xgboost

Release 1.4.0-SNAPSHOT

xgboost developers

Jan 05, 2021
# CONTENTS

1 Contents.......................................................... 3
   1.1 Installation Guide .......................................... 3
   1.2 Get Started with XGBoost ...................................... 12
   1.3 XGBoost Tutorials .............................................. 14
   1.4 Frequently Asked Questions .................................. 66
   1.5 XGBoost GPU Support ........................................... 67
   1.6 XGBoost Parameters ............................................. 72
   1.7 XGBoost Tree methods ......................................... 83
   1.8 XGBoost Python Package ....................................... 84
   1.9 XGBoost R Package ............................................. 142
   1.10 XGBoost JVM Package ......................................... 160
   1.11 XGBoost.jl ...................................................... 178
   1.12 XGBoost C Package ............................................ 179
   1.13 XGBoost C++ API .............................................. 179
   1.14 XGBoost Command Line version ............................... 179
   1.15 Contribute to XGBoost ........................................ 179

Python Module Index .................................................. 193

Index ........................................................................ 195
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.
1.1 Installation Guide

Note: Pre-built binary wheel for Python

If you are planning to use Python, consider installing XGBoost from a pre-built binary wheel, available from Python Package Index (PyPI). You may download and install it by running

```bash
# Ensure that you are downloading one of the following:
# * xgboost-{version}-py2.py3-none-manylinux1_x86_64.whl
# * xgboost-{version}-py2.py3-none-win_amd64.whl
pip3 install xgboost
```

- The binary wheel will support GPU algorithms (`gpu_hist`) on machines with NVIDIA GPUs. Please note that training with multiple GPUs is only supported for Linux platform. See XGBoost GPU Support.
- Currently, we provide binary wheels for 64-bit Linux, macOS and Windows.
- Nightly builds are available. You can go to this page, find the wheel with the commit id you want and install it with pip:

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

1.1.1 Building XGBoost from source

This page gives instructions on how to build and install XGBoost from scratch on various systems.

Note: Use of Git submodules

XGBoost uses Git submodules to manage dependencies. So when you clone the repo, remember to specify `--recursive` option:

```bash
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:

```bash
git submodule init
git submodule update
```
Please refer to *Trouble Shooting* section first if you have any problem during installation. If the instructions do not work for you, please feel free to ask questions at the user forum.

### Contents

- *Installation Guide*
  - Building XGBoost from source
  - Building the Shared Library
    * Building on Linux distributions
    * Building on OSX
      - Install with pip: simple method
      - Build from the source code - advanced method
    * Building on Windows
      - Compile XGBoost with Microsoft Visual Studio
    * Building with GPU support
    * Makefiles
- *Python Package Installation*
  - Building XGBoost library for Python for Windows with MinGW-w64 (Advanced)
- *R Package Installation*
  - Installing pre-packaged version
  - Installing the development version (Linux / Mac OSX)
  - Installing the development version with Visual Studio
  - Installing R package with GPU support
- *Trouble Shooting*
- *Building the Documentation*

### 1.1.2 Building the Shared Library

Our goal is to build the shared library:

- On Linux/OSX the target library is `libxgboost.so`
- 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). For building language specific package, see corresponding sections in this document. The minimal building requirement is

- A recent C++ compiler supporting C++11 (g++-5.0 or higher)
- CMake 3.13 or higher.

For a list of CMake options, see `#-- Options` in `CMakeLists.txt` on top level of source tree.
Building on Linux distributions

On Ubuntu, one builds XGBoost by running CMake:

```bash
git clone --recursive https://github.com/dmlc/xgboost
cd xgboost
mkdir build
cd build
cmake ..
make -j$(nproc)
```

Building on OSX

Install with pip: simple method

First, obtain the OpenMP library (`libomp`) with Homebrew (https://brew.sh/) to enable multi-threading (i.e. using multiple CPU threads for training):

```bash
brew install libomp
```

Then install XGBoost with `pip`:

```bash
pip3 install xgboost
```

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

Build from the source code - advanced method

Obtain `libomp` from Homebrew:

```bash
brew install libomp
```

Now clone the repository:

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

Create the `build/` directory and invoke CMake. After invoking CMake, you can build XGBoost with `make`:

```bash
mkdir build
cd build
cmake ..
make -j4
```

You may now continue to *Python Package Installation*. 
Building on Windows

You need to first clone the XGBoost repo with `--recursive` option, to clone the submodules. We recommend you use Git for Windows, as it comes with a standard Bash shell. This will highly ease the installation process.

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

XGBoost support compilation with Microsoft Visual Studio and MinGW.

Compile XGBoost with Microsoft Visual Studio

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.

Building with GPU support

XGBoost can be built with GPU support for both Linux and Windows using CMake. GPU support works with the Python package as well as the CLI version. See Installing 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
cmake .. -DUSE_CUDA=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 [here](#).

**Note:** Enabling distributed GPU training

By default, distributed GPU training is disabled and only a single GPU will be used. To enable distributed GPU training, set the option `USE_NCCL=ON`. Distributed GPU training depends on NCCL2, available at this [link](#). Since NCCL2 is only available for Linux machines, **distributed GPU training is available only for Linux**.

```
mkdir build
cd build
cmake .. -DUSE_CUDA=ON -DUSE_NCCL=ON -DNCC_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.)

**Note:** Visual Studio 2017 Win64 Generator may not work

Choosing the Visual Studio 2017 generator may cause compilation failure. When it happens, specify the 2015 compiler by adding the `-T` option:

```
cmake .. -G"Visual Studio 15 2017 Win64" -T v140,cuda=8.0 -DUSE_CUDA=ON
```

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`.

**Makefiles**

It’s only used for creating shorthands for running linters, performing packaging tasks etc. So the remaining makefiles are legacy.

**Python Package Installation**

The Python package is located at `python-package/`. 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. Setup tools 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)
```

(continues on next page)
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:

```python
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:

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

### Building XGBoost library for Python 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`.

### R Package Installation

#### Installing pre-packaged version

You can install XGBoost from CRAN just like any other R package:

```r
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.

### 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, `devtools::install_github` can no longer be used to install the latest version of R package. Thus, one has to run `git` to check out the code first:

```
git clone --recursive https://github.com/dmlc/xgboost
cd xgboost
git submodule init
git submodule update
mkdir build
cd build
cmake .. -DR_LIB=ON
make -j$(nproc)
make install
```

If all fails, try *Building the shared library* to see whether a problem is specific to R package or not. Notice that the R package is installed by CMake directly.

### Installing the development version with Visual Studio

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:
  ```
  %RTOOLS40\usr\bin% %RTOOLS40\mingw64\bin%
  ```
- If you are using R 3.x with RTools 3.x:
  ```
  %Rtools\bin%
  %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.
Installing 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
make --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.

**Trouble Shooting**

1. Compile failed after `git pull`

   Please first update the submodules, clean all and recompile:
   ```
git submodule update && make clean_all && make -j4
   ```

**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**
  - sphinx
  - breathe
  - guzzle_sphinx_theme
  - recommonmark
  - mock
  - sh
  - graphviz
  - matplotlib

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.

**1.2 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.

**1.2.1 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.
1.2.2 Python

```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 = {'max_depth':2, 'eta':1, 'objective':'binary:logistic' }
nnum_round = 2
bst = xgb.train(param, dtrain, num_round)
# make prediction
preds = bst.predict(dtest)
```

1.2.3 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)
```

1.2.4 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)
```

1.2.5 Scala

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

object XGBoostScalaExample {
  def main(args: Array[String]) {
    // read training 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")

1.3 XGBoost Tutorials

This section contains official tutorials inside XGBoost package. See Awesome XGBoost for more resources.

1.3.1 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 $\theta$. Usually we will use $\theta$ 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 $\theta$ 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 fits 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 $\Omega$ is the regularization term. The training loss measures how predictive our model is with respect to the training data. A common choice for $L$ is the **mean squared error**, which is given by

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

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

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

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

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

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

Decision Tree Ensembles

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

Input: age, gender, occupation, ...  
Like the computer game X

![Decision Tree Example](image)

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, $f$ 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

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

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

\[
\begin{align*}
\hat{y}_i^{(0)} &= 0 \\
\hat{y}_i^{(1)} &= f_1(x_i) = \hat{y}_i^{(0)} + f_1(x_i) \\
\hat{y}_i^{(2)} &= f_1(x_i) + f_2(x_i) = \hat{y}_i^{(1)} + f_2(x_i) \\
&\vdots \\
\hat{y}_i^{(t)} &= \sum_{k=1}^{t} f_k(x_i) = \hat{y}_i^{(t-1)} + f_t(x_i)
\end{align*}
\]

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

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

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:

\[
\text{obj}^{(t)} = \sum_{i=1}^{n} \left[ l(y_i, \hat{y}_i^{(t-1)}) + g_t f_t(x_i) + \frac{1}{2} h_t f_t^2(x_i) \right] + \Omega(f_t) + \text{constant}
\]

where the $g_t$ and $h_t$ are defined as

\[
\begin{align*}
g_t &= \partial_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)}) \\
h_t &= \partial^2_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)})
\end{align*}
\]

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

\[
\sum_{i=1}^{n} \left[ g_t f_t(x_i) + \frac{1}{2} h_t f_t^2(x_i) \right] + \Omega(f_t)
\]

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 $g_t$ and $h_t$. 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 $g_t$ and $h_t$ as input!
Model Complexity

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

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

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

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

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

The Structure Score

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

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

$$ = \sum_{j=1}^{T} [\left( \sum_{i \in I_j} g_i \right) w_j + \frac{1}{2} (\sum_{i \in I_j} h_i + \lambda) w_j^2] + \gamma T $$

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.
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 $\gamma$, we would do better not to add that branch. This is exactly the pruning techniques in tree based models! By using the principles of supervised learning, we can naturally come up with the reason these techniques work :)

For real valued data, we usually want to search for an optimal split. To efficiently do so, we place all the instances in sorted order, like the following picture.
A left to right scan is sufficient to calculate the structure score of all possible split solutions, and we can find the best split efficiently.

**Note:** Limitation of additive tree learning

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

---

**Final words on XGBoost**

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

---

### 1.3.2 Introduction to Model IO

In XGBoost 1.0.0, we introduced experimental 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. The support for binary format will be continued in the future until JSON format is no-longer experimental and has satisfying performance. 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 JSON format, which can be enabled by providing the file name with `.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 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.

---

1.3. XGBoost Tutorials
• R package: when the `xgb.Booster` object is persisted with the built-in functions `saveRDS` or `save`.

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` as file extension:

Listing 1: Python

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

Listing 2: R

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

While for memory snapshot, JSON is the default starting with xgboost 1.3.

### 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. To help easing the mitigation, we created a simple script for converting pickled XGBoost 0.90 Scikit-Learn interface object to XGBoost 1.0.0 native model. Please note that the script suits simple use cases, and it’s advised not to use pickle when stability is needed. It’s located in xgboost/doc/python with the name convert_090to100.py. See comments in the script for more details.

