"]].

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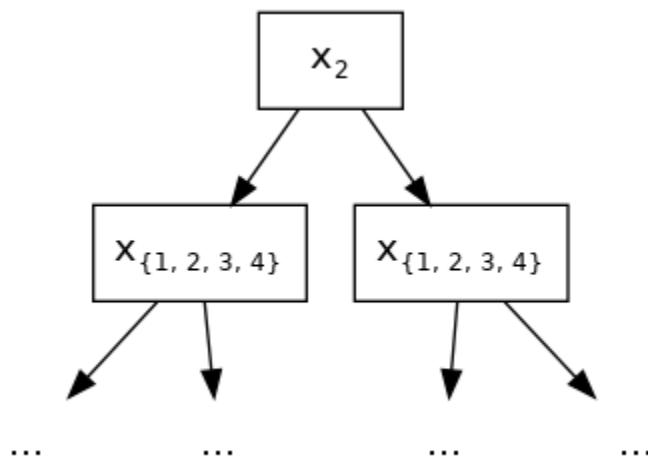
Fig. 1: {1, 2, 3, 4} represents the sets of legitimate split features.

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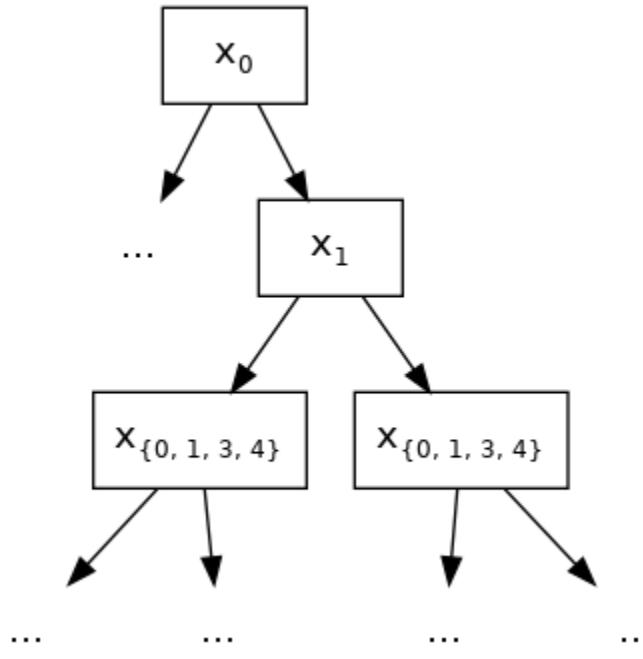
Fig. 2: {0, 1, 3, 4} represents the sets of legitimate split features.

1.4.12 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.

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

```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]])
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

```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

```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

```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	$\dfrac{\exp\left(-z^2/2\right)}{\sqrt{2\pi}}$
logistic	$\dfrac{e^z}{(1+e^z)^2}$
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*

1.4.13 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 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
mkdir build
cd build
# 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 .. -DCMAKE_INSTALL_PREFIX=$CONDA_PREFIX
# install XGBoost in your conda environment (usually under [your home directory]/
↪miniconda3)
make 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 3.13)
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.

```
# Nagivate to the build directory for your application
cd build
# Activate the Conda environment where we previously installed XGBoost
```

(continues on next page)

```
conda activate [env_name]
# Invoke CMake with CMAKE_PREFIX_PATH
cmake .. -DCMAKE_PREFIX_PATH=$CONDA_PREFIX
# Build your application
make
```

Usefull 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:

```
1  #define safe_xgboost(call) {  \
2    int err = (call); \
3    if (err != 0) { \
4      fprintf(stderr, "%s:%d: error in %s: %s\n", __FILE__, __LINE__, #call,␣
   →XGBGetLastError());  \
5      exit(1); \
6    } \
7  }
```

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.

```
1  #define safe_xgboost(call) {  \
2    int err = (call); \
3    if (err != 0) { \
4      throw new Exception(std::string(__FILE__) + ":" + std::to_string(__LINE__) + \
5                          ": error in " + #call + ":" + XGBGetLastError()));  \
6    } \
7  }
```

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:

```
1  #include <assert.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <xgboost/c_api.h>
5
```

(continued from previous page)

```c
6   int main(int argc, char** argv) {
7     int silent = 0;
8
9     BoosterHandle booster;
10
11    // do something with booster
12
13    //free the memory
14    XGBoosterFree(booster)
15
16    DMatrixHandle DMatrixHandle_param;
17
18    // do something with DMatrixHandle_param
19
20    // free the memory
21    XGDMatrixFree(DMatrixHandle_param);
22
23    return 0;
24  }
```

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 paramter value can be of any data type (e.g. int, char, float, double, etc), but they should always be encoded as strings.

```c
BoosterHandle booster;
XGBoosterSetParam(booster, "paramter_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()*

```c
DMatrixHandle data; // handle to DMatrix
// Load the dat 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()*

```c
1   // 1D matrix
2   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, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0 };
3
4   // 2D matrix
5   const int ROWS = 5, COLS = 3;
6   const int data2[ROWS][COLS] = { {1, 2, 3}, {2, 4, 6}, {3, -1, 9}, {4, 8, -1}, {2, 5, 1},
    ↪{0, 1, 5} };
7   DMatrixHandle dmatrix1, dmatrix2;
8   // Pass the matrix, no of rows & columns contained in the matrix variable
```

(continues on next page)

```
9   // here '0' represents the missing value in the matrix dataset
10  // dmatrix variable will contain the created DMatrix using it
11  safe_xgboost(XGDMatrixCreateFromMat(data1, 1, 50, 0, &dmatrix));
12  // here -1 represents the missing value in the matrix dataset
13  safe_xgboost(XGDMatrixCreateFromMat(data2, ROWS, COLS, -1, &dmatrix2));
```

3. Create a Booster object for training & testing on dataset using *XGBoosterCreate()*

```
1   BoosterHandle booster;
2   const int eval_dmats_size;
3   // We assume that training and test data have been loaded into 'train' and 'test'
4   DMatrixHandle eval_dmats[eval_dmats_size] = {train, test};
5   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()*.

```
1   const int ROWS=5, COLS=3;
2   const int data[ROWS][COLS] = { {1, 2, 3}, {2, 4, 6}, {3, -1, 9}, {4, 8, -1}, {2, 5, 1},
    →{0, 1, 5} };
3   DMatrixHandle dmatrix;
4
5   safe_xgboost(XGDMatrixCreateFromMat(data, ROWS, COLS, -1, &dmatrix));
6
7   // variable to store labels for the dataset created from above matrix
8   float labels[ROWS];
9
10  for (int i = 0; i < ROWS; i++) {
11    labels[i] = i;
12  }
13
14  // Loading the labels
15  safe_xgboost(XGDMatrixSetFloatInfo(dmatrix, "label", labels, ROWS));
16
17  // reading the labels and store the length of the result
18  bst_ulong result_len;
19
20  // labels result
21  const float *result;
22
23  safe_xgboost(XGDMatrixGetFloatInfo(dmatrix, "label", &result_len, &result));
24
25  for(unsigned int i = 0; i < result_len; i++) {
26    printf("label[%i] = %f\n", i, result[i]);
27  }
```

5. Set the parameters for the `Booster` object according to the requirement using *XGBoosterSetParam()* . Check out the full list of parameters available *here* .

```
1   BoosterHandle booster;
2   safe_xgboost(XGBoosterSetParam(booster, "booster", "gblinear"));
3   // default max_depth =6
4   safe_xgboost(XGBoosterSetParam(booster, "max_depth", "3"));
```

```
5  // default eta  = 0.3
6  safe_xgboost(XGBoosterSetParam(booster, "eta", "0.1"));
```

6. Train & evaluate the model using *XGBoosterUpdateOneIter()* and *XGBoosterEvalOneIter()* respectively.

```
1   int num_of_iterations = 20;
2   const char* eval_names[eval_dmats_size] = {"train", "test"};
3   const char* eval_result = NULL;
4
5   for (int i = 0; i < num_of_iterations; ++i) {
6     // Update the model performance for each iteration
7     safe_xgboost(XGBoosterUpdateOneIter(booster, i, train));
8
9     // Give the statistics for the learner for training & testing dataset in terms of␣
    ↪error after each iteration
10     safe_xgboost(XGBoosterEvalOneIter(booster, i, eval_dmats, eval_names, eval_dmats_size,␣
    ↪&eval_result));
11     printf("%s\n", eval_result);
12  }
```

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()*

```
1   char const config[] =
2       "{\"training\": false, \"type\": 0, "
3       "\"iteration_begin\": 0, \"iteration_end\": 0, \"strict_shape\": false}";
4   /* Shape of output prediction */
5   uint64_t const* out_shape;
6   /* Dimension of output prediction */
7   uint64_t out_dim;
8   /* Pointer to a thread local contigious array, assigned in prediction function. */
9   float const* out_result = NULL;
10  safe_xgboost(
11      XGBoosterPredictFromDMatrix(booster, dmatrix, config, &out_shape, &out_dim, &out_
    ↪result));
12
13  for (unsigned int i = 0; i < output_length; i++){
14    printf("prediction[%i] = %f \n", i, output_result[i]);
15  }
```

8. Get the number of features in your dataset using *XGBoosterGetNumFeature()*.

```
1   bst_ulong num_of_features = 0;
2
3   // Assuming booster variable of type BoosterHandle is already declared
4   // and dataset is loaded and trained on booster
5   // storing the results in num_of_features variable
6   safe_xgboost(XGBoosterGetNumFeature(booster, &num_of_features));
7
```

```
8  // Printing number of features by type conversion of num_of_features variable from bst_
   ↪ulong to unsigned long
9  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()*

```
1  BoosterHandle booster;
2  const char *model_path = "/path/of/model.json";
3
4  // create booster handle first
5  safe_xgboost(XGBoosterCreate(NULL, 0, &booster));
6
7  // set the model parameters here
8
9  // load model
10 safe_xgboost(XGBoosterLoadModel(booster, model_path));
11
12 // 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));
```

1.4.14 Text Input Format of DMatrix

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`. 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.

1.4.15 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` and `gamma`.
- The second way is to add randomness to make training robust to noise.
    - This includes `subsample` and `colsample_bytree`.
    - You can also reduce stepsize `eta`. Remember to increase `num_round` when you do so.

Faster training performance

There's a parameter called `tree_method`, set it to `hist` or `gpu_hist` for faster computation.

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

1.4.16 Using XGBoost External Memory Version

XGBoost supports loading data from external memory using builtin data parser. And starting from version 1.5, users can also define a custom iterator to load data in chunks. The feature is still experimental and not yet ready for production use. In this tutorial we will introduce both methods. Please note that training on data from external memory is not supported by `exact` tree method.

Data Iterator

Starting from XGBoost 1.5, users can define their own data loader using Python or C interface. There are some examples in the `demo` directory for quick start. This is a generalized version of text input external memory, where users no longer need to prepare a text file that XGBoost recognizes. To enable the feature, user need to define a data iterator with 2 class methods `next` and `reset` then pass it into `DMatrix` constructor.

```python
import os
from typing import List, Callable
import xgboost
from sklearn.datasets import load_svmlight_file

class Iterator(xgboost.DataIter):
  def __init__(self, svm_file_paths: List[str]):
    self._file_paths = svm_file_paths
    self._it = 0
    # XGBoost will generate some cache files under current directory with the prefix
    # "cache"
    super().__init__(cache_prefix=os.path.join(".", "cache"))

  def next(self, input_data: Callable):
    """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):
```

```python
        # return 0 to let XGBoost know this is the end of iteration
        return 0

    # input_data is a function passed in by XGBoost who has the exact same signature of
    # ``DMatrix``
    X, y = load_svmlight_file(self._file_paths[self._it])
    input_data(X, y)
    self._it += 1
    # Return 1 to let XGBoost know we haven't seen all the files yet.
    return 1

  def reset(self):
    """Reset the iterator to its beginning"""
    self._it = 0

it = Iterator(["file_0.svm", "file_1.svm", "file_2.svm"])
Xy = xgboost.DMatrix(it)

# Other tree methods including ``hist`` and ``gpu_hist`` also work, but has some caveats
# as noted in following sections.
booster = xgboost.train({"tree_method": "approx"}, Xy)
```

The above snippet is a simplifed version of `demo/guide-python/external_memory.py`. For an example in C, please see `demo/c-api/external-memory/`.

Text File Inputs

There is no big difference between using external memory version and in-memory version. The only difference is the filename format.

The external memory version takes in the following URI format:

```
filename#cacheprefix
```

The `filename` is the normal path to LIBSVM format file you want to load in, and `cacheprefix` is a path to a cache file that XGBoost will use for caching preprocessed data in binary form.

To load from csv files, use the following syntax:

```
filename.csv?format=csv&label_column=0#cacheprefix
```

where `label_column` should point to the csv column acting as the label.

To provide a simple example for illustration, extracting the code from demo/guide-python/external_memory.py. If you have a dataset stored in a file similar to `agaricus.txt.train` with LIBSVM format, the external memory support can be enabled by:

```
dtrain = DMatrix('../data/agaricus.txt.train#dtrain.cache')
```

XGBoost will first load `agaricus.txt.train` in, preprocess it, then write to a new file named `dtrain.cache` as an on disk cache for storing preprocessed data in an internal binary format. For more notes about text input formats, see *Text Input Format of DMatrix*.

For CLI version, simply add the cache suffix, e.g. `"../data/agaricus.txt.train#dtrain.cache"`.

GPU Version (GPU Hist tree method)

External memory is supported in GPU algorithms (i.e. when `tree_method` is set to `gpu_hist`).

If you are still getting out-of-memory errors after enabling external memory, try subsampling the data to further reduce GPU memory usage:

```python
param = {
  ...
  'subsample': 0.1,
  'sampling_method': 'gradient_based',
}
```

For more information, see this paper. Internally the tree method still concatenate all the chunks into 1 final histogram index due to performance reason, but in compressed format. So its scalability has an upper bound but still has lower memory cost in general.

CPU Version

For CPU histogram based tree methods (`approx`, `hist`) it's recommended to use `grow_policy=depthwise` for performance reason. Iterating over data batches is slow, with `depthwise` policy XGBoost can build a entire layer of tree nodes with a few iterations, while with `lossguide` XGBoost needs to iterate over the data set for each tree node.

1.4.17 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.

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-identy link function along with examples of using custom metric and objective with *scikit-learn* interface. with scikit-learn interface.

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:

```python
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:

```python
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 `feval` parameter:

```python
xgb.train({'tree_method': 'hist', 'seed': 1994,
           'disable_default_eval_metric': 1},
          dtrain=dtrain,
          num_boost_round=10,
          obj=squared_log,
          feval=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 identiy 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 recieve 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.

```python
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)
    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:

```python
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:

```python
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:

```python
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:

```python
booster = xgb.train(
    {
        "num_class": kClasses,
        "disable_default_eval_metric": True,
        "objective": "multi:softmax",
    },
    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*.

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:

```python
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`:

```python
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)
```

1.4.18 Categorical Data

Note: As of XGBoost 1.6, the feature is experimental and has limited features

---

Starting from version 1.5, XGBoost has experimental support for categorical data available for public testing. 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 `gpu_hist`, `approx`, and `hist`.
clf = xgb.XGBClassifier(tree_method="gpu_hist", enable_categorical=True)
# 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*.

Optimal Partitioning

New 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 or use quantized *DMatrix*. Also native interface supports data types other than dataframe, like `numpy/cupy array`. To use the native interface with categorical data, we need to pass the similar parameter to *DMatrix* 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.

Miscellaneous

By default, XGBoost assumes input categories 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.

1.4.19 Multiple Outputs

New 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.

Internally, 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.

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 on 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 fit into your memory. This usually means millions of instances. If you are running out of memory, checkout *external memory version* or *distributed version* of XGBoost.

1.5.4 Running XGBoost on platform X (Hadoop/Yarn, Mesos)

The distributed version of XGBoost is designed to be portable to various environment. Distributed XGBoost can be ported to any platform that supports rabit. You can directly run XGBoost on Yarn. In theory Mesos and other resource allocation engines can be easily supported as well.

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 of communication in machine learning is rather simple, in the sense that we can depend on a limited set of APIs (in our case rabit). 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 is 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.

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 specifed, 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. 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. For tree models, it is important to use consistent data formats during training and scoring.

1.6 XGBoost GPU Support

This page contains information about GPU algorithms supported in XGBoost.

Note: CUDA 11.0, 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

Specify the `tree_method` parameter as `gpu_hist`. For details around the `tree_method` parameter, see *tree method*.

Supported parameters

GPU accelerated prediction is enabled by default for the above mentioned `tree_method` parameters but can be switched to CPU prediction by setting `predictor` to `cpu_predictor`. This could be useful if you want to conserve GPU memory. Likewise when using CPU algorithms, GPU accelerated prediction can be enabled by setting `predictor` to `gpu_predictor`.

The device ordinal (which GPU to use if you have many of them) can be selected using the `gpu_id` parameter, which 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

```
param['gpu_id'] = 0
param['tree_method'] = 'gpu_hist'
```

Listing 25: With Scikit-Learn interface

```
XGBRegressor(tree_method='gpu_hist', gpu_id=0)
```

GPU-Accelerated SHAP values

XGBoost makes use of GPUTreeShap as a backend for computing shap values when the GPU predictor is selected.

```
model.set_param({"predictor": "gpu_predictor"})
shap_values = model.predict(dtrain, pred_contribs=True)
shap_interaction_values = model.predict(dtrain, pred_interactions=True)
```

See examples here.

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 here, also Python documentation *Dask API* for complete reference. For usage with `Spark` using Scala see *XGBoost4J-Spark-GPU Tutorial (version 1.6.1+)*. Lastly for distributed GPU training with `PySpark`, see *Distributed XGBoost with PySpark*.

Memory usage

The following are some guidelines on the device memory usage of the *gpu_hist* tree method.

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 or `xgboost.QuantileDMatrix` or *external memory version*. Note that when `external memory` is used for GPU hist, it's best to employ gradient based sampling as well. Last but not least, `inplace_predict` can be preferred over `predict` when data is already on GPU. Both `QuantileDMatrix` and `inplace_predict` are automatically enabled if you are using the scikit-learn interface.

CPU-GPU Interoperability

XGBoost models trained on GPUs can be used on CPU-only systems to generate predictions. For information about how to save and load an XGBoost model, 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

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: https://github.com/dmlc/xgboost/issues. For general questions please visit our user form: https://discuss.xgboost.ai/.

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.

- **Command line parameters** relate to behavior of CLI version of XGBoost.

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.

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 GPU memory. This option is only applicable when XGBoost is built (compiled) with the RMM plugin enabled. Valid values are `true` and `false`.

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.

- 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.

- 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.

- num_feature [set automatically by XGBoost, no need to be set by user]

  - Feature dimension used in boosting, set to maximum dimension of the feature

Parameters for Tree Booster

- eta [default=0.3, alias: learning_rate]

  - Step size shrinkage used in update to prevents 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.

  - range: $[0,\infty]$

- max_depth [default=6]

  - 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`]

  - 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 `gpu_hist`; other tree methods only support `uniform` sampling.

- `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.

  - `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.

  - range: $[0, \infty]$

- `alpha` [default=0, alias: `reg_alpha`]

  - L1 regularization term on weights. Increasing this value will make model more conservative.

- range: $[0, \infty]$

- `tree_method` string [default= `auto`]

  - The tree construction algorithm used in XGBoost. See description in the reference paper and *Tree Methods*.

  - XGBoost supports `approx`, `hist` and `gpu_hist` for distributed training. Experimental support for external memory is available for `approx` and `gpu_hist`.

  - Choices: `auto`, `exact`, `approx`, `hist`, `gpu_hist`, this is a combination of commonly used updaters. For other updaters like `refresh`, set the parameter `updater` directly.

    * `auto`: Use heuristic to choose the fastest method.

      · For small dataset, exact greedy (`exact`) will be used.

      · For larger dataset, approximate algorithm (`approx`) will be chosen. It's recommended to try `hist` and `gpu_hist` for higher performance with large dataset. (`gpu_hist`)has support for `external memory`.

      · Because old behavior is always use exact greedy in single machine, user will get a message when approximate algorithm is chosen to notify this choice.

    * `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.

    * `gpu_hist`: GPU implementation of `hist` 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`: Grow tree with GPU.

    * `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`, `approx` or `gpu_hist`.

    - – Choices: `depthwise`, `lossguide`

        - * `depthwise`: split at nodes closest to the root.

        - * `lossguide`: split at nodes with highest loss change.

- `max_leaves` [default=0]

    - – Maximum number of nodes to be added. Not used by `exact` tree method.

- `max_bin`, [default=256]

    - – Only used if `tree_method` is set to `hist`, `approx` or `gpu_hist`.

    - – Maximum number of discrete bins to bucket continuous features.

    - – Increasing this number improves the optimality of splits at the cost of higher computation time.

- `predictor`, [default= `auto`]

    - – The type of predictor algorithm to use. Provides the same results but allows the use of GPU or CPU.

        - * `auto`: Configure predictor based on heuristics.

        - * `cpu_predictor`: Multicore CPU prediction algorithm.

        - * `gpu_predictor`: Prediction using GPU. Used when `tree_method` is `gpu_hist`. When `predictor` is set to default value `auto`, the `gpu_hist` tree method is able to provide GPU based prediction without copying training data to GPU memory. If `gpu_predictor` is explicitly specified, then all data is copied into GPU, only recommended for performing prediction tasks.

- `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.

Parameters for Categorical Feature

These parameters are only used for training with categorical data. See *Categorical Data* for more information.

Note: These parameters are experimental. `exact` tree method is not yet supported.

- `max_cat_to_onehot`

  New 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`

  New in version 1.7.0.

  - Maximum number of categories considered for each split. Used only by partition-based splits for preventing over-fitting.

Additional parameters for Dart Booster (`booster=dart`)

Note: Using `predict()` with DART booster

If the booster object is DART type, `predict()` will perform dropouts, i.e. only some of the trees will be evaluated. This will produce incorrect results if `data` is not the training data. To obtain correct results on test sets, set `iteration_range` to a nonzero value, e.g.

```
preds = bst.predict(dtest, iteration_range=(0, num_round))
```

- `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`)

- `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.

- `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.

- `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.

  - `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.

  - `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.

  - `aft_loss_distribution`: Probability Density Function used by `survival:aft` objective and `aft-nloglik` metric.

  - `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:pairwise`: Use LambdaMART to perform pairwise ranking where the pairwise loss is minimized

  - `rank:ndcg`: Use LambdaMART to perform list-wise ranking where Normalized Discounted Cumulative Gain (NDCG) is maximized

  - `rank:map`: Use LambdaMART to perform list-wise ranking where Mean Average Precision (MAP) is maximized

  - `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, global bias

- – The parameter is automatically estimated for selected objectives before training. To disable the estimation, specify a real number argument.

- – For sufficient number of iterations, changing this value will not have too much effect.

- • `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 ranking)

  - – 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 one

  - – 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.

    - * `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.

* ndcg: Normalized Discounted Cumulative Gain

* map: Mean Average Precision

* ndcg@n, map@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, NDCG and MAP will evaluate the score of a list without any positive samples as 1. By adding "-" in the evaluation metric XGBoost will evaluate these score 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. This parameter is ignored in R package, use *set.seed()* instead.

- 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]

1.7.4 Command Line Parameters

The following parameters are only used in the console version of XGBoost

- `num_round`
    - The number of rounds for boosting
- `data`
    - The path of training data
- `test:data`
    - The path of test data to do prediction
- `save_period` [default=0]
    - The period to save the model. Setting `save_period=10` means that for every 10 rounds XGBoost will save the model. Setting it to 0 means not saving any model during the training.
- `task` [default= `train`] options: `train`, `pred`, `eval`, `dump`
    - `train`: training using data
    - `pred`: making prediction for test:data
    - `eval`: for evaluating statistics specified by `eval[name]=filename`
    - `dump`: for dump the learned model into text format
- `model_in` [default=NULL]
    - Path to input model, needed for `test`, `eval`, `dump` tasks. If it is specified in training, XGBoost will continue training from the input model.
- `model_out` [default=NULL]
    - Path to output model after training finishes. If not specified, XGBoost will output files with such names as `0003.model` where `0003` is number of boosting rounds.
- `model_dir` [default= `models/`]
    - The output directory of the saved models during training
- `fmap`
    - Feature map, used for dumping model
- `dump_format` [default= `text`] options: `text`, `json`
    - Format of model dump file
- `name_dump` [default= `dump.txt`]
    - Name of model dump file
- `name_pred` [default= `pred.txt`]
    - Name of prediction file, used in pred mode
- `pred_margin` [default=0]

– Predict margin instead of transformed probability

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 `preds_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.

- When using `preds_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 eqauls 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 eqauls 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 Predictor

There are 2 predictors in XGBoost (3 if you have the one-api plugin enabled), namely `cpu_predictor` and `gpu_predictor`. The default option is `auto` so that XGBoost can employ some heuristics for saving GPU memory during training. They might have slight different outputs due to floating point errors.

1.8.4 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.

1.8.5 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).

1.8.6 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.

1.8.7 Categorical Data

Other than users performing encoding, XGBoost has experimental support for categorical data using `gpu_hist` and `gpu_predictor`. No special operation needs to be done on input test data since the information about categories is encoded into the model during training.

1.8.8 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:

```python
def predict_fn(clf: xgb.XGBClassifier, X):
    X = preprocess(X)
    clf.set_params(predictor="gpu_predictor")  # NOT safe!
    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, ...)
```

1.9 Tree Methods

For training boosted tree models, there are 2 parameters used for choosing algorithms, namely `updater` and `tree_method`. XGBoost has 4 builtin tree methods, namely `exact`, `approx`, `hist` and `gpu_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 has 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`: Vanilla gradient boosting tree algorithm described in reference paper. During each split finding procedure, it iterates over all entries of input data. It's more accurate (among other greedy methods) but slow in computation performance. Also it doesn't support distributed training as XGBoost employs row spliting data distribution while `exact` tree method works on a sorted column format. This tree method can be used with parameter `tree_method` set to `exact`.

1.9.2 Approximated Solutions

As `exact` tree method is slow in performance and not scalable, 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 below.

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`.

3. `gpu_hist` tree method: The `gpu_hist` tree method is a GPU implementation of `hist`, with additional support for gradient based sampling. The algorithm can be accessed by setting `tree_method` to `gpu_hist`.

1.9.3 Implications

Some objectives like `reg:squarederror` have constant hessian. In this case, `hist` or `gpu_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 `(gpu)_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 have slight differences 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 trianing, 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`, `hist` and `gpu_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	Hist	GPU Hist
grow_policy	Depthwise	depthwise/lossguide	depthwise/lossguide	depthwise/lossguide
max_leaves	F	T	T	T
sampling method	uniform	uniform	uniform	gradient_based/uniform
categorical data	F	T	T	T
External memory	F	T	T	P
Distributed	F	T	T	T

Features/parameters that are not mentioned here are universally supported for all 4 tree methods (for instance, column sampling and constraints). The *P* in external memory means partially supported. 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*
    - *Supported data structures for various XGBoost functions*
        * *Markers*
        * *Table Header*
        * *Support Matrix*
- *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:

```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 complete list of supported data types, please reference the *Supported data structures for various XGBoost functions*. 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 with 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 will make loading faster:

```
dtrain = xgb.DMatrix('train.svm.txt')
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)
```

When performing ranking tasks, the number of weights should be equal to number of groups.

- To load a LIBSVM text file or a XGBoost binary file into *DMatrix*:

```
dtrain = xgb.DMatrix('train.svm.txt')
dtest = xgb.DMatrix('test.svm.buffer')
```

The parser in XGBoost has limited functionality. When using Python interface, it's recommended to use sklearn `load_svmlight_file` or other similar utilites than XGBoost's builtin parser.

- To load a CSV file into *DMatrix*:

```
# label_column specifies the index of the column containing the true label
dtrain = xgb.DMatrix('train.csv?format=csv&label_column=0')
dtest = xgb.DMatrix('test.csv?format=csv&label_column=0')
```

The parser in XGBoost has limited functionality. When using Python interface, it's recommended to use pandas `read_csv` or other similar utilites than XGBoost's builtin parser.

Supported data structures for various XGBoost functions

Markers

- T: Supported.

- F: Not supported.

- NE: Invalid type for the use case. For instance, *pd.Series* can not be multi-target label.

- NPA: Support with the help of numpy array.

- CPA: Support with the help of cupy array.

- SciCSR: Support with the help of scripy sparse CSR. 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
dlpack	CPA	CPA		CPA	FF	FF
datatable.Frame	T	FF		NPA	FF	
datatable.Table	T	FF		NPA	FF	
modin.DataFrame	NPA	FF	NPA	NPA	FF	
modin.Series	NPA	FF	NPA	NPA	FF	
pyarrow.Table	T	F		NPA	FF	
pyarrow.dataset.Dataset	T	F			F	
__array__	NPA	F	NPA	NPA	H	
Others	SciCSR	F		F	F	

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.

```
bst.save_model('0001.model')
```

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.bin')  # load 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.

```python
# Use "gpu_hist" for training the model.
reg = xgb.XGBRegressor(tree_method="gpu_hist")
# 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()
```

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*

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.

New in version 1.4.0.

>   **Parameters**
>   **new_config** (*Dict[str, Any]*) – Keyword arguments representing the parameters and their values

>   **Return type**
>   *Iterator*[None]

Example

```python
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.
>
> New 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.
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.

New 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
```

Core Data Structure

Core XGBoost Library.

class xgboost.**DMatrix**(*data*, *label=None*, ***, *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=False*, *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 *Supported data structures for various XGBoost functions* for a list of supported input types.
> - **label** (*array_like*) – Label of the training data.
> - **weight** (*array_like*) – 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** (*array_like*) – Base margin used for boosting from existing model.
> - **missing** (*float, optional*) – Value in the input data which needs to be present as a missing value. If None, defaults to np.nan.
> - **silent** (*boolean, optional*) – Whether print messages during construction
> - **feature_names** (*list, optional*) – Set names for features.
> - **feature_types** (*FeatureTypes*) – Set types for features. When *enable_categorical* is set to *True*, string "c" represents categorical data type while "q" represents numerical feature type. 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.