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": {
```
json_object = {
    "grow_gpu_hist": {
        "gpu_hist_train_param": {
            "debug_synchronize": "0",
            "gpu_batch_nrows": "0",
            "single_precision_histogram": "0"
        },
        "train_param": {
            "alpha": "0",
            "cache_opt": "1",
            "colsample_bylevel": "1",
            "colsample_bynode": "1",
            "colsample_bytree": "1",
            "default_direction": "learn",
            ...
            "subsample": "1"
        }
    },
    "learner_train_param": {
        "booster": "gbtree",
        "disable_default_eval_metric": "0",
        "dsplit": "auto",
        "objective": "reg:squarederror"
    },
    "metrics": [],
    "objective": {
        "name": "reg:squarederror",
        "reg_loss_param": {
            "scale_pos_weight": "1"
        }
    }
}

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 initial draft of JSON schema for the output model (not serialization, which will not be stable as noted above). It’s subject to change due to the beta status. 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.

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

(continues on next page)
"loss_changes": {
  "type": "array",
  "items": {
    "type": "number"
  }
},
"sum_hessian": {
  "type": "array",
  "items": {
    "type": "number"
  }
},
"base_weights": {
  "type": "array",
  "items": {
    "type": "number"
  }
},
"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"
  }
},
"default_left": {
  "type": "array",
  "items": {
    "type": "boolean"
  }
},
"categories": {
  "type": "array",
  "items": {
    "type": "number"
  }
},
"node_id": 1,
"split": 0,
"weight": 0,
"gain": 0,
"decrease_hessian": 0,
"node_depth": 1,
"children": [
  0,
  1
],
"left_split_index": -1,
"right_split_index": -1,
"left_child": 0,
"right_child": 1,
"left_weight": 0,
"right_weight": 0,
"leafweight": 0,
"depth": 1,
"left_leaf": false,
"right_leaf": false,
"left_child_type": "split",
"right_child_type": "leaf",
"node_locked": false,
"left_leaf_locked": false,
"right_leaf_locked": false,
"left_child_is_leaf": false,
"right_child_is_leaf": true,
"is_leaf": true,
"is_split": false,
"type": "integer"
},
"categories_nodes": {
  "type": "array",
  "items": {
    "type": "integer"
  }
},
"categories_segments": {
  "type": "array",
  "items": {
    "type": "integer"
  }
},
"categorical_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"
    },
    "size_leaf_vector": {
      "type": "string"
    }
  }
},
"tree_param": {
  "type": "object",
  "properties": {
    "num_nodes": {
      "type": "string"
    },
    "size_leaf_vector": {
      "type": "string"
    },
    "num_feature": {
      "type": "string"
    }
  }
},
"reg_loss_param": {
  "type": "object",
  "properties": {
    "scale_pos_weight": {
      "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": {
    "softmax_type": {
      "type": "string"
    }
  }
}
"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", "const": 1 },
      { "type": "number", "minimum": 0 },
      { "type": "number", "minimum": 0 }
    ],
    "minItems": 3,
    "maxItems": 3
  },
  "learner": {
    "type": "object",
    "properties": {
      "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"
      ]
    }
  ]
}
```


```json
]
},
{
  "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" }
      }
    }
  }
}
```

(continues on next page)

```
},
  "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"
  ],
},
  {
    "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"]
```

(continues on next page)
{ "type": "object", "properties": { "name": { "const": "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": { } }

(continues on next page)
```json
{
    "base_score": { "type": "string" },
    "num_class": { "type": "string" },
    "num_feature": { "type": "string" }
}
```

### Future Plans

Right now using the JSON format incurs longer serialisation time, we have been working on optimizing the JSON implementation to close the gap between binary format and JSON format.

### 1.3.3 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.

### 1.3.4 Distributed XGBoost on Kubernetes

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

1.3.5 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. Right now it is still under construction and may change (with proper warnings) in the future. 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
- Working with other clusters
- Threads
- Working with asyncio
- Evaluation and Early Stopping
- Why is the initialization of DaskDMatrix so slow and throws weird errors
- Memory Usage
- Limitations
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

cluster = dask.distributed.LocalCluster(n_workers=4, threads_per_worker=1)
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 `dask.distributed.LocalCluster`, then connect a `dask.distributed.Client` to this cluster, setting up an environment for later computation.

We then create a `DaskDMatrix` object and pass it to `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 `xgb.dask.predict`:

```python
prediction = xgb.dask.predict(client, output, dtrain)
```

Or equivalently, pass `output['booster']`:

```python
prediction = xgb.dask.predict(client, output['booster'], dtrain)
```

Here `prediction` is a `dask Array` object containing predictions from model.

Alternatively, XGBoost also implements the Scikit-Learn interface with `DaskXGBClassifier` and `DaskXGBRegressor`. See `xgboost/demo/dask` 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 inplace predict support on `dask` interface, which can help reducing both memory usage and prediction time.

```python
# dtrain is the DaskDMatrix defined above.
prediction = xgb.dask.predict(client, booster, dtrain)
```
or equivalently:

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

Also for inplace prediction:

```python
booster.set_param({'predictor': 'gpu_predictor'})
# where X is a dask DataFrame or dask Array.
prediction = xgb.dask.inplace_predict(client, booster, X)
```

### 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})
```

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

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

```python
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:

```python
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
    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.

```python
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.

```python
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.

```python
booster = result["booster"]
print(booster.best_iteration)
best_model = booster[:booster.best_iteration]
```
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 practices on reducing memory usage with dask and xgboost.

- In a distributed workflow, 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. DaskDeviceQuantileDMatrix as a drop in replacement for DaskDMatrix to reduce overall memory usage. See demo/dask/gpu_training.py for an example.

- Use inplace prediction when possible.

References:

Limitations

Basic functionality including model training and generating classification and regression predictions have been implemented. However, there are still some other limitations we haven’t addressed yet:

- Label encoding for the DaskXGBClassifier classifier may not be supported. So users need to encode their training labels into discrete values first.

- Ranking is not yet supported.

1.3.6 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 an instruction of new tree booster dart.
Original paper

Rashmi Korlakai Vinayak, Ran Gilad-Bachrach. “DART: Dropouts meet Multiple Additive Regression Trees.” JMLR.

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 $D = \sum_{i \in K} F_i$ be the leaf scores of dropped trees and $F_m = \eta \ddot{F}_m$ be the leaf scores of a new tree.
- The objective function is as follows:
  \[
  \text{Obj} = \sum_{j=1}^{n} L \left( y_j, \hat{y}_j^{m-1} - D_j + \ddot{F}_m \right) + \Omega \left( \ddot{F}_m \right).
  \]
- $D$ and $F_m$ are overshooting, so using scale factor
  \[
  \hat{y}_j^m = \sum_{i \in K} F_i + a \left( \sum_{i \in K} F_i + bF_m \right).
  \]

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.
    \[
    a \left( \sum_{i \in K} F_i + \frac{1}{k} F_m \right) = a \left( \sum_{i \in K} F_i + \frac{\eta}{k} \ddot{F}_m \right)
    \approx a \left( 1 + \frac{\eta}{k} \right) D
    = a \frac{k + \eta}{k} D = D,
    \]
    \[
    a = \frac{k}{k + \eta}
    \]
  - **forest**: New trees have the same weight of sum of dropped trees (forest).
\[
a \left( \sum_{i \in K} F_i + F_m \right) = a \left( \sum_{i \in K} F_i + \eta F_m \right)
\]
\[
\sim a (1 + \eta) D
\]
\[
= a(1 + \eta)D = D;
\]
\[
a = \frac{1}{1 + \eta}.
\]

- **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)
```

### 1.3.7 Monotonic Constraints

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

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

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

declining whenever \( x \leq x' \) is an **increasing constraint**; or

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

declining whenever \( x \leq x' \) is a **decreasing constraint**.

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

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

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

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

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

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

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

**Enforcing Monotonic Constraints in XGBoost**

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

Suppose the following code fits your model without monotonicity constraints

```python
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

```python
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.

**Choice of tree construction algorithm.** To use monotonic constraints, be sure to set the `tree_method` parameter to one of `exact`, `hist`, and `gpu_hist`.

**Note for the ‘hist’ tree construction algorithm.** If `tree_method` is set to either `hist` 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.

1.3. XGBoost Tutorials 45
1.3.8 Random Forests 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:

```python
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:

```python
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.

1.3.9 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 \((\[0, 1\], \[2, 3, 4\])\).

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:

```python
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:

```python
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)
```

Choice of tree construction algorithm. To use feature interaction constraints, be sure to set the `tree_method` parameter to one of the following: `exact`, `hist`, `approx` or `gpu_hist`. Support for `gpu_hist` and `approx` is added only in 1.0.0.

Advanced topic

The intuition behind interaction constraint is simple. User 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 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 following diagram, root splits at feature 2. because all its descendants should be able to interact with it, so at the second layer all 4 features are legitimate split candidates for further splitting, disregarding specified constraint sets.

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 same with \([\[0, 1, 2\]]\). Since no matter which feature is chosen for split in root node, all its descendants have to include every feature as legitimate split candidates to avoid violating interaction constraints.

For one last example, we use \([\[0, 1\], \[1, 3, 4\]]\) and choose feature 0 as split for root node. At the second layer of 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 second layer splits at feature 1. But due to the fact that 1 also belongs to second constraint set $\{1, 3, 4\}$, at third layer, we need to include all features as candidates to comply with its ascendants.

1.3.10 Survival Analysis with Accelerated Failure Time

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

**What is survival analysis?**

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

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

The second bullet point is crucial and we should dwell on it more. As you may have guessed from the name, one of the earliest applications of survival analysis is to model mortality of a given population. Let’s take NCCTG Lung Cancer Dataset as an example. The first 8 columns represent features and the last column, Time to death, represents the label.
Fig. 2: \( \{0, 1, 3, 4\} \) represents the sets of legitimate split features.
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, +∞), 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, +∞), 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:

\[ \ln Y = \langle w, x \rangle + \sigma Z \]

where

- \( x \) is a vector in \( \mathbb{R}^d \) representing the features.
- \( w \) 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 \( \langle w, x \rangle \) away from the true log label \( \ln Y \).
- \( \sigma \) is a parameter that scales the size of \( Z \).

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

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

**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 $[a, a]$.

<table>
<thead>
<tr>
<th>Censoring type</th>
<th>Interval form</th>
<th>Lower bound finite?</th>
<th>Upper bound finite?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uncensored</td>
<td>$[a, a]$</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Right-censored</td>
<td>$(a, +\infty)$</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Left-censored</td>
<td>$[0, b]$</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Interval-censored</td>
<td>$[a, b]$</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

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 3: 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)

datast.set_float_info('label_lower_bound', y_lower_bound)
datast.set_float_info('label_upper_bound', y_upper_bound)
```

**Listing 4: 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)

datast.set_float_info('label_lower_bound', y_lower_bound)
datast.set_float_info('label_upper_bound', y_upper_bound)
```

Now we are ready to invoke the training API:
Listing 5: Python

```python
define parameters
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}

train the model
bst = xgb.train(params, dtrain, num_boost_round=5,
                evals=[(dtrain, 'train')])
```

Listing 6: R

```r
define parameters
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)

create a watchlist
watchlist <- list(train = dtrain)

train the model
bst <- xgb.train(params, dtrain, nrounds=5, watchlist)
```

We set the `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 $\sigma$.

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

<table>
<thead>
<tr>
<th><code>aft_loss_distribution</code></th>
<th>Probability Density Function (PDF)</th>
</tr>
</thead>
<tbody>
<tr>
<td>normal</td>
<td>$\frac{\exp(-z^2/2)}{\sqrt{2\pi}}$</td>
</tr>
<tr>
<td>logistic</td>
<td>$\frac{(1+e^z)^2}{e^z}$</td>
</tr>
<tr>
<td>extreme</td>
<td>$e^z e^{-\exp z}$</td>
</tr>
</tbody>
</table>

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`.

### 1.3.11 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 usefull 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.

- **Requirements**
  - Install XGBoost on conda environment
  - Configure CMakeList.txt file of your application to link with XGBoost
• Usefull Tips To Remember
• Sample examples along with Code snippet to use C API functions

Requirements

Install CMake - Follow the cmake installation documentation for instructions. Install Conda - Follow the conda installation documentation for instructions

Install XGBoost on conda environment

Run the following commands on your terminal. The below commands will install the XGBoost in your XGBoost folder of the repository cloned

```bash
# 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 .. -DBUILD_STATIC_LIB=ON -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:

```bash
# Nagivate to the build directory for your application
cd build
# Activate the Conda environment where we previously installed XGBoost
conda activate [env_name]
# Invoke CMake with CMAKE_PREFIX_PATH

cmake .. -DCMAKE_INSTALL_PREFIX=$CONDA_PREFIX
make
```

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.
Usefull Tips To Remember

Below are some usefull 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:

   ```c
   #define safe_xgboost(call) { \
   int err = (call); \
   if (err != 0) { \
     fprintf(stderr, "%s:%d: error in %s: %s\n", __FILE__, __LINE__, #call, \
     XGBGetLastError()); \
     exit(1); \
   } \
   }
   
   In your application, wrap all C API function calls with the macro as follows:

   ```c
   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.

   ```c
   #define safe_xgboost(call) { \
   int err = (call); \
   if (err != 0) { \
     throw new Exception(std::string(__FILE__) + ":\n     " + std::to_string(__LINE__) + \
     ": error in " + #call + ":" + XGBGetLastError()); \
   } \
   }
   
   c. Assertion technique: It works both in C/ C++. If expression evaluates to 0 (false), then the expression, source code filename, and line number are sent to the standard error, and then abort() function is called. It can be used to test assumptions made by you in the code.

   ```c
   DMatrixHandle dmat;
   assert( XGDMatrixCreateFromFile("training_data.libsvm", 0, &dmat) == 0);
   ```

2. Always remember to free the allocated space by BoosterHandle & DMatrixHandle appropriately:

   ```c
   #include <assert.h>
   #include <stdio.h>
   #include <stdlib.h>
   #include <xgboost/c_api.h>

   int main(int arg, char** argv) {
   int silent = 0;
   
   BoosterHandle booster;
   // do something with booster
   //free the memory
   XGBoosterFree(booster)
   DMatrixHandle DMatrixHandle_param;
   (continues on next page)
   ```
3. For tree models, it is important to use consistent data formats during training and scoring/predicting otherwise it will result in wrong outputs. Example if we our training data is in dense matrix format then your prediction dataset should also be a dense matrix or if training in libsvm format then dataset for prediction should also be in libsvm format.

4. Always use strings for setting values to the parameters in booster handle object. The parameter value can be of any data type (e.g. int, char, float, double, etc), but they should always be encoded as strings.

```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 function

```c
// 1D matrix
const int data1[] = { 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, -1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

// 2D matrix
const int ROWS = 5, COLS = 3;
const int data2[ROWS][COLS] = { {1, 2, 3}, {2, 4, 6}, {3, -1, 9}, {4, 8, -1}, {2, 5, -1} , {0, 1, 5} };;
DMatrixHandle dmatrix1, dmatrix2;
// Pass the matrix, no of rows & columns contained in the matrix variable
// here '0' represents the missing value in the matrix dataset
// dmatrix variable will contain the created DM atrix using it
safe_xgboost(XGDMatrixCreateFromMat(data1, 1, 50, 0, &dmatrix1));
// here -1 represents the missing value in the matrix dataset
safe_xgboost(XGDMatrixCreateFromMat(data2, ROWS, COLS, -1, &dmatrix2));
```

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

```c
BoosterHandle booster;
cost int eval_dmats_size;
// We assume that training and test data have been loaded into 'train' and 'test'
DMatrixHandle eval_dmats[eval_dmats_size] = {train, test};
safe_xgboost(XGBoosterCreate(eval_dmats, eval_dmats_size, &booster));
4. For each DMatrix object, set the labels using XGDMatrixSetFloatInfo. Later you can access the label using XGDMatrixGetFloatInfo.

```c
const int ROWS = 5, COLS = 3;
const int data[ROWS][COLS] = {{1, 2, 3}, {2, 4, 6}, {3, -1, 9}, {4, 8, -1}, {2, 5, 1} -> {0, 1, 5}};
DMatrixHandle dmatrix;
// variable to store labels for the dataset created from above matrix
float labels[ROWS];
for (int i = 0; i < ROWS; i++) {
    labels[i] = i;
}
// Loading the labels
safe_xgboost(XGDMatrixSetFloatInfo(dmatrix, "labels", labels, ROWS));

// reading the labels and store the length of the result
bst_ulong result_len;

// labels result
const float *result;

safe_xgboost(XGDMatrixGetFloatInfo(dmatrix, "labels", &result_len, &result));
for (unsigned int i = 0; i < result_len; i++) {
    printf("label[%i] = %f\n", i, result[i]);
}
```

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

```c
BoosterHandle booster;
safe_xgboost(XGBoosterSetParam(booster, "booster", "gblinear"));
// default max_depth = 6
safe_xgboost(XGBoosterSetParam(booster, "max_depth", "3"));
// default eta = 0.3
safe_xgboost(XGBoosterSetParam(booster, "eta", "0.1"));
```

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

```c
int num_of_iterations = 20;
const char* eval_names[eval_dmats_size] = {"train", "test"};
const char* eval_result = NULL;
for (int i = 0; i < num_of_iterations; ++i) {
    // Update the model performance for each iteration
    safe_xgboost(XGBoosterUpdateOneIter(booster, i, train));
    // Give the statistics for the learner for training & testing dataset in terms of error after each iteration
    safe_xgboost(XGBoosterEvalOneIter(booster, i, eval_dmats, eval_names, eval_dmats_size, &eval_result));
    printf("%s\n", eval_result);
}
```
Note: For customized loss function, use XGBoosterBoostOneIter function instead and manually specify the gradient and 2nd order gradient.

7. Predict the result on a test set using XGBoosterPredict

```c
bst_ulong output_length;
const float *output_result;
safe_xgboost(XGBoosterPredict(booster, test, 0, 0, &output_length, &output_result));
for (unsigned int i = 0; i < output_length; i++) {
    printf("prediction[%i] = %f \n", i, output_result[i]);
}
```

8. Free all the internal structure used in your code using XGDMatrixFree and XGBoosterFree. This step is important to prevent memory leak.

```c
safe_xgboost(XGDMatrixFree(dmatrix));
safe_xgboost(XGBoosterFree(booster));
```

9. Get the number of features in your dataset using XGBoosterGetNumFeature.

```c
bst_ulong num_of_features = 0;
// Assuming booster variable of type BoosterHandle is already declared
// and dataset is loaded and trained on booster
// storing the results in num_of_features variable
safe_xgboost(XGBoosterGetNumFeature(booster, &num_of_features));
// Printing number of features by type conversion of num_of_features variable from bst_ulong to unsigned long
printf("num_feature: %lu\n", (unsigned long)(num_of_features));
```

10. Load the model using XGBoosterLoadModel function

```c
BoosterHandle booster;
const char *model_path = "/path/of/model";
// create booster handle first
safe_xgboost(XGBoosterCreate(NULL, 0, &booster));
// set the model parameters here
// load model
safe_xgboost(XGBoosterLoadModel(booster, model_path));
// predict the model here
```
1.3.12 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 7: train.txt

<table>
<thead>
<tr>
<th>Label</th>
<th>Feature 1</th>
<th>Feature 2</th>
<th>Feature 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>101:1.2</td>
<td>102:0.3</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1:2.1</td>
<td>10001:300</td>
<td>10002:400</td>
</tr>
<tr>
<td>0</td>
<td>0:1.3</td>
<td>0:0.3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0:0.01</td>
<td>1:0.3</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0:0.2</td>
<td>1:0.3</td>
<td></td>
</tr>
</tbody>
</table>

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 8: train.txt.group

<table>
<thead>
<tr>
<th>Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

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 9: train.txt.weight
```

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>0.5</td>
<td>1</td>
<td>0.5</td>
</tr>
</tbody>
</table>

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:
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:

```plaintext
Listing 11: 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:

```plaintext
Listing 12: 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.
1.3.13 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
1.3.14 Using XGBoost External Memory Version

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**

External memory is fully 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:

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

For more information, see this paper.

**Distributed Version**

The external memory mode naturally works on distributed version, you can simply set path like

```
data = "hdfs://path-to-data/#dtrain.cache"
```

XGBoost will cache the data to the local position. When you run on YARN, the current folder is temporary so that you can directly use `dtrain.cache` to cache to current folder.
Limitations

- The hist tree method hasn’t been tested thoroughly with external memory support (see this issue).
- OSX is not tested.

1.3.15 Custom Objective and Evaluation Metric

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.
- The customized functions defined here are only applicable to single node training. Distributed environment requires syncing with xgboost.rabit, the interface is subject to change hence beyond the scope of this tutorial.
- We also plan to improve the interface for multi-classes objective in the future.

In the following sections, we will provide a step by step walk through of implementing 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.

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.'''
```

(continues on next page)
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},
    dtrain=dtrain,
    num_boost_round=10,
    obj=squared_log)
```

Notice that in our definition of the objective, whether we subtract the labels from the prediction or the other way around is important. If you find the training error goes up instead of down, this might be the reason.

**Customized Metric Function**

So after having a customized objective, we might also need a corresponding metric to monitor our model’s performance. As mentioned above, the default metric for SLE is RMSLE. Similarly we define another callback like function as the new metric:

```
def rmsle(predt: np.ndarray, dtrain: xgb.DMatrix) -> Tuple[str, float]:
    ''' Root mean squared log error metric.'''
    y = dtrain.get_label()
    predt[predt < -1] = -1 + 1e-6
    elements = np.power(np.log1p(y) - np.log1p(predt), 2)
    return 'PyRMSLE', float(np.sqrt(np.sum(elements) / len(y)))
```

Since we are demonstrating in Python, the metric or objective needs not be a function, any callable object should suffice. Similarly to the objective function, our metric also accepts `predt` and `dtrain` as inputs, but returns the name of metric itself and a floating point value as result. After passing it into XGBoost as argument of `feval` parameter:

```
xgb.train({'tree_method': 'hist', 'seed': 1994,
            'disableDefaultEvalMetric': 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:
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 custom_rmsle.py.

**Multi-class objective function**

A similar demo for multi-class objective function is also available, see demo/guide-python/custom_softmax.py for details.

### 1.4 Frequently Asked Questions

This document contains frequently asked questions about XGBoost.

#### 1.4.1 How to tune parameters

See Parameter Tuning Guide.

#### 1.4.2 Description on the model

See Introduction to Boosted Trees.

#### 1.4.3 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.

#### 1.4.4 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.
1.4.5 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.

1.4.6 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.

1.4.7 Do you support LambdaMART?

Yes, XGBoost implements LambdaMART. Checkout the objective section in parameters.

1.4.8 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.

1.4.9 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.

1.4.10 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.

1.5 XGBoost GPU Support

This page contains information about GPU algorithms supported in XGBoost. To install GPU support, checkout the Installation Guide.

Note: CUDA 10.0, Compute Capability 3.5 required

The GPU algorithms in XGBoost require a graphics card with compute capability 3.5 or higher, with CUDA toolkits 10.0 or later. (See this list to look up compute capability of your GPU card.)
1.5.1 CUDA Accelerated Tree Construction Algorithms

Tree construction (training) and prediction can be accelerated with CUDA-capable GPUs.

Usage

Specify the `tree_method` parameter as one of the following algorithms.

## Algorithms

<table>
<thead>
<tr>
<th>tree_method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gpu_hist</td>
<td>Equivalent to the XGBoost fast histogram algorithm. Much faster and uses considerably less memory. NOTE: May run very slowly on GPUs older than Pascal architecture.</td>
</tr>
</tbody>
</table>

## Supported parameters

<table>
<thead>
<tr>
<th>parameter</th>
<th>gpu_hist</th>
</tr>
</thead>
<tbody>
<tr>
<td>subsample</td>
<td>✓</td>
</tr>
<tr>
<td>sampling_method</td>
<td>✓</td>
</tr>
<tr>
<td>colsample_bytree</td>
<td>✓</td>
</tr>
<tr>
<td>colsample_bylevel</td>
<td>✓</td>
</tr>
<tr>
<td>max_bin</td>
<td>✓</td>
</tr>
<tr>
<td>gamma</td>
<td>✓</td>
</tr>
<tr>
<td>gpu_id</td>
<td>✓</td>
</tr>
<tr>
<td>predictor</td>
<td>✓</td>
</tr>
<tr>
<td>grow_policy</td>
<td>✓</td>
</tr>
<tr>
<td>monotone_constraints</td>
<td>✓</td>
</tr>
<tr>
<td>interaction_constraints</td>
<td>✓</td>
</tr>
<tr>
<td>single_precision_histogram</td>
<td>✓</td>
</tr>
</tbody>
</table>

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 experimental parameter `single_precision_histogram` can be set to True to enable building histograms using single precision. This may improve speed, in particular on older architectures.

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 and R packages. See Installation Guide for details.

Listing 13: Python example

```python
param['gpu_id'] = 0
param['tree_method'] = 'gpu_hist'
```
Listing 14: With Scikit-Learn interface

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

**GPU-Accelerated SHAP values**

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

```python
model.set_param({'predictor': 'gpu_predictor'})
shap_values = model.predict(dtrain, pred_contras=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. For getting started see our tutorial *Distributed XGBoost with Dask* and worked examples here, also Python documentation *Dask API* for complete reference.