- **nthread** (`integer, optional`) – Number of threads to use for loading data when parallelization is applicable. If -1, uses maximum threads available on the system.

- **group** (`array_like`) – Group size for all ranking group.

- **qid** (`array_like`) – Query ID for data samples, used for ranking.

- **label_lower_bound** (`array_like`) – Lower bound for survival training.

- **label_upper_bound** (`array_like`) – Upper bound for survival training.

- **feature_weights** (`array_like, optional`) – Set feature weights for column sampling.

- **enable_categorical** (`boolean, optional`) – New in version 1.3.0.

    Note: This parameter is experimental

    Experimental support of specializing for categorical features. Do not set to True unless you are interested in development. Also, JSON/UBJSON serialization format is required.

- **data_split_mode** (`DataSplitMode`) –

property feature_names: Optional[Sequence[str]]

Get feature names (column labels).

> **Returns**
> > feature_names
>
> **Return type**
> > list or None

property feature_types: Optional[Sequence[str]]

Get feature types (column types).

> **Returns**
> > feature_types
>
> **Return type**
> > list or None

get_base_margin()

Get the base margin of the DMatrix.

> **Return type**
> > base_margin

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.

> New 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

Returns
　　info – a numpy array of float information of the data

Return type
　　array

get_group()

　　Get the group of the DMatrix.

　　　　Return type
　　　　　　group

get_label()

　　Get the label of the DMatrix.

　　　　Returns
　　　　　　label

　　　　Return type
　　　　　　array

get_uint_info(*field*)

　　Get unsigned integer property from the DMatrix.

　　　　Parameters
　　　　　　field (`str`) – The field name of the information

　　　　Returns
　　　　　　info – a numpy array of unsigned integer information of the data

　　　　Return type
　　　　　　array

get_weight()

　　Get the weight of the DMatrix.

　　　　Returns
　　　　　　weight

　　　　Return type
　　　　　　array

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.

　　New 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** (`Optional[Any]`) –
> - **weight** (`Optional[Any]`) –
> - **base_margin** (`Optional[Any]`) –
> - **group** (`Optional[Any]`) –
> - **qid** (`Optional[Any]`) –
> - **label_lower_bound** (`Optional[Any]`) –
> - **label_upper_bound** (`Optional[Any]`) –
> - **feature_names** (`Optional[Sequence[str]]`) –
> - **feature_types** (`Optional[Sequence[str]]`) –
> - **feature_weights** (`Optional[Any]`) –
>
> **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** (`Union[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=False*,
data_split_mode=DataSplitMode.ROW)

> Bases: `DMatrix`
>
> A DMatrix variant that generates quantilized data directly from input for `hist` and `gpu_hist` tree methods. 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.
>
> ---
>
> New in version 1.7.0.
>
> > **Parameters**
> >
> > - **max_bin** (`Optional[int]`) – The number of histogram bin, should be consistent with the training parameter `max_bin`.
> >
> > - **ref** (`Optional[DMatrix]`) – 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
> >
> > - **data** (*Any*) – Data source of DMatrix. See *Supported data structures for various XGBoost functions* for a list of supported input types.
> >
> > - **label** (*array_like*) – Label of the training data.
> >
> > - **weight** (*array_like*) – 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** (*array_like*) – Base margin used for boosting from existing model.

- **missing** ([*float,*](#) *optional*) – Value in the input data which needs to be present as a missing value. If None, defaults to np.nan.

- **silent** (*boolean,* *optional*) – Whether print messages during construction

- **feature_names** ([*list,*](#) *optional*) – Set names for features.

- **feature_types** (*FeatureTypes*) – Set types for features. When *enable_categorical* is set to *True*, string "c" represents categorical data type while "q" represents numerical feature type. 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.

- **nthread** (*integer,* *optional*) – Number of threads to use for loading data when parallelization is applicable. If -1, uses maximum threads available on the system.

- **group** (*array_like*) – Group size for all ranking group.

- **qid** (*array_like*) – Query ID for data samples, used for ranking.

- **label_lower_bound** (*array_like*) – Lower bound for survival training.

- **label_upper_bound** (*array_like*) – Upper bound for survival training.

- **feature_weights** (*array_like,* *optional*) – Set feature weights for column sampling.

- **enable_categorical** (*boolean,* *optional*) – New in version 1.3.0.

Note: This parameter is experimental

Experimental support of specializing for categorical features. Do not set to True unless you are interested in development. Also, JSON/UBJSON serialization format is required.

- **data_split_mode** (*DataSplitMode*) –

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** ([*dict*](#)) – Parameters for boosters.

- **cache** ([*list*](#)) – List of cache items.

- **model_file** (*string/os.PathLike/Booster/bytearray*) – Path to the model file if it's string or PathLike.

attr(*key*)

Get attribute string from the Booster.

> **Parameters**
>> **key** (`str`) – The key to get attribute from.
>
> **Returns**
>> **value** – The attribute value of the key, returns None if attribute do not exist.
>
> **Return type**
>> str

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.

boost(*dtrain*, *grad*, *hess*)

> Boost the booster for one iteration, with customized gradient statistics. Like *xgboost.Booster.*
> *update()*, this function should not be called directly by users.
>
>> **Parameters**
>>
>> - **dtrain** (`DMatrix`) – The training DMatrix.
>>
>> - **grad** (`ndarray`) – The first order of gradient.
>>
>> - **hess** (`ndarray`) – The second order of gradient.
>>
>> **Return type**
>>> None

copy()

> Copy the booster object.
>
>> **Returns**
>>> **booster** – 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** (`string or os.PathLike`) – Output file name.
>>
>> - **fmap** (`string or os.PathLike, optional`) – Name of the file containing feature map
>>   names.
>>
>> - **with_stats** (`bool, optional`) – Controls whether the split statistics are output.
>>
>> - **dump_format** (`string, optional`) – 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** (`Optional[Callable[[ndarray, DMatrix], Tuple[str, float]]]`) – Custom evaluation function.
> >
> > - **output_margin** (`bool`) –
> >
> > **Returns**
> > > **result** – Evaluation result string.
> >
> > **Return type**
> > > str

property feature_names: Optional[Sequence[str]]

> Feature names for this booster. Can be directly set by input data or by assignment.

property feature_types: Optional[Sequence[str]]

> Feature types for this booster. Can be directly set by input data or by assignment. See *DMatrix* for details.

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** (`Union[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. 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** (`Union[str, PathLike]`) – The name of feature map file

Return type

> *Dict*[str, *Union*[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** (`Union[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, *Union*[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 or os.PathLike (optional)`) – The name of feature map file.
- **bin** (`int, default None`) – 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, default True`) – Return pd.DataFrame when pandas is installed. If False or pandas is not installed, return numpy ndarray.
- **bins** (`Optional[int]`) –

Returns

- *a histogram of used splitting values for the specified feature*

- *either as numpy array or pandas DataFrame.*

Return type
    Union[*ndarray*, *DataFrame*]

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, 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.

```
booster.set_param({"predictor": "gpu_predictor"})
booster.inplace_predict(cupy_array)

booster.set_param({"predictor": "cpu_predictor"})
booster.inplace_predict(numpy_array)
```

New in version 1.1.0.

Parameters

- **data** (*numpy.ndarray/scipy.sparse.csr_matrix/cupy.ndarray/*) – cudf.DataFrame/pd.DataFrame The input data, must not be a view for numpy array. Set `predictor` to `gpu_predictor` for running prediction on CuPy array or CuDF DataFrame.

- **iteration_range** (*Tuple[int, int]*) – 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** (*Optional[Any]*) – See `xgboost.DMatrix` for details.

    New in version 1.4.0.

- **strict_shape** (*bool*) – See `xgboost.Booster.predict()` for details.

    New 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*.

New in version 1.0.0.

Parameters
    config (*str*) –

Return type
None

load_model(*fname*)

Load the model from a file or bytearray. Path to file can be local or as an URI.

The model is loaded from XGBoost format which is universal among the various XGBoost interfaces. Auxiliary attributes of the Python Booster object (such as feature_names) will not be loaded when using binary format. To save those attributes, use JSON/UBJ instead. See *Model IO* for more info.

```
model.load_model("model.json")
# or
model.load_model("model.ubj")
```

Parameters
fname (*Union[str, bytearray, PathLike]*) – 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*, *ntree_limit=0*, *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 user 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.

- **ntree_limit** (*int*) – Deprecated, use *iteration_range* instead.

- **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.

  New in version 1.0.0.

- **iteration_range** (*Tuple[int, int]*) – 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.

  New 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.

  New in version 1.4.0.

> **Returns**
> > **prediction**
>
> **Return type**
> > numpy array

save_config()

> Output internal parameter configuration of Booster as a JSON string.
>
> New 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) will not be saved when using binary format. To save those attributes, use JSON/UBJ instead. See *Model IO* for more info.

```
model.save_model("model.json")
# or
model.save_model("model.ubj")
```

Parameters
    fname (`string or os.PathLike`) – Output file name

Return type
    None

save_raw(*raw_format='deprecated'*)
    Save the model to a in memory buffer representation instead of file.

    Parameters
        raw_format (`str`) – Format of output buffer. Can be *json*, *ubj* or *deprecated*. Right now the default is *deprecated* but it will be changed to *ubj* (univeral binary json) in the future.

    Return type
        An in memory buffer representation of the model

set_attr(***kwargs*)
    Set the attribute of the Booster.

    Parameters
        **kwargs (`Optional[str]`) – 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/list/str`) – list of key,value pairs, dict of key to value or simply str key

        • **value** (`optional`) – 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 or os.PathLike (optional)`) – The name of feature map file.

    Return type
        DataFrame

update(*dtrain*, *iteration*, *fobj=None*)
    Update for one iteration, with objective function calculated internally. This function should not be called directly by users.

    Parameters

        • **dtrain** (`DMatrix`) – Training data.

        • **iteration** (`int`) – Current iteration number.

        • **fobj** (`function`) – Customized objective function.

    Return type
        None

Learning API

Training Library containing training routines.

xgboost.**train**(*params*, *dtrain*, *num_boost_round=10*, *, *evals=None*, *obj=None*, *feval=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** (`Optional[Sequence[Tuple[DMatrix, str]]]`) – List of validation sets for which metrics will evaluated during training. Validation metrics will help us track the performance of the model.

- **obj** (`Optional[Callable[[ndarray, DMatrix], Tuple[ndarray, ndarray]]]`) – Custom objective function. See *Custom Objective* for details.

- **feval** (`Optional[Callable[[ndarray, DMatrix], Tuple[str, float]]]`) – Deprecated since version 1.6.0: Use *custom_metric* instead.

- **maximize** (`bool`) – Whether to maximize feval.

- **early_stopping_rounds** (`Optional[int]`) – 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 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** (`Optional[Dict[str, Dict[str, Union[List[float], List[Tuple[float, float]]]]]]`) – 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** (`Optional[Union[bool, int]]`) – 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** (`Optional[Union[str, PathLike, Booster, bytearray]]`) – Xgb model to be loaded before training (allows training continuation).

- **callbacks** (*Optional*[*Sequence*[*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)]
    xgboost.train(params, Xy, callbacks=callbacks)
```

- **custom_metric** (*Optional*[*Callable*[[*ndarray*, *DMatrix*], *Tuple*[*str*, *float*]]]) – Custom metric function. See *Custom Metric* for details.

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, feval=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.

- **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** (*Optional*[*Callable*[[*ndarray*, *DMatrix*], *Tuple*[*ndarray*, *ndarray*]]]) – Custom objective function. See *Custom Objective* for details.

- **feval** (*function*) – Deprecated since version 1.6.0: Use *custom_metric* instead.

- **maximize** (*bool*) – Whether to maximize feval.

- **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** (*Optional[Sequence[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)]
    xgboost.train(params, Xy, callbacks=callbacks)
```

- **shuffle** (*bool*) – Shuffle data before creating folds.

- **custom_metric** (*Optional[Callable[[ndarray, DMatrix], Tuple[str, float]]]*) – 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: XGBModel, RegressorMixin

> Implementation of the scikit-learn API for XGBoost regression.

> **Parameters**

> - **n_estimators** (*int*) – Number of gradient boosted trees. Equivalent to number of boosting rounds.

> - **max_depth** (*Optional[int]*) – Maximum tree depth for base learners.

> - **max_leaves** – Maximum number of leaves; 0 indicates no limit.

> - **max_bin** – If using histogram-based algorithm, maximum number of bins per feature

- **grow_policy** – Tree growing policy. 0: favor splitting at nodes closest to the node, i.e. grow depth-wise. 1: favor 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, Callable[[numpy.ndarray, numpy.ndarray], Tuple[numpy.ndarray, numpy.ndarray]], NoneType]*) – Specify the learning task and the corresponding learning objective or a custom objective function to be used (see note below).

- **booster** (*Optional[str]*) – Specify which booster to use: gbtree, gblinear or dart.

- **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** –

  Sampling method. Used only by *gpu_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** (*Optional[float]*) – The initial prediction score of all instances, global bias.

- **random_state** (*Optional[Union[numpy.random.RandomState, int]]*) – Random number seed.

Note: Using gblinear booster with shotgun updater is nondeterministic as it uses Hogwild algorithm.

- **missing** (`float, default np.nan`) – Value in the data which needs to be present as a missing value.

- **num_parallel_tree** (`Optional[int]`) – Used for boosting random forest.

- **monotone_constraints** (`Optional[Union[Dict[str, int], str]]`) – Constraint of variable monotonicity. See *tutorial* for more information.

- **interaction_constraints** (`Optional[Union[str, List[Tuple[str]]]]`) – 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.

- **gpu_id** (`Optional[int]`) – Device ordinal.

- **validate_parameters** (`Optional[bool]`) – Give warnings for unknown parameter.

- **predictor** (`Optional[str]`) – Force XGBoost to use specific predictor, available choices are [cpu_predictor, gpu_predictor].

- **enable_categorical** (`bool`) – New in version 1.5.0.

Note: This parameter is experimental

Experimental support for categorical data. When enabled, cudf/pandas.DataFrame should be used to specify categorical data type. Also, JSON/UBJSON serialization format is required.

- **feature_types** (`Optional[FeatureTypes]`) – New in version 1.7.0.

Used for specifying feature types without constructing a dataframe. See `DMatrix` for details.

- **max_cat_to_onehot** (`Optional[int]`) – New 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]`) – New 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.

- **eval_metric** (`Optional[Union[str, List[str], Callable]]`) – New 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 doc/parameter.rst), 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* for more.

  Note: This parameter replaces *eval_metric* in `fit()` method. The old one receives untransformed prediction regardless of whether custom objective is being used.

  ```python
  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]`) – New 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()`.

  The method returns the model from the last iteration (not the best one). 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.

  If early stopping occurs, the model will have three additional fields: `best_score`, `best_iteration` and `best_ntree_limit`.

  Note: This parameter replaces *early_stopping_rounds* in `fit()` method.

- **callbacks** (`Optional[List[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** (*dict, optional*) – 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`:

y_true: array_like of shape [n_samples]
    The target values

y_pred: array_like of shape [n_samples]
    The predicted values

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

---

apply(*X*, *ntree_limit=0*, *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** (*array_like, shape=[n_samples, n_features]*) – Input features matrix.
>
> - **iteration_range** (*Optional[Tuple[int, int]]*) – See *predict()*.
>
> - **ntree_limit** (*int*) – Deprecated, use `iteration_range` instead.
>
> **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 $[0; 2^{**}(\text{self.max\_depth+1}))$, 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*, *eval_metric=None*, *early_stopping_rounds=None*, *verbose=True*, *xgb_model=None*, *sample_weight_eval_set=None*, *base_margin_eval_set=None*, *feature_weights=None*, *callbacks=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*) – Feature matrix. See *Supported data structures for various XGBoost functions* for a list of supported types.

  When the `tree_method` is set to `hist` or `gpu_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 `gpu_hist` is used for training, then the data is first processed on CPU then transferred to GPU.

- **y** (*Any*) – Labels

- **sample_weight** (*Optional[Any]*) – instance weights

- **base_margin** (*Optional[Any]*) – global bias for each instance.

- **eval_set** (*Optional[Sequence[Tuple[Any, Any]]]*) – 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_metric** (*str, list of str, or callable, optional*) – Deprecated since version 1.6.0: Use *eval_metric* in __init__() or *set_params()* instead.

- **early_stopping_rounds** (*int*) – Deprecated since version 1.6.0: Use *early_stopping_rounds* in __init__() or *set_params()* instead.

- **verbose** (*Optional[Union[bool, int]]*) – 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** (*Optional[Union[Booster, XGBModel, str]]*) – file name of stored XGBoost model or 'Booster' instance XGBoost model to be loaded before training (allows training continuation).

- **sample_weight_eval_set** (*Optional[Sequence[Any]]*) – 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** (*Optional[Sequence[Any]]*) – 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** (*Optional[Any]*) – 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.

- **callbacks** (*Optional[Sequence[TrainingCallback]]*) – Deprecated since version 1.6.0: Use *callbacks* 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_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

Note: Intercept is defined only for linear learners

Intercept (bias) is 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**
> > > **intercept_**
> >
> > **Return type**
> > > array of shape `(1,)` or `[n_classes]`

load_model(*fname*)

> Load the model from a file or bytearray. Path to file can be local or as an URI.
>
> The model is loaded from XGBoost format which is universal among the various XGBoost interfaces. Auxiliary attributes of the Python Booster object (such as feature_names) will not be loaded when using binary format. To save those attributes, use JSON/UBJ instead. See *Model IO* for more info.
>
> ```
> model.load_model("model.json")
> # or
> model.load_model("model.ubj")
> ```
>
> > **Parameters**
> > > **fname** (*Union[str, bytearray, PathLike]*) – 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*, *ntree_limit=None*, *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. For tree models, when data is on GPU, like cupy array or cuDF dataframe and *predictor* is not specified, the prediction is run on GPU automatically, otherwise it will run on CPU.

Note: This function is only thread safe for *gbtree* and *dart*.

Parameters

- **X** (*Any*) – Data to predict with.
- **output_margin** (*bool*) – Whether to output the raw untransformed margin value.
- **ntree_limit** (*Optional[int]*) – Deprecated, use *iteration_range* instead.
- **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** (*Optional[Any]*) – Margin added to prediction.
- **iteration_range** (*Optional[Tuple[int, int]]*) – 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.

  New 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) will not be saved when using binary format. To save those attributes, use JSON/UBJ instead. See *Model IO* for more info.

```
model.save_model("model.json")
# or
model.save_model("model.ubj")
```

Parameters
  fname (*string or os.PathLike*) – Output file name

Return type
  None

score(*X*, *y*, *sample_weight=None*)

Return the coefficient of determination of the prediction.

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)` wrt. *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_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](#)) –

class xgboost.**XGBClassifier**(***, *objective='binary:logistic'*, *use_label_encoder=None*, ***kwargs*)

Bases: XGBModel, [ClassifierMixin](#)

Implementation of the scikit-learn API for XGBoost classification.

Parameters

- **n_estimators** ([int](#)) – Number of boosting rounds.

- **max_depth** (`Optional[int]`) – Maximum tree depth for base learners.

- **max_leaves** – Maximum number of leaves; 0 indicates no limit.

- **max_bin** – If using histogram-based algorithm, maximum number of bins per feature

- **grow_policy** – Tree growing policy. 0: favor splitting at nodes closest to the node, i.e. grow depth-wise. 1: favor 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, Callable[[numpy.ndarray, numpy.ndarray], Tuple[numpy.ndarray, numpy.ndarray]], NoneType]`) – Specify the learning

task and the corresponding learning objective or a custom objective function to be used (see note below).

- **booster** (*Optional[str]*) – Specify which booster to use: gbtree, gblinear or dart.

- **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** –

  Sampling method. Used only by *gpu_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** (*Optional[float]*) – The initial prediction score of all instances, global bias.

- **random_state** (*Optional[Union[numpy.random.RandomState, int]]*) – Random number seed.

Note: Using gblinear booster with shotgun updater is nondeterministic as it uses Hogwild algorithm.

- **missing** (*float, default np.nan*) – Value in the data which needs to be present as a missing value.

- **num_parallel_tree** (*Optional[int]*) – Used for boosting random forest.

- **monotone_constraints** (*Optional[Union[Dict[str, int], str]]*) – Constraint of variable monotonicity. See *tutorial* for more information.

- **interaction_constraints** (`Optional[Union[str, List[Tuple[str]]]]`) – 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.

- **gpu_id** (`Optional[int]`) – Device ordinal.

- **validate_parameters** (`Optional[bool]`) – Give warnings for unknown parameter.

- **predictor** (`Optional[str]`) – Force XGBoost to use specific predictor, available choices are [cpu_predictor, gpu_predictor].

- **enable_categorical** (*bool*) – New in version 1.5.0.

Note: This parameter is experimental

Experimental support for categorical data. When enabled, cudf/pandas.DataFrame should be used to specify categorical data type. Also, JSON/UBJSON serialization format is required.

- **feature_types** (`Optional[FeatureTypes]`) – New in version 1.7.0.

Used for specifying feature types without constructing a dataframe. See `DMatrix` for details.

- **max_cat_to_onehot** (`Optional[int]`) – New 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]`) – New 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.

- **eval_metric** (`Optional[Union[str, List[str], Callable]]`) – New 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 doc/parameter.rst), 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* for more.

Note: This parameter replaces *eval_metric* in *fit()* method. The old one receives untransformed prediction regardless of whether custom objective is being used.

```
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]*) – New 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()*.

  The method returns the model from the last iteration (not the best one). 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.

  If early stopping occurs, the model will have three additional fields: *best_score*, *best_iteration* and best_ntree_limit.

  Note: This parameter replaces *early_stopping_rounds* in *fit()* method.

- **callbacks** (*Optional[List[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** (*dict, optional*) – 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`:

y_true: array_like of shape [n_samples]
    The target values

y_pred: array_like of shape [n_samples]
    The predicted values

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

- **use_label_encoder** (*Optional[bool]*) –

apply(*X*, *ntree_limit=0*, *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** (*array_like, shape=[n_samples, n_features]*) – Input features matrix.
- **iteration_range** (*Optional[Tuple[int, int]]*) – See *predict()*.
- **ntree_limit** (*int*) – Deprecated, use `iteration_range` instead.

    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*, *eval_metric=None*, *early_stopping_rounds=None*, *verbose=True*, *xgb_model=None*, *sample_weight_eval_set=None*, *base_margin_eval_set=None*, *feature_weights=None*, *callbacks=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*) – Feature matrix. See *Supported data structures for various XGBoost functions* for a list of supported types.
> >
> >   When the tree_method is set to hist or gpu_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 `gpu_hist` is used for training, then the data is first processed on CPU then transferred to GPU.

- **y** (*Any*) – Labels

- **sample_weight** (*Optional[Any]*) – instance weights

- **base_margin** (*Optional[Any]*) – global bias for each instance.

- **eval_set** (*Optional[Sequence[Tuple[Any, Any]]]*) – 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_metric** (*str, list of str, or callable, optional*) – Deprecated since version 1.6.0: Use *eval_metric* in __init__() or *set_params()* instead.

- **early_stopping_rounds** (*int*) – Deprecated since version 1.6.0: Use *early_stopping_rounds* in __init__() or *set_params()* instead.

- **verbose** (*Optional[Union[bool, int]]*) – 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** (*Optional[Union[Booster, str, XGBModel]]*) – file name of stored XGBoost model or 'Booster' instance XGBoost model to be loaded before training (allows training continuation).

- **sample_weight_eval_set** (*Optional[Sequence[Any]]*) – 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** (*Optional[Sequence[Any]]*) – 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** (*Optional[Any]*) – 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.

- **callbacks** (*Optional[Sequence[TrainingCallback]]*) – Deprecated since version 1.6.0: Use *callbacks* 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_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

>   **Note:** Intercept is defined only for linear learners

>   Intercept (bias) is 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**
>>>   intercept_

>>   **Return type**
>>>   array of shape `(1,)` or `[n_classes]`

load_model(*fname*)

>   Load the model from a file or bytearray. Path to file can be local or as an URI.

>   The model is loaded from XGBoost format which is universal among the various XGBoost interfaces. Auxiliary attributes of the Python Booster object (such as feature_names) will not be loaded when using binary format. To save those attributes, use JSON/UBJ instead. See *Model IO* for more info.

```
model.load_model("model.json")
# or
model.load_model("model.ubj")
```

>>   **Parameters**
>>>   **fname** (*Union[str, bytearray, PathLike]*) – 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*, *ntree_limit=None*, *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. For tree models, when data is on GPU, like cupy array or cuDF dataframe and *predictor* is not specified, the prediction is run on GPU automatically, otherwise it will run on CPU.

Note: This function is only thread safe for *gbtree* and *dart*.

Parameters

- **X** (*Any*) – Data to predict with.

- **output_margin** (*bool*) – Whether to output the raw untransformed margin value.

- **ntree_limit** (*Optional[int]*) – Deprecated, use *iteration_range* instead.

- **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** (*Optional[Any]*) – Margin added to prediction.

- **iteration_range** (*Optional[Tuple[int, int]]*) – 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.

  New in version 1.4.0.

Return type

  prediction

predict_proba(*X*, *ntree_limit=None*, *validate_features=True*, *base_margin=None*, *iteration_range=None*)

  Predict the probability of each *X* example being of a given class.

Note: This function is only thread safe for *gbtree* and *dart*.

Parameters

- **X** (*array_like*) – Feature matrix. See *Supported data structures for various XGBoost functions* for a list of supported types.

- **ntree_limit** (*int*) – Deprecated, use *iteration_range* instead.

- **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** (*array_like*) – Margin added to prediction.

- **iteration_range** (*Optional[Tuple[int, int]]*) – 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) will not be saved when using binary format. To save those attributes, use JSON/UBJ instead. See *Model IO* for more info.

```
model.save_model("model.json")
# or
model.save_model("model.ubj")
```

> **Parameters**
> > **fname** (`string or os.PathLike`) – Output file name
>
> **Return type**
> > None

score(*X*, *y*, *sample_weight=None*)

> Return the mean accuracy on the given test 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)` wrt. *y*.
>
> **Return type**
> > float

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`) –

class xgboost.**XGBRanker**(***, *objective='rank:pairwise'*, ***kwargs*)

> Bases: `XGBModel`, `XGBRankerMixIn`
>
> Implementation of the Scikit-Learn API for XGBoost Ranking.
>
> **Parameters**
>
> - **n_estimators** (`int`) – Number of gradient boosted trees. Equivalent to number of boosting rounds.
>
> - **max_depth** (`Optional[int]`) – Maximum tree depth for base learners.

- **max_leaves** – Maximum number of leaves; 0 indicates no limit.

- **max_bin** – If using histogram-based algorithm, maximum number of bins per feature

- **grow_policy** – Tree growing policy. 0: favor splitting at nodes closest to the node, i.e. grow depth-wise. 1: favor 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, Callable[[numpy.ndarray, numpy.ndarray], Tuple[numpy.ndarray, numpy.ndarray]], NoneType]*) – Specify the learning task and the corresponding learning objective or a custom objective function to be used (see note below).

- **booster** (*Optional[str]*) – Specify which booster to use: gbtree, gblinear or dart.

- **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** –

  Sampling method. Used only by *gpu_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** (*Optional[float]*) – The initial prediction score of all instances, global bias.

- **random_state** (*Optional[Union[numpy.random.RandomState, int]]*) – Random number seed.

Note: Using gblinear booster with shotgun updater is nondeterministic as it uses Hogwild algorithm.

- **missing** (`float, default np.nan`) – Value in the data which needs to be present as a missing value.

- **num_parallel_tree** (`Optional[int]`) – Used for boosting random forest.

- **monotone_constraints** (`Optional[Union[Dict[str, int], str]]`) – Constraint of variable monotonicity. See *tutorial* for more information.

- **interaction_constraints** (`Optional[Union[str, List[Tuple[str]]]]`) – 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.

- **gpu_id** (`Optional[int]`) – Device ordinal.

- **validate_parameters** (`Optional[bool]`) – Give warnings for unknown parameter.

- **predictor** (`Optional[str]`) – Force XGBoost to use specific predictor, available choices are [cpu_predictor, gpu_predictor].

- **enable_categorical** (`bool`) – New in version 1.5.0.

Note: This parameter is experimental

Experimental support for categorical data. When enabled, cudf/pandas.DataFrame should be used to specify categorical data type. Also, JSON/UBJSON serialization format is required.

- **feature_types** (`Optional[FeatureTypes]`) – New in version 1.7.0.

Used for specifying feature types without constructing a dataframe. See `DMatrix` for details.

- **max_cat_to_onehot** (`Optional[int]`) – New 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]`) – New 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.

- **eval_metric** (*Optional[Union[str, List[str], Callable]]*) – New 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 doc/parameter.rst), 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* for more.

  Note: This parameter replaces *eval_metric* in `fit()` method. The old one receives untransformed prediction regardless of whether custom objective is being used.

  ```python
  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]*) – New 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()`.

  The method returns the model from the last iteration (not the best one). 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.

  If early stopping occurs, the model will have three additional fields: `best_score`, `best_iteration` and `best_ntree_limit`.

  Note: This parameter replaces *early_stopping_rounds* in `fit()` method.

- **callbacks** (*Optional[List[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** (*dict, optional*) – 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: The default objective for XGBRanker is "rank:pairwise"

Note: A custom objective function is currently not supported by XGBRanker. Likewise, a custom metric function is not supported either.

Note: Query group information is required for ranking tasks by either using the *group* parameter or *qid* parameter in *fit* method. This information is not required in 'predict' method and 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 query group first. *group* must be an array that contains the size of each query group. When fitting the model with the *qid* parameter, your data does not need sorting. *qid* must be an array that contains the group of 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.

apply(*X*, *ntree_limit=0*, *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** (`array_like, shape=[n_samples, n_features]`) – Input features matrix.

- **iteration_range** (`Optional[Tuple[int, int]]`) – See `predict()`.

- **ntree_limit** (`int`) – Deprecated, use `iteration_range` instead.

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*, *eval_metric=None*, *early_stopping_rounds=None*, *verbose=False*, *xgb_model=None*, *sample_weight_eval_set=None*, *base_margin_eval_set=None*, *feature_weights=None*, *callbacks=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 *Supported data structures for various XGBoost functions* for a list of supported types.

  When the tree_method is set to hist or gpu_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 gpu_hist is used for training, then the data is first processed on CPU then transferred to GPU.

- **y** (*Any*) – Labels

- **group** (*Optional[Any]*) – 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** (*Optional[Any]*) – Query ID for each training sample. Should have the size of n_samples. If this is set to None, then user must provide group.

- **sample_weight** (*Optional[Any]*) – 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** (*Optional[Any]*) – Global bias for each instance.

- **eval_set** (*Optional[Sequence[Tuple[Any, Any]]]*) – 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** (*Optional[Sequence[Any]]*) – 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** (*Optional[Sequence[Any]]*) – A list in which eval_qid[i] is the array containing query ID of i-th pair in **eval_set**.

- **eval_metric** (*str, list of str, optional*) – Deprecated since version 1.6.0: use *eval_metric* in __init__() or *set_params()* instead.

- **early_stopping_rounds** (`int`) – Deprecated since version 1.6.0: use *early_stopping_rounds* in __init__() or `set_params()` instead.

- **verbose** (`Optional[Union[bool, int]]`) – 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** (`Optional[Union[Booster, str, XGBModel]]`) – file name of stored XGBoost model or 'Booster' instance XGBoost model to be loaded before training (allows training continuation).

- **sample_weight_eval_set** (`Optional[Sequence[Any]]`) – 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** (`Optional[Sequence[Any]]`) – 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** (`Optional[Any]`) – 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.

- **callbacks** (`Optional[Sequence[TrainingCallback]]`) – Deprecated since version 1.6.0: Use *callbacks* in __init__() or `set_params()` instead.

   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_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

Note: Intercept is defined only for linear learners

Intercept (bias) is 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
intercept_

Return type
array of shape `(1,)` or `[n_classes]`

load_model(*fname*)

Load the model from a file or bytearray. Path to file can be local or as an URI.

The model is loaded from XGBoost format which is universal among the various XGBoost interfaces. Auxiliary attributes of the Python Booster object (such as feature_names) will not be loaded when using binary format. To save those attributes, use JSON/UBJ instead. See *Model IO* for more info.

```
model.load_model("model.json")
# or
model.load_model("model.ubj")
```

Parameters
fname (*Union[str, bytearray, PathLike]*) – 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*, *ntree_limit=None*, *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. For tree models, when data is on GPU, like cupy array or cuDF dataframe and *predictor* is not specified, the prediction is run on GPU automatically, otherwise it will run on CPU.

Note: This function is only thread safe for *gbtree* and *dart*.

Parameters

- **X** (*Any*) – Data to predict with.

- **output_margin** (*bool*) – Whether to output the raw untransformed margin value.

- **ntree_limit** (*Optional[int]*) – Deprecated, use *iteration_range* instead.

- **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** (*Optional[Any]*) – Margin added to prediction.

- **iteration_range** (*Optional[Tuple[int, int]]*) – 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.

  New 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) will not be saved when using binary format. To save those attributes, use JSON/UBJ instead. See *Model IO* for more info.

```
model.save_model("model.json")
# or
model.save_model("model.ubj")
```

> **Parameters**
> **fname** (*string or os.PathLike*) – Output file name
>
> **Return type**
> None

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*) –

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.

> **Parameters**
>
> - **n_estimators** (*int*) – Number of trees in random forest to fit.
>
> - **max_depth** (*Optional[int]*) – Maximum tree depth for base learners.

- **max_leaves** – Maximum number of leaves; 0 indicates no limit.

- **max_bin** – If using histogram-based algorithm, maximum number of bins per feature

- **grow_policy** – Tree growing policy. 0: favor splitting at nodes closest to the node, i.e. grow depth-wise. 1: favor 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, Callable[[numpy.ndarray, numpy.ndarray], Tuple[numpy.ndarray, numpy.ndarray]], NoneType]*) – Specify the learning task and the corresponding learning objective or a custom objective function to be used (see note below).

- **booster** (*Optional[str]*) – Specify which booster to use: gbtree, gblinear or dart.

- **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** –

  Sampling method. Used only by *gpu_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** (*Optional[float]*) – The initial prediction score of all instances, global bias.

- **random_state** (*Optional[Union[numpy.random.RandomState, int]]*) – Random number seed.

Note: Using gblinear booster with shotgun updater is nondeterministic as it uses Hogwild algorithm.

- **missing** (`float, default np.nan`) – Value in the data which needs to be present as a missing value.

- **num_parallel_tree** (`Optional[int]`) – Used for boosting random forest.

- **monotone_constraints** (`Optional[Union[Dict[str, int], str]]`) – Constraint of variable monotonicity. See *tutorial* for more information.

- **interaction_constraints** (`Optional[Union[str, List[Tuple[str]]]]`) – 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.

- **gpu_id** (`Optional[int]`) – Device ordinal.

- **validate_parameters** (`Optional[bool]`) – Give warnings for unknown parameter.

- **predictor** (`Optional[str]`) – Force XGBoost to use specific predictor, available choices are [cpu_predictor, gpu_predictor].

- **enable_categorical** (`bool`) – New in version 1.5.0.

Note: This parameter is experimental

Experimental support for categorical data. When enabled, cudf/pandas.DataFrame should be used to specify categorical data type. Also, JSON/UBJSON serialization format is required.

- **feature_types** (`Optional[FeatureTypes]`) – New in version 1.7.0.

Used for specifying feature types without constructing a dataframe. See `DMatrix` for details.

- **max_cat_to_onehot** (`Optional[int]`) – New 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]`) – New 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.

- **eval_metric** (*Optional[Union[str, List[str], Callable]]*) – New 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 doc/parameter.rst), 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* for more.

  Note: This parameter replaces *eval_metric* in `fit()` method. The old one receives untransformed prediction regardless of whether custom objective is being used.

  ```
  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]*) – New 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()`.

  The method returns the model from the last iteration (not the best one). 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.

  If early stopping occurs, the model will have three additional fields: `best_score`, `best_iteration` and `best_ntree_limit`.

  Note: This parameter replaces *early_stopping_rounds* in `fit()` method.

- **callbacks** (*Optional[List[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** (*dict, optional*) – 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`:

y_true: array_like of shape [n_samples]
The target values

y_pred: array_like of shape [n_samples]
The predicted values

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

---

apply(*X*, *ntree_limit=0*, *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** (*array_like, shape=[n_samples, n_features]*) – Input features matrix.

- **iteration_range** (*Optional[Tuple[int, int]]*) – See *predict()*.

- **ntree_limit** (*int*) – Deprecated, use `iteration_range` instead.

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*, *eval_metric=None*, *early_stopping_rounds=None*, *verbose=True*, *xgb_model=None*, *sample_weight_eval_set=None*, *base_margin_eval_set=None*, *feature_weights=None*, *callbacks=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*) – Feature matrix. See *Supported data structures for various XGBoost functions* for a list of supported types.

  When the `tree_method` is set to `hist` or `gpu_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 `gpu_hist` is used for training, then the data is first processed on CPU then transferred to GPU.

- **y** (*Any*) – Labels

- **sample_weight** (*Optional[Any]*) – instance weights

- **base_margin** (*Optional[Any]*) – global bias for each instance.

- **eval_set** (*Optional[Sequence[Tuple[Any, Any]]]*) – 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_metric** (*str, list of str, or callable, optional*) – Deprecated since version 1.6.0: Use *eval_metric* in __init__() or *set_params()* instead.

- **early_stopping_rounds** (*int*) – Deprecated since version 1.6.0: Use *early_stopping_rounds* in __init__() or *set_params()* instead.

- **verbose** (*Optional[Union[bool, int]]*) – 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** (*Optional[Union[Booster, str, XGBModel]]*) – file name of stored XGBoost model or 'Booster' instance XGBoost model to be loaded before training (allows training continuation).

- **sample_weight_eval_set** (*Optional[Sequence[Any]]*) – 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** (*Optional[Sequence[Any]]*) – 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** (*Optional[Any]*) – 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.

- **callbacks** (*Optional[Sequence[TrainingCallback]]*) – Deprecated since version 1.6.0: Use *callbacks* 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_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
>
>   ---
>
>   **Note:** Intercept is defined only for linear learners
>
>   Intercept (bias) is 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**
>   >   **intercept_**
>   >
>   > **Return type**
>   >   array of shape (1,) or [n_classes]

load_model(*fname*)

>   Load the model from a file or bytearray. Path to file can be local or as an URI.
>
>   The model is loaded from XGBoost format which is universal among the various XGBoost interfaces. Auxiliary attributes of the Python Booster object (such as feature_names) will not be loaded when using binary format. To save those attributes, use JSON/UBJ instead. See *Model IO* for more info.
>
>   ```
>   model.load_model("model.json")
>   # or
>   model.load_model("model.ubj")
>   ```
>
>   > **Parameters**
>   >   **fname** (*Union[str, bytearray, PathLike]*) – 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*, *ntree_limit=None*, *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. For tree models, when data is on GPU, like cupy array or cuDF dataframe and *predictor* is not specified, the prediction is run on GPU automatically, otherwise it will run on CPU.

Note: This function is only thread safe for *gbtree* and *dart*.

Parameters

- **X** (*Any*) – Data to predict with.

- **output_margin** (*bool*) – Whether to output the raw untransformed margin value.

- **ntree_limit** (*Optional[int]*) – Deprecated, use *iteration_range* instead.

- **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** (*Optional[Any]*) – Margin added to prediction.

- **iteration_range** (*Optional[Tuple[int, int]]*) – 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.

  New 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) will not be saved when using binary format. To save those attributes, use JSON/UBJ instead. See *Model IO* for more info.

```
model.save_model("model.json")
# or
model.save_model("model.ubj")
```

Parameters
 fname (*string or os.PathLike*) – Output file name

Return type
 None

score(*X*, *y*, *sample_weight=None*)

Return the coefficient of determination of the prediction.

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) wrt. *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_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*) –

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.

  Parameters

- **n_estimators** (*int*) – Number of trees in random forest to fit.

- **max_depth** (*Optional[int]*) – Maximum tree depth for base learners.

- **max_leaves** – Maximum number of leaves; 0 indicates no limit.

- **max_bin** – If using histogram-based algorithm, maximum number of bins per feature

- **grow_policy** – Tree growing policy. 0: favor splitting at nodes closest to the node, i.e. grow depth-wise. 1: favor 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, Callable[[numpy.ndarray, numpy.ndarray], Tuple[numpy.ndarray, numpy.ndarray]], NoneType]*) – Specify the learning task and the corresponding learning objective or a custom objective function to be used (see note below).

- **booster** (*Optional[str]*) – Specify which booster to use: gbtree, gblinear or dart.

- **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** –

  Sampling method. Used only by *gpu_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** (*Optional[float]*) – The initial prediction score of all instances, global bias.

- **random_state** (*Optional[Union[numpy.random.RandomState, int]]*) – Random number seed.

Note: Using gblinear booster with shotgun updater is nondeterministic as it uses Hogwild algorithm.

- **missing** (*float, default np.nan*) – Value in the data which needs to be present as a missing value.

- **num_parallel_tree** (*Optional[int]*) – Used for boosting random forest.

- **monotone_constraints** (`Optional[Union[Dict[str, int], str]]`) – Constraint of variable monotonicity. See *tutorial* for more information.

- **interaction_constraints** (`Optional[Union[str, List[Tuple[str]]]]`) – 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.

- **gpu_id** (`Optional[int]`) – Device ordinal.

- **validate_parameters** (`Optional[bool]`) – Give warnings for unknown parameter.

- **predictor** (`Optional[str]`) – Force XGBoost to use specific predictor, available choices are [cpu_predictor, gpu_predictor].

- **enable_categorical** (*bool*) – New in version 1.5.0.

Note: This parameter is experimental

Experimental support for categorical data. When enabled, cudf/pandas.DataFrame should be used to specify categorical data type. Also, JSON/UBJSON serialization format is required.

- **feature_types** (`Optional[FeatureTypes]`) – New in version 1.7.0.

Used for specifying feature types without constructing a dataframe. See `DMatrix` for details.

- **max_cat_to_onehot** (`Optional[int]`) – New 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]`) – New 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.

- **eval_metric** (`Optional[Union[str, List[str], Callable]]`) – New 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 doc/parameter.rst), 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* for more.

Note: This parameter replaces *eval_metric* in `fit()` method. The old one receives untransformed prediction regardless of whether custom objective is being used.

```python
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]*) – New 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()`.

  The method returns the model from the last iteration (not the best one). 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.

  If early stopping occurs, the model will have three additional fields: `best_score`, `best_iteration` and `best_ntree_limit`.

  Note: This parameter replaces *early_stopping_rounds* in `fit()` method.

- **callbacks** (*Optional[List[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.

```python
for params in parameters_grid:
    # be sure to (re)initialize the callbacks before each run
    callbacks = [xgb.callback.LearningRateScheduler(custom_rates)]
```

(continues on next page)

```
reg = xgboost.XGBRegressor(**params, callbacks=callbacks)
reg.fit(X, y)
```

- **kwargs** (*dict, optional*) – 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`:

y_true: array_like of shape [n_samples]
    The target values

y_pred: array_like of shape [n_samples]
    The predicted values

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

---

apply(*X*, *ntree_limit=0*, *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** (*array_like, shape=[n_samples, n_features]*) – Input features matrix.
>
> - **iteration_range** (*Optional[Tuple[int, int]]*) – See *predict()*.
>
> - **ntree_limit** (*int*) – Deprecated, use `iteration_range` instead.
>
> **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*, *eval_metric=None*, *early_stopping_rounds=None*, *verbose=True*, *xgb_model=None*, *sample_weight_eval_set=None*, *base_margin_eval_set=None*, *feature_weights=None*, *callbacks=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*) – Feature matrix. See *Supported data structures for various XGBoost functions* for a list of supported types.

> When the `tree_method` is set to `hist` or `gpu_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 `gpu_hist` is used for training, then the data is first processed on CPU then transferred to GPU.

- **y** (*Any*) – Labels

- **sample_weight** (*Optional[Any]*) – instance weights

- **base_margin** (*Optional[Any]*) – global bias for each instance.

- **eval_set** (*Optional[Sequence[Tuple[Any, Any]]]*) – 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_metric** (*str, list of str, or callable, optional*) – Deprecated since version 1.6.0: Use *eval_metric* in __init__() or *set_params()* instead.

- **early_stopping_rounds** (*int*) – Deprecated since version 1.6.0: Use *early_stopping_rounds* in __init__() or *set_params()* instead.

- **verbose** (*Optional[Union[bool, int]]*) – 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** (*Optional[Union[Booster, str, XGBModel]]*) – file name of stored XGBoost model or 'Booster' instance XGBoost model to be loaded before training (allows training continuation).

- **sample_weight_eval_set** (*Optional[Sequence[Any]]*) – 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** (*Optional[Sequence[Any]]*) – 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** (*Optional[Any]*) – 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.

- **callbacks** (*Optional[Sequence[TrainingCallback]]*) – Deprecated since version 1.6.0: Use *callbacks* 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_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

Note: Intercept is defined only for linear learners

Intercept (bias) is 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**
> > > **intercept_**
> >
> > **Return type**
> > > array of shape (1,) or [n_classes]

load_model(*fname*)

> Load the model from a file or bytearray. Path to file can be local or as an URI.
>
> The model is loaded from XGBoost format which is universal among the various XGBoost interfaces. Auxiliary attributes of the Python Booster object (such as feature_names) will not be loaded when using binary format. To save those attributes, use JSON/UBJ instead. See *Model IO* for more info.

```
model.load_model("model.json")
# or
model.load_model("model.ubj")
```

> > **Parameters**
> > > **fname** (*Union[str, bytearray, PathLike]*) – 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*, *ntree_limit=None*, *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. For tree models, when data is on GPU, like cupy array or cuDF dataframe and *predictor* is not specified, the prediction is run on GPU automatically, otherwise it will run on CPU.

Note: This function is only thread safe for *gbtree* and *dart*.

Parameters

- **X** (*Any*) – Data to predict with.

- **output_margin** (*bool*) – Whether to output the raw untransformed margin value.

- **ntree_limit** (*Optional[int]*) – Deprecated, use *iteration_range* instead.

- **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** (*Optional[Any]*) – Margin added to prediction.

- **iteration_range** (*Optional[Tuple[int, int]]*) – 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.

  New in version 1.4.0.

Return type
  prediction

predict_proba(*X*, *ntree_limit=None*, *validate_features=True*, *base_margin=None*, *iteration_range=None*)

Predict the probability of each *X* example being of a given class.

Note: This function is only thread safe for *gbtree* and *dart*.

Parameters

- **X** (*array_like*) – Feature matrix. See *Supported data structures for various XGBoost functions* for a list of supported types.

- **ntree_limit** (*int*) – Deprecated, use *iteration_range* instead.

- **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** (*array_like*) – Margin added to prediction.

- **iteration_range** (*Optional[Tuple[int, int]]*) – 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 inter-faces. Auxiliary attributes of the Python Booster object (such as feature_names) will not be saved when using binary format. To save those attributes, use JSON/UBJ instead. See *Model IO* for more info.

```
model.save_model("model.json")
# or
model.save_model("model.ubj")
```

> **Parameters**
>> **fname** (`string or os.PathLike`) – Output file name
>
> **Return type**
>> None

score(*X*, *y*, *sample_weight=None*)

> Return the mean accuracy on the given test 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)` wrt. *y*.
>
> **Return type**
>> float

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*) –

Plotting API

Plotting Library.

xgboost.**plot_importance**(*booster*, *ax=None*, *height=0.2*, *xlim=None*, *ylim=None*, *title='Feature importance'*, *xlabel='F 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** (`Booster, XGBModel or dict`) – Booster or XGBModel instance, or dict taken by Booster.get_fscore()
>
> - **ax** (`matplotlib Axes, default None`) – 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, default "weight"`) – 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, default None`) – Maximum number of top features displayed on plot. If None, all features will be displayed.
>
> - **height** (`float, default 0.2`) – Bar height, passed to ax.barh()
>
> - **xlim** (`tuple, default None`) – Tuple passed to axes.xlim()
>
> - **ylim** (`tuple, default None`) – Tuple passed to axes.ylim()
>
> - **title** (`str, default "Feature importance"`) – Axes title. To disable, pass None.
>
> - **xlabel** (`str, default "F score"`) – X axis title label. To disable, pass None.
>
> - **ylabel** (`str, default "Features"`) – Y axis title label. To disable, pass None.
>
> - **fmap** (`str or os.PathLike (optional)`) – The name of feature map file.
>
> - **show_values** (`bool, default True`) – Show values on plot. To disable, pass False.
>
> - **values_format** (`str, default "{v}"`) – 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=0*, *rankdir=None*, *ax=None*, ***kwargs*)

Plot specified tree.

> **Parameters**

- **booster** (`Booster, `*`XGBModel`*) – Booster or XGBModel instance

- **fmap** (`str (optional)`) – The name of feature map file

- **num_trees** (`int, `*`default 0`*) – Specify the ordinal number of target tree

- **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.

- **kwargs** (`Any`) – Other keywords passed to to_graphviz

  Returns
      ax

  Return type
      matplotlib Axes

xgboost.**to_graphviz**(*booster*, *fmap=''*, *num_trees=0*, *rankdir=None*, *yes_color=None*, *no_color=None*, *condition_node_params=None*, *leaf_node_params=None*, ***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 (optional)`) – The name of feature map file

- **num_trees** (`int, `*`default 0`*) – Specify the ordinal number of target tree

- **rankdir** (`str, `*`default "UT"`*) – Passed to graphviz via graph_attr

- **yes_color** (`str, `*`default '#0000FF'`*) – Edge color when meets the node condition.

- **no_color** (`str, `*`default '#FF0000'`*) – Edge color when doesn't meet the node condition.

- **condition_node_params** (`dict, `*`optional`*) – Condition node configuration for for graphviz. Example:

  ```
  {'shape': 'box',
   'style': 'filled,rounded',
   'fillcolor': '#78bceb'}
  ```

- **leaf_node_params** (`dict, `*`optional`*) – Leaf node configuration for graphviz. Example:

  ```
  {'shape': 'box',
   'style': 'filled',
   'fillcolor': '#e48038'}
  ```

- ****kwargs** (`dict, `*`optional`*) – 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.

New in version 1.3.0.

> **after_iteration**(*model*, *epoch*, *evals_log*)
>
> Run after each iteration. Return True when training should stop.
>
> > **Parameters**
> >
> > - **model** (*Any*) –
> >
> > - **epoch** (*int*) –
> >
> > - **evals_log** (*Dict[str, Dict[str, Union[List[float], List[Tuple[float, float]]]]]*) –
> >
> > **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. Return True when training should stop.
>
> > **Parameters**
> >
> > - **model** (*Any*) –
> >
> > - **epoch** (*int*) –
> >
> > - **evals_log** (*Dict[str, Dict[str, Union[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*)

Bases: *TrainingCallback*

Print the evaluation result at each iteration.

New in version 1.3.0.

> **Parameters**
>
> > - **metric** – Extra user defined metric.
> >
> > - **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.

after_iteration(*model*, *epoch*, *evals_log*)

> Run after each iteration. Return True when training should stop.
>
> > **Parameters**
> >
> > > - **model** (*Any*) –
> > >
> > > - **epoch** (*int*) –
> > >
> > > - **evals_log** (*Dict[str, Dict[str, Union[List[float], List[Tuple[float, float]]]]]*) –
> >
> > **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. Return True when training should stop.
>
> > **Parameters**
> >
> > > - **model** (*Any*) –
> > >
> > > - **epoch** (*int*) –
> > >
> > > - **evals_log** (*Dict[str, Dict[str, Union[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.**EarlyStopping**(*rounds*, *metric_name=None*, *data_name=None*, *maximize=None*, *save_best=False*, *min_delta=0.0*)

> Bases: *TrainingCallback*
>
> Callback function for early stopping

New in version 1.3.0.

> **Parameters**
>
> - **rounds** (`int`) – Early stopping rounds.
>
> - **metric_name** (`Optional[str]`) – Name of metric that is used for early stopping.
>
> - **data_name** (`Optional[str]`) – Name of dataset that is used for early stopping.
>
> - **maximize** (`Optional[bool]`) – Whether to maximize evaluation metric. None means auto (discouraged).
>
> - **save_best** (`Optional[bool]`) – Whether training should return the best model or the last model.
>
> - **min_delta** (`float`) – Minimum absolute change in score to be qualified as an improvement.
>
>   New in version 1.5.0.

```
es = xgboost.callback.EarlyStopping(
    rounds=2,
    abs_tol=1e-3,
    save_best=True,
    maximize=False,
    data_name="validation_0",
    metric_name="mlogloss",
)
clf = xgboost.XGBClassifier(tree_method="gpu_hist", 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. Return True when training should stop.

> **Parameters**
>
> - **model** (`Any`) –
>
> - **epoch** (`int`) –
>
> - **evals_log** (`Dict[str, Dict[str, Union[List[float], List[Tuple[float, float]]]]]`) –
>
> **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. Return True when training should stop.

> **Parameters**
>
> - **model** (`Any`) –

- **epoch** (*int*) –

- **evals_log** (`Dict[str, Dict[str, Union[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.**LearningRateScheduler**(*learning_rates*)

>   Bases: [`TrainingCallback`](#)

>   Callback function for scheduling learning rate.

>   New in version 1.3.0.

>   **Parameters**
>   **learning_rates** (`Union[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. Return True when training should stop.

>   **Parameters**

>   - **model** (*Any*) –

>   - **epoch** (*int*) –

>   - **evals_log** (`Dict[str, Dict[str, Union[List[float], List[Tuple[float, float]]]]]`) –

>   **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. Return True when training should stop.

>   **Parameters**

>   - **model** (*Any*) –

>   - **epoch** (*int*) –

> > > - **evals_log** (*Dict[str, Dict[str, Union[List[float], List[Tuple[float, float]]]]]*) –

> > **Return type**
> > > [bool](https://...)

> **before_training**(*model*)
> > Run before training starts.

> > **Parameters**
> > > **model** (*Any*) –

> > **Return type**
> > > *Any*

class xgboost.callback.**TrainingCheckPoint**(*directory*, *name='model'*, *as_pickle=False*, *iterations=100*)

> Bases: *TrainingCallback*

> Checkpointing operation.

> New in version 1.3.0.

> > **Parameters**

> > - **directory** (*Union[str, PathLike]*) – Output model directory.
> > - **name** (*str*) – pattern of output model file. Models will be saved as name_0.json, name_1.json, name_2.json ….
> > - **as_pickle** (*bool*) – When set to True, all training parameters will be saved in pickle format, instead of saving only the model.
> > - **iterations** (*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. Return True when training should stop.

> > **Parameters**

> > - **model** (*Any*) –
> > - **epoch** (*int*) –
> > - **evals_log** (*Dict[str, Dict[str, Union[List[float], List[Tuple[float, float]]]]]*) –

> > **Return type**
> > > [bool](https://...)

> **after_training**(*model*)
> > Run after training is finished.

> > **Parameters**
> > > **model** (*Any*) –

> > **Return type**
> > > *Any*

> **before_iteration**(*model*, *epoch*, *evals_log*)
> > Run before each iteration. Return True when training should stop.

> > **Parameters**

> > - **model** (*Any*) –

- **epoch** (`int`) –

- **evals_log** (`Dict[str, Dict[str, Union[List[float], List[Tuple[float, float]]]]]`) –

> **Return type**
>   bool

before_training(*model*)

> Run before training starts.
>
> > **Parameters**
> >   **model** (`Any`) –
> >
> > **Return type**
> >   *Any*

Dask API

Dask extensions for distributed training

See *Distributed XGBoost with Dask* for simple tutorial. Also *XGBoost Dask Feature Walkthrough* for some examples.

There are two sets of APIs in this module, one is the functional API including `train` and `predict` methods. Another is stateful Scikit-Learner wrapper inherited from single-node Scikit-Learn interface.

The implementation is heavily influenced by dask_xgboost: https://github.com/dask/dask-xgboost

Optional dask configuration

- **xgboost.scheduler_address**: Specify the scheduler address, see *Troubleshooting*.

  New in version 1.6.0.

  ```
  dask.config.set({"xgboost.scheduler_address": "192.0.0.100"})
  # We can also specify the port.
  dask.config.set({"xgboost.scheduler_address": "192.0.0.100:12345"})
  ```

class xgboost.dask.**DaskDMatrix**(*client*, *data*, *label=None*, *, *weight=None*, *base_margin=None*, *missing=None*, *silent=False*, *feature_names=None*, *feature_types=None*, *group=None*, *qid=None*, *label_lower_bound=None*, *label_upper_bound=None*, *feature_weights=None*, *enable_categorical=False*)

> Bases: `object`
>
> DMatrix holding on references to Dask DataFrame or Dask Array. Constructing a *DaskDMatrix* forces all lazy computation to be carried out. Wait for the input data explicitly if you want to see actual computation of constructing *DaskDMatrix*.
>
> See doc for `xgboost.DMatrix` constructor for other parameters. DaskDMatrix accepts only dask collection.
>
> ---
>
> **Note:** DaskDMatrix does not repartition or move data between workers. It's the caller's responsibility to balance the data.
>
> ---
>
> New in version 1.0.0.

Parameters

- **client** (`distributed.Client`) – Specify the dask client used for training. Use default client returned from dask if it's set to None.

- **data** (`Union[da.Array, dd.DataFrame]`) –

- **label** (`Optional[Union[da.Array, dd.DataFrame, dd.Series]]`) –

- **weight** (`Optional[Union[da.Array, dd.DataFrame, dd.Series]]`) –

- **base_margin** (`Optional[Union[da.Array, dd.DataFrame, dd.Series]]`) –

- **missing** (`Optional[float]`) –

- **silent** (`bool`) –

- **feature_names** (`Optional[Sequence[str]]`) –

- **feature_types** (`Optional[Sequence[str]]`) –

- **group** (`Optional[Union[da.Array, dd.DataFrame, dd.Series]]`) –

- **qid** (`Optional[Union[da.Array, dd.DataFrame, dd.Series]]`) –

- **label_lower_bound** (`Optional[Union[da.Array, dd.DataFrame, dd.Series]]`) –

- **label_upper_bound** (`Optional[Union[da.Array, dd.DataFrame, dd.Series]]`) –

- **feature_weights** (`Optional[Union[da.Array, dd.DataFrame, dd.Series]]`) –

- **enable_categorical** (`bool`) –

num_col()

> Get the number of columns (features) in the DMatrix.