**Objective functions**

Most of the objective functions implemented in XGBoost can be run on GPU. Following table shows current support status.

<table>
<thead>
<tr>
<th>Objectives</th>
<th>GPU support</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg:squarederror</td>
<td>✓</td>
</tr>
<tr>
<td>reg:squaredlogerror</td>
<td>✓</td>
</tr>
<tr>
<td>reg:logistic</td>
<td>✓</td>
</tr>
<tr>
<td>reg:pseudohubererror</td>
<td>✓</td>
</tr>
<tr>
<td>binary:logistic</td>
<td>✓</td>
</tr>
<tr>
<td>binary:logitraw</td>
<td>✓</td>
</tr>
<tr>
<td>binary:hinge</td>
<td>✓</td>
</tr>
<tr>
<td>count:poisson</td>
<td>✓</td>
</tr>
<tr>
<td>reg:gamma</td>
<td>✓</td>
</tr>
<tr>
<td>reg:tweedie</td>
<td>✓</td>
</tr>
<tr>
<td>multi:softmax</td>
<td>✓</td>
</tr>
<tr>
<td>multi:softprob</td>
<td>✓</td>
</tr>
<tr>
<td>survival:cox</td>
<td>✓</td>
</tr>
<tr>
<td>survival:aft</td>
<td>✓</td>
</tr>
<tr>
<td>rank:pairwise</td>
<td>✓</td>
</tr>
<tr>
<td>rank:ndcg</td>
<td>✓</td>
</tr>
<tr>
<td>rank:map</td>
<td>✓</td>
</tr>
</tbody>
</table>

Objective will run on GPU if GPU updater (gpu_hist), otherwise they will run on CPU by default. For unsupported objectives XGBoost will fall back to using CPU implementation by default. Note that when using GPU ranking objective, the result is not deterministic due to the non-associative aspect of floating point summation.
Metric functions

Following table shows current support status for evaluation metrics on the GPU.

<table>
<thead>
<tr>
<th>Metric</th>
<th>GPU Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>rmse</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>rmsle</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>mae</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>mape</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>mphe</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>logloss</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>error</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>merror</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>mlogloss</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>auc</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>aucpr</td>
<td>✓</td>
</tr>
<tr>
<td>ndcg</td>
<td>✓</td>
</tr>
<tr>
<td>map</td>
<td>✓</td>
</tr>
<tr>
<td>poisson-nloglik</td>
<td>✓</td>
</tr>
<tr>
<td>gamma-nloglik</td>
<td>✓</td>
</tr>
<tr>
<td>cox-nloglik</td>
<td>✓</td>
</tr>
<tr>
<td>aft-nloglik</td>
<td>✓</td>
</tr>
<tr>
<td>interval-regression-accuracy</td>
<td>✓</td>
</tr>
<tr>
<td>gamma-deviance</td>
<td>✓</td>
</tr>
<tr>
<td>tweedie-nloglik</td>
<td>✓</td>
</tr>
</tbody>
</table>

Similar to objective functions, default device for metrics is selected based on tree updater and predictor (which is selected based on tree updater).

Benchmarks

You can run benchmarks on synthetic data for binary classification:

```python
# python tests/benchmark/benchmark_tree.py --tree_method=gpu_hist
# python tests/benchmark/benchmark_tree.py --tree_method=hist
```

Training time on 1,000,000 rows x 50 columns of random data with 500 boosting iterations and 0.25/0.75 test/train split with AMD Ryzen 7 2700 8 core @3.20GHz and Nvidia 1080ti yields the following results:

<table>
<thead>
<tr>
<th>tree_method</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>gpu_hist</td>
<td>12.57</td>
</tr>
<tr>
<td>hist</td>
<td>36.01</td>
</tr>
</tbody>
</table>
Memory usage

The following are some guidelines on the device memory usage of the `gpu_hist` updater.

If you train xgboost in a loop you may notice xgboost is not freeing device memory after each training iteration. This is because memory is allocated over the lifetime of the booster object and does not get freed until the booster is freed. A workaround is to serialise the booster object after training. See `demo/gpu_acceleration/memory.py` for a simple example.

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.

In some cases the full CSR matrix stored in floating point needs to be allocated on the device. This currently occurs for prediction in multiclass classification. If this is a problem consider setting `predictor='cpu_predictor'`. This also occurs when the external data itself comes from a source on device e.g. a cudf DataFrame. These are known issues we hope to resolve.

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.

The quantile finding algorithm also uses some amount of working device memory. It is able to operate in batches, but is not currently well optimised for sparse data.

If you are getting out-of-memory errors on a big dataset, try the external memory version.

Developer notes

The application may be profiled with annotations by specifying USE_NTVX to cmake and providing the path to the stand-alone nvtx header via NVTX_HEADER_DIR. Regions covered by the ‘Monitor’ class in cuda code will automatically appear in the nsight profiler.

1.5.2 References


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/.

1.6 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.
1.6.1 Global Configuration

The following parameters can be set in the global scope, using xgb.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).

1.6.2 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. The feature is still experimental. It’s expected to have some false positives.

- **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_pbuffer** [set automatically by XGBoost, no need to be set by user]
  - Size of prediction buffer, normally set to number of training instances. The buffers are used to save the prediction results of last boosting step.

- **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,∞]

- **max_depth** [default=6]
  - Maximum depth of a tree. Increasing this value will make the model more complex and more likely to overfit. 0 is only accepted in lossguided growing policy when tree_method is set as hist or gpu_hist and it indicates no limit on depth. Beware that XGBoost aggressively consumes memory when training a deep tree.
  - range: [0,∞] (0 is only accepted in lossguided growing policy when tree_method is set as hist or gpu_hist)

- **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,∞]

- **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,∞]

- **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.

On Python interface, when using `hist`, `gpu_hist` or `exact` tree method, 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.

- **alpha** [default=0, alias: `reg_alpha`]
  - L1 regularization term on weights. Increasing this value will make model more conservative.

- **tree_method** string [default= `auto`]
  - The tree construction algorithm used in XGBoost. See description in the reference paper and XGBoost 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** [default=0.03]

- Only used for **tree_method=approx**.
- This roughly translates into $O(1 / \text{sketch eps})$ number of bins. Compared to directly select number of bins, this comes with theoretical guarantee with sketch accuracy.
- Usually user does not have to tune this. But consider setting to a lower number for more accurate enumeration of split candidates.
- range: (0, 1)

**scale_pos_weight** [default=1]

- Control the balance of positive and negative weights, useful for unbalanced classes. A typical value to consider: $\text{sum(negative instances) / sum(positive instances)}$. See Parameters Tuning for more discussion. Also, see Higgs Kaggle competition demo for examples: R, py1, py2, py3.

**updater** [default=grow_colmaker, prune]

- 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_local_histmaker**: based on local 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).

- In a distributed setting, the implicit updater sequence value would be adjusted to **grow_histmaker, prune** by default, and you can set **tree_method** as **hist** to use **grow_histmaker**.

**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`.  
  - 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. Only relevant when `grow_policy=lossguide` is set.

• **max_bin** [default=256]  
  - Only used if `tree_method` is set to `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 tutorial 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 tutorial for more information.

**Additional parameters for hist and gpu_hist tree method**

• **single_precision_histogram** [default=``false``]  
  - Use single precision to build histograms instead of double precision.
Additional parameters for gpu_hist tree method

- deterministic_histogram, [default=`true`]  
  - Build histogram on GPU deterministically. Histogram building is not deterministic due to the non-associative aspect of floating point summation. We employ a pre-rounding routine to mitigate the issue, which may lead to slightly lower accuracy. Set to `false` to disable it.

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 `ntree_limit` to a nonzero value, e.g.

```
preds = bst.predict(dtest, ntree_limit=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 $1 / (k + \text{learning\_rate})$.  
        - Dropped trees are scaled by a factor of $k / (k + \text{learning\_rate})$.  
      * forest: new trees have the same weight of sum of dropped trees (forest).  
        - Weight of new trees are $1 / (1 + \text{learning\_rate})$.  
        - Dropped trees are scaled by a factor of $1 / (1 + \text{learning\_rate})$.  
  - 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 $O(\text{num\_feature}^2)$ 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 $O(\text{num\_feature} \times \text{top\_k})$.
    - 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.

Parameters for Tweedie Regression (objective=reg:tweedie)

- tweedie_variance_power [default=1.5]
  - Parameter that controls the variance of the Tweedie distribution $\text{var}(y) \sim E(y)^{\text{tweedie\_variance\_power}}$
    - range: (1,2)
    - Set closer to 2 to shift towards a gamma distribution
    - Set closer to 1 to shift towards a Poisson distribution.
1.6.3 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.
  - 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 = $\exp$($\text{marginal\_prediction}$) in the proportional hazard function $h(t) = h0(t) \cdot HR$).
  - 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 $n_{data} \times n_{class}$, which can be further reshaped to $n_{data} \times n_{class}$ 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** [default=0.5]
  - The initial prediction score of all instances, global bias
  - 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
  * **rmrse**: root mean square log error: \( \sqrt{\frac{1}{N} \left[ \log(pred + 1) - \log(label + 1) \right]^2} \). Default metric of **reg:squaredlogerror** objective. This metric reduces errors generated by outliers in dataset. But because log function is employed, rmrse 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**: Area under the curve. Available for binary classification and learning-to-rank tasks.
  * **aucpr**: Area under the PR curve. Available for binary classification and learning-to-rank tasks.
  * **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 deterministically via iterator number, this option will be switched on automatically on distributed mode.

1.6.4 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

1.7 XGBoost 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 grow_local_histmaker, 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.

1.7.1 Exact Solution

Exact means XGBoost considers all candidates from data for tree splitting, but underlying the objective is still inter-
preted as a Taylor expansion.

1. exact: Vanilla tree boosting tree algorithm described in reference paper. During each split finding procedure, it 
iterates over every entry 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.

1.7.2 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. grow_local_histmaker updater: An approximation tree method described in reference paper. This up-
dater is rarely used in practice so it’s 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’s faster than exact in some applications, but still slow in 
computation.

2. approx tree method: An approximation tree method described in reference paper. Different from 
grow_local_histmaker, it runs sketching before building each tree using all the rows (rows belonging 
to the root) instead of per-node dataset. Similar to grow_local_histmaker updater, hessian is used as 
weights during sketch. The algorithm can be accessed by setting tree_method to approx.

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.

4. 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.
1.7.3 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.

1.7.4 Other Updaters

1. **Pruner**: It prunes the built tree by `gamma` parameter. `pruner` is usually used as part of other tree methods.
2. **Refresh**: Refresh the statistic of built trees on a new training dataset.
3. **Sync**: Synchronize the tree among workers when running distributed training.

1.7.5 Removed Updaters

2 Updaters were removed during development due to maintainability. We describe them here solely for the interest of documentation. First one is 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. Second one is `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.

1.8 XGBoost Python Package

This page contains links to all the python related documents on python package. To install the package package, checkout `Installation Guide`.

1.8.1 Contents

**Python Package Introduction**

This document gives a basic walkthrough of xgboost python package.

**List of other Helpful Links**

- Python walkthrough code collections
- `Python API Reference`
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:

- LibSVM text format file
- Comma-separated values (CSV) file
- NumPy 2D array
- SciPy 2D sparse array
- cuDF DataFrame
- Pandas data frame, and
- XGBoost binary buffer file.

(See Text Input Format of DMatrix for detailed description of text input format.)

The data is stored in a DMatrix object.

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

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

- To load a CSV file into DMatrix:

  ```python
  # 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')
  ```

**Note:** Categorical features not supported

Note that XGBoost does not provide specialization for categorical features; if your data contains categorical features, load it as a NumPy array first and then perform corresponding preprocessing steps like one-hot encoding.

**Note:** Use Pandas to load CSV files with headers

Currently, the DMLC data parser cannot parse CSV files with headers. Use Pandas (see below) to read CSV files with headers.

- To load a NumPy array into DMatrix:

  ```python
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`:

```python
csr = scipy.sparse.csr_matrix((dat, (row, col)))
dtrain = xgb.DMatrix(csr)
```

To load a Pandas data frame into `DMatrix`:

```python
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:

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

Missing values can be replaced by a default value in the `DMatrix` constructor:

```python
dtrain = xgb.DMatrix(data, label=label, missing=-999.0)
```

Weights can be set when needed:

```python
w = np.random.rand(5, 1)
dtrain = xgb.DMatrix(data, label=label, missing=-999.0, weight=w)
```

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

### Setting Parameters

XGBoost can use either a list of pairs or a dictionary to set parameters. For instance:

- Booster parameters

```python
param = {'max_depth': 2, 'eta': 1, 'objective': 'binary:logistic'}
param['nthread'] = 4
param['eval_metric'] = 'auc'
```

- You can also specify multiple eval metrics:

```python
param['eval_metric'] = ['auc', 'ams@0']
# alternatively:
# plst = param.items()
# plst += [('eval_metric', 'ams@0')]
```

- Specify validations set to watch performance

```python
evallist = [(dtest, 'eval'), (dtrain, 'train')]
```
Training

Training a model requires a parameter list and data set.