> **Return type**
>> number of columns

class xgboost.dask.**DaskQuantileDMatrix**(*client*, *data*, *label=None*, *, *weight=None*, *base_margin=None*, *missing=None*, *silent=False*, *feature_names=None*, *feature_types=None*, *max_bin=None*, *ref=None*, *group=None*, *qid=None*, *label_lower_bound=None*, *label_upper_bound=None*, *feature_weights=None*, *enable_categorical=False*)

Bases: `DaskDMatrix`

Parameters

- **client** (`distributed.Client`) –

- **data** (`Union[da.Array, dd.DataFrame]`) –

- **label** (`Optional[Union[da.Array, dd.DataFrame, dd.Series]]`) –

- **weight** (`Optional[Union[da.Array, dd.DataFrame, dd.Series]]`) –

- **base_margin** (`Optional[Union[da.Array, dd.DataFrame, dd.Series]]`) –

- **missing** (`Optional[float]`) –

- **silent** (`bool`) –

- **feature_names** (`Optional[Sequence[str]]`) –

- **feature_types** (`Optional[Union[Any, List[Any]]]`) –

- **max_bin** (*Optional[int]*) –
- **ref** (*Optional[DMatrix]*) –
- **group** (*Optional[Union[da.Array, dd.DataFrame, dd.Series]]*) –
- **qid** (*Optional[Union[da.Array, dd.DataFrame, dd.Series]]*) –
- **label_lower_bound** (*Optional[Union[da.Array, dd.DataFrame, dd.Series]]*) –
- **label_upper_bound** (*Optional[Union[da.Array, dd.DataFrame, dd.Series]]*) –
- **feature_weights** (*Optional[Union[da.Array, dd.DataFrame, dd.Series]]*) –
- **enable_categorical** (*bool*) –

num_col()

> Get the number of columns (features) in the DMatrix.

> > **Return type**
> > > number of columns

xgboost.dask.train(*client*, *params*, *dtrain*, *num_boost_round=10*, ***, *evals=None*, *obj=None*, *feval=None*, *early_stopping_rounds=None*, *xgb_model=None*, *verbose_eval=True*, *callbacks=None*, *custom_metric=None*)

Train XGBoost model.

New in version 1.0.0.

Note: Other parameters are the same as *xgboost.train()* except for *evals_result*, which is returned as part of function return value instead of argument.

Parameters

- **client** (*distributed.Client*) – Specify the dask client used for training. Use default client returned from dask if it's set to None.
- **params** (*Dict[str, Any]*) –
- **dtrain** (*DaskDMatrix*) –
- **num_boost_round** (*int*) –
- **evals** (*Optional[Sequence[Tuple[DaskDMatrix, str]]]*) –
- **obj** (*Optional[Callable[[ndarray, DMatrix], Tuple[ndarray, ndarray]]]*) –
- **feval** (*Optional[Callable[[ndarray, DMatrix], Tuple[str, float]]]*) –
- **early_stopping_rounds** (*Optional[int]*) –
- **xgb_model** (*Optional[Booster]*) –
- **verbose_eval** (*Union[int, bool]*) –
- **callbacks** (*Optional[Sequence[TrainingCallback]]*) –
- **custom_metric** (*Optional[Callable[[ndarray, DMatrix], Tuple[str, float]]]*) –

Returns

> **results** – A dictionary containing trained booster and evaluation history. *history* field is the same as *eval_result* from *xgboost.train*.

```
{'booster': xgboost.Booster,
 'history': {'train': {'logloss': ['0.48253', '0.35953']},
             'eval': {'logloss': ['0.480385', '0.357756']}}}}
```

Return type

> dict

xgboost.dask.**predict**(*client*, *model*, *data*, *output_margin=False*, *missing=nan*, *pred_leaf=False*, *pred_contribs=False*, *approx_contribs=False*, *pred_interactions=False*, *validate_features=True*, *iteration_range=(0, 0)*, *strict_shape=False*)

Run prediction with a trained booster.

Note: Using `inplace_predict` might be faster when some features are not needed. See *xgboost.Booster.predict()* for details on various parameters. When output has more than 2 dimensions (shap value, leaf with strict_shape), input should be `da.Array` or `DaskDMatrix`.

New in version 1.0.0.

Parameters

- **client** (*Optional[distributed.Client]*) – Specify the dask client used for training. Use default client returned from dask if it's set to None.
- **model** (*Union[TrainReturnT, Booster, distributed.Future]*) – The trained model. It can be a distributed.Future so user can pre-scatter it onto all workers.
- **data** (*Union[DaskDMatrix, da.Array, dd.DataFrame]*) – Input data used for prediction. When input is a dataframe object, prediction output is a series.
- **missing** (*float*) – Used when input data is not DaskDMatrix. Specify the value considered as missing.
- **output_margin** (*bool*) –
- **pred_leaf** (*bool*) –
- **pred_contribs** (*bool*) –
- **approx_contribs** (*bool*) –
- **pred_interactions** (*bool*) –
- **validate_features** (*bool*) –
- **iteration_range** (*Tuple[int, int]*) –
- **strict_shape** (*bool*) –

Returns

> **prediction** – When input data is `dask.array.Array` or `DaskDMatrix`, the return value is an array, when input data is `dask.dataframe.DataFrame`, return value can be `dask.dataframe.Series`, `dask.dataframe.DataFrame`, depending on the output shape.

Return type

> dask.array.Array/dask.dataframe.Series

xgboost.dask.**inplace_predict**(*client*, *model*, *data*, *iteration_range=(0, 0)*, *predict_type='value'*, *missing=nan*, *validate_features=True*, *base_margin=None*, *strict_shape=False*)

>   Inplace prediction. See doc in *xgboost.Booster.inplace_predict()* for details.

>   New in version 1.1.0.

>   **Parameters**

>> *   **client** (*Optional[distributed.Client]*) – Specify the dask client used for training. Use default client returned from dask if it's set to None.

>> *   **model** (*Union[TrainReturnT, Booster, distributed.Future]*) – See *xgboost.dask.predict()* for details.

>> *   **data** (*Union[da.Array, dd.DataFrame]*) – dask collection.

>> *   **iteration_range** (*Tuple[int, int]*) – See *xgboost.Booster.predict()* for details.

>> *   **predict_type** (*str*) – See *xgboost.Booster.inplace_predict()* for details.

>> *   **missing** (*float*) – Value in the input data which needs to be present as a missing value. If None, defaults to np.nan.

>> *   **base_margin** (*Optional[Union[da.Array, dd.DataFrame, dd.Series]]*) – See *xgboost.DMatrix* for details.

>>   New in version 1.4.0.

>> *   **strict_shape** (*bool*) – See *xgboost.Booster.predict()* for details.

>>   New in version 1.4.0.

>> *   **validate_features** (*bool*) –

>   **Returns**

>>   When input data is dask.array.Array, the return value is an array, when input data is dask.dataframe.DataFrame, return value can be dask.dataframe.Series, dask.dataframe.DataFrame, depending on the output shape.

>   **Return type**

>>   prediction

class xgboost.dask.**DaskXGBClassifier**(*max_depth=None*, *max_leaves=None*, *max_bin=None*, *grow_policy=None*, *learning_rate=None*, *n_estimators=100*, *verbosity=None*, *objective=None*, *booster=None*, *tree_method=None*, *n_jobs=None*, *gamma=None*, *min_child_weight=None*, *max_delta_step=None*, *subsample=None*, *sampling_method=None*, *colsample_bytree=None*, *colsample_bylevel=None*, *colsample_bynode=None*, *reg_alpha=None*, *reg_lambda=None*, *scale_pos_weight=None*, *base_score=None*, *random_state=None*, *missing=nan*, *num_parallel_tree=None*, *monotone_constraints=None*, *interaction_constraints=None*, *importance_type=None*, *gpu_id=None*, *validate_parameters=None*, *predictor=None*, *enable_categorical=False*, *feature_types=None*, *max_cat_to_onehot=None*, *max_cat_threshold=None*, *eval_metric=None*, *early_stopping_rounds=None*, *callbacks=None*, ***kwargs*)

>   Bases: DaskScikitLearnBase, ClassifierMixin

>   Implementation of the scikit-learn API for XGBoost classification.

Parameters

- **n_estimators** (`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. 0: favor splitting at nodes closest to the node, i.e. grow depth-wise. 1: favor 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, Callable[[numpy.ndarray, numpy.ndarray], Tuple[numpy.ndarray, numpy.ndarray]], NoneType]`) – Specify the learning task and the corresponding learning objective or a custom objective function to be used (see note below).

- **booster** (`Optional[str]`) – Specify which booster to use: gbtree, gblinear or dart.

- **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 *gpu_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** (`Optional[float]`) – The initial prediction score of all instances, global bias.

- **random_state** (`Optional[Union[numpy.random.RandomState, int]]`) – Random number seed.

Note: Using gblinear booster with shotgun updater is nondeterministic as it uses Hogwild algorithm.

- **missing** (`float, default np.nan`) – Value in the data which needs to be present as a missing value.

- **num_parallel_tree** (`Optional[int]`) – Used for boosting random forest.

- **monotone_constraints** (`Optional[Union[Dict[str, int], str]]`) – Constraint of variable monotonicity. See *tutorial* for more information.

- **interaction_constraints** (`Optional[Union[str, List[Tuple[str]]]]`) – 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.

- **gpu_id** (`Optional[int]`) – Device ordinal.

- **validate_parameters** (`Optional[bool]`) – Give warnings for unknown parameter.

- **predictor** (`Optional[str]`) – Force XGBoost to use specific predictor, available choices are [cpu_predictor, gpu_predictor].

- **enable_categorical** (`bool`) – New in version 1.5.0.

Note: This parameter is experimental

Experimental support for categorical data. When enabled, cudf/pandas.DataFrame should be used to specify categorical data type. Also, JSON/UBJSON serialization format is required.

- **feature_types** (`Optional[FeatureTypes]`) – New in version 1.7.0.

Used for specifying feature types without constructing a dataframe. See `DMatrix` for details.

- **max_cat_to_onehot** (`Optional[int]`) – New 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, *en-*

able_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]*) – New 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.

- **eval_metric** (*Optional[Union[str, List[str], Callable]]*) – New 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 doc/parameter.rst), 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* for more.

Note: This parameter replaces *eval_metric* in `fit()` method. The old one receives untransformed prediction regardless of whether custom objective is being used.

```python
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]*) – New 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()`.

  The method returns the model from the last iteration (not the best one). 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.

  If early stopping occurs, the model will have three additional fields: *best_score*, *best_iteration* and best_ntree_limit.

> **Note:** This parameter replaces *early_stopping_rounds* in `fit()` method.

- **callbacks** (`Optional[List[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** (`dict, optional`) – 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.

apply(*X*, *ntree_limit=None*, *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** (`array_like, shape=[n_samples, n_features]`) – Input features matrix.
- **iteration_range** (`Optional[Tuple[int, int]]`) – See `predict()`.
- **ntree_limit** (`Optional[int]`) – Deprecated, use `iteration_range` instead.

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 client: `distributed.Client`

The dask client used in this model. The *Client* object can not be serialized for transmission, so if task is

launched from a worker instead of directly from the client process, this attribute needs to be set at that worker.

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*, *eval_metric=None*, *early_stopping_rounds=None*, *verbose=True*, *xgb_model=None*, *sample_weight_eval_set=None*, *base_margin_eval_set=None*, *feature_weights=None*, *callbacks=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** (*Union[da.Array, dd.DataFrame]*) – Feature matrix. See *Supported data structures for various XGBoost functions* for a list of supported types.

  When the `tree_method` is set to `hist` or `gpu_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 `gpu_hist` is used for training, then the data is first processed on CPU then transferred to GPU.

- **y** (*Union[da.Array, dd.DataFrame, dd.Series]*) – Labels

- **sample_weight** (*Optional[Union[da.Array, dd.DataFrame, dd.Series]]*) – instance weights

- **base_margin** (*Optional[Union[da.Array, dd.DataFrame, dd.Series]]*) – global bias for each instance.

- **eval_set** (*Optional[Sequence[Tuple[Union[da.Array, dd.DataFrame, dd.Series], Union[da.Array, dd.DataFrame, dd.Series]]]]*) – 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_metric** (*str, list of str, or callable, optional*) – Deprecated since version 1.6.0: Use *eval_metric* in `__init__()` or *set_params()* instead.

- **early_stopping_rounds** (*int*) – Deprecated since version 1.6.0: Use *early_stopping_rounds* in `__init__()` or *set_params()* instead.

- **verbose** (*Union[int, bool]*) – 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** (*Optional[Union[Booster, XGBModel]]*) – file name of stored XGBoost model or 'Booster' instance XGBoost model to be loaded before training (allows training continuation).

- **sample_weight_eval_set** (*Optional[Sequence[Union[da.Array, dd.DataFrame, dd.Series]]]*) – 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** (*Optional[Sequence[Union[da.Array, dd.DataFrame, dd.Series]]]*) – 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** (*Optional[Union[da.Array, dd.DataFrame, dd.Series]]*) – 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.

- **callbacks** (*Optional[Sequence[TrainingCallback]]*) – Deprecated since version 1.6.0: Use *callbacks* in `__init__()` or *set_params()* instead.

Return type
  DaskXGBClassifier

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_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

   Note: Intercept is defined only for linear learners

   Intercept (bias) is 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
       intercept_

   Return type
       array of shape (1,) or [n_classes]

load_model(*fname*)

   Load the model from a file or bytearray. Path to file can be local or as an URI.

   The model is loaded from XGBoost format which is universal among the various XGBoost interfaces.
   Auxiliary attributes of the Python Booster object (such as feature_names) will not be loaded when using
   binary format. To save those attributes, use JSON/UBJ instead. See *Model IO* for more info.

```
model.load_model("model.json")
# or
model.load_model("model.ubj")
```

   Parameters
       fname (*Union[str, bytearray, PathLike]*) – 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*, *ntree_limit=None*, *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. For tree models, when data is on GPU, like cupy array or cuDF dataframe and *predictor* is not specified, the prediction is run on GPU automatically, otherwise it will run on CPU.

Note: This function is only thread safe for *gbtree* and *dart*.

Parameters

- **X** (*Union[da.Array, dd.DataFrame]*) – Data to predict with.

- **output_margin** (*bool*) – Whether to output the raw untransformed margin value.

- **ntree_limit** (*Optional[int]*) – Deprecated, use *iteration_range* instead.

- **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** (*Optional[Union[da.Array, dd.DataFrame, dd.Series]]*) – Margin added to prediction.

- **iteration_range** (*Optional[Tuple[int, int]]*) – 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.

    New in version 1.4.0.

Return type
:   prediction

predict_proba(*X*, *ntree_limit=None*, *validate_features=True*, *base_margin=None*, *iteration_range=None*)
:   Predict the probability of each *X* example being of a given class.

Note: This function is only thread safe for *gbtree* and *dart*.

Parameters

- **X** (*array_like*) – Feature matrix. See *Supported data structures for various XGBoost functions* for a list of supported types.

- **ntree_limit** (*int*) – Deprecated, use *iteration_range* instead.

- **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** (*array_like*) – Margin added to prediction.

- **iteration_range** (*Optional[Tuple[int, int]]*) – 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) will not be saved when using binary format. To save those attributes, use JSON/UBJ instead. See *Model IO* for more info.

```
model.save_model("model.json")
# or
model.save_model("model.ubj")
```

Parameters
fname (`string or os.PathLike`) – Output file name

Return type
None

score(*X*, *y*, *sample_weight=None*)

Return the mean accuracy on the given test 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)` wrt. *y*.

Return type
float

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*) –

class xgboost.dask.**DaskXGBRegressor**(*max_depth=None*, *max_leaves=None*, *max_bin=None*, *grow_policy=None*, *learning_rate=None*, *n_estimators=100*, *verbosity=None*, *objective=None*, *booster=None*, *tree_method=None*, *n_jobs=None*, *gamma=None*, *min_child_weight=None*, *max_delta_step=None*, *subsample=None*, *sampling_method=None*, *colsample_bytree=None*, *colsample_bylevel=None*, *colsample_bynode=None*, *reg_alpha=None*, *reg_lambda=None*, *scale_pos_weight=None*, *base_score=None*, *random_state=None*, *missing=nan*, *num_parallel_tree=None*, *monotone_constraints=None*, *interaction_constraints=None*, *importance_type=None*, *gpu_id=None*, *validate_parameters=None*, *predictor=None*, *enable_categorical=False*, *feature_types=None*, *max_cat_to_onehot=None*, *max_cat_threshold=None*, *eval_metric=None*, *early_stopping_rounds=None*, *callbacks=None*, ***kwargs*)

> Bases: DaskScikitLearnBase, `RegressorMixin`

> Implementation of the Scikit-Learn API for XGBoost.

> > **Parameters**

> > > - **n_estimators** (`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. 0: favor splitting at nodes closest to the node, i.e. grow depth-wise. 1: favor 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, Callable[[numpy.ndarray, numpy.ndarray], Tuple[numpy.ndarray, numpy.ndarray]], NoneType]`) – Specify the learning task and the corresponding learning objective or a custom objective function to be used (see note below).

> > > - **booster** (`Optional[str]`) – Specify which booster to use: gbtree, gblinear or dart.

> > > - **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 *gpu_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** (`Optional[`[`float`]`]`) – The initial prediction score of all instances, global bias.

- **random_state** (`Optional[Union[`[`numpy.random.RandomState`]`, `[`int`]`]]`) – Random number seed.

Note: Using gblinear booster with shotgun updater is nondeterministic as it uses Hogwild algorithm.

- **missing** (`float, default np.nan`) – Value in the data which needs to be present as a missing value.

- **num_parallel_tree** (`Optional[`[`int`]`]`) – Used for boosting random forest.

- **monotone_constraints** (`Optional[Union[Dict[`[`str`]`, `[`int`]`], `[`str`]`]]`) – Constraint of variable monotonicity. See *tutorial* for more information.

- **interaction_constraints** (`Optional[Union[`[`str`]`, `List[Tuple[`[`str`]`]]]]`) – 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.

- **gpu_id** (`Optional[`[`int`]`]`) – Device ordinal.

- **validate_parameters** (`Optional[`[`bool`]`]`) – Give warnings for unknown parameter.

- **predictor** (`Optional[`[`str`]`]`) – Force XGBoost to use specific predictor, available choices are [cpu_predictor, gpu_predictor].

- **enable_categorical** (*bool*) – New in version 1.5.0.

Note: This parameter is experimental

Experimental support for categorical data. When enabled, cudf/pandas.DataFrame should be used to specify categorical data type. Also, JSON/UBJSON serialization format is required.

- **feature_types** (*Optional[FeatureTypes]*) – New in version 1.7.0.

  Used for specifying feature types without constructing a dataframe. See DMatrix for details.

- **max_cat_to_onehot** (*Optional[int]*) – New 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]*) – New 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.

- **eval_metric** (*Optional[Union[str, List[str], Callable]]*) – New 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 doc/parameter.rst), 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* for more.

Note: This parameter replaces *eval_metric* in *fit()* method. The old one receives untransformed prediction regardless of whether custom objective is being used.

```
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]`) – New 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()*.

  The method returns the model from the last iteration (not the best one). 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.

  If early stopping occurs, the model will have three additional fields: *best_score*, *best_iteration* and best_ntree_limit.

  Note: This parameter replaces *early_stopping_rounds* in *fit()* method.

- **callbacks** (`Optional[List[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** (`dict, optional`) – 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.

  ---

apply(*X*, *ntree_limit=None*, *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** (`array_like, shape=[n_samples, n_features]`) – Input features matrix.

   - **iteration_range** (`Optional[Tuple[int, int]]`) – See *predict()*.

- **ntree_limit** (*Optional[int]*) – Deprecated, use `iteration_range` instead.

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 client: distributed.Client

> The dask client used in this model. The *Client* object can not be serialized for transmission, so if task is launched from a worker instead of directly from the client process, this attribute needs to be set at that worker.

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*, *eval_metric=None*, *early_stopping_rounds=None*, *verbose=True*, *xgb_model=None*, *sample_weight_eval_set=None*, *base_margin_eval_set=None*, *feature_weights=None*, *callbacks=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** (*Union[da.Array, dd.DataFrame]*) – Feature matrix. See *Supported data structures for various XGBoost functions* for a list of supported types.

  When the tree_method is set to hist or gpu_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 gpu_hist is used for training, then the data is first processed on CPU then transferred to GPU.

- **y** (*Union[da.Array, dd.DataFrame, dd.Series]*) – Labels

- **sample_weight** (*Optional[Union[da.Array, dd.DataFrame, dd.Series]]*) – instance weights

- **base_margin** (*Optional[Union[da.Array, dd.DataFrame, dd.Series]]*) – global bias for each instance.

- **eval_set** (*Optional[Sequence[Tuple[Union[da.Array, dd.DataFrame, dd.Series], Union[da.Array, dd.DataFrame, dd.Series]]]]*) – 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_metric** (*str, list of str, or callable, optional*) – Deprecated since version 1.6.0: Use *eval_metric* in __init__() or *set_params()* instead.

- **early_stopping_rounds** (*int*) – Deprecated since version 1.6.0: Use *early_stopping_rounds* in __init__() or *set_params()* instead.

- **verbose** (*Union[int, bool]*) – 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** (*Optional[Union[Booster, XGBModel]]*) – file name of stored XGBoost model or 'Booster' instance XGBoost model to be loaded before training (allows training continuation).

- **sample_weight_eval_set** (*Optional[Sequence[Union[da.Array, dd.DataFrame, dd.Series]]]*) – 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** (*Optional[Sequence[Union[da.Array, dd.DataFrame, dd.Series]]]*) – 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** (*Optional[Union[da.Array, dd.DataFrame, dd.Series]]*) – Weight for each feature, defines the probability of each feature being selected when col-sample is being used. All values must be greater than 0, otherwise a *ValueError* is thrown.

- **callbacks** (*Optional[Sequence[TrainingCallback]]*) – Deprecated since version 1.6.0: Use *callbacks* in __init__() or *set_params()* instead.

> **Return type**
> *DaskXGBRegressor*

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_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

Note: Intercept is defined only for linear learners

Intercept (bias) is 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**
> intercept_

> **Return type**
> array of shape (1,) or [n_classes]

---

load_model(*fname*)

> Load the model from a file or bytearray. Path to file can be local or as an URI.
>
> The model is loaded from XGBoost format which is universal among the various XGBoost interfaces. Auxiliary attributes of the Python Booster object (such as feature_names) will not be loaded when using binary format. To save those attributes, use JSON/UBJ instead. See *Model IO* for more info.
>
> ```
> model.load_model("model.json")
> # or
> model.load_model("model.ubj")
> ```
>
> > **Parameters**
> > > **fname** (*Union[str, bytearray, PathLike]*) – 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*, *ntree_limit=None*, *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. For tree models, when data is on GPU, like cupy array or cuDF dataframe and *predictor* is not specified, the prediction is run on GPU automatically, otherwise it will run on CPU.
>
> ---
>
> **Note:** This function is only thread safe for *gbtree* and *dart*.
>
> ---
>
> > **Parameters**
> >
> > - **X** (*Union[da.Array, dd.DataFrame]*) – Data to predict with.
> >
> > - **output_margin** (*bool*) – Whether to output the raw untransformed margin value.
> >
> > - **ntree_limit** (*Optional[int]*) – Deprecated, use *iteration_range* instead.
> >
> > - **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** (*Optional[Union[da.Array, dd.DataFrame, dd.Series]]*) – Margin added to prediction.
> >
> > - **iteration_range** (*Optional[Tuple[int, int]]*) – 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.
> >
> >   New 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) will not be saved when using binary format. To save those attributes, use JSON/UBJ instead. See *Model IO* for more info.

```
model.save_model("model.json")
# or
model.save_model("model.ubj")
```

> **Parameters**
> **fname** (`string or os.PathLike`) – Output file name
>
> **Return type**
> None

score(*X*, *y*, *sample_weight=None*)

Return the coefficient of determination of the prediction.

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)` wrt. *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_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*) –

---

class xgboost.dask.**DaskXGBRanker**(*, *objective='rank:pairwise'*, ***kwargs*)

Bases: DaskScikitLearnBase, XGBRankerMixIn

Implementation of the Scikit-Learn API for XGBoost Ranking.

New in version 1.4.0.

Parameters

- **n_estimators** (`int`) – Number of gradient boosted trees. Equivalent to number of boosting rounds.

- **max_depth** (`Optional[int]`) – Maximum tree depth for base learners.

- **max_leaves** – Maximum number of leaves; 0 indicates no limit.

- **max_bin** – If using histogram-based algorithm, maximum number of bins per feature

- **grow_policy** – Tree growing policy. 0: favor splitting at nodes closest to the node, i.e. grow depth-wise. 1: favor 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`, `Callable[[numpy.ndarray, numpy.ndarray]`, `Tuple[numpy.ndarray, numpy.ndarray]], NoneType]`) – Specify the learning task and the corresponding learning objective or a custom objective function to be used (see note below).

- **booster** (`Optional[str]`) – Specify which booster to use: gbtree, gblinear or dart.

- **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** –

  Sampling method. Used only by *gpu_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** (*Optional[float]*) – The initial prediction score of all instances, global bias.

- **random_state** (*Optional[Union[numpy.random.RandomState, int]]*) – Random number seed.

Note: Using gblinear booster with shotgun updater is nondeterministic as it uses Hogwild algorithm.

- **missing** (*float, default np.nan*) – Value in the data which needs to be present as a missing value.

- **num_parallel_tree** (*Optional[int]*) – Used for boosting random forest.

- **monotone_constraints** (*Optional[Union[Dict[str, int], str]]*) – Constraint of variable monotonicity. See *tutorial* for more information.

- **interaction_constraints** (*Optional[Union[str, List[Tuple[str]]]]*) – 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.

- **gpu_id** (*Optional[int]*) – Device ordinal.

- **validate_parameters** (*Optional[bool]*) – Give warnings for unknown parameter.

- **predictor** (*Optional[str]*) – Force XGBoost to use specific predictor, available choices are [cpu_predictor, gpu_predictor].

- **enable_categorical** (*bool*) – New in version 1.5.0.

Note: This parameter is experimental

Experimental support for categorical data. When enabled, cudf/pandas.DataFrame should be used to specify categorical data type. Also, JSON/UBJSON serialization format is required.

- **feature_types** (*Optional[FeatureTypes]*) – New in version 1.7.0.

Used for specifying feature types without constructing a dataframe. See DMatrix for details.

- **max_cat_to_onehot** (*Optional[int]*) – New 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]*) – New 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.

- **eval_metric** (*Optional[Union[str, List[str], Callable]]*) – New 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 doc/parameter.rst), 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* for more.

Note: This parameter replaces *eval_metric* in *fit()* method. The old one receives untransformed prediction regardless of whether custom objective is being used.

```python
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]*) – New 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()*.

The method returns the model from the last iteration (not the best one). 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.

If early stopping occurs, the model will have three additional fields: *best_score*, *best_iteration* and best_ntree_limit.

Note: This parameter replaces *early_stopping_rounds* in *fit()* method.

- **callbacks** (*Optional[List[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** (*dict, optional*) – 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: For dask implementation, group is not supported, use qid instead.

---

apply(*X*, *ntree_limit=None*, *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** (*array_like, shape=[n_samples, n_features]*) – Input features matrix.
- **iteration_range** (*Optional[Tuple[int, int]]*) – See *predict()*.
- **ntree_limit** (*Optional[int]*) – Deprecated, use iteration_range instead.

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 client: [distributed.Client](#)

The dask client used in this model. The *Client* object can not be serialized for transmission, so if task is launched from a worker instead of directly from the client process, this attribute needs to be set at that worker.

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*, *eval_metric=None*, *early_stopping_rounds=None*, *verbose=False*, *xgb_model=None*, *sample_weight_eval_set=None*, *base_margin_eval_set=None*, *feature_weights=None*, *callbacks=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** (*Union[da.Array, dd.DataFrame]*) – Feature matrix. See *Supported data structures for various XGBoost functions* for a list of supported types.
>
>   When the `tree_method` is set to `hist` or `gpu_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 `gpu_hist` is used for training, then the data is first processed on CPU then transferred to GPU.
>
> - **y** (*Union[da.Array, dd.DataFrame, dd.Series]*) – Labels
>
> - **group** (*Optional[Union[da.Array, dd.DataFrame, dd.Series]]*) – 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** (*Optional[Union[da.Array, dd.DataFrame, dd.Series]]*) – Query ID for each training sample. Should have the size of n_samples. If this is set to None, then user must provide group.
>
> - **sample_weight** (*Optional[Union[da.Array, dd.DataFrame, dd.Series]]*) – 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** (*Optional[Union[da.Array, dd.DataFrame, dd.Series]]*) – Global bias for each instance.
>
> - **eval_set** (*Optional[Sequence[Tuple[Union[da.Array, dd.DataFrame, dd.Series], Union[da.Array, dd.DataFrame, dd.Series]]]]*) – 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** (*Optional[Sequence[Union[da.Array, dd.DataFrame, dd.Series]]]*) – 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** (*Optional[Sequence[Union[da.Array, dd.DataFrame, dd.Series]]]*) – A list in which `eval_qid[i]` is the array containing query ID of `i`-th pair in **eval_set**.
>
> - **eval_metric** (*str, list of str, optional*) – Deprecated since version 1.6.0: use *eval_metric* in __init__() or *set_params()* instead.
>
> - **early_stopping_rounds** (*int*) – Deprecated since version 1.6.0: use *early_stopping_rounds* in __init__() or *set_params()* instead.

- **verbose** (`Union[int, bool]`) – 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** (`Optional[Union[Booster, XGBModel]]`) – file name of stored XGBoost model or 'Booster' instance XGBoost model to be loaded before training (allows training continuation).

- **sample_weight_eval_set** (`Optional[Sequence[Union[da.Array, dd.DataFrame, dd.Series]]]`) – 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** (`Optional[Sequence[Union[da.Array, dd.DataFrame, dd.Series]]]`) – 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** (`Optional[Union[da.Array, dd.DataFrame, dd.Series]]`) – Weight for each feature, defines the probability of each feature being selected when col-sample is being used. All values must be greater than 0, otherwise a *ValueError* is thrown.

- **callbacks** (`Optional[Sequence[TrainingCallback]]`) – Deprecated since version 1.6.0: Use *callbacks* in __init__() or *set_params()* instead.

    Return type
    DaskXGBRanker

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_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
>
> ---
>
> **Note:** Intercept is defined only for linear learners
>
> Intercept (bias) is 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**
> >
> > intercept_
> >
> > **Return type**
> >
> > array of shape (1,) or [n_classes]

load_model(*fname*)

> Load the model from a file or bytearray. Path to file can be local or as an URI.
>
> The model is loaded from XGBoost format which is universal among the various XGBoost interfaces. Auxiliary attributes of the Python Booster object (such as feature_names) will not be loaded when using binary format. To save those attributes, use JSON/UBJ instead. See *Model IO* for more info.
>
> ```
> model.load_model("model.json")
> # or
> model.load_model("model.ubj")
> ```
>
> > **Parameters**
> >
> > **fname** (*Union[str, bytearray, PathLike]*) – 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*, *ntree_limit=None*, *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. For tree models, when data is on GPU, like cupy array or cuDF dataframe and *predictor* is not specified, the prediction is run on GPU automatically, otherwise it will run on CPU.
>
> ---
>
> **Note:** This function is only thread safe for *gbtree* and *dart*.
>
> ---
>
> > **Parameters**
> >
> > - **X** (*Union[da.Array, dd.DataFrame]*) – Data to predict with.
> >
> > - **output_margin** (*bool*) – Whether to output the raw untransformed margin value.

- **ntree_limit** (`Optional[int]`) – Deprecated, use *iteration_range* instead.

- **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** (`Optional[Union[da.Array, dd.DataFrame, dd.Series]]`) – Margin added to prediction.

- **iteration_range** (`Optional[Tuple[int, int]]`) – 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.

    New 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) will not be saved when using binary format. To save those attributes, use JSON/UBJ instead. See *Model IO* for more info.

```
model.save_model("model.json")
# or
model.save_model("model.ubj")
```

Parameters
    fname (`string or os.PathLike`) – Output file name

Return type
    None

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*) –

class xgboost.dask.**DaskXGBRFRegressor**(***, *learning_rate=1*, *subsample=0.8*, *colsample_bynode=0.8*, *reg_lambda=1e-05*, ***kwargs*)

Bases: *DaskXGBRegressor*

Implementation of the Scikit-Learn API for XGBoost Random Forest Regressor.

New in version 1.4.0.

Parameters

- **n_estimators** (*int*) – Number of trees in random forest to fit.

- **max_depth** (`Optional[int]`) – Maximum tree depth for base learners.

- **max_leaves** – Maximum number of leaves; 0 indicates no limit.

- **max_bin** – If using histogram-based algorithm, maximum number of bins per feature

- **grow_policy** – Tree growing policy. 0: favor splitting at nodes closest to the node, i.e. grow depth-wise. 1: favor 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, Callable[[numpy.ndarray, numpy.ndarray], Tuple[numpy.ndarray, numpy.ndarray]], NoneType]*) – Specify the learning task and the corresponding learning objective or a custom objective function to be used (see note below).

- **booster** (*Optional[str]*) – Specify which booster to use: gbtree, gblinear or dart.

- **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** –

  Sampling method. Used only by *gpu_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** (*Optional[float]*) – The initial prediction score of all instances, global bias.

- **random_state** (*Optional[Union[numpy.random.RandomState, int]]*) – Random number seed.

> **Note:** Using gblinear booster with shotgun updater is nondeterministic as it uses Hogwild algorithm.

- **missing** (`float, default np.nan`) – Value in the data which needs to be present as a missing value.

- **num_parallel_tree** (`Optional[int]`) – Used for boosting random forest.

- **monotone_constraints** (`Optional[Union[Dict[str, int], str]]`) – Constraint of variable monotonicity. See *tutorial* for more information.

- **interaction_constraints** (`Optional[Union[str, List[Tuple[str]]]]`) – 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.

- **gpu_id** (`Optional[int]`) – Device ordinal.

- **validate_parameters** (`Optional[bool]`) – Give warnings for unknown parameter.

- **predictor** (`Optional[str]`) – Force XGBoost to use specific predictor, available choices are [cpu_predictor, gpu_predictor].

- **enable_categorical** (`bool`) – New in version 1.5.0.

> **Note:** This parameter is experimental

  Experimental support for categorical data. When enabled, cudf/pandas.DataFrame should be used to specify categorical data type. Also, JSON/UBJSON serialization format is required.

- **feature_types** (`Optional[FeatureTypes]`) – New in version 1.7.0.

  Used for specifying feature types without constructing a dataframe. See `DMatrix` for details.

- **max_cat_to_onehot** (`Optional[int]`) – New 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]`) – New 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.

- **eval_metric** (`Optional[Union[str, List[str], Callable]]`) – New 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 doc/parameter.rst), 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* for more.

  Note: This parameter replaces *eval_metric* in `fit()` method. The old one receives untransformed prediction regardless of whether custom objective is being used.

```python
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]`) – New 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()`.

  The method returns the model from the last iteration (not the best one). 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.

  If early stopping occurs, the model will have three additional fields: `best_score`, `best_iteration` and `best_ntree_limit`.

  Note: This parameter replaces *early_stopping_rounds* in `fit()` method.

- **callbacks** (`Optional[List[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** (*dict, optional*) – 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`:

y_true: array_like of shape [n_samples]
    The target values

y_pred: array_like of shape [n_samples]
    The predicted values

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

---

apply(*X*, *ntree_limit=None*, *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** (*array_like, shape=[n_samples, n_features]*) – Input features matrix.

- **iteration_range** (*Optional[Tuple[int, int]]*) – See *predict()*.

- **ntree_limit** (*Optional[int]*) – Deprecated, use `iteration_range` instead.

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 client: `distributed.Client`

> The dask client used in this model. The *Client* object can not be serialized for transmission, so if task is launched from a worker instead of directly from the client process, this attribute needs to be set at that worker.

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*, *eval_metric=None*, *early_stopping_rounds=None*, *verbose=True*, *xgb_model=None*, *sample_weight_eval_set=None*, *base_margin_eval_set=None*, *feature_weights=None*, *callbacks=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** (*Union[da.Array, dd.DataFrame]*) – Feature matrix. See *Supported data structures for various XGBoost functions* for a list of supported types.
>
>   When the `tree_method` is set to `hist` or `gpu_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 `gpu_hist` is used for training, then the data is first processed on CPU then transferred to GPU.
>
> - **y** (*Union[da.Array, dd.DataFrame, dd.Series]*) – Labels
>
> - **sample_weight** (*Optional[Union[da.Array, dd.DataFrame, dd.Series]]*) – instance weights
>
> - **base_margin** (*Optional[Union[da.Array, dd.DataFrame, dd.Series]]*) – global bias for each instance.
>
> - **eval_set** (*Optional[Sequence[Tuple[Union[da.Array, dd.DataFrame, dd.Series], Union[da.Array, dd.DataFrame, dd.Series]]]]*) – 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_metric** (*str, list of str, or callable, optional*) – Deprecated since version 1.6.0: Use *eval_metric* in `__init__()` or *set_params()* instead.
>
> - **early_stopping_rounds** (*int*) – Deprecated since version 1.6.0: Use *early_stopping_rounds* in `__init__()` or *set_params()* instead.
>
> - **verbose** (*Union[int, bool]*) – 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** (*Optional[Union[Booster, XGBModel]]*) – file name of stored XGBoost model or 'Booster' instance XGBoost model to be loaded before training (allows training continuation).
>
> - **sample_weight_eval_set** (*Optional[Sequence[Union[da.Array, dd.DataFrame, dd.Series]]]*) – 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** (*Optional[Sequence[Union[da.Array, dd.DataFrame, dd.Series]]]*) – 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** (*Optional[Union[da.Array, dd.DataFrame, dd.Series]]*) – 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.
>
> - **callbacks** (*Optional[Sequence[TrainingCallback]]*) – Deprecated since version 1.6.0: Use *callbacks* in `__init__()` or *set_params()* instead.

> > **Return type**
> > *DaskXGBRFRegressor*

> **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_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
> >
> > ---
> >
> > **Note:** Intercept is defined only for linear learners
> >
> > Intercept (bias) is 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**
> > > **intercept_**
> > >
> > > **Return type**
> > > array of shape `(1,)` or `[n_classes]`

> **load_model**(*fname*)
>
> > Load the model from a file or bytearray. Path to file can be local or as an URI.
> >
> > The model is loaded from XGBoost format which is universal among the various XGBoost interfaces. Auxiliary attributes of the Python Booster object (such as feature_names) will not be loaded when using binary format. To save those attributes, use JSON/UBJ instead. See *Model IO* for more info.
> >
> > ```
> > model.load_model("model.json")
> > # or
> > model.load_model("model.ubj")
> > ```

> **Parameters**
>> **fname** (`Union[str, bytearray, PathLike]`) – 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*, *ntree_limit=None*, *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. For tree models, when data is on GPU, like cupy array or cuDF dataframe and *predictor* is not specified, the prediction is run on GPU automatically, otherwise it will run on CPU.
>
> ---
>
> **Note:** This function is only thread safe for *gbtree* and *dart*.
>
> ---
>
> **Parameters**
>
>> - **X** (`Union[da.Array, dd.DataFrame]`) – Data to predict with.
>>
>> - **output_margin** (`bool`) – Whether to output the raw untransformed margin value.
>>
>> - **ntree_limit** (`Optional[int]`) – Deprecated, use *iteration_range* instead.
>>
>> - **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** (`Optional[Union[da.Array, dd.DataFrame, dd.Series]]`) – Margin added to prediction.
>>
>> - **iteration_range** (`Optional[Tuple[int, int]]`) – 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.
>>
>>   New 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) will not be saved when using binary format. To save those attributes, use JSON/UBJ instead. See *Model IO* for more info.
>
> ```
> model.save_model("model.json")
> # or
> model.save_model("model.ubj")
> ```
>
> **Parameters**
>> **fname** (`string or os.PathLike`) – Output file name

> **Return type**
> None

score(*X*, *y*, *sample_weight=None*)

Return the coefficient of determination of the prediction.

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)` wrt. *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_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](#)) –

class xgboost.dask.**DaskXGBRFClassifier**(***, *learning_rate=1*, *subsample=0.8*, *colsample_bynode=0.8*, *reg_lambda=1e-05*, ***kwargs*)

Bases: [DaskXGBClassifier](#)

Implementation of the Scikit-Learn API for XGBoost Random Forest Classifier.

> New in version 1.4.0.

> **Parameters**
>
> - **n_estimators** ([int](#)) – Number of trees in random forest to fit.

- **max_depth** (`Optional[int]`) – Maximum tree depth for base learners.

- **max_leaves** – Maximum number of leaves; 0 indicates no limit.

- **max_bin** – If using histogram-based algorithm, maximum number of bins per feature

- **grow_policy** – Tree growing policy. 0: favor splitting at nodes closest to the node, i.e. grow depth-wise. 1: favor 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`, `Callable[[numpy.ndarray`, `numpy.ndarray]`, `Tuple[numpy.ndarray`, `numpy.ndarray]]`, `NoneType]`) – Specify the learning task and the corresponding learning objective or a custom objective function to be used (see note below).

- **booster** (`Optional[str]`) – Specify which booster to use: gbtree, gblinear or dart.

- **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** –

  Sampling method. Used only by *gpu_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** (`Optional[float]`) – The initial prediction score of all instances, global bias.

- **random_state** (*Optional[Union[*numpy.random.RandomState, *int]]*) – Random number seed.

> **Note:** Using gblinear booster with shotgun updater is nondeterministic as it uses Hogwild algorithm.

- **missing** (*float, default np.nan*) – Value in the data which needs to be present as a missing value.

- **num_parallel_tree** (*Optional[*int*]*) – Used for boosting random forest.

- **monotone_constraints** (*Optional[Union[Dict[*str, int*], str]]*) – Constraint of variable monotonicity. See *tutorial* for more information.

- **interaction_constraints** (*Optional[Union[*str, List[Tuple[*str*]]]]*) – 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.

- **gpu_id** (*Optional[*int*]*) – Device ordinal.

- **validate_parameters** (*Optional[*bool*]*) – Give warnings for unknown parameter.

- **predictor** (*Optional[*str*]*) – Force XGBoost to use specific predictor, available choices are [cpu_predictor, gpu_predictor].

- **enable_categorical** (*bool*) – New in version 1.5.0.

> **Note:** This parameter is experimental

Experimental support for categorical data. When enabled, cudf/pandas.DataFrame should be used to specify categorical data type. Also, JSON/UBJSON serialization format is required.

- **feature_types** (*Optional[FeatureTypes]*) – New in version 1.7.0.

Used for specifying feature types without constructing a dataframe. See DMatrix for details.

- **max_cat_to_onehot** (*Optional[*int*]*) – New 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*]*) – New 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.

- **eval_metric** (*Optional[Union[str, List[str], Callable]]*) – New 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 doc/parameter.rst), 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* for more.

  Note: This parameter replaces *eval_metric* in `fit()` method. The old one receives untransformed prediction regardless of whether custom objective is being used.

  ```python
  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]*) – New 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()`.

  The method returns the model from the last iteration (not the best one). 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.

  If early stopping occurs, the model will have three additional fields: `best_score`, `best_iteration` and `best_ntree_limit`.

  Note: This parameter replaces *early_stopping_rounds* in `fit()` method.

- **callbacks** (*Optional[List[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** (*dict, optional*) – 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`:

y_true: array_like of shape [n_samples]
    The target values

y_pred: array_like of shape [n_samples]
    The predicted values

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

apply(*X*, *ntree_limit=None*, *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** (*array_like, shape=[n_samples, n_features]*) – Input features matrix.

- **iteration_range** (*Optional[Tuple[int, int]]*) – See *predict()*.

- **ntree_limit** (*Optional[int]*) – Deprecated, use `iteration_range` instead.

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 client: `distributed.Client`

The dask client used in this model. The *Client* object can not be serialized for transmission, so if task is launched from a worker instead of directly from the client process, this attribute needs to be set at that worker.

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*, *eval_metric=None*,
early_stopping_rounds=None, *verbose=True*, *xgb_model=None*, *sample_weight_eval_set=None*,
base_margin_eval_set=None, *feature_weights=None*, *callbacks=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** (*Union[da.Array, dd.DataFrame]*) – Feature matrix. See *Supported data structures for various XGBoost functions* for a list of supported types.

  When the `tree_method` is set to `hist` or `gpu_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 `gpu_hist` is used for training, then the data is first processed on CPU then transferred to GPU.

- **y** (*Union[da.Array, dd.DataFrame, dd.Series]*) – Labels

- **sample_weight** (*Optional[Union[da.Array, dd.DataFrame, dd.Series]]*) – instance weights

- **base_margin** (*Optional[Union[da.Array, dd.DataFrame, dd.Series]]*) – global bias for each instance.

- **eval_set** (*Optional[Sequence[Tuple[Union[da.Array, dd.DataFrame, dd.Series], Union[da.Array, dd.DataFrame, dd.Series]]]]*) – 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_metric** (*str, list of str, or callable, optional*) – Deprecated since version 1.6.0: Use *eval_metric* in `__init__()` or *set_params()* instead.

- **early_stopping_rounds** (*int*) – Deprecated since version 1.6.0: Use *early_stopping_rounds* in `__init__()` or *set_params()* instead.

- **verbose** (*Union[int, bool]*) – 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** (*Optional[Union[Booster, XGBModel]]*) – file name of stored XGBoost model or 'Booster' instance XGBoost model to be loaded before training (allows training continuation).

- **sample_weight_eval_set** (*Optional[Sequence[Union[da.Array, dd.DataFrame, dd.Series]]]*) – 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** (*Optional[Sequence[Union[da.Array, dd.DataFrame, dd.Series]]]*) – 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** (*Optional[Union[da.Array, dd.DataFrame, dd.Series]]*) – Weight for each feature, defines the probability of each feature being selected when col-sample is being used. All values must be greater than 0, otherwise a *ValueError* is thrown.

- **callbacks** (*Optional[Sequence[TrainingCallback]]*) – Deprecated since version 1.6.0: Use *callbacks* in `__init__()` or *set_params()* instead.

> > **Return type**
> > *DaskXGBRFClassifier*

> **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_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
>
> ---
>
> **Note:** Intercept is defined only for linear learners
>
> Intercept (bias) is 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**
> > **intercept_**
> >
> > **Return type**
> > array of shape (1,) or [n_classes]

> **load_model**(*fname*)
> > Load the model from a file or bytearray. Path to file can be local or as an URI.
> >
> > The model is loaded from XGBoost format which is universal among the various XGBoost interfaces. Auxiliary attributes of the Python Booster object (such as feature_names) will not be loaded when using binary format. To save those attributes, use JSON/UBJ instead. See *Model IO* for more info.
> >
> > ```
> > model.load_model("model.json")
> > # or
> > model.load_model("model.ubj")
> > ```

Parameters
    fname (`Union[str, bytearray, PathLike]`) – 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*, *ntree_limit=None*, *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. For tree models, when data is on GPU, like cupy array or cuDF dataframe and *predictor* is not specified, the prediction is run on GPU automatically, otherwise it will run on CPU.

Note: This function is only thread safe for *gbtree* and *dart*.

Parameters

- **X** (`Union[da.Array, dd.DataFrame]`) – Data to predict with.

- **output_margin** (`bool`) – Whether to output the raw untransformed margin value.

- **ntree_limit** (`Optional[int]`) – Deprecated, use *iteration_range* instead.

- **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** (`Optional[Union[da.Array, dd.DataFrame, dd.Series]]`) – Margin added to prediction.

- **iteration_range** (`Optional[Tuple[int, int]]`) – 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.

    New in version 1.4.0.

Return type
    prediction

predict_proba(*X*, *ntree_limit=None*, *validate_features=True*, *base_margin=None*, *iteration_range=None*)

Predict the probability of each *X* example being of a given class.

Note: This function is only thread safe for *gbtree* and *dart*.

Parameters

- **X** (`array_like`) – Feature matrix. See *Supported data structures for various XGBoost functions* for a list of supported types.

- **ntree_limit** (`int`) – Deprecated, use *iteration_range* instead.

- **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** (*array_like*) – Margin added to prediction.

- **iteration_range** (`Optional[Tuple[int, int]]`) – 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) will not be saved when using binary format. To save those attributes, use JSON/UBJ instead. See *Model IO* for more info.

```
model.save_model("model.json")
# or
model.save_model("model.ubj")
```

Parameters

fname (`string or os.PathLike`) – Output file name

Return type

None

score(*X*, *y*, *sample_weight=None*)

Return the mean accuracy on the given test 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)` wrt. *y*.

Return type

float

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*) –

PySpark API

PySpark XGBoost integration interface

class xgboost.spark.**SparkXGBClassifier**(***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 predict method.

> SparkXGBClassifier doesn't support setting *gpu_id* but support another param *use_gpu*, see doc below for more details.

> 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**
>>> - **callbacks** – The export and import of the callback functions are at best effort. For details, see `xgboost.spark.SparkXGBClassifier.callbacks` param doc.
>>> - **raw_prediction_col** – The *output_margin=True* is implicitly supported by the *rawPredictionCol* output column, which is always returned with the predicted margin values.
>>> - **validation_indicator_col** – 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** – 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.
>>> - **xgb_model** – Set the value to be the instance returned by `xgboost.spark.SparkXGBClassifierModel.get_booster()`.
>>> - **num_workers** – Integer that specifies the number of XGBoost workers to use. Each XGBoost worker corresponds to one spark task.
>>> - **use_gpu** – Boolean that specifies whether the executors are running on GPU instances.
>>> - **base_margin_col** – 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. Note: this isn't available for distributed training.

- **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 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** (*Union[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.
>
> New 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*.
>
> New 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 (*Union[str, Param[T]]*) –

        Return type

            Union[*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** (*Union[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** (*Union[str, Param[Any]]*) –
>
> **Return type**
>> bool

isSet(*param*)

Checks whether a param is explicitly set by user.

> **Parameters**
>> **param** (*Union[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.

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.

uid

> A unique id for the object.

write()

> Return the writer for saving the estimator.

class xgboost.spark.**SparkXGBClassifierModel**(*xgb_sklearn_model=None*)

> Bases: _SparkXGBModel, HasProbabilityCol, HasRawPredictionCol, HasContribPredictionCol
>
> The model returned by *xgboost.spark.SparkXGBClassifier.fit()*
>
> ---
>
> **Note:** This API is experimental.
>
> ---
>
> **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** (*Union[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 (`Union[str, Param[T]]`) –

Return type
    Union[*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.

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.

hasDefault(*param*)

> Checks whether a param has a default value.
>
> > **Parameters**
> >
> > > **param** (*Union[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 (`Union[str, Param[Any]]`) –

  Return type
   [bool](#)

isSet(*param*)

 Checks whether a param is explicitly set by user.

  Parameters
   param (`Union[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.

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

transform(*dataset*, *params=None*)

 Transforms the input dataset with optional parameters.

 New 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.

class xgboost.spark.**SparkXGBRegressor**(***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 predict method.
>
> SparkXGBRegressor doesn't support setting *gpu_id* but support another param *use_gpu*, see doc below for more details.
>
> 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.
>
> **callbacks:**
>> The export and import of the callback functions are at best effort. For details, see `xgboost.spark.SparkXGBRegressor.callbacks` param doc.
>
> **validation_indicator_col**
>> 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:**
>> 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.
>
> **xgb_model:**
>> Set the value to be the instance returned by `xgboost.spark.SparkXGBRegressorModel.get_booster()`.
>
> **num_workers:**
>> Integer that specifies the number of XGBoost workers to use. Each XGBoost worker corresponds to one spark task.
>
> **use_gpu:**
>> Boolean that specifies whether the executors are running on GPU instances.

base_margin_col:

> 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. Note: this isn't available for distributed training.

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 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** (`Union[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.
>
> New 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*.
>
> New 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** (*Union[str, Param[T]]*) –
>
> **Return type**
> *Union*[*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** (*Union[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** (`Union[str, Param[Any]]`) –

Return type
> bool

isSet(*param*)

Checks whether a param is explicitly set by user.

Parameters
> **param** (`Union[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.

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.

uid

> A unique id for the object.

write()

> Return the writer for saving the estimator.

class xgboost.spark.**SparkXGBRegressorModel**(*xgb_sklearn_model=None*)

> Bases: _SparkXGBModel
>
> The model returned by *xgboost.spark.SparkXGBRegressor.fit()*
>
> ---
>
> **Note:** This API is experimental.
>
> ---
>
> **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** (*Union[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** (`Union[str, Param[T]]`) –
> >
> > **Return type**
> >
> > > *Union*[*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.

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.

hasDefault(*param*)

> Checks whether a param has a default value.
>
> > **Parameters**
> > **param** (`Union[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** (`Union[str, Param[Any]]`) –
> >
> > **Return type**
> > bool

isSet(*param*)

> Checks whether a param is explicitly set by user.
>
> > **Parameters**
> > **param** (`Union[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.

> **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

> **transform**(*dataset*, *params=None*)

>> Transforms the input dataset with optional parameters.

>> New 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.

class xgboost.spark.**SparkXGBRanker**(***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 predict method.

SparkXGBRanker doesn't support setting *gpu_id* but support another param *use_gpu*, see doc below for more details.

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

- **callbacks** – The export and import of the callback functions are at best effort. For details, see `xgboost.spark.SparkXGBRanker.callbacks` param doc.

- **validation_indicator_col** – For params related to *xgboost.XGBRanker* training with evaluation dataset's supervision, set `xgboost.spark.XGBRanker.validation_indicator_col` parameter instead of setting the *eval_set* parameter in *xgboost.XGBRanker* fit method.

- **weight_col** – 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.

- **xgb_model** – Set the value to be the instance returned by [*xgboost.spark.SparkXGBRankerModel.get_booster()*](#).

- **num_workers** – Integer that specifies the number of XGBoost workers to use. Each XGBoost worker corresponds to one spark task.

- **use_gpu** – Boolean that specifies whether the executors are running on GPU instances.

- **base_margin_col** – 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** – To specify the qid of the training and validation dataset, set `xgboost.spark.SparkXGBRanker.qid_col` parameter instead of setting *qid* / *group*, *eval_qid* / *eval_group* in the *xgboost.XGBRanker* fit method.

- **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),
```

(continues on next page)

```
...            (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** (*Union[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.
>
> New 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*.
>
> New 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** (`Union[str, Param[T]]`) –
> >
> > **Return type**
> > > *Union*[*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** (`Union[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** (`Union[str, Param[Any]]`) –

> **Return type**
> [bool](#)

isSet(*param*)

Checks whether a param is explicitly set by user.

> **Parameters**
> **param** (`Union[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.

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.

uid

A unique id for the object.

write()

> Return the writer for saving the estimator.

class xgboost.spark.**SparkXGBRankerModel**(*xgb_sklearn_model=None*)

> Bases: _SparkXGBModel
>
> The model returned by [*xgboost.spark.SparkXGBRanker.fit()*](#)

Note: This API is experimental.

> **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** ([*Union[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** (*Union[str, Param[T]]*) –
>
> > **Return type**
> > *Union*[*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.

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.

hasDefault(*param*)

> Checks whether a param has a default value.

>> **Parameters**
>>> **param** (`Union[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** (`Union[str, Param[Any]]`) –

>> **Return type**
>>> bool

isSet(*param*)

> Checks whether a param is explicitly set by user.

>> **Parameters**
>>> **param** (`Union[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.

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

transform(*dataset*, *params=None*)

>   Transforms the input dataset with optional parameters.
>
>   New 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.

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:

```python
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)
    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='Train')

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*.

Model

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
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)

# 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
```

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.

XGBoost Python Feature Walkthrough

This is a collection of examples for using the XGBoost Python package.

Demo for using xgboost with sklearn

```python
import multiprocessing

from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import GridSearchCV

import xgboost as xgb

if __name__ == "__main__":
    print("Parallel Parameter optimization")
    X, y = fetch_california_housing(return_X_y=True)
    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_)
```

Total running time of the script: (0 minutes 0.000 seconds)

Demo for obtaining leaf index

```python
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'))
dtest = xgb.DMatrix(os.path.join(CURRENT_DIR, '../data/agaricus.txt.test'))
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 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)
```

Total running time of the script: (0 minutes 0.000 seconds)

This script demonstrate how to access the eval metrics

```python
import os

import xgboost as xgb

CURRENT_DIR = os.path.dirname(__file__)
dtrain = xgb.DMatrix(os.path.join(CURRENT_DIR, '../data/agaricus.txt.train'))
dtest = xgb.DMatrix(os.path.join(CURRENT_DIR, '../data/agaricus.txt.test'))
```

```python
param = [('max_depth', 2), ('objective', 'binary:logistic'), ('eval_metric', 'logloss'),
→('eval_metric', 'error')]

num_round = 2
watchlist = [(dtest,'eval'), (dtrain,'train')]

evals_result = {}
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))
        print('      - {}'.format(e_mtr_vals))

print('')
print('Access complete dictionary:')
print(evals_result)
```

Total running time of the script: (0 minutes 0.000 seconds)

Demo for gamma regression

```python
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
# 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))))
```

Total running time of the script: (0 minutes 0.000 seconds)

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'))
dtest = xgb.DMatrix(os.path.join(CURRENT_DIR, '../data/agaricus.txt.test'))
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)
```

Total running time of the script: (0 minutes 0.000 seconds)

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}
```

```python
labels, y = np.unique(y, return_inverse=True)

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)
# 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)],
        eval_metric='logloss',
        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)
```

Total running time of the script: (0 minutes 0.000 seconds)

Demo for using feature weight to change column sampling

New in version 1.3.0.

```python
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)

    kRows = 4196
```

```python
    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)
```

Total running time of the script: (0 minutes 0.000 seconds)

Demo for GLM

```python
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
##
CURRENT_DIR = os.path.dirname(__file__)
dtrain = xgb.DMatrix(os.path.join(CURRENT_DIR, '../data/agaricus.txt.train'))
dtest = xgb.DMatrix(os.path.join(CURRENT_DIR, '../data/agaricus.txt.test'))
# 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))))
```

Total running time of the script: (0 minutes 0.000 seconds)

Demo for prediction using number of trees

```python
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")
```

```python
def native_interface():
    # load data in do training
    dtrain = xgb.DMatrix(train)
    dtest = xgb.DMatrix(test)
    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():
    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)))
    )


if __name__ == "__main__":
    native_interface()
    sklearn_interface()
```

Total running time of the script: (0 minutes 0.000 seconds)

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.

```python
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))
    )
)
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
```

```python
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
```

Total running time of the script: (0 minutes 0.000 seconds)

Getting started with categorical data

Experimental support for categorical data. After 1.5 XGBoost *gpu_hist* tree method has experimental support for one-hot encoding based tree split, and in 1.6 *approx* support was added.

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.

Also, see *the tutorial* for using XGBoost with categorical data.

New in version 1.5.0.