```python
num_round = 10
bst = xgb.train(param, dtrain, num_round, evallist)
```

After training, the model can be saved.

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

The model and its feature map can also be dumped to a text file.

```python
# 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:

```python
bst = xgb.Booster({'nthread': 4})  # init model
bst.load_model('model.bin')  # load data
```

Methods including `update` and `boost` from `xgb.Booster` are designed for internal usage only. The wrapper function `xgb.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.

```python
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 three additional fields: `bst.best_score`, `bst.best_iteration` and `bst.best_ntree_limit`. Note that `xgb.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.

```python
# 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_ntree_limit`.

1.8. XGBoost Python Package
ypred = bst.predict(dtest, ntree_limit=bst.best_ntree_limit)

**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.

```python
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`.

```python
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.

```python
xgb.to_graphviz(bst, num_trees=2)
```

**Python API Reference**

This page gives the Python API reference of xgboost, please also refer to Python Package Introduction for more information about python package.

- **Global Configuration**
- **Core Data Structure**
- **Learning API**
- **Scikit-Learn API**
- **Plotting API**
- **Callback API**
- **Dask 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 https://xgboost.readthedocs.io/en/stable/parameter.html 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` (**new_config**) – Keyword arguments representing the parameters and their values

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
```

See also:

- `set_config` Set global XGBoost configuration
- `get_config` Get current values of the global configuration

```python
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 https://xgboost.readthedocs.io/en/stable/parameter.html 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

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().
```

(continues on next page)
# The context manager will restore the previous value of the global configuration upon exiting.

```python
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
```

```python
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 https://xgboost.readthedocs.io/en/stable/parameter.html 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

```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
```

Core Data Structure

Core XGBoost Library.

```python
class xgboost.DMatrix(data=None, label=None, weight=None, base_margin=None, missing=None, silent=False, feature_names=None, feature_types=None, nthread=None, enable_categorical=False)
```

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** 
  (os.PathLike/string/numpy.array/scipy.sparse/pd.DataFrame) – DMatrix Data source of DMatrix. When data is string or os.PathLike type, it represents the path libsvm format txt file, csv file (by specifying uri parameter ‘path_to_csv?format=csv’), or binary file that xgboost can read from.

• **label** (list, numpy 1-D array or cudf.DataFrame, optional) – Label of the training data.

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

• **weight** (list, numpy 1-D array or cudf.DataFrame, optional) – 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.

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

• **feature_names** (list, optional) – Set names for features.

• **feature_types** (list, optional) – Set types for features.

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

• **enable_categorical** (boolean, optional) – New in version 1.3.0. Experimental support of specializing for categorical features. Do not set to True unless you are interested in development. Currently it’s only available for gpu_hist tree method with 1 vs rest (one hot) categorical split. Also, JSON serialization format, gpu_predictor and pandas input are required.

**property feature_names**
Get feature names (column labels).

  Returns feature_names
  Return type list or None

**property feature_types**
Get feature types (column types).

  Returns feature_types
  Return type list or None

**get_base_margin()**
Get the base margin of the DMatrix.

  Returns base_margin
  Return type float

**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_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.

Returns number of columns

Return type int

**num_row()**
Get the number of rows in the DMatrix.

Returns number of rows

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.

**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

**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

```python
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

```python
set_group(group)
```

Set group size of DMatrix (used for ranking).

**Parameters**

• **group** (*array like*) – Group size of each group

```python
set_info(*, label=None, weight=None, base_margin=None, group=None, label_lower_bound=None, label_upper_bound=None, feature_names=None, feature_types=None, feature_weights=None)
```

Set meta info for DMatrix.

```python
set_label(label)
```

Set label of DMatrix

**Parameters**

• **label** (*array like*) – The label information to be set into DMatrix

```python
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

```python
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.

---

```python
slice(rindex, allow_groups=False)
```

Slice the DMatrix and return a new DMatrix that only contains `rindex`.

**Parameters**

• **rindex** (*Union[List[int], numpy.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`

```python
class xgboost.DeviceQuantileDMatrix(data, label=None, weight=None, base_margin=None, missing=None, silent=False, feature_names=None, feature_types=None, nthread=None, max_bin=256)
```

Bases: `xgboost.core.DMatrix`

---

1.8. XGBoost Python Package
Device memory Data Matrix used in XGBoost for training with tree_method='gpu_hist’. Do not use this for test/validation tasks as some information may be lost in quantisation. This DMatrix is primarily designed to save memory in training from device memory inputs by avoiding intermediate storage. Set max_bin to control the number of bins during quantisation.

You can construct DeviceQuantileDMatrix from cupy/cudf/dlpack.

New in version 1.1.0.

Parameters

- **data** *(os.PathLike/string/numpy.array/scipy.sparse/pd.DataFrame/cudf.DataFrame/cupy.array/dlpack)* – Data source of DMatrix. When data is string or os.PathLike type, it represents the path libsvm format txt file, csv file (by specifying uri parameter 'path_to_csv?format=csv'), or binary file that xgboost can read from.

- **label** *(list, numpy 1-D array or cudf.DataFrame, optional)* – Label of the training data.

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

- **weight** *(list, numpy 1-D array or cudf.DataFrame, optional)* – 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.

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

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

- **feature_types** *(list, optional)* – Set types for features.

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

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

   Experimental support of specializing for categorical features. Do not set to True unless you are interested in development. Currently it’s only available for gpu_hist tree method with 1 vs rest (one hot) categorical split. Also, JSON serialization format, gpu_predictor and pandas input are required.

```python
class xgboost.Booster(params=None, cache=(), 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.core.Booster.update()`, this function should not be called directly by users.

**Parameters**
- **dtrain** *(DMatrix)* – The training DMatrix.
- **grad** *(list)* – The first order of gradient.
- **hess** *(list)* – The second order of gradient.

**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’.

**eval** *(data, name='eval', iteration=0)*
Evaluate the model on mat.

**Parameters**
- **data** *(DMatrix)* – The dmatrix storing the input.
- **name** *(str, optional)* – The name of the dataset.
- **iteration** *(int, optional)* – The current iteration number.

**Returns**
- **result** – Evaluation result string.

**Return type**
*str*

**eval_set** *(evals, iteration=0, feval=None)*
Evaluate a set of data.
Parameters

- **evals**(list of tuples (DMatrix, string)) – List of items to be evaluated.
- **iteration**(int) – Current iteration.
- **feval**(function) – Custom evaluation function.

Returns **result** – Evaluation result string.

Return type **str**

**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**(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. Can be ‘text’, ‘json’ or ‘dot’.

**get_fscore**(fmap='')

Get feature importance of each feature.

Note: Feature importance is defined only for tree boosters

Feature importance is only defined when the decision tree model is chosen as base learner (booster=gbtree). It is not defined for other base learner types, such as linear learners (booster=gblinear).

Note: Zero-importance features will not be included

Keep in mind that this function does not include zero-importance feature, i.e. those features that have not been used in any split conditions.

**Parameters** **fmap**(str or os.PathLike (optional)) – The name of feature map file

**get_score**(fmap='', 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.

Note: Feature importance is defined only for tree boosters

Feature importance is only defined when the decision tree model is chosen as base learner (booster=gbtree). 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.

- **importance_type (str, default 'weight')** – One of the importance types defined above.

**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.

Returns

- a histogram of used splitting values for the specified feature

- either as numpy array or pandas DataFrame.

**inplace_predict (data, iteration_range=(0, 0), predict_type='value', missing=nan)**

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.

```python
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)** – 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) (open set) rounds are used in this prediction.

- **predict_type (str)** –
  - **value** Output model prediction values.
  - **margin** Output the raw untransformed margin value.

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

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.

load_model(name)
    Load the model from a file or bytestream. 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. See:
    for more info.

    Parameters
    name (string, os.PathLike, or a memory buffer) – Input file
    name or memory buffer(see also save_raw)

num_boosted_rounds()
    Get number of boosted rounds. For gblinear this is reset to 0 after serializing the model.

    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)
    Predict with data.

    Parameters
    • data (DMatrix) – The dmatrix storing the input.
    • output_margin (bool) – Whether to output the raw untransformed margin value.
    • ntree_limit (int) – Limit number of trees in the prediction; defaults to 0 (use all
    trees).
    • pred_leaf (bool) – When this option is on, the output will be a matrix of (n_sample,
    n_trees) 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 (n_sample,
    n_features + 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

    Note:
    This function is not thread safe except for gbtree booster.

    For gbtree booster, the thread safety is guaranteed by locks. For lock free prediction use
    inplace_predict instead. Also, the safety does not hold when used in conjunction with other
    methods.

    When using booster other than gbtree, predict can only be called from one thread. If you want to
    run prediction using multiple thread, call bst.copy() to make copies of model object and then call
    predict().
- **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.

New in version 1.0.0.

:[param .. note:: Using *predict* () with DART booster: If the booster object is DART type, *predict* () will not perform dropouts, i.e. all the trees will be evaluated. If you want to obtain result with dropouts, provide *training=True*.

Returns prediction

Return type numpy array

**save_config()**

Output internal parameter configuration of Booster as a JSON string.

New in version 1.0.0.

**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. See: [https://xgboost.readthedocs.io/en/latest/tutorials/saving_model.html](https://xgboost.readthedocs.io/en/latest/tutorials/saving_model.html) for more info.

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

**save_raw()**

Save the model to a in memory buffer representation instead of file.

Returns

Return type a in memory buffer representation of the model

**set_attr (**)kwargs**

Set the attribute of the Booster.

Parameters **kwargs** – The attributes to set. Setting a value to None deletes an attribute.

**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
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.

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.

Learning API

Training Library containing training routines.

xgboost.train (params, dtrain, num_boost_round=10, evals=(), obj=None, feval=None, maximize=None, early_stopping_rounds=None, evals_result=None, verbose_eval=True, xgb_model=None, callbacks=None)

Train a booster with given parameters.

Parameters

- params (dict) – Booster params.
- dtrain (DMatrix) – Data to be trained.
- num_boost_round (int) – Number of boosting iterations.
- evals (list of pairs (DMatrix, string)) – List of validation sets for which metrics will evaluated during training. Validation metrics will help us track the performance of the model.
- obj (function) – Customized objective function.
- feval (function) – Customized evaluation function.
- maximize (bool) – Whether to maximize feval.
- early_stopping_rounds (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). 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 three additional fields: bst.best_score, bst.best_iteration and bst.best_ntree_limit. (Use bst.best_ntree_limit to get the correct value if num_parallel_tree and/or num_class appears in the parameters)
- evals_result (dict) – 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 \texttt{evals_result} returns
\begin{verbatim}
{'train': {'logloss': ['0.48253', '0.35953']},
 'eval': {'logloss': ['0.480385', '0.357756']}}
\end{verbatim}

\begin{itemize}
\item \texttt{verbose_eval (bool or int)} – Requires at least one item in \texttt{evals}. If \texttt{verbose_eval} is True then the evaluation metric on the validation set is printed at each boosting stage. If \texttt{verbose_eval} is an integer then the evaluation metric on the validation set is printed at every given \texttt{verbose_eval} boosting stage. The last boosting stage / the boosting stage found by using \texttt{early_stopping_rounds} is also printed. Example: with \texttt{verbose_eval=4} and at least one item in \texttt{evals}, an evaluation metric is printed every 4 boosting stages, instead of every boosting stage.

\item \texttt{xgb_model (file name of stored xgb model or 'Booster' instance)} – Xgb model to be loaded before training (allows training continuation).

\item \texttt{callbacks (list of callback functions)} – List of callback functions that are applied at end of each iteration. It is possible to use predefined callbacks by using \texttt{Callback API}. Example:
\begin{verbatim}
[xgb.callback.reset_learning_rate(custom_rates)]
\end{verbatim}
\end{itemize}

\textbf{Returns \texttt{Booster}}

\textbf{Return type} a trained booster model

\begin{verbatim}
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)
\end{verbatim}

Cross-validation with given parameters.

\textbf{Parameters}

\begin{itemize}
\item \texttt{params (dict)} – Booster params.
\item \texttt{dtrain (DMatrix)} – Data to be trained.
\item \texttt{num_boost_round (int)} – Number of boosting iterations.
\item \texttt{nfold (int)} – Number of folds in CV.
\item \texttt{stratified (bool)} – Perform stratified sampling.
\item \texttt{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 \texttt{n} folds, \texttt{folds} should be a length \texttt{n} list of tuples. Each tuple is \texttt{(in, out)} where \texttt{in} is a list of indices to be used as the training samples for the \texttt{n} th fold and \texttt{out} is a list of indices to be used as the testing samples for the \texttt{n} th fold.
\item \texttt{metrics (string or list of strings)} – Evaluation metrics to be watched in CV.
\item \texttt{obj (function)} – Custom objective function.
\item \texttt{feval (function)} – Custom evaluation function.
\item \texttt{maximize (bool)} – Whether to maximize \texttt{feval}.
\item \texttt{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 (list of callback functions)** – List of callback functions that are applied at end of each iteration. It is possible to use predefined callbacks by using `Callback API`. Example:

```
[xgb.callback.reset_learning_rate(custom_rates)]
```

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

**Returns** evaluation history

**Return type** list(string)

### Scikit-Learn API

Scikit-Learn Wrapper interface for XGBoost.

**class xgboost.XGBRegressor(**kwargs)**

**Bases:** xgboost.sklearn.XGBModel, object

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 (int)** – Maximum tree depth for base learners.
- **learning_rate (float)** – Boosting learning rate (xgb’s “eta”)
- **verbosity (int)** – The degree of verbosity. Valid values are 0 (silent) - 3 (debug).
- **objective (string or callable)** – Specify the learning task and the corresponding learning objective or a custom objective function to be used (see note below).
- **booster (string)** – Specify which booster to use: gbtree, gblinear or dart.
- **tree_method (string)** – 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 parameters document.
- **n_jobs (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 (float)** – Minimum loss reduction required to make a further partition on a leaf node of the tree.

- **min_child_weight (float)** – Minimum sum of instance weight(hessian) needed in a child.

- **max_delta_step (int)** – Maximum delta step we allow each tree’s weight estimation to be.

- **subsample (float)** – Subsample ratio of the training instance.

- **colsample_bytree (float)** – Subsample ratio of columns when constructing each tree.

- **colsample_bylevel (float)** – Subsample ratio of columns for each level.

- **colsample_bynode (float)** – Subsample ratio of columns for each split.

- **reg_alpha (float)** – L1 regularization term on weights

- **reg_lambda (float)** – L2 regularization term on weights

- **scale_pos_weight (float)** – Balancing of positive and negative weights.

- **base_score** – The initial prediction score of all instances, global bias.

- **random_state (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 (int)** – Used for boosting random forest.

- **monotone_constraints (str)** – Constraint of variable monotonicity. See tutorial for more information.

- **interaction_constraints (str)** – 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 tutorial for more information

- **importance_type (string, default ”gain”)** – The feature importance type for the feature_importances_ property: either “gain”, “weight”, “cover”, “total_gain” or “total_cover”.

- ****kwargs (dict, optional)** – Keyword arguments for XGBoost Booster object. Full documentation of parameters can be found here: https://github.com/dmlc/xgboost/blob/master/doc/parameter.rst. 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)

Return the predicted leaf every tree for each sample.

**Parameters**

- **X** (array_like, shape=[n_samples, n_features]) – Input features matrix.
- **ntree_limit** (int) – Limit number of trees in the prediction; defaults to 0 (use all trees).

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 coef_**

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.

Returns **evals_result**

**Return type** dictionary
Example

```python
param_dist = {'objective':'binary:logistic', 'n_estimators':2}
clf = xgb.XGBModel(**param_dist)
clf.fit(X_train, y_train,
       eval_set=[(X_train, y_train), (X_test, y_test)],
       eval_metric='logloss',
       verbose=True)

evals_result = clf.evals_result()
```

The variable `evals_result` will contain:

```json
{'validation_0': {'logloss': ['0.604835', '0.531479']},
 'validation_1': {'logloss': ['0.41965', '0.17686']}}
```

**property feature_importances_**

Feature importances property

*Note:* Feature importance is only defined for tree boosters

Feature importance is only defined when the decision tree model is chosen as base learner (`booster=gbtree`). It is not defined for other base learner types, such as linear learners (`booster=gblinear`).

Returns `feature_importances_`

Return type array of shape [n_features]

```python
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,
    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` (array_like) – Feature matrix
- `y` (array_like) – Labels
- `sample_weight` (array_like) – instance weights
- `base_margin` (array_like) – global bias for each instance.
- `eval_set` (list, optional) – 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) – If a str, should be a built-in evaluation metric to use. See doc/parameter.rst. If a list of str, should be the list of multiple built-in evaluation metrics to use. If callable, a custom evaluation metric. The call signature is `func(y_predicted, y_true)` where `y_true` will be a DMatrix object such that you may need to call the `get_label` method. It must return a str, value pair where the str is a name for the evaluation and value is the value of the evaluation function. The callable custom objective is always minimized.
- **early_stopping_rounds** *(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 **eval_set**. 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: `clf.best_score`, `clf.best_iteration` and `clf.best_ntree_limit`.

- **verbose** *(bool)* – If **verbose** and an evaluation set is used, writes the evaluation metric measured on the validation set to stderr.

- **xgb_model** *(str)* – file name of stored XGBoost model or ‘Booster’ instance XGBoost model to be loaded before training (allows training continuation).

- **sample_weight_eval_set** *(list, optional)* – A list of the form \([L_1, L_2, \ldots, L_n]\), where each \(L_i\) is a list of instance weights on the i-th validation set.

- **feature_weights** *(array_like)* – 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. Only available for **hist**, **gpu_hist** and **exact** tree methods.

- **callbacks** *(list of callback functions)* – List of callback functions that are applied at end of each iteration. It is possible to use predefined callback functions by using **Callback API**. Example:

```
callbacks = [xgb.callback.EarlyStopping(rounds=early_stopping_rounds,
                                          save_best=True)]
```

**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.

**get_params** *(deep=True)*
Get parameters.

**get_xgb_params**()
Get xgboost specific parameters.

**property intercept_**
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.

The model is loaded from 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 loaded.

Parameters

- **fname** (string) – Input file name.

predict (data, output_margin=False, ntree_limit=None, validate_features=True, base_margin=None)
Predict with data.

Note: This function is not thread safe.
For each booster object, predict can only be called from one thread. If you want to run prediction using multiple thread, call xgb.copy() to make copies of model object and then call predict().

\[
preds = bst.predict(dtest, ntree_limit=num_round)
\]

Parameters

- **data** (numpy.array/scipy.sparse) – Data to predict with
- **output_margin** (bool) – Whether to output the raw untransformed margin value.
- **ntree_limit** (int) – Limit number of trees in the prediction; defaults to best_ntree_limit if defined (i.e. it has been trained with early stopping), otherwise 0 (use all trees).
- **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.

Returns prediction

Return type numpy array

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.

Note: See:

Parameters

- **fname** (string) – Output file name

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.

Returns

Return type self
class xgboost.XGBClassifier(**kwargs)
    Bases: xgboost.sklearn.XGBModel, object

Implementation of the scikit-learn API for XGBoost classification.

Parameters

- **n_estimators (int)** – Number of boosting rounds.
- **use_label_encoder (bool)** – (Deprecated) Use the label encoder from scikit-learn to encode the labels. For new code, we recommend that you set this parameter to False.
- **max_depth (int)** – Maximum tree depth for base learners.
- **learning_rate (float)** – Boosting learning rate (xgb’s “eta”)
- **verbosity (int)** – The degree of verbosity. Valid values are 0 (silent) - 3 (debug).
- **objective (string or callable)** – Specify the learning task and the corresponding learning objective or a custom objective function to be used (see note below).
- **booster (string)** – Specify which booster to use: gbtree, gblinear or dart.
- **tree_method (string)** – 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 parameters document.
- **n_jobs (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 (float)** – Minimum loss reduction required to make a further partition on a leaf node of the tree.
- **min_child_weight (float)** – Minimum sum of instance weight(hessian) needed in a child.
- **max_delta_step (int)** – Maximum delta step we allow each tree’s weight estimation to be.
- **subsample (float)** – Subsample ratio of the training instance.
- **colsample_bytree (float)** – Subsample ratio of columns when constructing each tree.
- **colsample_bylevel (float)** – Subsample ratio of columns for each level.
- **colsample_bynode (float)** – Subsample ratio of columns for each split.
- **reg_alpha (float (xgb’s alpha))** – L1 regularization term on weights
- **reg_lambda (float (xgb’s lambda))** – L2 regularization term on weights
- **scale_pos_weight (float)** – Balancing of positive and negative weights.
- **base_score** – The initial prediction score of all instances, global bias.
- **random_state (int)** – Random number seed.

Note: Using gblinear booster with shotgun updater is nondeterministic as it uses Hogwild algorithm.
• **kwargs (dict, optional) – Keyword arguments for XGBoost Booster object. Full documentation of parameters can be found here: https://github.com/dmlc/xgboost/blob/master/doc/parameter.rst. 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)

Return the predicted leaf every tree for each sample.

**Parameters**

• X (array_like, shape=[n_samples, n_features]) – Input features matrix.

• ntree_limit (int) – Limit number of trees in the prediction; defaults to 0 (use all trees).

**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 coef_**

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.

**Returns evals_result**

**Return type** dictionary

**Example**

```python
param_dist = {'objective': 'binary:logistic', 'n_estimators': 2}

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)

evals_result = clf.evals_result()
```

The variable evals_result will contain

```python
{'validation_0': {'logloss': ['0.604835', '0.531479']},
 'validation_1': {'logloss': ['0.41965', '0.17686']}}
```

**property feature_importances_**

Feature importances property

**Note:** Feature importance is defined only for tree boosters

Feature importance is only defined when the decision tree model is chosen as base learner (booster=gbtree). It is not defined for other base learner types, such as linear learners (booster=gblinear).