```python
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]:
    """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)
```

```python
        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 must be pandas DataFrame or cudf
    # DataFrame with categorical features
    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="gpu_hist", enable_categorical=True, max_cat_to_onehot=5
    )
    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="gpu_hist")
    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()
    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()
```

Total running time of the script: (0 minutes 0.000 seconds)

Demo for using cross validation

```python
import os

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'))
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.
# as a example, we try to set scale_pos_weight
def fpreproc(dtrain, dtest, param):
    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
##
```

```python
print('running cross validation, with customized loss function')
def logregobj(preds, dtrain):
    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, dtrain):
    labels = dtrain.get_label()
    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, feval=evalerror)
```

Total running time of the script: (0 minutes 0.000 seconds)

Collection of examples for using sklearn interface

Created on 1 Apr 2015

@author: Jamie Hall

```python
import pickle

import numpy as np
from sklearn.datasets import fetch_california_housing, load_digits, load_iris
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):
```

```python
    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")
X, y = fetch_california_housing(return_X_y=True)
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)
clf.fit(X_train, y_train, early_stopping_rounds=10, eval_metric="auc",
        eval_set=[(X_test, y_test)])
```

Total running time of the script: (0 minutes 0.000 seconds)

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.

> New in version 1.5.0.

See *the tutorial* for more details.

```python
import os
import tempfile
```

```python
from typing import Callable, List, Tuple

import numpy as np
from sklearn.datasets import make_regression

import xgboost


def make_batches(
    n_samples_per_batch: int, n_features: int, n_batches: int, tmpdir: str,
) -> List[Tuple[str, str]]:
    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(tmpdir, "X-" + str(i) + ".npy")
        y_path = os.path.join(tmpdir, "y-" + str(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, file_paths: List[Tuple[str, str]]):
        self._file_paths = file_paths
        self._it = 0
        # XGBoost will generate some cache files under current directory with the prefix
        # "cache"
        super().__init__(cache_prefix=os.path.join(".", "cache"))

    def load_file(self) -> Tuple[np.ndarray, np.ndarray]:
        X_path, y_path = self._file_paths[self._it]
        X = np.load(X_path)
        y = np.load(y_path)
        assert X.shape[0] == y.shape[0]
        return X, y

    def next(self, input_data: Callable) -> int:
        """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 0 to let XGBoost know this is the end of iteration
            return 0

        # input_data is a function passed in by XGBoost who has the similar signature to
        # the ``DMatrix`` constructor.
        X, y = self.load_file()
        input_data(data=X, label=y)
```

```python
        self._it += 1
        return 1

    def reset(self) -> None:
        """Reset the iterator to its beginning"""
        self._it = 0


def main(tmpdir: str) -> xgboost.Booster:
    # generate some random data for demo
    files = make_batches(1024, 17, 31, tmpdir)
    it = Iterator(files)
    # For non-data arguments, specify it here once instead of passing them by the `next`
    # method.
    missing = np.NaN
    Xy = xgboost.DMatrix(it, missing=missing, enable_categorical=False)

    # Other tree methods including ``hist`` and ``gpu_hist`` also work, see tutorial in
    # doc for details.
    booster = xgboost.train(
        {"tree_method": "approx", "max_depth": 2},
        Xy,
        evals=[(Xy, "Train")],
        num_boost_round=10,
    )
    return booster


if __name__ == "__main__":
    with tempfile.TemporaryDirectory() as tmpdir:
        main(tmpdir)
```

Total running time of the script: (0 minutes 0.000 seconds)

Demo for using data iterator with Quantile DMatrix

New 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 used 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 XG-Boost 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.

```python
import cupy
import numpy

import xgboost

COLS = 64
```

```python
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):
        '''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(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):
        return cupy.concatenate(self._data)

    def as_array_labels(self):
        return cupy.concatenate(self._labels)

    def as_array_weights(self):
        return cupy.concatenate(self._weights)

    def data(self):
        '''Utility function for obtaining current batch of data.'''
        return self._data[self.it]

    def labels(self):
        '''Utility function for obtaining current batch of label.'''
        return self._labels[self.it]

    def weights(self):
        return self._weights[self.it]

    def reset(self):
        '''Reset the iterator'''
        self.it = 0

    def next(self, input_data):
        '''Yield next batch of data.'''
```

```python
        if self.it == len(self._data):
            # Return 0 when there's no more batch.
            return 0
        input_data(data=self.data(), label=self.labels(),
                   weight=self.weights())
        self.it += 1
        return 1


def main():
    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 meethod must be one of the `hist` or `gpu_hist`. We use `gpu_hist` for GPU
    # input here.
    reg_with_it = xgboost.train(
        {"tree_method": "gpu_hist"}, m_with_it, num_boost_round=rounds
    )
    predict_with_it = reg_with_it.predict(m_with_it)

    reg = xgboost.train({"tree_method": "gpu_hist"}, m, num_boost_round=rounds)
    predict = reg.predict(m)

    numpy.testing.assert_allclose(predict_with_it, predict, rtol=1e6)


if __name__ == '__main__':
    main()
```

Total running time of the script: (0 minutes 0.000 seconds)

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.

```python
import numpy as np
from sklearn.datasets import fetch_california_housing

import xgboost as xgb


def main():
    n_rounds = 32

    X, y = fetch_california_housing(return_X_y=True)

    # 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": "gpu_hist", "max_depth": 6},
        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,
        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: xgb.callback.EvaluationMonitor.EvalsLog = {}
    refreshed = xgb.train(
        {"process_type": "update", "updater": "refresh", "refresh_leaf": False},
        Xy_refresh,
        num_boost_round=n_rounds,
        xgb_model=booster,
```

(continues on next page)

```python
        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()
```

Total running time of the script: (0 minutes 0.000 seconds)

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

Also, see the tutorial for using XGBoost with categorical data: *Categorical Data*.

```python
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"]

    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": "gpu_hist",
    "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"""
```

(continues on next page)

```python
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, random_state=1994, test_size=0.2
    )
    # 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:
    """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)
```

Total running time of the script: (0 minutes 0.000 seconds)

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.

```python
import argparse
from typing import Dict, List, Tuple

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) -> None:
    s = 25
    plt.scatter(y[:, 0], y[:, 1], c="navy", s=s, edgecolor="black", label="data")
    plt.scatter(
        y_predt[:, 0], y_predt[:, 1], c="cornflowerblue", s=s, edgecolor="black"
    )
    plt.xlim([-1, 2])
    plt.ylim([-1, 2])
    plt.show()


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(plot_result: bool):
    """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=64)
    reg.fit(X, y, eval_set=[(X, y)])

    y_predt = reg.predict(X)
    if plot_result:
        plot_predt(y, y_predt, "multi")


def custom_rmse_model(plot_result: bool) -> None:
    """Train using Python implementation of Squared Error."""
```

(continues on next page)

```python
    # As the experimental support status, custom objective doesn't support matrix as
    # gradient and hessian, which will be changed in future release.
    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).reshape(y.size)

    def hessian(predt: np.ndarray, dtrain: xgb.DMatrix) -> np.ndarray:
        """Compute the hessian for squared error."""
        return np.ones(predt.shape).reshape(predt.size)

    def squared_log(
        predt: np.ndarray, dtrain: xgb.DMatrix
    ) -> Tuple[np.ndarray, np.ndarray]:
        grad = gradient(predt, dtrain)
        hess = hessian(predt, dtrain)
        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.sum(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],
        },
        dtrain=Xy,
        num_boost_round=100,
        obj=squared_log,
        evals=[(Xy, "Train")],
        evals_result=results,
        custom_metric=rmse,
    )

    y_predt = booster.inplace_predict(X)
    if plot_result:
        plot_predt(y, y_predt, "multi")


if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--plot", choices=[0, 1], type=int, default=1)
    args = parser.parse_args()
```

```
    # Train with builtin RMSE objective
    rmse_model(args.plot == 1)
    # Train with custom objective.
    custom_rmse_model(args.plot == 1)
```

Total running time of the script: (0 minutes 0.000 seconds)

Collection of examples for using xgboost.spark estimator interface

@author: Weichen Xu

```python
import sklearn.datasets
from pyspark.ml.evaluation import MulticlassClassificationEvaluator, RegressionEvaluator
from pyspark.ml.linalg import Vectors
from pyspark.sql import 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, y):
    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)

# 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)}"
)
```

```python
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
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()
```

Total running time of the script: (0 minutes 0.000 seconds)

Demo for training continuation

```python
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)
    clf.fit(X, y, eval_set=[(X, y)], eval_metric="logloss")
    print("Total boosted rounds:", clf.get_booster().num_boosted_rounds())

    # 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)
    clf.fit(X, y, eval_set=[(X, y)], eval_metric="logloss")
    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)
    clf.fit(X, y, eval_set=[(X, y)], eval_metric="logloss", 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
    )
```

(continues on next page)

```python
    n_estimators = 512

    X, y = load_breast_cancer(return_X_y=True)
    clf = xgboost.XGBClassifier(n_estimators=n_estimators)
    clf.fit(X, y, eval_set=[(X, y)], eval_metric="logloss", callbacks=[early_stop])
    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)
    # Reinitialize the early stop callback
    early_stop = xgboost.callback.EarlyStopping(
        rounds=early_stopping_rounds, save_best=True
    )
    clf.fit(X, y, eval_set=[(X, y)], eval_metric="logloss", callbacks=[early_stop])
    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)
    clf.fit(
        X,
        y,
        eval_set=[(X, y)],
        eval_metric="logloss",
        callbacks=[early_stop],
        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)
```

Total running time of the script: (0 minutes 0.000 seconds)

Demo for using and defining callback functions

New in version 1.3.0.

```python
import argparse
import os
import tempfile

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.

    '''
    def __init__(self, rounds):
        self.fig = plt.figure()
        self.ax = self.fig.add_subplot(111)
        self.rounds = rounds
        self.lines = {}
        self.fig.show()
        self.x = np.linspace(0, self.rounds, self.rounds)
        plt.ion()

    def _get_key(self, data, metric):
        return f'{data}-{metric}'

    def after_iteration(self, model, epoch, evals_log):
        '''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():
```

```python
            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():
    '''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': 'gpu_hist'
        },
        D_train,
        evals=[(D_train, 'Train'), (D_valid, 'Valid')],
        num_boost_round=num_boost_round,
        callbacks=[plotting])


def check_point_callback():
    # only for demo, set a larger value (like 100) in practice as checkpointing is quite
    # slow.
    rounds = 2

    def check(as_pickle):
        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, 'model_' + str(i) + '.json')
            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
```

```python
    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,
                                                      iterations=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,
                                                      iterations=rounds,
                                                      as_pickle=True,
                                                      name='model')
        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()
```

Total running time of the script: (0 minutes 0.000 seconds)

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* for detailed tutorial and notes.

```python
import argparse

import numpy as np
from matplotlib import pyplot as plt

import xgboost as xgb

np.random.seed(1994)
```

```python
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):
    '''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):
    '''Loss function.  Computing the gradient and approximated hessian (diagonal).
    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, 1), 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, 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 = labels[r]
        p = softmax(predt[r, :])
        for c in range(predt.shape[1]):
            assert target >= 0 or target <= kClasses
```

```python
            g = p[c] - 1.0 if c == target else p[c]
            g = g * weights[r]
            h = max((2.0 * p[c] * (1.0 - p[c]) * weights[r]).item(), eps)
            grad[r, c] = g
            hess[r, c] = h

    # Right now (XGBoost 1.0.0), reshaping is necessary
    grad = grad.reshape((kRows * kClasses, 1))
    hess = hess.reshape((kRows * kClasses, 1))
    return grad, hess


def predict(booster: xgb.Booster, X):
    '''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)
    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):
    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, native_results):
```

```python
    fig, axs = plt.subplots(2, 1)
    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):
    custom_results = {}
    # Use our custom objective function
    booster_custom = xgb.train({'num_class': kClasses,
                                '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 = {}
    # Use the same objective function defined in XGBoost.
    booster_native = xgb.train({'num_class': kClasses,
                                "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__':
```

```
    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)
```

Total running time of the script: (0 minutes 0.000 seconds)

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.

```python
import argparse
from time import time
from typing import Dict, List, Tuple

import matplotlib
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

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)
```

```python
    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]]]:
    '''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',
```

```python
        '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
        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,
```

```python
                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, rmsle_evals, py_rmsle_evals):
    fig, axs = plt.subplots(3, 1)
    ax0: matplotlib.axes.Axes = axs[0]
    ax1: matplotlib.axes.Axes = axs[1]
    ax2: matplotlib.axes.Axes = 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):
    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()


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)
```

Total running time of the script: (0 minutes 0.000 seconds)

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

```python
from dask import array as da
from dask.distributed import Client, LocalCluster

import xgboost


def main(client):
    # 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 = xgboost.dask.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)
```

Total running time of the script: (0 minutes 0.000 seconds)

Use scikit-learn regressor interface with GPU histogram tree method

```python
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

import xgboost


def main(client):
    # generate some random data for demonstration
    n = 100
    m = 1000000
    partition_size = 10000
    X = da.random.random((m, n), partition_size)
    y = da.random.random(m, partition_size)

    regressor = xgboost.dask.DaskXGBRegressor(verbosity=1)
    regressor.set_params(tree_method='gpu_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__':
    # With dask cuda, one can scale up XGBoost to arbitrary GPU clusters.
    # `LocalCUDACluster` used here is only for demonstration purpose.
    with LocalCUDACluster() as cluster:
        with Client(cluster) as client:
            main(client)
```

Total running time of the script: (0 minutes 0.000 seconds)

Example of training with Dask on CPU

```python
from dask import array as da
from dask.distributed import Client, LocalCluster
from xgboost.dask import DaskDMatrix

import xgboost as xgb


def main(client):
    # generate some random data for demonstration
    m = 100000
    n = 100
    X = da.random.random(size=(m, n), chunks=100)
    y = da.random.random(size=(m, ), chunks=100)

    # 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 = xgb.dask.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 = xgb.dask.predict(client, bst, dtrain)
    print('Evaluation history:', history)
    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)
```

Total running time of the script: (0 minutes 0.000 seconds)

Example of training survival model with Dask on CPU

```python
import os

import dask.dataframe as dd
from dask.distributed import Client, LocalCluster
from xgboost.dask import DaskDMatrix

import xgboost as xgb


def main(client):
    # 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
    # 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 = xgb.dask.train(client,
                            params,
                            dtrain,
                            num_boost_round=100,
                            evals=[(dtrain, 'train')])
    bst = output['booster']
    history = output['history']
```

```python
    # you can pass output directly into `predict` too.
    prediction = xgb.dask.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)
```

Total running time of the script: (0 minutes 0.000 seconds)

Example of training with Dask on GPU

```python
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.dask import DaskDMatrix

import xgboost as xgb
from xgboost import dask as dxgb


def using_dask_matrix(client: Client, X, y):
    # 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 = xgb.dask.train(client,
                            {'verbosity': 2,
                             # Golden line for GPU training
                             'tree_method': 'gpu_hist'},
                            dtrain,
                            num_boost_round=4, evals=[(dtrain, 'train')])
    bst = output['booster']
    history = output['history']

    # you can pass output directly into `predict` too.
    prediction = xgb.dask.predict(client, bst, dtrain)
```

```python
    print('Evaluation history:', history)
    return prediction


def using_quantile_device_dmatrix(client: Client, X, y):
    """`DaskQuantileDMatrix` is a data type specialized for `gpu_hist`, tree
     method that reduces memory overhead.  When training on GPU pipeline, it's
     preferred over `DaskDMatrix`.

    .. versionadded:: 1.2.0

    """
    # Input must be on GPU for `DaskQuantileDMatrix`.
    X = dask_cudf.from_dask_dataframe(dd.from_dask_array(X))
    y = dask_cudf.from_dask_dataframe(dd.from_dask_array(y))

    # `DaskQuantileDMatrix` is used instead of `DaskDMatrix`, be careful
    # that it can not be used for anything else other than training.
    dtrain = dxgb.DaskQuantileDMatrix(client, X, y)
    output = xgb.dask.train(
        client, {"verbosity": 2, "tree_method": "gpu_hist"}, dtrain, num_boost_round=4
    )

    prediction = xgb.dask.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:
        with Client(cluster) as client:
            # generate some random data for demonstration
            m = 100000
            n = 100
            X = da.random.random(size=(m, n), chunks=10000)
            y = da.random.random(size=(m, ), chunks=10000)

            print('Using DaskQuantileDMatrix')
            from_ddqdm = using_quantile_device_dmatrix(client, X, y)
            print('Using DMatrix')
            from_dmatrix = using_dask_matrix(client, X, y)
```

Total running time of the script: (0 minutes 0.000 seconds)

Example of using callbacks with Dask

```python
import numpy as np
from dask.distributed import Client, LocalCluster
from dask_ml.datasets import make_regression
from dask_ml.model_selection import train_test_split
from xgboost.dask import DaskDMatrix

import xgboost as xgb


def probability_for_going_backward(epoch):
    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, target_metric, maximize, seed):
        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, epoch, evals_log):
        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)

```python
def main(client):
    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 = xgb.dask.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)
```

Total running time of the script: (0 minutes 0.000 seconds)

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.

```python
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
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)})
```

```python
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')
```

Total running time of the script: (0 minutes 0.000 seconds)

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.

```python
import matplotlib.pyplot as plt
import numpy as np

import xgboost as xgb

plt.rcParams.update({'font.size': 13})

# Function to visualize censored labels
def plot_censored_labels(X, y_lower, y_upper):
    def replace_inf(x, target_value):
        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),
               label='Range for y', color='gray')

# 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))
```

```python
# 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 = []
def plot_intermediate_model_callback(env):
    """Custom callback to plot intermediate models"""
    # Compute y_pred = prediction using the intermediate model, at current boosting␣
→iteration
    y_pred = env.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)

    # Plot ranged labels as well as predictions by the model
    plt.subplot(5, 3, env.iteration + 1)
    plot_censored_labels(X, y_lower, y_upper)
    y_pred_grid_pts = env.model.predict(xgb.DMatrix(grid_pts))
    plt.plot(grid_pts, y_pred_grid_pts, 'r-', label='XGBoost AFT model', linewidth=4)
    plt.title('Iteration {}'.format(env.iteration), 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')

res = {}
plt.figure(figsize=(12,13))
bst = xgb.train(params, dmat, 15, [(dmat, 'train')], evals_result=res,
                callbacks=[plot_intermediate_model_callback])
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')
```

```python
plt.legend(loc='best')
plt.tight_layout()

plt.show()
```

Total running time of the script: (0 minutes 0.000 seconds)

Demo for survival analysis (regression) with Optuna.

Demo for survival analysis (regression) using Accelerated Failure Time (AFT) model, using Optuna to tune hyperparameters

```python
import numpy as np
import optuna
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)
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_
→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')
```

Total running time of the script: (0 minutes 0.000 seconds)

1.11 XGBoost R Package

You have found the XGBoost R Package!

1.11.1 Get Started

- Checkout the *Installation Guide* contains instructions to install xgboost, and *Tutorials* for examples on how to use XGBoost for various tasks.

- Read the API documentation.

- Please visit Walk-through Examples.

1.11.2 Tutorials

XGBoost R Tutorial

Introduction

XGBoost is short for e**X**treme **G**radient **Boost**ing package.

The purpose of this Vignette is to show you how to use **XGBoost** to build a model and make predictions.

It is an efficient and scalable implementation of gradient boosting framework by @friedman2000additive and @friedman2001greedy. Two solvers are included:

- *linear* model ;

- *tree learning* algorithm.

It supports various objective functions, including *regression*, *classification* and *ranking*. The package is made to be extendible, so that users are also allowed to define their own objective functions easily.

It has been used to win several Kaggle competitions.

It has several features:

- Speed: it can automatically do parallel computation on *Windows* and *Linux*, with *OpenMP*. It is generally over 10 times faster than the classical `gbm`.

- Input Type: it takes several types of input data:

    - *Dense* Matrix: *R*'s *dense* matrix, i.e. `matrix` ;

    - *Sparse* Matrix: *R*'s *sparse* matrix, i.e. `Matrix::dgCMatrix` ;

    - Data File: local data files ;

    - `xgb.DMatrix`: its own class (recommended).

- Sparsity: it accepts *sparse* input for both *tree booster* and *linear booster*, and is optimized for *sparse* input ;

- Customization: it supports customized objective functions and evaluation functions.

Installation

GitHub version

For weekly updated version (highly recommended), install from *GitHub*:

```
install.packages("drat", repos="https://cran.rstudio.com")
drat:::addRepo("dmlc")
install.packages("xgboost", repos="http://dmlc.ml/drat/", type = "source")
```

> *Windows* users will need to install Rtools first.

CRAN version

The version 0.4-2 is on CRAN, and you can install it by:

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

Formerly available versions can be obtained from the CRAN archive

Learning

For the purpose of this tutorial we will load **XGBoost** package.

```
require(xgboost)
```

Dataset presentation

In this example, we are aiming to predict whether a mushroom can be eaten or not (like in many tutorials, example data are the same as you will use on in your every day life :-).

Mushroom data is cited from UCI Machine Learning Repository. @Bache+Lichman:2013.

Dataset loading

We will load the `agaricus` datasets embedded with the package and will link them to variables.

The datasets are already split in:

- `train`: will be used to build the model ;
- `test`: will be used to assess the quality of our model.

Why *split* the dataset in two parts?

In the first part we will build our model. In the second part we will want to test it and assess its quality. Without dividing the dataset we would test the model on the data which the algorithm have already seen.

```
data(agaricus.train, package='xgboost')
data(agaricus.test, package='xgboost')
train <- agaricus.train
test <- agaricus.test
```

In the real world, it would be up to you to make this division between `train` and `test` data. The way to do it is out of scope for this article, however `caret` package may help.

Each variable is a `list` containing two things, `label` and `data`:

```
str(train)
```

```
## List of 2
##  $ data :Formal class 'dgCMatrix' [package "Matrix"] with 6 slots
##   .. ..@ i       : int [1:143286] 2 6 8 11 18 20 21 24 28 32 ...
##   .. ..@ p       : int [1:127] 0 369 372 3306 5845 6489 6513 8380 8384 10991 ...
##   .. ..@ Dim     : int [1:2] 6513 126
##   .. ..@ Dimnames:List of 2
##   .. .. ..$ : NULL
##   .. .. ..$ : chr [1:126] "cap-shape=bell" "cap-shape=conical" "cap-shape=convex"
→"cap-shape=flat" ...
##   .. ..@ x       : num [1:143286] 1 1 1 1 1 1 1 1 1 1 ...
##   .. ..@ factors : list()
##  $ label: num [1:6513] 1 0 0 1 0 0 0 1 0 0 ...
```

`label` is the outcome of our dataset meaning it is the binary *classification* we will try to predict.

Let's discover the dimensionality of our datasets.

```
dim(train$data)
```

```
## [1] 6513  126
```

```
dim(test$data)
```

```
## [1] 1611  126
```

This dataset is very small to not make the **R** package too heavy, however **XGBoost** is built to manage huge datasets very efficiently.

As seen below, the `data` are stored in a `dgCMatrix` which is a *sparse* matrix and `label` vector is a `numeric` vector (`{0,1}`):

```
class(train$data)[1]
```

```
## [1] "dgCMatrix"
```

```
class(train$label)
```

```
## [1] "numeric"
```

Basic Training using XGBoost

This step is the most critical part of the process for the quality of our model.

Basic training

We are using the `train` data. As explained above, both `data` and `label` are stored in a `list`.

In a *sparse* matrix, cells containing `0` are not stored in memory. Therefore, in a dataset mainly made of `0`, memory size is reduced. It is very common to have such a dataset.

We will train decision tree model using the following parameters:

- `objective = "binary:logistic"`: we will train a binary classification model ;
- `max.depth = 2`: the trees won't be deep, because our case is very simple ;
- `nthread = 2`: the number of CPU threads we are going to use;
- `nrounds = 2`: there will be two passes on the data, the second one will enhance the model by further reducing the difference between ground truth and prediction.

```
bstSparse <- xgboost(data = train$data, label = train$label, max.depth = 2, eta = 1,
→nthread = 2, nrounds = 2, objective = "binary:logistic")
```

```
## [0]      train-error:0.046522
## [1]      train-error:0.022263
```

The more complex the relationship between your features and your `label` is, the more passes you need.

Parameter variations

Dense matrix

Alternatively, you can put your dataset in a *dense* matrix, i.e. a basic **R** matrix.

```
bstDense <- xgboost(data = as.matrix(train$data), label = train$label, max.depth = 2,
→eta = 1, nthread = 2, nrounds = 2, objective = "binary:logistic")
```

```
## [0]      train-error:0.046522
## [1]      train-error:0.022263
```

xgb.DMatrix

XGBoost offers a way to group them in a `xgb.DMatrix`. You can even add other meta data in it. This will be useful for the most advanced features we will discover later.

```
dtrain <- xgb.DMatrix(data = train$data, label = train$label)
bstDMatrix <- xgboost(data = dtrain, max.depth = 2, eta = 1, nthread = 2, nrounds = 2,
→objective = "binary:logistic")
```

```
## [0]          train-error:0.046522
## [1]          train-error:0.022263
```

Verbose option

XGBoost has several features to help you view the learning progress internally. The purpose is to help you to set the best parameters, which is the key of your model quality.

One of the simplest way to see the training progress is to set the `verbose` option (see below for more advanced techniques).

```
# verbose = 0, no message
bst <- xgboost(data = dtrain, max.depth = 2, eta = 1, nthread = 2, nrounds = 2,␣
→objective = "binary:logistic", verbose = 0)
```

```
# verbose = 1, print evaluation metric
bst <- xgboost(data = dtrain, max.depth = 2, eta = 1, nthread = 2, nrounds = 2,␣
→objective = "binary:logistic", verbose = 1)
```

```
## [0]          train-error:0.046522
## [1]          train-error:0.022263
```

```
# verbose = 2, also print information about tree
bst <- xgboost(data = dtrain, max.depth = 2, eta = 1, nthread = 2, nrounds = 2,␣
→objective = "binary:logistic", verbose = 2)
```

```
## [11:41:01] amalgamation/../src/tree/updater_prune.cc:74: tree pruning end, 1 roots, 6␣
→extra nodes, 0 pruned nodes, max_depth=2
## [0]          train-error:0.046522
## [11:41:01] amalgamation/../src/tree/updater_prune.cc:74: tree pruning end, 1 roots, 4␣
→extra nodes, 0 pruned nodes, max_depth=2
## [1]          train-error:0.022263
```

Basic prediction using XGBoost

Perform the prediction

The purpose of the model we have built is to classify new data. As explained before, we will use the `test` dataset for this step.

```
pred <- predict(bst, test$data)

# size of the prediction vector
print(length(pred))
```

```
## [1] 1611
```

```
# limit display of predictions to the first 10
print(head(pred))
```

```
## [1] 0.28583017 0.92392391 0.28583017 0.28583017 0.05169873 0.92392391
```

These numbers doesn't look like *binary classification* {0,1}. We need to perform a simple transformation before being able to use these results.

Transform the regression in a binary classification

The only thing that **XGBoost** does is a *regression*. **XGBoost** is using label vector to build its *regression* model.

How can we use a *regression* model to perform a binary classification?

If we think about the meaning of a regression applied to our data, the numbers we get are probabilities that a datum will be classified as 1. Therefore, we will set the rule that if this probability for a specific datum is > 0.5 then the observation is classified as 1 (or 0 otherwise).

```
prediction <- as.numeric(pred > 0.5)
print(head(prediction))
```

```
## [1] 0 1 0 0 0 1
```

Measuring model performance

To measure the model performance, we will compute a simple metric, the *average error*.

```
err <- mean(as.numeric(pred > 0.5) != test$label)
print(paste("test-error=", err))
```

```
## [1] "test-error= 0.0217256362507759"
```

> Note that the algorithm has not seen the test data during the model construction.

Steps explanation:

1. `as.numeric(pred > 0.5)` applies our rule that when the probability (<=> regression <=> prediction) is > 0.5 the observation is classified as 1 and 0 otherwise ;

2. `probabilityVectorPreviouslyComputed != test$label` computes the vector of error between true data and computed probabilities ;

3. `mean(vectorOfErrors)` computes the *average error* itself.

The most important thing to remember is that **to do a classification, you just do a regression to the** label **and then apply a threshold**.

Multiclass classification works in a similar way.

This metric is **0.02** and is pretty low: our yummly mushroom model works well!

Advanced features

Most of the features below have been implemented to help you to improve your model by offering a better understanding of its content.

Dataset preparation

For the following advanced features, we need to put data in `xgb.DMatrix` as explained above.

```
dtrain <- xgb.DMatrix(data = train$data, label=train$label)
dtest <- xgb.DMatrix(data = test$data, label=test$label)
```

Measure learning progress with xgb.train

Both `xgboost` (simple) and `xgb.train` (advanced) functions train models.

One of the special features of `xgb.train` is the capacity to follow the progress of the learning after each round. Because of the way boosting works, there is a time when having too many rounds lead to overfitting. You can see this feature as a cousin of a cross-validation method. The following techniques will help you to avoid overfitting or optimizing the learning time in stopping it as soon as possible.

One way to measure progress in the learning of a model is to provide to **XGBoost** a second dataset already classified. Therefore it can learn on the first dataset and test its model on the second one. Some metrics are measured after each round during the learning.

> in some way it is similar to what we have done above with the average error. The main difference is that above it was after building the model, and now it is during the construction that we measure errors.

For the purpose of this example, we use `watchlist` parameter. It is a list of `xgb.DMatrix`, each of them tagged with a name.

```
watchlist <- list(train=dtrain, test=dtest)

bst <- xgb.train(data=dtrain, max.depth=2, eta=1, nthread = 2, nrounds=2,␣
→watchlist=watchlist, objective = "binary:logistic")
```

```
## [0]    train-error:0.046522    test-error:0.042831
## [1]    train-error:0.022263    test-error:0.021726
```

XGBoost has computed at each round the same average error metric seen above (we set `nrounds` to 2, that is why we have two lines). Obviously, the `train-error` number is related to the training dataset (the one the algorithm learns from) and the `test-error` number to the test dataset.

Both training and test error related metrics are very similar, and in some way, it makes sense: what we have learned from the training dataset matches the observations from the test dataset.