**Returns feature_importances_**

**Return type** array of shape [n_features]

**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, 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** (*array_like*) – Feature matrix
- **y** (*array_like*) – Labels
- **sample_weight** (*array_like*) – instance weights
- **base_margin** (*array_like*) – global bias for each instance.
- **eval_set** (*list, optional*) – 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*) – If a str, should be a built-in evaluation metric to use. See doc/parameter.rst. If a list of str, should be the list of multiple built-in evaluation metrics to use. If callable, a custom evaluation metric. The call signature is `func(y_predicted, y_true)` where `y_true` will be a DMatrix object such that you may need to call the `get_label` method. It must return a str, value pair where the str is a name for the evaluation and value is the value of the evaluation function. The callable custom objective is always minimized.
- **early_stopping_rounds** (*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 `eval_set`. 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: `clf.best_score`, `clf.best_iteration` and `clf.best_ntree_limit`.
- **verbose** (*bool*) – If `verbose` and an evaluation set is used, writes the evaluation metric measured on the validation set to stderr.
- **xgb_model** (*str*) – file name of stored XGBoost model or ‘Booster’ instance XGBoost model to be loaded before training (allows training continuation).
- **sample_weight_eval_set** (*list, optional*) – A list of the form `[L_1, L_2, ..., L_n]`, where each L_i is a list of instance weights on the i-th validation set.
- **feature_weights** (*array_like*) – 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. Only available for hist, gpu_hist and exact tree methods.
- **callbacks** (*list of callback functions*) – List of callback functions that are applied at end of each iteration. It is possible to use predefined callbacks by using `Callback API`. Example:

```python
callbacks = [xgb.callback.EarlyStopping(rounds=early_stopping_rounds, save_best=True)]
```

**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.

**get_params**(deep=True)
Get parameters.

**get_xgb_params()**
Get xgboost specific parameters.

**property intercept_**
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.

The model is loaded from 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 loaded.

**Parameters**

**fname**(string) – Input file name.

**predict**(data, output_margin=False, ntree_limit=None, validate_features=True, base_margin=None)
Predict with data.

---

**Note:** This function is not thread safe.

For each booster object, predict can only be called from one thread. If you want to run prediction using multiple thread, call xgb.copy() to make copies of model object and then call predict().

```python
preds = bst.predict(dtest, ntree_limit=num_round)
```

**Parameters**

- **data**(array_like) – Feature matrix.
- **output_margin**(bool) – Whether to output the raw untransformed margin value.
- **ntree_limit**(int) – Limit number of trees in the prediction; defaults to best_ntree_limit if defined (i.e. it has been trained with early stopping), otherwise 0 (use all trees).
- **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.

**Returns** prediction

**Return type** numpy array

---
predict_proba \((X, ntree\_limit=None, validate\_features=False, base\_margin=None)\)
Predict the probability of each \(X\) example being of a given class.

**Note:** This function is not thread safe
For each booster object, predict can only be called from one thread. If you want to run prediction using multiple thread, call `xgb.copy()` to make copies of model object and then call predict

**Parameters**
- \(X\) (*array_like*) – Feature matrix.
- \(ntree\_limit\) (*int*) – Limit number of trees in the prediction; defaults to best_ntree_limit if defined (i.e. it has been trained with early stopping), otherwise 0 (use all trees).
- \(validate\_features\) (*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.

**Returns**
- prediction – 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**
numpy array

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.

**Note:** See:

**Parameters**
- \(fname\) (*string*) – Output file name

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.

**Returns**
- self

**class** xgboost.XGBRanker(**kwargs)**
**Bases:** xgboost.sklearn.XGBModel
Implementation of the Scikit-Learn API for XGBoost Ranking.

**Parameters**
- \(n\_estimators\) (*int*) – Number of gradient boosted trees. Equivalent to number of boosting rounds.
- \(max\_depth\) (*int*) – Maximum tree depth for base learners.
- \(learning\_rate\) (*float*) – Boosting learning rate (xgb’s “eta”)
• **verbosity** *(int)* – The degree of verbosity. Valid values are 0 (silent) - 3 (debug).

• **objective** *(string or callable)* – Specify the learning task and the corresponding learning objective or a custom objective function to be used (see note below).

• **booster** *(string)* – Specify which booster to use: gbtree, gblinear or dart.

• **tree_method** *(string)* – 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 parameters document.

• **n_jobs** *(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** *(float)* – Minimum loss reduction required to make a further partition on a leaf node of the tree.

• **min_child_weight** *(float)* – Minimum sum of instance weight(hessian) needed in a child.

• **max_delta_step** *(int)* – Maximum delta step we allow each tree’s weight estimation to be.

• **subsample** *(float)* – Subsample ratio of the training instance.

• **colsample_bytree** *(float)* – Subsample ratio of columns when constructing each tree.

• **colsample_bylevel** *(float)* – Subsample ratio of columns for each level.

• **colsample_bynode** *(float)* – Subsample ratio of columns for each split.

• **reg_alpha** *(float (xgb's alpha))* – L1 regularization term on weights

• **reg_lambda** *(float (xgb's lambda))* – L2 regularization term on weights

• **scale_pos_weight** *(float)* – Balancing of positive and negative weights.

• **base_score** – The initial prediction score of all instances, global bias.

• **random_state** *(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** *(int)* – Used for boosting random forest.

• **monotone_constraints** *(str)* – Constraint of variable monotonicity. See tutorial for more information.

• **interaction_constraints** *(str)* – 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 tutorial for more information

• **importance_type** *(string, default "gain")* – The feature importance type for the feature_importances_ property: either “gain”, “weight”, “cover”, “total_gain” or “total_cover”.
• **kwargs (dict, optional) – Keyword arguments for XGBoost Booster object. Full documentation of parameters can be found here: https://github.com/dmlc/xgboost/blob/master/doc/parameter.rst. Attempting to set a parameter via the constructor args and **kwargs dict simultaneously will result in a TypeError.

**Note:** **kwargs unsupported by scikit-learn

**kwargs is unsupported by scikit-learn. We do not guarantee that parameters passed via this argument will interact properly with scikit-learn.

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

**Note:** Query group information is required for ranking tasks.

Before fitting the model, your data need to be sorted by query group. When fitting the model, you need to provide an additional array that contains the size of each query group.

For example, if your original data look like:

<table>
<thead>
<tr>
<th>qid</th>
<th>label</th>
<th>features</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>x_1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>x_2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>x_3</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>x_4</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>x_5</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>x_6</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>x_7</td>
</tr>
</tbody>
</table>

then your group array should be [3, 4].

**apply** *(X, ntree_limit=0)*

Return the predicted leaf every tree for each sample.

**Parameters**

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

- **ntree_limit** *(int) – Limit number of trees in the prediction; defaults to 0 (use all trees).*

**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 coef_**

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.

**Returns** `evals_result`

**Return type** dictionary

**Example**

```python
param_dist = {'objective':'binary:logistic', 'n_estimators':2}
clf = xgb.XGBModel(**param_dist)
clf.fit(X_train, y_train,
        eval_set=[(X_train, y_train), (X_test, y_test)],
        eval_metric='logloss',
        verbose=True)
evals_result = clf.evals_result()
```

The variable `evals_result` will contain:

```python
{'validation_0': {'logloss': [0.604835, 0.531479]},
 'validation_1': {'logloss': [0.41965, 0.17686]}}
```

**Property** `feature_importances_`

Feature importances property

**Note:** Feature importance is defined only for tree boosters

Feature importance is only defined when the decision tree model is chosen as base learner (*booster=gbtree*). It is not defined for other base learner types, such as linear learners (*booster=gblinear*).

**Returns** `feature_importances_`

**Return type** array of shape `[n_features]`

**`fit()`**

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

- \( X (\text{array_like}) \) – Feature matrix
- \( y (\text{array_like}) \) – Labels
- \( \text{group} (\text{array_like}) \) – Size of each query group of training data. Should have as many elements as the query groups in the training data
- \( \text{sample_weight} (\text{array_like}) \) – Query group weights

**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.

- \( \text{base_margin} (\text{array_like}) \) – Global bias for each instance.
- \( \text{eval_set} (\text{list}, \text{optional}) \) – 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.
- \( \text{sample_weight_eval_set} (\text{list}, \text{optional}) \) – A list of the form \([L_1, L_2, \ldots, L_n]\), 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.

- \( \text{eval_group} (\text{list of arrays}, \text{optional}) \) – A list in which \(\text{eval_group}[i]\) is the list containing the sizes of all query groups in the \(i\)-th pair in \(\text{eval_set}\).
- \( \text{eval_metric} (\text{str}, \text{list of str}, \text{optional}) \) – If a str, should be a built-in evaluation metric to use. See doc/parameter.rst. If a list of str, should be the list of multiple built-in evaluation metrics to use. The custom evaluation metric is not yet supported for the ranker.
- \( \text{early_stopping_rounds} (\text{int}) \) – Activates early stopping. Validation metric needs to improve at least once in every \(\text{early_stopping_rounds}\) round(s) to continue training. Requires at least one item in \(\text{eval_set}\). The method returns the model from the last iteration (not the best one). If there’s more than one item in \(\text{eval_set}\), the last entry will be used for early stopping. If there’s more than one metric in \(\text{eval_metric}\), the last metric will be used for early stopping. If early stopping occurs, the model will have three additional fields: \text{clf.best_score}, \text{clf.best_iteration} and \text{clf.best_ntree_limit}.
- \( \text{verbose} (\text{bool}) \) – If \(\text{verbose}\) and an evaluation set is used, writes the evaluation metric measured on the validation set to stderr.
- \( \text{xgb_model} (\text{str}) \) – file name of stored XGBoost model or ‘Booster’ instance XGBoost model to be loaded before training (allows training continuation).
- \( \text{feature_weights} (\text{array_like}) \) – Weight for each feature, defines the probability of each feature being selected when \(\text{colsample}\) is being used. All values must be greater
than 0, otherwise a `ValueError` is thrown. Only available for `hist`, `gpu_hist` and `exact` tree methods.

- `callbacks (list of callback functions)` – List of callback functions that are applied at end of each iteration. It is possible to use predefined callbacks by using `Callback API`. Example:

```python
callbacks = [xgb.callback.EarlyStopping(rounds=early_stopping_rounds,
                                         save_best=True)]
```

**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.

**get_params**(deep=True)
Get parameters.

**get_xgb_params()**
Get xgboost specific parameters.

**property intercept_**
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**  
a numpy array of shape (1,) or [n_classes]

**load_model**(fname)
Load the model from a file.

The model is loaded from 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 loaded.

**Parameters**  
fname (string) – Input file name.

**predict**(data, output_margin=False, ntree_limit=0, validate_features=True, base_margin=None)
Predict with `data`.

**Note:**  
This function is not thread safe.

For each booster object, predict can only be called from one thread. If you want to run prediction using multiple thread, call `xgb.copy()` to make copies of model object and then call `predict()`.

```python
preds = bst.predict(dtest, ntree_limit=num_round)
```
Parameters

- **data** *(numpy.array/scipy.sparse)* – Data to predict with
- **output_margin** *(bool)* – Whether to output the raw untransformed margin value.
- **ntree_limit** *(int)* – Limit number of trees in the prediction; defaults to best_ntree_limit if defined (i.e. it has been trained with early stopping), otherwise 0 (use all trees).
- **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.

Returns prediction

Return type: numpy array

**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.

Parameters **fname** *(string)* – Output file name

**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.

Returns

Return type: self

**class** xgboost.XGBRFRegressor(**kwargs**)

**scikit-learn API for XGBoost random forest regression.**

Parameters

- **n_estimators** *(int)* – Number of trees in random forest to fit.
- **max_depth** *(int)* – Maximum tree depth for base learners.
- **learning_rate** *(float)* – Boosting learning rate (xgb’s “eta”)
- **verbosity** *(int)* – The degree of verbosity. Valid values are 0 (silent) - 3 (debug).
- **objective** *(string or callable)* – Specify the learning task and the corresponding learning objective or a custom objective function to be used (see note below).
- **booster** *(string)* – Specify which booster to use: gbtree, gblinear or dart.
- **tree_method** *(string)* – 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 parameters document.
• **n_jobs** (*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** (*float*) – Minimum loss reduction required to make a further partition on a leaf node of the tree.

• **min_child_weight** (*float*) – Minimum sum of instance weight(hessian) needed in a child.

• **max_delta_step** (*int*) – Maximum delta step we allow each tree’s weight estimation to be.

• **subsample** (*float*) – Subsample ratio of the training instance.

• **colsample_bytree** (*float*) – Subsample ratio of columns when constructing each tree.

• **colsample_bylevel** (*float*) – Subsample ratio of columns for each level.

• **colsample_bynode** (*float*) – Subsample ratio of columns for each split.

• **reg_alpha** (*float* (xgb’s alpha)) – L1 regularization term on weights

• **reg_lambda** (*float* (xgb’s lambda)) – L2 regularization term on weights

• **scale_pos_weight** (*float*) – Balancing of positive and negative weights.