If with your own dataset you do not have such results, you should think about how you divided your dataset in training and test. May be there is something to fix. Again, `caret` package may help.

For a better understanding of the learning progression, you may want to have some specific metric or even use multiple evaluation metrics.

```
bst <- xgb.train(data=dtrain, max.depth=2, eta=1, nthread = 2, nrounds=2,␣
→watchlist=watchlist, eval.metric = "error", eval.metric = "logloss", objective =
→"binary:logistic")
```

```
## [0]          train-error:0.046522          train-logloss:0.233376          test-error:0.
→042831          test-logloss:0.226686
## [1]          train-error:0.022263          train-logloss:0.136658          test-error:0.
→021726          test-logloss:0.137874
```

eval.metric allows us to monitor two new metrics for each round, logloss and error.

Linear boosting

Until now, all the learnings we have performed were based on boosting trees. **XGBoost** implements a second algorithm, based on linear boosting. The only difference with the previous command is booster = "gblinear" parameter (and removing eta parameter).

```
bst <- xgb.train(data=dtrain, booster = "gblinear", nthread = 2, nrounds=2,
→watchlist=watchlist, eval.metric = "error", eval.metric = "logloss", objective =
→"binary:logistic")
```

```
## [0]          train-error:0.024720          train-logloss:0.184616          test-error:0.
→022967          test-logloss:0.184234
## [1]          train-error:0.004146          train-logloss:0.069885          test-error:0.
→003724          test-logloss:0.068081
```

In this specific case, *linear boosting* gets slightly better performance metrics than a decision tree based algorithm.

In simple cases, this will happen because there is nothing better than a linear algorithm to catch a linear link. However, decision trees are much better to catch a non linear link between predictors and outcome. Because there is no silver bullet, we advise you to check both algorithms with your own datasets to have an idea of what to use.

Manipulating xgb.DMatrix

Save / Load

Like saving models, xgb.DMatrix object (which groups both dataset and outcome) can also be saved using xgb.DMatrix.save function.

```
xgb.DMatrix.save(dtrain, "dtrain.buffer")
```

```
## [1] TRUE
```

```
# to load it in, simply call xgb.DMatrix
dtrain2 <- xgb.DMatrix("dtrain.buffer")
```

```
## [11:41:01] 6513x126 matrix with 143286 entries loaded from dtrain.buffer
```

```
bst <- xgb.train(data=dtrain2, max.depth=2, eta=1, nthread = 2, nrounds=2,
→watchlist=watchlist, objective = "binary:logistic")
```

```
## [0]          train-error:0.046522          test-error:0.042831
## [1]          train-error:0.022263          test-error:0.021726
```

Information extraction

Information can be extracted from an `xgb.DMatrix` using `getinfo` function. Hereafter we will extract `label` data.

```
label = getinfo(dtest, "label")
pred <- predict(bst, dtest)
err <- as.numeric(sum(as.integer(pred > 0.5) != label))/length(label)
print(paste("test-error=", err))
```

```
## [1] "test-error= 0.0217256362507759"
```

View feature importance/influence from the learnt model

Feature importance is similar to R gbm package's relative influence (rel.inf).

```
importance_matrix <- xgb.importance(model = bst)
print(importance_matrix)
xgb.plot.importance(importance_matrix = importance_matrix)
```

View the trees from a model

You can dump the tree you learned using `xgb.dump` into a text file.

```
xgb.dump(bst, with_stats = TRUE)
```

```
##  [1] "booster[0]"
##  [2] "0:[f28<-1.00136e-05] yes=1,no=2,missing=1,gain=4000.53,cover=1628.25"
##  [3] "1:[f55<-1.00136e-05] yes=3,no=4,missing=3,gain=1158.21,cover=924.5"
##  [4] "3:leaf=1.71218,cover=812"
##  [5] "4:leaf=-1.70044,cover=112.5"
##  [6] "2:[f108<-1.00136e-05] yes=5,no=6,missing=5,gain=198.174,cover=703.75"
##  [7] "5:leaf=-1.94071,cover=690.5"
##  [8] "6:leaf=1.85965,cover=13.25"
##  [9] "booster[1]"
## [10] "0:[f59<-1.00136e-05] yes=1,no=2,missing=1,gain=832.545,cover=788.852"
## [11] "1:[f28<-1.00136e-05] yes=3,no=4,missing=3,gain=569.725,cover=768.39"
## [12] "3:leaf=0.784718,cover=458.937"
## [13] "4:leaf=-0.96853,cover=309.453"
## [14] "2:leaf=-6.23624,cover=20.4624"
```

You can plot the trees from your model using ```xgb.plot.tree``

```
xgb.plot.tree(model = bst)
```

if you provide a path to `fname` parameter you can save the trees to your hard drive.

Save and load models

Maybe your dataset is big, and it takes time to train a model on it? May be you are not a big fan of losing time in redoing the same task again and again? In these very rare cases, you will want to save your model and load it when required.

Helpfully for you, **XGBoost** implements such functions.

```
# save model to binary local file
xgb.save(bst, "xgboost.model")
```

```
## [1] TRUE
```

> xgb.save function should return TRUE if everything goes well and crashes otherwise.

An interesting test to see how identical our saved model is to the original one would be to compare the two predictions.

```
# load binary model to R
bst2 <- xgb.load("xgboost.model")
pred2 <- predict(bst2, test$data)

# And now the test
print(paste("sum(abs(pred2-pred))=", sum(abs(pred2-pred))))
```

```
## [1] "sum(abs(pred2-pred))= 0"
```

> result is 0? We are good!

In some very specific cases, like when you want to pilot **XGBoost** from caret package, you will want to save the model as a *R* binary vector. See below how to do it.

```
# save model to R's raw vector
rawVec <- xgb.save.raw(bst)

# print class
print(class(rawVec))
```

```
## [1] "raw"
```

```
# load binary model to R
bst3 <- xgb.load(rawVec)
pred3 <- predict(bst3, test$data)

# pred3 should be identical to pred
print(paste("sum(abs(pred3-pred))=", sum(abs(pred3-pred))))
```

```
## [1] "sum(abs(pred3-pred))= 0"
```

> Again 0? It seems that XGBoost works pretty well!

References

Understand your dataset with XGBoost

Introduction

The purpose of this Vignette is to show you how to use **XGBoost** to discover and understand your own dataset better.

This Vignette is not about predicting anything (see XGBoost presentation). We will explain how to use **XGBoost** to highlight the *link* between the *features* of your data and the *outcome*.

Package loading:

```
require(xgboost)
require(Matrix)
require(data.table)
if (!require('vcd')) install.packages('vcd')
```

> **VCD** package is used for one of its embedded dataset only.

Preparation of the dataset

Numeric VS categorical variables

XGBoost manages only `numeric` vectors.

What to do when you have *categorical* data?

A *categorical* variable has a fixed number of different values. For instance, if a variable called *Colour* can have only one of these three values, *red*, *blue* or *green*, then *Colour* is a *categorical* variable.

> In **R**, a *categorical* variable is called `factor`.
>
> Type `?factor` in the console for more information.

To answer the question above we will convert *categorical* variables to `numeric` one.

Conversion from categorical to numeric variables

Looking at the raw data

In this Vignette we will see how to transform a *dense* `data.frame` (*dense* = few zeroes in the matrix) with *categorical* variables to a very *sparse* matrix (*sparse* = lots of zero in the matrix) of `numeric` features.

The method we are going to see is usually called one-hot encoding.

The first step is to load `Arthritis` dataset in memory and wrap it with `data.table` package.

```
data(Arthritis)
df <- data.table(Arthritis, keep.rownames = FALSE)
```

> `data.table` is 100% compliant with **R** `data.frame` but its syntax is more consistent and its performance for large dataset is best in class (`dplyr` from **R** and `Pandas` from **Python** included). Some parts of **XGBoost R** package use `data.table`.

The first thing we want to do is to have a look to the first lines of the `data.table`:

```
head(df)
```

```
##    ID Treatment  Sex Age Improved
## 1: 57   Treated Male  27     Some
## 2: 46   Treated Male  29     None
## 3: 77   Treated Male  30     None
## 4: 17   Treated Male  32   Marked
## 5: 36   Treated Male  46   Marked
## 6: 23   Treated Male  58   Marked
```

Now we will check the format of each column.

```
str(df)
```

```
## Classes 'data.table' and 'data.frame':        84 obs. of  5 variables:
##  $ ID       : int  57 46 77 17 36 23 75 39 33 55 ...
##  $ Treatment: Factor w/ 2 levels "Placebo","Treated": 2 2 2 2 2 2 2 2 2 2 ...
##  $ Sex      : Factor w/ 2 levels "Female","Male": 2 2 2 2 2 2 2 2 2 2 ...
##  $ Age      : int  27 29 30 32 46 58 59 59 63 63 ...
##  $ Improved : Ord.factor w/ 3 levels "None"<"Some"<..: 2 1 1 3 3 3 1 3 1 1 ...
##  - attr(*, ".internal.selfref")=<externalptr>
```

2 columns have `factor` type, one has `ordinal` type.

> `ordinal` variable :
>
>   • can take a limited number of values (like `factor`) ;
>
>   • these values are ordered (unlike `factor`). Here these ordered values are: `Marked` > `Some` > `None`

Creation of new features based on old ones

We will add some new *categorical* features to see if it helps.

Grouping per 10 years

For the first feature we create groups of age by rounding the real age.

Note that we transform it to `factor` so the algorithm treat these age groups as independent values.

Therefore, 20 is not closer to 30 than 60. To make it short, the distance between ages is lost in this transformation.

```
head(df[,AgeDiscret := as.factor(round(Age/10,0))])
```

```
##    ID Treatment  Sex Age Improved AgeDiscret
## 1: 57   Treated Male  27     Some          3
## 2: 46   Treated Male  29     None          3
## 3: 77   Treated Male  30     None          3
## 4: 17   Treated Male  32   Marked          3
## 5: 36   Treated Male  46   Marked          5
## 6: 23   Treated Male  58   Marked          6
```

Random split in two groups

Following is an even stronger simplification of the real age with an arbitrary split at 30 years old. I choose this value **based on nothing**. We will see later if simplifying the information based on arbitrary values is a good strategy (you may already have an idea of how well it will work. . .).

```
head(df[,AgeCat:= as.factor(ifelse(Age > 30, "Old", "Young"))])
```

```
##     ID Treatment  Sex Age Improved AgeDiscret AgeCat
## 1: 57   Treated Male  27     Some          3  Young
## 2: 46   Treated Male  29     None          3  Young
## 3: 77   Treated Male  30     None          3  Young
## 4: 17   Treated Male  32   Marked          3    Old
## 5: 36   Treated Male  46   Marked          5    Old
## 6: 23   Treated Male  58   Marked          6    Old
```

Risks in adding correlated features

These new features are highly correlated to the `Age` feature because they are simple transformations of this feature.

For many machine learning algorithms, using correlated features is not a good idea. It may sometimes make prediction less accurate, and most of the time make interpretation of the model almost impossible. GLM, for instance, assumes that the features are uncorrelated.

Fortunately, decision tree algorithms (including boosted trees) are very robust to these features. Therefore we have nothing to do to manage this situation.

Cleaning data

We remove ID as there is nothing to learn from this feature (it would just add some noise).

```
df[,ID:=NULL]
```

We will list the different values for the column `Treatment`:

```
levels(df[,Treatment])
```

```
## [1] "Placebo" "Treated"
```

One-hot encoding

Next step, we will transform the categorical data to dummy variables. This is the one-hot encoding step.

The purpose is to transform each value of each *categorical* feature in a *binary* feature {0, 1}.

For example, the column `Treatment` will be replaced by two columns, `Placebo`, and `Treated`. Each of them will be *binary*. Therefore, an observation which has the value `Placebo` in column `Treatment` before the transformation will have after the transformation the value 1 in the new column `Placebo` and the value 0 in the new column `Treated`. The column `Treatment` will disappear during the one-hot encoding.

Column `Improved` is excluded because it will be our `label` column, the one we want to predict.

```
sparse_matrix <- sparse.model.matrix(Improved~.-1, data = df)
head(sparse_matrix)
```

```
## 6 x 10 sparse Matrix of class "dgCMatrix"
##
## 1 . 1 1 27 1 . . . . 1
## 2 . 1 1 29 1 . . . . 1
## 3 . 1 1 30 1 . . . . 1
## 4 . 1 1 32 1 . . . . .
## 5 . 1 1 46 . . 1 . . .
## 6 . 1 1 58 . . . 1 . .
```

Formulae `Improved~.-1` used above means transform all *categorical* features but column `Improved` to binary values. The `-1` is here to remove the first column which is full of `1` (this column is generated by the conversion). For more information, you can type `?sparse.model.matrix` in the console.

Create the output `numeric` vector (not as a sparse `Matrix`):

```
output_vector = df[,Improved] == "Marked"
```

1. set `Y` vector to `0`;

2. set `Y` to 1 for rows where `Improved == Marked` is TRUE ;

3. return `Y` vector.

Build the model

The code below is very usual. For more information, you can look at the documentation of `xgboost` function (or at the vignette XGBoost presentation).

```
bst <- xgboost(data = sparse_matrix, label = output_vector, max.depth = 4,
               eta = 1, nthread = 2, nrounds = 10,objective = "binary:logistic")
```

```
## [0]       train-error:0.202381
## [1]       train-error:0.166667
## [2]       train-error:0.166667
## [3]       train-error:0.166667
## [4]       train-error:0.154762
## [5]       train-error:0.154762
## [6]       train-error:0.154762
## [7]       train-error:0.166667
## [8]       train-error:0.166667
## [9]       train-error:0.166667
```

You can see some `train-error:   0.XXXXX` lines followed by a number. It decreases. Each line shows how well the model explains your data. Lower is better.

A model which fits too well may overfit (meaning it copy/paste too much the past, and won't be that good to predict the future).

Here you can see the numbers decrease until line 7 and then increase.

It probably means we are overfitting. To fix that I should reduce the number of rounds to `nrounds = 4`. I will let things like that because I don't really care for the purpose of this example :-)

Feature importance

Measure feature importance

Build the feature importance data.table

In the code below, `sparse_matrix@Dimnames[[2]]` represents the column names of the sparse matrix. These names are the original values of the features (remember, each binary column == one value of one *categorical* feature).

```
importance <- xgb.importance(feature_names = sparse_matrix@Dimnames[[2]], model = bst)
head(importance)
```

```
##                    Feature         Gain       Cover  Frequency
## 1:                    Age 0.622031651 0.67251706 0.67241379
## 2: TreatmentPlacebo 0.285750607 0.11916656 0.10344828
## 3:                SexMale 0.048744054 0.04522027 0.08620690
## 4:           AgeDiscret6 0.016604647 0.04784637 0.05172414
## 5:           AgeDiscret3 0.016373791 0.08028939 0.05172414
## 6:           AgeDiscret4 0.009270558 0.02858801 0.01724138
```

> The column `Gain` provide the information we are looking for.
>
> As you can see, features are classified by `Gain`.

`Gain` is the improvement in accuracy brought by a feature to the branches it is on. The idea is that before adding a new split on a feature X to the branch there was some wrongly classified elements, after adding the split on this feature, there are two new branches, and each of these branch is more accurate (one branch saying if your observation is on this branch then it should be classified as 1, and the other branch saying the exact opposite).

`Cover` measures the relative quantity of observations concerned by a feature.

`Frequency` is a simpler way to measure the `Gain`. It just counts the number of times a feature is used in all generated trees. You should not use it (unless you know why you want to use it).

Improvement in the interpretability of feature importance data.table

We can go deeper in the analysis of the model. In the `data.table` above, we have discovered which features counts to predict if the illness will go or not. But we don't yet know the role of these features. For instance, one of the question we may want to answer would be: does receiving a placebo treatment helps to recover from the illness?

One simple solution is to count the co-occurrences of a feature and a class of the classification.

For that purpose we will execute the same function as above but using two more parameters, `data` and `label`.

```
importanceRaw <- xgb.importance(feature_names = sparse_matrix@Dimnames[[2]], model = bst,
↪ data = sparse_matrix, label = output_vector)

# Cleaning for better display
importanceClean <- importanceRaw[,`:=`(Cover=NULL, Frequency=NULL)]

head(importanceClean)
```

```
##                    Feature        Split        Gain RealCover RealCover %
## 1: TreatmentPlacebo -1.00136e-05 0.28575061         7   0.2500000
```

(continues on next page)

```
## 2:          Age        61.5 0.16374034     12    0.4285714
## 3:          Age          39 0.08705750      8    0.2857143
## 4:          Age        57.5 0.06947553     11    0.3928571
## 5:      SexMale -1.00136e-05 0.04874405      4    0.1428571
## 6:          Age        53.5 0.04620627     10    0.3571429
```

In the table above we have removed two not needed columns and select only the first lines.

First thing you notice is the new column `Split`. It is the split applied to the feature on a branch of one of the tree. Each split is present, therefore a feature can appear several times in this table. Here we can see the feature `Age` is used several times with different splits.

How the split is applied to count the co-occurrences? It is always <. For instance, in the second line, we measure the number of persons under 61.5 years with the illness gone after the treatment.

The two other new columns are `RealCover` and `RealCover %`. In the first column it measures the number of observations in the dataset where the split is respected and the label marked as 1. The second column is the percentage of the whole population that `RealCover` represents.

Therefore, according to our findings, getting a placebo doesn't seem to help but being younger than 61 years may help (seems logic).

You may wonder how to interpret the `< 1.00001` on the first line. Basically, in a sparse `Matrix`, there is no `0`, therefore, looking for one hot-encoded categorical observations validating the rule `< 1.00001` is like just looking for `1` for this feature.

Plotting the feature importance

All these things are nice, but it would be even better to plot the results.

```
xgb.plot.importance(importance_matrix = importanceRaw)
```

```
## Error in xgb.plot.importance(importance_matrix = importanceRaw): Importance matrix is
→not correct (column names issue)
```

Feature have automatically been divided in 2 clusters: the interesting features... and the others.

Depending of the dataset and the learning parameters you may have more than two clusters. Default value is to limit them to `10`, but you can increase this limit. Look at the function documentation for more information.

According to the plot above, the most important features in this dataset to predict if the treatment will work are :

- the Age ;

- having received a placebo or not ;

- the sex is third but already included in the not interesting features group ;

- then we see our generated features (AgeDiscret). We can see that their contribution is very low.

Do these results make sense?

Let's check some **Chi2** between each of these features and the label.

Higher **Chi2** means better correlation.

```
c2 <- chisq.test(df$Age, output_vector)
print(c2)
```

```
##
##          Pearson's Chi-squared test
##
## data:  df$Age and output_vector
## X-squared = 35.475, df = 35, p-value = 0.4458
```

Pearson correlation between Age and illness disappearing is **35.48**.

```
c2 <- chisq.test(df$AgeDiscret, output_vector)
print(c2)
```

```
##
##          Pearson's Chi-squared test
##
## data:  df$AgeDiscret and output_vector
## X-squared = 8.2554, df = 5, p-value = 0.1427
```

Our first simplification of Age gives a Pearson correlation is **8.26**.

```
c2 <- chisq.test(df$AgeCat, output_vector)
print(c2)
```

```
##
##          Pearson's Chi-squared test with Yates' continuity correction
##
## data:  df$AgeCat and output_vector
## X-squared = 2.3571, df = 1, p-value = 0.1247
```

The perfectly random split I did between young and old at 30 years old have a low correlation of **2.36**. It's a result we may expect as may be in my mind > 30 years is being old (I am 32 and starting feeling old, this may explain that), but for the illness we are studying, the age to be vulnerable is not the same.

Morality: don't let your *gut* lower the quality of your model.

In *data science* expression, there is the word *science* :-)

Conclusion

As you can see, in general *destroying information by simplifying it won't improve your model*. **Chi2** just demonstrates that.

But in more complex cases, creating a new feature based on existing one which makes link with the outcome more obvious may help the algorithm and improve the model.

The case studied here is not enough complex to show that. Check Kaggle website for some challenging datasets. However it's almost always worse when you add some arbitrary rules.

Moreover, you can notice that even if we have added some not useful new features highly correlated with other features, the boosting tree algorithm have been able to choose the best one, which in this case is the Age.

Linear models may not be that smart in this scenario.

Special Note: What about Random Forests™?

As you may know, Random Forests algorithm is cousin with boosting and both are part of the ensemble learning family.

Both train several decision trees for one dataset. The *main* difference is that in Random Forests, trees are independent and in boosting, the tree `N+1` focus its learning on the loss (`<=>` what has not been well modeled by the tree `N`).

This difference have an impact on a corner case in feature importance analysis: the *correlated features*.

Imagine two features perfectly correlated, feature `A` and feature `B`. For one specific tree, if the algorithm needs one of them, it will choose randomly (true in both boosting and Random Forests).

However, in Random Forests this random choice will be done for each tree, because each tree is independent from the others. Therefore, approximatively, depending of your parameters, 50% of the trees will choose feature `A` and the other 50% will choose feature `B`. So the *importance* of the information contained in `A` and `B` (which is the same, because they are perfectly correlated) is diluted in `A` and `B`. So you won't easily know this information is important to predict what you want to predict! It is even worse when you have 10 correlated features...

In boosting, when a specific link between feature and outcome have been learned by the algorithm, it will try to not refocus on it (in theory it is what happens, reality is not always that simple). Therefore, all the importance will be on feature `A` or on feature `B` (but not both). You will know that one feature have an important role in the link between the observations and the label. It is still up to you to search for the correlated features to the one detected as important if you need to know all of them.

If you want to try Random Forests algorithm, you can tweak XGBoost parameters!

Warning: this is still an experimental parameter.

For instance, to compute a model with 1000 trees, with a 0.5 factor on sampling rows and columns:

```
data(agaricus.train, package='xgboost')
data(agaricus.test, package='xgboost')
train <- agaricus.train
test <- agaricus.test

#Random Forest - 1000 trees
bst <- xgboost(data = train$data, label = train$label, max.depth = 4, num_parallel_tree
  = 1000, subsample = 0.5, colsample_bytree =0.5, nrounds = 1, objective =
  "binary:logistic")
```

```
## [0]    train-error:0.002150
```

```
#Boosting - 3 rounds
bst <- xgboost(data = train$data, label = train$label, max.depth = 4, nrounds = 3,␣
→objective = "binary:logistic")
```

```
## [0]       train-error:0.006142
## [1]       train-error:0.006756
## [2]       train-error:0.001228
```

Note that the parameter `round` is set to `1`.

Random Forests is a trademark of Leo Breiman and Adele Cutler and is licensed exclusively to Salford Systems for the commercial release of the software.

1.12 XGBoost JVM Package

You have found the XGBoost JVM Package!

1.12.1 Installation

Checkout the *Installation Guide* for how to install jvm package, or *Building from Source* on how to build it form source.

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:

```java
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:

```java
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:

```java
float[] data = new float[] {1f,2f,3f,4f,5f,6f};
int nrow = 3;
int ncol = 2;
float missing = 0.0f;
DMatrix dmat = new DMatrix(data, nrow, ncol, missing);
```

- To set weight:

```java
float[] weights = new float[] {1f,2f,1f};
dmat.setWeight(weights);
```

Setting Parameters

To set parameters, parameters are specified as a Map:

```java
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.bin");
```

- Generaing model dump with feature map

```
// 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

```
Booster booster = XGBoost.loadModel("model.bin");
```

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).

```
DMatrix dtest = new DMatrix("test.svm.txt");
// predict
float[][] predicts = booster.predict(dtest);
// predict leaf
float[][] leafPredicts = booster.predictLeaf(dtest, 0);
```

XGBoost4J-Spark Tutorial (version 0.9+)

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 XGBoost/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

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 2.4+

XGBoost4J-Spark now requires **Apache Spark 2.4+**. 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.

Installation from maven repo

Note: Use of Python in XGBoost4J-Spark

By default, we use the tracker in Python package to drive the training with XGBoost4J-Spark. It requires Python 3.6+. We also have an experimental Scala version of tracker which can be enabled by passing the parameter `tracker_conf` as `scala`.

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),
```

(continues on next page)

```scala
    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.

```scala
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 to contain 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.

```scala
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.

Note: There is no need to assemble feature columns from version 1.6.1+. Instead, users can specify an array of feature column names by `setFeaturesCol(value:  Array[String])` and XGBoost4j-Spark will do it.

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).
      setFeaturesCol("features").
      setLabelCol("classIndex")
```

Note: Missing values with Spark's VectorAssembler

If given a Dataset with enough features having a value of 0 Spark's VectorAssembler transformer class will return a SparseVector where the absent values are meant to indicate a value of 0. This conflicts with XGBoost's default to treat values absent from the SparseVector as missing. The model would effectively be treating 0 as missing but not declaring that to be so which can lead to confusion when using the trained model on other platforms. To avoid this, XGBoost will raise an exception if it receives a SparseVector and the "missing" parameter has not been explicitly set to 0. To workaround this issue the user has three options:

1. Explicitly convert the Vector returned from VectorAssembler to a DenseVector to return the zeros to the dataset. If doing this with missing values encoded as NaN, you will want to set `setHandleInvalid = "keep"` on VectorAssembler in order to keep the NaN values in the dataset. You would then set the "missing" parameter to whatever you want to be treated as missing. However this may cause a large amount of memory use if your dataset is very sparse. For example:

---

val assembler = new VectorAssembler().setInputCols(feature_names.toArray).setOutputCol("features").setHandleInvalid("keep")

// conversion to dense vector using Array()

---

val featurePipeline = new Pipeline().setStages(Array(assembler)) val featureModel = featurePipeline.fit(df_training) val featureDf = featureModel.transform(df_training)

val xgbParam = Map("eta" -> 0.1f,
    "max_depth" -> 2, "objective" -> "multi:softprob", "num_class" -> 3, "num_round" -> 100, "num_workers" ->
    2, "allow_non_zero_for_missing" -> "true", "missing" -> -999)

val xgb = new XGBoostClassifier(xgbParam) val xgbclassifier = xgb.fit(featureDf)

2. Before calling VectorAssembler you can transform the values you want to represent missing into an irregular value that is not 0, NaN, or Null and set the "missing" parameter to 0. The irregular value should ideally be chosen to be outside the range of values that your features have.

3. Do not use the VectorAssembler class and instead use a custom way of constructing a SparseVector that allows for specifying sparsity to indicate a non-zero value. You can then set the "missing" parameter to whatever sparsity indicates in your Dataset. If this approach is taken you can pass the parameter `"allow_non_zero_for_missing_value"` -> `true` to bypass XGBoost's assertion that "missing" must be zero when given a SparseVector.

Option 1 is recommended if memory constraints are not an issue. Option 3 requires more work to get set up but is guaranteed to give you correct results while option 2 will be quicker to set up but may be difficult to find a good irregular value that does not conflict with your feature values.

Note: Using a non-default missing value when using other bindings of XGBoost.

When XGBoost is saved in native format only the booster itself is saved, the value of the missing parameter is not saved alongside the model. Thus, if a non-default missing parameter is used to train the model in Spark the user should take care to use the same missing parameter when using the saved model in another binding.

Training

XGBoost supports both regression and classification. While we use Iris dataset in this tutorial to show how we use XGBoost/XGBoost4J-Spark to resolve a multi-classes classification problem, the usage in Regression is very similar to classification.

To train a XGBoost model for classification, we need to claim a XGBoostClassifier first:

```scala
import ml.dmlc.xgboost4j.scala.spark.XGBoostClassifier
val xgbParam = Map("eta" -> 0.1f,
      "max_depth" -> 2,
      "objective" -> "multi:softprob",
      "num_class" -> 3,
      "num_round" -> 100,
      "num_workers" -> 2)
val xgbClassifier = new XGBoostClassifier(xgbParam).
      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 traning would be early stopped at 15th iteration.

Training with Evaluation Sets

You can also monitor the performance of the model during training with multiple evaluation datasets. By specifying `eval_sets` or call `setEvalSets` over a XGBoostClassifier or XGBoostRegressor, you can pass in multiple evaluation datasets typed as a Map from String to DataFrame.

Prediction

XGBoost4j-Spark supports two ways for model serving: batch prediction and single instance prediction.

Batch Prediction

When we get a model, either XGBoostClassificationModel or XGBoostRegressionModel, 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`).

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|
|[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 or XGBoostRegressionModel support make 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. 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 from version 1.7.0+.

```
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 from version 1.7.0+.

```
val xgbClassificationModelPath = "/tmp/xgbClassificationModel"
xgbClassificationModel.write.overwrite().save(xgbClassificationModelPath)
```

```
import xgboost as xgb
bst = xgb.Booster({'nthread': 4})
bst.load_model("/tmp/xgbClassificationModel/data/XGBoostClassificationModel")
```

Before version 1.7.0, XGBoost4j-Spark needs to export model to local manually by:

```
val nativeModelPath = "/tmp/nativeModel"
xgbClassificationModel.nativeBooster.saveModel(nativeModelPath)
```

Then we can load this model with single node Python XGBoost:

```
import xgboost as xgb
bst = xgb.Booster({'nthread': 4})
bst.load_model(nativeModelPath)
```

Note: Using HDFS and S3 for exporting the models with nativeBooster.saveModel()

When interacting with other language bindings, XGBoost also supports saving-models-to and loading-models-from file systems other than the local one. You can use HDFS and S3 by prefixing the path with `hdfs://` and `s3://` respectively. However, for this capability, you must do **one** of the following:

1. Build XGBoost4J-Spark with the steps described in *here*, but turning USE_HDFS (or USE_S3, etc. in the same place) switch on. With this approach, you can reuse the above code example by replacing "nativeModelPath" with a HDFS path.

   - However, if you build with USE_HDFS, etc. you have to ensure that the involved shared object file, e.g. libhdfs.so, is put in the LIBRARY_PATH of your cluster. To avoid the complicated cluster environment configuration, choose the other option.

2. Use bindings of HDFS, S3, etc. to pass model files around. Here are the steps (taking HDFS as an example):

   - Create a new file with

   ```scala
   val outputStream = fs.create("hdfs_path")
   ```

   where "fs" is an instance of org.apache.hadoop.fs.FileSystem class in Hadoop.

   - Pass the returned OutputStream in the first step to nativeBooster.saveModel():

   ```scala
   xgbClassificationModel.nativeBooster.saveModel(outputStream)
   ```

   - Download file in other languages from HDFS and load with the pre-built (without the requirement of libhdfs.so) version of XGBoost. (The function "download_from_hdfs" is a helper function to be implemented by the user)

   ```python
   import xgboost as xgb
   bst = xgb.Booster({'nthread': 4})
   local_path = download_from_hdfs("hdfs_path")
   bst.load_model(local_path)
   ```

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:

```scala
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:

```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.

```scala
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 Index-ToString:

```scala
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:

```scala
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.

```scala
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.

```scala
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))
    .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:

```
xgbClassifier.setTimeoutRequestWorkers(60000L)
```

or pass in `timeout_request_workers` in `xgbParamMap` when building XGBoostClassifier:

```scala
val xgbParam = Map("eta" -> 0.1f,
    "max_depth" -> 2,
    "objective" -> "multi:softprob",
    "num_class" -> 3,
    "num_round" -> 100,
    "num_workers" -> 2,
    "timeout_request_workers" -> 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.

XGBoost4J-Spark-GPU Tutorial (version 1.6.1+)

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*

Build an ML Application with XGBoost4J-Spark-GPU

Add XGBoost to Your Project

Before we go into the tour of how to use XGBoost4J-Spark-GPU, you should first consult *Installation from Maven repository* in order to add XGBoost4J-Spark-GPU as a dependency for your project. We provide both stable releases and snapshots.

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

```scala
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)
```

In the first line, 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:

```scala
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)
```

<div align="right">(continues on next page)</div>

```
    .withColumnRenamed("tmpClassName", labelName)
    .randomSplit(Array(0.7, 0.3), seed = 1)

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

The GPU version of XGBoost-Spark supports both regression and classification models. Although we use the Iris dataset in this tutorial to show how we use `XGBoost/XGBoost4J-Spark-GPU` to resolve a multi-classes classification problem, the usage in Regression is very similar to classification.

To train a XGBoost model for classification, we need to claim a XGBoostClassifier first:

```scala
import ml.dmlc.xgboost4j.scala.spark.XGBoostClassifier
val xgbParam = Map(
    "objective" -> "multi:softprob",
    "num_class" -> 3,
    "num_round" -> 100,
    "tree_method" -> "gpu_hist",
    "num_workers" -> 1)

val featuresNames = schema.fieldNames.filter(name => name != labelName)

val xgbClassifier = new XGBoostClassifier(xgbParam)
    .setFeaturesCol(featuresNames)
    .setLabelCol(labelName)
```

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:

```scala
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.

```scala
val xgbClassificationModel = xgbClassifier.fit(train)
```

Prediction

When we get a model, either a XGBoostClassificationModel or a XGBoostRegressionModel, 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`).

```scala
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|
|         5.1|        2.5|             3.0|              1.1|    1|[3.16666603088378.
→..|[0.98853939771652...|       0.0|
```

(continues on next page)

```
|        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

Here's an example to submit an end-to-end XGBoost-4j-Spark-GPU Spark application to an Apache Spark Standalone cluster, assuming the application main class is Iris and the application jar is iris-1.0.0.jar

```
cudf_version=22.02.0
rapids_version=22.02.0
xgboost_version=1.6.1
main_class=Iris
app_jar=iris-1.0.0.jar

spark-submit \
  --master $master \
  --packages ai.rapids:cudf:${cudf_version},com.nvidia:rapids-4-spark_2.12:${rapids_
→version},ml.dmlc:xgboost4j-gpu_2.12:${xgboost_version},ml.dmlc:xgboost4j-spark-gpu_2.
→12:${xgboost_version} \
  --conf spark.executor.cores=12 \
  --conf spark.task.cpus=1 \
  --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, cudf, xgboost4j-gpu, 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.

XGBoost4J Java API

XGBoost4J Scala API

XGBoost4J-Spark Scala API

XGBoost4J-Flink Scala API

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 DMatrix

typedef void *`BoosterHandle`

> handle to Booster

Functions

void `XGBoostVersion`(int *major, int *minor, int *patch)

> Return the version of the XGBoost library being currently used.
>
> 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 shared library.
>
> > **Parameters**
> > out – string encoded JSON object containing build flags and dependency version.
> >
> > **Returns**
> > 0 for success, -1 for failure

const char *`XGBGetLastError`()

> get string message of the last error
>
> all function in this file will return 0 when success and -1 when an error occurred, XGBGetLastError can be called to retrieve the error
>
> this function is thread safe and can be called by different thread
>
> > **Returns**
> > const char* error information

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 for success, -1 for failure

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 for success, -1 for failure

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 for success, -1 for failure

DMatrix

group **DMatrix**

> DMatrix is the baisc 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.
> >
> >   - silent (optional): Whether to print message during loading. Default to true.
> >
> >   - data_split_mode (optional): Whether to split by row or column. In distributed mode, the file is split accordingly; otherwise this is only an indicator on how the file was split beforehand. Default to row.
> >
> > - **out** – a loaded data matrix
> >
> > **Returns**
> >
> > 0 when success, -1 when failure happens

int **XGDMatrixCreateFromCSREx**(const size_t *indptr, const unsigned *indices, const float *data, size_t nindptr, size_t nelem, size_t num_col, *DMatrixHandle* *out)

> create a matrix content from CSR format
>
> > **Parameters**
> >
> > - **indptr** – pointer to row headers
> >
> > - **indices** – findex
> >
> > - **data** – fvalue
> >
> > - **nindptr** – number of rows in the matrix + 1
> >
> > - **nelem** – number of nonzero elements in the matrix
> >
> > - **num_col** – number of columns; when it's set to kAdapterUnknownSize, then guess from data
> >
> > - **out** – 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 matrix 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** – Number of columns.
> >
> > - **config** – JSON encoded configuration. Required values are:
> >
> >   - missing: Which value to represent missing value.
> >
> >   - nthread (optional): Number of threads used for initializing DMatrix.
> >
> > - **out** – created dmatrix

Returns

0 when success, -1 when failure happens

int **XGDMatrixCreateFromDense**(char const *data, char const *config, *DMatrixHandle* *out)

Create a matrix from dense array.

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.

- **out** – created dmatrix

Returns

0 when success, -1 when failure happens

int **XGDMatrixCreateFromCSCEx**(const size_t *col_ptr, const unsigned *indices, const float *data, size_t nindptr, size_t nelem, size_t num_row, *DMatrixHandle* *out)

create a matrix content from CSC format

Parameters

- **col_ptr** – pointer to col headers

- **indices** – findex

- **data** – fvalue

- **nindptr** – number of rows in the matrix + 1

- **nelem** – number of nonzero elements in the matrix

- **num_row** – number of rows; when it's set to 0, then guess from data

- **out** – 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 **XGDMatrixCreateFromDT**(void **data, const char **feature_stypes, bst_ulong nrow, bst_ulong ncol, *DMatrixHandle* *out, int nthread)

create matrix content from python data table

Parameters

- **data** – pointer to pointer to column data

- **feature_stypes** – pointer to strings

- **nrow** – number of rows

- **ncol** – number columns

- **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)

Parameters

- **data** – Array of JSON encoded **cuda_array_interface** for each column.

- **config** – JSON encoded configuration. Required values are:

  - missing: Which value to represent missing value.

  - nthread (optional): Number of threads used for initializing DMatrix.

- **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.

- • **out** – created dmatrix

> **Returns**
>> 0 when success, -1 when failure happens

int **XGImportArrowRecordBatch**(*DataIterHandle* data_handle, void *ptr_array, void *ptr_schema)

int **XGDMatrixCreateFromArrowCallback**(*XGDMatrixCallbackNext* *next, char const *config,
DMatrixHandle *out)

> Construct DMatrix from arrow using callbacks. Arrow related C API is not stable and subject to change in the future.

> **Parameters**

- • **next** – Callback function for fetching arrow records.

- • **config** – JSON encoded configuration. Required values are:

> – missing: Which value to represent missing value.

> – nbatch: Number of batches in arrow table.

> – nthread (optional): Number of threads used for initializing DMatrix.

- • **out** – The 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 space in data matrix
>
> > **Returns**
> >
> > > 0 when success, -1 when failure happens

int **XGDMatrixSaveBinary**(*DMatrixHandle* handle, const char *fname, int silent)

> load a data matrix into binary file
>
> > **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 *c_interface_str)

> Set content in array interface to a content in info.
>
> > **Parameters**
> >
> > > - **handle** – a instance of data matrix
> > > - **field** – field name.
> > > - **c_interface_str** – JSON string representation of array interface.
> >
> > **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)

> set uint32 vector to a content in info
>
> > **Parameters**
> >
> > > - **handle** – a instance of data matrix
> > > - **field** – field name
> > > - **array** – pointer to unsigned int vector
> > > - **len** – length of array
> >
> > **Returns**
> >
> > > 0 when success, -1 when failure happens

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 **XGDMatrixSetDenseInfo**(*DMatrixHandle* handle, const char *field, void const *data, bst_ulong size, int type)

Set meta info from dense matrix. Valid field names are:

- label

- weight

- base_margin

- group

- label_lower_bound

- label_upper_bound

- feature_weights

Parameters

- **handle** – An instance of data matrix

- **field** – Field name

- **data** – Pointer to consecutive memory storing data.

- **size** – Size of the data, this is relative to size of type. (Meaning NOT number of bytes.)

- **type** – Indicator of data type. This is defined in xgboost::DataType enum class.

  - float = 1

  - double = 2

  - uint32_t = 3

  - uint64_t = 4

Returns

0 when success, -1 when failure happens

int **XGDMatrixSetGroup**(*DMatrixHandle* handle, const unsigned *group, bst_ulong len)

(deprecated) Use XGDMatrixSetUIntInfo instead. Set group of the training matrix

Parameters

- **handle** – a instance of data matrix

- **group** – pointer to group size

- **len** – length of array

> **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 number of rows.

>> **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 number of columns

>> **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 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 **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.

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

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 accept 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 hist and GPU Hist. 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. 1 by a `concat` from external library to make the data into a blob for normal DMatrix initialization, another 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. 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

Proxy that callers can use to pass data to XGBoost

- *XGProxyDMatrixCreate*
- *XGDMatrixCallbackNext*
- *DataIterResetCallback*
- *XGProxyDMatrixSetDataCudaArrayInterface*
- *XGProxyDMatrixSetDataCudaColumnar*
- *XGProxyDMatrixSetDataDense*
- *XGProxyDMatrixSetDataCSR*
- ... (data setters)

Typedefs

typedef void ***DataIterHandle**

> handle to a external data iterator

typedef void ***DataHolderHandle**

> handle to a 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 external iterator.

Functions

int **XGDMatrixCreateFromDataIter**(*DataIterHandle* data_handle, *XGBCallbackDataIterNext* *callback,
const char *cache_info, *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.

- **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 free by XGDMatrixFree.

Second set of callback functions, used by constructing Quantile DMatrix or external memory DMatrix using custom iterator.

Parameters

out – The created Device Quantile 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 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.
  - **out** – The created Device Quantile DMatrix

Returns

0 when success, -1 when failure happens

int **XGDeviceQuantileDMatrixCreateFromCallback**(*DataIterHandle* iter, *DMatrixHandle* proxy, *DataIterResetCallback* *reset, *XGDMatrixCallbackNext* *next, float missing, int nthread, int max_bin, *DMatrixHandle* *out)

Create a Device Quantile DMatrix with data iterator.

Deprecated:

since 1.7.0

See also:

XGQuantileDMatrixCreateFromCallback()

int **XGProxyDMatrixSetDataCudaArrayInterface**(*DMatrixHandle* handle, const char *c_interface_str)

Set data on a DMatrix proxy.

Parameters

- **handle** – A DMatrix proxy created by *XGProxyDMatrixCreate*

- **c_interface_str** – Null terminated JSON document string representation of CUDA array interface.

Returns

0 when success, -1 when failure happens

int **XGProxyDMatrixSetDataCudaColumnar**(*DMatrixHandle* handle, const char *c_interface_str)

Set data on a DMatrix proxy.

Parameters

- **handle** – A DMatrix proxy created by *XGProxyDMatrixCreate*

- **c_interface_str** – Null terminated JSON document string representation of CUDA array interface, with an array of columns.

Returns

0 when success, -1 when failure happens

int **XGProxyDMatrixSetDataDense**(*DMatrixHandle* handle, char const *c_interface_str)

Set data on a DMatrix proxy.

Parameters

- **handle** – A DMatrix proxy created by *XGProxyDMatrixCreate*

- **c_interface_str** – 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.

Functions

int **XGBoosterCreate**(const *DMatrixHandle* dmats[], bst_ulong len, *BoosterHandle* *out)

> create xgboost learner

> > **Parameters**
> >
> > - **dmats** – matrices that are set to be cached
> >
> > - **len** – length of dmats
> >
> > - **out** – handle to the result booster
> >
> > **Returns**
> > 0 when success, -1 when failure happens

int **XGBoosterFree**(*BoosterHandle* handle)

> free obj in handle

> > **Parameters**
> > **handle** – handle to be freed

> > **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)

update the model, by directly specify gradient and second order gradient, this can be used to replace UpdateOneIter, to support customized loss function

Parameters

- **handle** – handle

- **dtrain** – training data

- **grad** – gradient statistics

- **hess** – second order gradient statistics

- **len** – length of grad/hess array

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 **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 feature info in DMatrix.

> Accepted fields are:

> • feature_name

> • feature_type

> Caller is responsible for copying out the data, before 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.
>
> > **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 **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.
>
> > **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* m, bst_ulong const **out_shape, bst_ulong *out_dim, const float **out_result)

> Inplace prediction from CUDA Dense matrix (cupy in Python).
>
> > **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

- **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 **XGBoosterPredictFromCudaColumnar**(*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 CUDA dense dataframe (cuDF in Python).

Parameters

- **handle** – Booster handle

- **values** – List of **cuda_array_interface** for all columns encoded in JSON list.

- **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

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 are introduced in 1.0.0, and are not yet stable.

- Functions with the term "Serialization" are combined of above two. They are used in situations like check-pointing, or continuing training task in distributed environment. In these cases the task must be carried out without any user intervention.

Functions

int **XGBoosterLoadModel**(*BoosterHandle* handle, const char *fname)

    Load model from existing file.

        Parameters
- **handle** – handle
- **fname** – File URI or file name.

        Returns
        0 when success, -1 when failure happens

int **XGBoosterSaveModel**(*BoosterHandle* handle, const char *fname)

    Save model into existing file.

        Parameters
- **handle** – handle
- **fname** – File URI or file name.

        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.

"format": str

- json: Output booster will be encoded as JSON.
- ubj: Output booster will be encoded as Univeral binary JSON.
- deprecated: Output booster will be encoded as old custom binary format. Do not use this format except for compatibility reasons.

Parameters

- **handle** – handle

- **config** – JSON encoded string storing parameters for the function. Following keys are expected in the JSON document:

- **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 **XGBoosterGetModelRaw**(*BoosterHandle* handle, bst_ulong *out_len, const char **out_dptr)

Save booster to a buffer with in binary format.

Deprecated:

since 1.6.0

See also:

XGBoosterSaveModelToBuffer()

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 **XGBoosterLoadRabitCheckpoint**(*BoosterHandle* handle, int *version)

Initialize the booster from rabit checkpoint. This is used in distributed training API.

Parameters

- **handle** – handle

- **version** – The output version of the model.

Returns

0 when success, -1 when failure happens

int **XGBoosterSaveRabitCheckpoint**(*BoosterHandle* handle)

> Save the current checkpoint to rabit.
>
> > **Parameters**
> >   **handle** – handle
> >
> > **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.

Functions

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 before using anything.
>
> The additional configuration is not required. Usually the communicator will detect settings from environment variables.
>
> > **Parameters**
> >   **config** – JSON encoded configuration. Accepted JSON keys are:

- xgboost_communicator: The type of the communicator. Can be set as an environment variable.

  - rabit: Use Rabit. This is the default if the type is unspecified.

  - mpi: Use MPI.

  - federated: Use the gRPC interface for Federated Learning. Only applicable to the Rabit communicator (these are case-sensitive):

- rabit_tracker_uri: Hostname of the tracker.

- rabit_tracker_port: Port number of the tracker.

- rabit_task_id: ID of the current task, can be used to obtain deterministic rank assignment.

- rabit_world_size: Total number of workers.

- rabit_hadoop_mode: Enable Hadoop support.

- rabit_tree_reduce_minsize: Minimal size for tree reduce.

- rabit_reduce_ring_mincount: Minimal count to perform ring reduce.

- rabit_reduce_buffer: Size of the reduce buffer.

- rabit_bootstrap_cache: Size of the bootstrap cache.

- rabit_debug: Enable debugging.

- rabit_timeout: Enable timeout.

- rabit_timeout_sec: Timeout in seconds.

- rabit_enable_tcp_no_delay: Enable TCP no delay on Unix platforms. Only applicable to the Rabit communicator (these are case-sensitive, and can be set as environment variables):

- 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 rank assignment.

- DMLC_ROLE: Role of the current task, "worker" or "server".

- DMLC_NUM_ATTEMPT: Number of attempts after task failure.

- DMLC_WORKER_CONNECT_RETRY: Number of retries to connect to the tracker. 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: 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.

Returns

    0 for success, -1 for failure.

---

int **XGCommunicatorFinalize**(void)

> Finalize the collective communicator.
>
> Call this function after you finished all jobs.
>
> > **Returns**
> >
> > > 0 for success, -1 for failure.

int **XGCommunicatorGetRank**(void)

> Get rank of current process.
>
> > **Returns**
> >
> > > Rank of the worker.

int **XGCommunicatorGetWorldSize**(void)

> Get 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 communicator.
>
> This function can be used to communicate the information of the progress to the user who monitors the communicator.
>
> > **Parameters**
> >
> > > **message** – The message to be printed.
> >
> > **Returns**
> >
> > > 0 for success, -1 for failure.

int **XGCommunicatorGetProcessorName**(const char **name_str)

> Get the name of the processor.
>
> > **Parameters**
> >
> > > **name_str** – Pointer to received returned processor name.
> >
> > **Returns**
> >
> > > 0 for success, -1 for failure.

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.
> > >
> > > - **root** – The process rank to broadcast from.

Returns
0 for success, -1 for failure.

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

```
vector<int> data(10);
...
Allreduce(&data[0], 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 for success, -1 for failure.

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 XGBoost Command Line version

See XGBoost Command Line walkthrough.

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 the 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 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 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 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 use BuildKite to organize CI pipelines.

The cloud-hosted test farm has recurring operating expenses. It utilizes a leading cloud provider (AWS) to accommodate variable workload. BuildKite 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 exprenses for the cloud test farm (https://buildkite.com/xgboost)

- Cost of domain https://xgboost-ci.net

- Monthly cost of using BuildKite

- Hosting cost of the User Forum (https://discuss.xgboost.ai)

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

> – *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++11 features such as smart pointers, braced initializers, lambda functions, and `std::thread`.
- Use Doxygen to document all the interface code.
- 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*.

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 `tests/ci_build` 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.

Linter

We use pylint and cpplint to enforce style convention and find potential errors. Linting is especially useful for Python, as we can catch many errors that would have otherwise occured at run-time.

To run this check locally, run the following command from the top level source tree:

```
cd /path/to/xgboost/
make lint
```

This command requires the Python packages pylint and cpplint.

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, run the following command from the top level source tree:

```
cd /path/to/xgboost/
python3 tests/ci_build/tidy.py
```

Also, the script accepts two optional integer arguments, namely `--cpp` and `--cuda`. By default they are both set to 1, meaning that both C++ and CUDA code will be checked. If the CUDA toolkit is not installed on your machine, you'll encounter an error. To exclude CUDA source from linting, use:

```
cd /path/to/xgboost/
python3 tests/ci_build/tidy.py --cuda=0
```

Similarly, if you want to exclude C++ source from linting:

```
cd /path/to/xgboost/
python3 tests/ci_build/tidy.py --cpp=0
```

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 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*
    - *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 directory tests/python/ or tests/python-gpu/ (if you are testing GPU code). 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

```
make Rcheck
```

at the root of the project directory.

JVM packages

As part of the building process, tests are run:

```
mvn package
```

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.)

C++: Google Test

To build and run C++ unit tests enable tests while running CMake:

```
mkdir build
cd build
cmake -DGOOGLE_TEST=ON -DUSE_DMLC_GTEST=ON  ..
make
make test
```

To enable tests for CUDA code, add -DUSE_CUDA=ON and -DUSE_NCCL=ON (CUDA toolkit required):

```
mkdir build
cd build
cmake -DGOOGLE_TEST=ON -DUSE_DMLC_GTEST=ON -DUSE_CUDA=ON -DUSE_NCCL=ON ..
make
make test
```

One can also run all unit test using ctest tool which provides higher flexibility. For example:

```
ctest --verbose
```

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;leak" /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;leak" \
-DSANITIZER_PATH=/path/to/sanitizers /path/to/xgboost
```

How to use sanitizers with CUDA support

Runing 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.

1.17.5 Documentation and Examples

Contents

- *Documents*
- *Examples*

Documents

- Python and C documentation is built using Sphinx.
- Each document is written in reStructuredText.
- You can build document locally to see the effect, by running

```
make html
```

inside the doc/ directory. The online document is hosted by Read the Docs where the imported project is managed by Hyunsu Cho and Jiaming Yuan.

Examples

- Use cases and examples will be in demo.

- 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.

1.17.6 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.

1.17.7 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 `tests/ci_build/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, CRAN, and Maven packages.

   - The pip package is maintained by Hyunsu Cho and Jiaming Yuan. There's a helper script for downloading pre-built wheels and R packages `xgboost/dev/release-pypi-r.py` along with simple instructions for using `twine`.

   - The CRAN package is maintained by Tong He and Jiaming Yuan.

   - The Maven package is maintained by Nan Zhu and Hyunsu Cho.

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.

References

[1] https://stat.ethz.ch/pipermail/r-package-devel/2022q4/008610.html

1.17.8 Automated testing in XGBoost project

This document collects tips for using the Continuous Integration (CI) service of the XGBoost project.

Contents

- *GitHub Actions*
- *Reproduce CI testing environments using Docker containers*
    - *Prerequisites*
    - *Building and Running Docker containers locally*
- *Update pipeline definitions for BuildKite CI*
- *Managing Elastic CI Stack with BuildKite*
    - *Worker Image Pipeline*
    - *EC2 Autoscaling Groups*

GitHub Actions

The configuration files are located under the directory .github/workflows.

Most of the tests listed in the configuration files run automatically for every incoming pull requests and every update to branches. A few tests however require manual activation:

- R tests with noLD option: 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). To invoke this test suite for a particular pull request, simply add a review comment `/gha run r-nold-test`. (Ordinary comment won't work. It needs to be a review comment.)

GitHub Actions is also used to build Python wheels targeting MacOS Intel and Apple Silicon. See .github/workflows/python_wheels.yml. The `python_wheels` pipeline sets up environment variables prefixed `CIBW_*` to indicate the target OS and processor. The pipeline then invokes the script `build_python_wheels.sh`, which in turns calls `cibuildwheel` to build the wheel. The `cibuildwheel` is a library that sets up a suitable Python environment for each OS and processor target. Since we don't have Apple Silion machine in GitHub Actions, cross-compilation

is needed; `cibuildwheel` takes care of the complex task of cross-compiling a Python wheel. (Note that `cibuildwheel` will call `setup.py bdist_wheel`. Since XGBoost has a native library component, `setup.py` contains a glue code to call CMake and a C++ compiler to build the native library on the fly.)

Reproduce CI testing environments using Docker containers

In our CI pipelines, we use Docker containers extensively to package many software packages together. You can reproduce the same testing environment as the CI pipelines by running Docker 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/install-guide.html#installing-on-ubuntu-and-debian The runtime lets you access NVIDIA GPUs inside a Docker container.

Building and Running Docker containers locally

For your convenience, we provide the wrapper script `tests/ci_build/ci_build.sh`. You can use it as follows:

```
tests/ci_build/ci_build.sh <CONTAINER_TYPE> <DOCKER_BINARY> --build-arg <BUILD_ARG> \
  <COMMAND> ...
```

where:

- `<CONTAINER_TYPE>` is the identifier for the container. The wrapper script will use the container definition (Dockerfile) located at `tests/ci_build/Dockerfile.<CONTAINER_TYPE>`. For example, setting the container type to `gpu` will cause the script to load the Dockerfile `tests/ci_build/Dockerfile.gpu`.

- `<DOCKER_BINARY>` must be either `docker` or `nvidia-docker`. Choose `nvidia-docker` as long as you need to run any GPU code.

- `<BUILD_ARG>` is a build argument to be passed to Docker. Must be of form `VAR=VALUE`. Example: `--build-arg CUDA_VERSION_ARG=11.0`. You can pass multiple `--build-arg`.

- `<COMMAND>` is the command to run inside the Docker container. This can be more than one argument. Example: `tests/ci_build/build_via_cmake.sh -DUSE_CUDA=ON -DUSE_NCCL=ON`.

Optionally, you can set the environment variable `CI_DOCKER_EXTRA_PARAMS_INIT` to pass extra arguments to Docker. For example:

```
# Allocate extra space in /dev/shm to enable NCCL
export CI_DOCKER_EXTRA_PARAMS_INIT='--shm-size=4g'
# Run multi-GPU test suite
tests/ci_build/ci_build.sh gpu nvidia-docker --build-arg CUDA_VERSION_ARG=11.0 \
  tests/ci_build/test_python.sh mgpu
```

To pass multiple extra arguments:

```
export CI_DOCKER_EXTRA_PARAMS_INIT='-e VAR1=VAL1 -e VAR2=VAL2 -e VAR3=VAL3'
```

Update pipeline definitions for BuildKite CI

BuildKite is a SaaS (Software as a Service) platform that orchestrates cloud machines to host CI pipelines. The Build-Kite platform allows us to define CI pipelines as a declarative YAML file.

The pipeline definitions are found in `tests/buildkite/`:

- `tests/buildkite/pipeline-win64.yml`: This pipeline builds and tests XGBoost for the Windows platform.

- `tests/buildkite/pipeline-mgpu.yml`: This pipeline builds and tests XGBoost with access to multiple NVIDIA GPUs.

- `tests/buildkite/pipeline.yml`: This pipeline builds and tests XGBoost with access to a single NVIDIA GPU. Most tests are located here.

Managing Elastic CI Stack with BuildKite

BuildKite allows us to define cloud resources in a declarative fashion. Every configuration step is now documented explicitly as code.

Prerequisite: You should have some knowledge of CloudFormation. CloudFormation lets us define a stack of cloud resources (EC2 machines, Lambda functions, S3 etc) using a single YAML file.

Prerequisite: Gain access to the XGBoost project's AWS account (`admin@xgboost-ci.net`), and then set up a credential pair in order to provision resources on AWS. See Creating an IAM user in your AWS account.

- Option 1. Give full admin privileges to your IAM user. This is the simplest option.

- Option 2. Give limited set of permissions to your IAM user, to reduce the possibility of messing up other resources. For this, use the script `tests/buildkite/infrastructure/service-user/create_service_user.py`.

Worker Image Pipeline

Building images for worker machines used to be a chore: you'd provision an EC2 machine, SSH into it, and manually install the necessary packages. This process is not only laborous but also error-prone. You may forget to install a package or change a system configuration.

No more. Now we have an automated pipeline for building images for worker machines.

- Run `tests/buildkite/infrastructure/worker-image-pipeline/create_worker_image_pipelines.py` in order to provision CloudFormation stacks named `buildkite-linux-amd64-gpu-worker` and `buildkite-windows-gpu-worker`. They are pipelines that create AMIs (Amazon Machine Images) for Linux and Windows workers, respectively.

- Navigate to the CloudFormation web console to verify that the image builder pipelines have been provisioned. It may take some time.

- Once they pipelines have been fully provisioned, run the script `tests/buildkite/infrastructure/worker-image-pipeline/run_pipelines.py` to execute the pipelines. New AMIs will be uploaded to the EC2 service. You can locate them in the EC2 console.

- Make sure to modify `tests/buildkite/infrastructure/aws-stack-creator/metadata.py` to use the correct AMI IDs. (For `linux-amd64-cpu` and `linux-arm64-cpu`, use the AMIs provided by BuildKite. Consult the `AWSRegion2AMI` section of https://s3.amazonaws.com/buildkite-aws-stack/latest/aws-stack.yml.)

EC2 Autoscaling Groups

In EC2, you can create auto-scaling groups, where you can dynamically adjust the number of worker instances according to workload. When a pull request is submitted, the following steps take place:

1. GitHub sends a signal to the registered webhook, which connects to the BuildKite server.

2. BuildKite sends a signal to a Lambda function named `Autoscaling`.

3. The Lambda function sends a signal to the auto-scaling group. The group scales up and adds additional worker instances.

4. New worker instances run the test jobs. Test results are reported back to BuildKite.

5. When the test jobs complete, BuildKite sends a signal to `Autoscaling`, which in turn requests the autoscaling group to scale down. Idle worker instances are shut down.

To set up the auto-scaling group, run the script `tests/buildkite/infrastructure/aws-stack-creator/create_stack.py`. Check the CloudFormation web console to verify successful provision of auto-scaling groups.

PYTHON MODULE INDEX

X

W

X