• **base_score** – The initial prediction score of all instances, global bias.

• **random_state** (*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** (*int*) – Used for boosting random forest.

• **monotone_constraints** (*str*) – Constraint of variable monotonicity. See tutorial for more information.

• **interaction_constraints** (*str*) – 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 tutorial for more information

• **importance_type** (*string*, default ”gain”) – The feature importance type for the feature_importances_ property: either ”gain”, “weight”, “cover”, “total_gain” or “total_cover”.

• ****kwargs** (*dict*, *optional*) – Keyword arguments for XGBoost Booster object. Full documentation of parameters can be found here: https://github.com/dmlc/xgboost/blob/master/doc/parameter.rst. 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)*

Return the predicted leaf every tree for each sample.

**Parameters**

- `X` (array_like, shape=[n_samples, n_features]) — Input features matrix.
- `ntree_limit` (int) — Limit number of trees in the prediction; defaults to 0 (use all trees).

**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 coef_**

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.

**Returns** `evals_result`

**Return type** dictionary
Example

```python
param_dist = {'objective': 'binary:logistic', 'n_estimators': 2}
clf = xgb.XGBModel(**param_dist)
clf.fit(X_train, y_train,
        eval_set=[(X_train, y_train), (X_test, y_test)],
        eval_metric='logloss',
        verbose=True)

evals_result = clf.evals_result()
```

The variable `evals_result` will contain:

```python
{'validation_0': {'logloss': ['0.604835', '0.531479']},
 'validation_1': {'logloss': ['0.41965', '0.17686']}}
```

**property feature_importances_**

Feature importances property

**Note:** Feature importance is defined only for tree boosters

Feature importance is only defined when the decision tree model is chosen as base learner (`booster=gbtree`). It is not defined for other base learner types, such as linear learners (`booster=gblinear`).

**Returns** `feature_importances_`

**Return type** array of shape `[n_features]`

**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, 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 (array_like)` – Feature matrix
- `y (array_like)` – Labels
- `sample_weight (array_like)` – instance weights
- `base_margin (array_like)` – global bias for each instance.
- `eval_set (list, optional)` – 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)` – If a str, should be a built-in evaluation metric to use. See doc/parameter.rst. If a list of str, should be the list of multiple built-in evaluation metrics to use. If callable, a custom evaluation metric. The call signature is `func(y_predicted, y_true)` where `y_true` will be a DMatrix object such that you may need to call the `get_label` method. It must return a str, value pair where the str is a name for the evaluation and value is the value of the evaluation function. The callable custom objective is always minimized.
• **early_stopping_rounds** *(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 **eval_set**. 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: **clf.best_score**, **clf.best_iteration** and **clf.best_ntree_limit**.

• **verbose** *(bool)* – If **verbose** and an evaluation set is used, writes the evaluation metric measured on the validation set to stderr.

• **xgb_model** *(str)* – file name of stored XGBoost model or ‘Booster’ instance XGBoost model to be loaded before training (allows training continuation).

• **sample_weight_eval_set** *(list, optional)* – A list of the form [L_1, L_2, ..., L_n], where each L_i is a list of instance weights on the i-th validation set.

• **feature_weights** *(array_like)* – 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. Only available for **hist**, **gpu_hist** and **exact** tree methods.

• **callbacks** *(list of callback functions)* – List of callback functions that are applied at end of each iteration. It is possible to use predefined callback functions by using **Callback API**. Example:

```python
callbacks = [xgb.callback.EarlyStopping(rounds=early_stopping_rounds, save_best=True)]
```

g**et_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

g**et_num_boosting_rounds** ()
Gets the number of xgboost boosting rounds.

g**et_params** *(deep=True)*
Get parameters.

g**et_xgb_params** ()
Get xgboost specific parameters.

**property intercept_**
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.

The model is loaded from 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 loaded.

Parameters

- **fname** (string) – Input file name.

predict(data, output_margin=False, ntree_limit=None, validate_features=True, base_margin=None)

Predict with data.

Note: This function is not thread safe.

For each booster object, predict can only be called from one thread. If you want to run prediction using multiple thread, call xgb.copy() to make copies of model object and then call predict().

```
preds = bst.predict(dtest, ntree_limit=num_round)
```

Parameters

- **data** (numpy.array/scipy.sparse) – Data to predict with
- **output_margin** (bool) – Whether to output the raw untransformed margin value.
- **ntree_limit** (int) – Limit number of trees in the prediction; defaults to best_ntree_limit if defined (i.e. it has been trained with early stopping), otherwise 0 (use all trees).
- **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.

Returns prediction

Return type numpy array

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.

Note: See:


Parameters

- **fname** (string) – Output file name

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.

Returns

Return type self
class xgboost.XGBRFClassifier(**kwargs)
Bases: xgboost.sklearn.XGBClassifier

scikit-learn API for XGBoost random forest classification.

Parameters

- **n_estimators (int)** – Number of trees in random forest to fit.
- **use_label_encoder (bool)** – (Deprecated) Use the label encoder from scikit-learn to encode the labels. For new code, we recommend that you set this parameter to False.
- **max_depth (int)** – Maximum tree depth for base learners.
- **learning_rate (float)** – Boosting learning rate (xgb’s “eta”)
- **verbosity (int)** – The degree of verbosity. Valid values are 0 (silent) - 3 (debug).
- **objective (string or callable)** – Specify the learning task and the corresponding learning objective or a custom objective function to be used (see note below).
- **booster (string)** – Specify which booster to use: gbtree, gblinear or dart.
- **tree_method (string)** – 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 parameters document.
- **n_jobs (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 (float)** – Minimum loss reduction required to make a further partition on a leaf node of the tree.
- **min_child_weight (float)** – Minimum sum of instance weight(hessian) needed in a child.
- **max_delta_step (int)** – Maximum delta step we allow each tree’s weight estimation to be.
- **subsample (float)** – Subsample ratio of the training instance.
- **colsample_bytree (float)** – Subsample ratio of columns when constructing each tree.
- **colsample_bylevel (float)** – Subsample ratio of columns for each level.
- **colsample_bynode (float)** – Subsample ratio of columns for each split.
- **reg_alpha (float (xgb’s alpha))** – L1 regularization term on weights
- **reg_lambda (float (xgb’s lambda))** – L2 regularization term on weights
- **scale_pos_weight (float)** – Balancing of positive and negative weights.
- **base_score** – The initial prediction score of all instances, global bias.
- **random_state (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** (*int*) – Used for boosting random forest.

• **monotone_constraints** (*str*) – Constraint of variable monotonicity. See tutorial for more information.

• **interaction_constraints** (*str*) – Constraints for interaction representing permitted interactions. The constraints must be specified in the form of a nest list, e.g. 

```plaintext
[[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** (*string*, default "gain") – The feature importance type for the feature_importances_ property: either "gain", “weight”, “cover”, “total_gain” or “total_cover”.

• ****kwargs** (*dict, optional*) – Keyword arguments for XGBoost Booster object. Full documentation of parameters can be found here: [https://github.com/dmlc/xgboost/blob/master/doc/parameter.rst](https://github.com/dmlc/xgboost/blob/master/doc/parameter.rst). 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*)

Return the predicted leaf every tree for each sample.

**Parameters**

- **X** (array_like, shape=[n_samples, n_features]) – Input features matrix.
- **ntree_limit** (*int*) – Limit number of trees in the prediction; defaults to 0 (use all trees).

**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 

```plaintext
[0; 2**(self.max_depth+1)),
```

possibly with gaps in the numbering.

**Return type** array_like, shape=[n_samples, n_trees]

**property coef_**

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.

Returns evals_result

Return type dictionary

Example

```python
param_dist = {'objective':'binary:logistic', 'n_estimators':2}
clf = xgb.XGBClassifier(**param_dist)
clf.fit(X_train, y_train,
    eval_set=[(X_train, y_train), (X_test, y_test)],
    eval_metric=['logloss',
                 'logloss'],
    verbose=True)
evals_result = clf.evals_result()
```

The variable evals_result will contain

```json
{'validation_0': {'logloss': ['0.604835', '0.531479']},
 'validation_1': {'logloss': ['0.41965', '0.17686']}}
```

property feature_importances_

Feature importances property

Note: Feature importance is defined only for tree boosters

Feature importance is only defined when the decision tree model is chosen as base learner (booster=gbtree). It is not defined for other base learner types, such as linear learners (booster=gblinear).

Returns feature_importances_

Return type array of shape [n_features]

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, 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** (`array_like`) – Feature matrix
- **y** (`array_like`) – Labels
- **sample_weight** (`array_like`) – instance weights
- **base_margin** (`array_like`) – global bias for each instance.
- **eval_set** (`list`, optional) – 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) – If a str, should be a built-in evaluation metric to use. See doc/parameter.rst. If a list of str, should be the list of multiple built-in evaluation metrics to use. If callable, a custom evaluation metric. The call signature is `func(y_predicted, y_true)` where `y_true` will be a DMatrix object such that you may need to call the `get_label` method. It must return a str, value pair where the str is a name for the evaluation and value is the value of the evaluation function. The callable custom objective is always minimized.
- **early_stopping_rounds** (`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 `eval_set`. 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: `clf.best_score`, `clf.best.iteration` and `clf.best.ntree_limit`.
- **verbose** (`bool`) – If `verbose` and an evaluation set is used, writes the evaluation metric measured on the validation set to stderr.
- **xgb_model** (`str`) – file name of stored XGBoost model or ‘Booster’ instance XGBoost model to be loaded before training (allows training continuation).
- **sample_weight_eval_set** (`list`, optional) – A list of the form `[L_1, L_2, ... , L_n]`, where each L_i is a list of instance weights on the i-th validation set.
- **feature_weights** (`array_like`) – 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. Only available for `hist`, `gpu_hist` and `exact` tree methods.
- **callbacks** (`list of callback functions`) – List of callback functions that are applied at end of each iteration. It is possible to use predefined callbacks by using `Callback API`. Example:

```python
callbacks = [xgb.callback.EarlyStopping(rounds=early_stopping_rounds,
                     save_best=True)]
```

**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.

**get_params** (deep=True)
Get parameters.

**get_xgb_params** ()
Get xgboost specific parameters.

**property intercept_**
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.

The model is loaded from 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 loaded.

**Parameters**  
**fname** (string) – Input file name.

**predict** (data, output_margin=False, ntree_limit=None, validate_features=True, base_margin=None)
Predict with data.

**Note:** This function is not thread safe.

For each booster object, predict can only be called from one thread. If you want to run prediction using multiple thread, call xgb.copy() to make copies of model object and then call predict().

```python
preds = bst.predict(dtest, ntree_limit=num_round)
```

**Parameters**

- **data** (array_like) – Feature matrix.
- **output_margin** (bool) – Whether to output the raw untransformed margin value.
- **ntree_limit** (int) – Limit number of trees in the prediction; defaults to best_ntree_limit if defined (i.e. it has been trained with early stopping), otherwise 0 (use all trees).
- **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.

**Returns**  prediction

**Return type**  numpy array
**predict_proba** *(X, ntree_limit=None, validate_features=False, base_margin=None)*

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

**Note:** This function is not thread safe

For each booster object, predict can only be called from one thread. If you want to run prediction using multiple thread, call `xgb.copy()` to make copies of model object and then call predict

**Parameters**

- **X** *(array_like)* – Feature matrix.
- **ntree_limit** *(int)* – Limit number of trees in the prediction; defaults to best_ntree_limit if defined (i.e. it has been trained with early stopping), otherwise 0 (use all trees).
- **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.

**Returns** *prediction* – 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** numpy array

**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.

**Note:** See:


**Parameters** *fname* *(string)* – Output file name

**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.

**Returns**

**Return type** self
Plotting API

Plotting Library.

```python
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, **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.
- `**kwargs` – Other keywords passed to ax.barh()

Returns `ax`

Return type matplotlib Axes

```python
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
xgboost, Release 1.4.0-SNAPSHOT

- **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** – 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 graphviz. Example:

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

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

  ```python
  {'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

```python
xgboost.callback.TrainingCallback()
    Interface for training callback.
    New in version 1.3.0.
```

```python
xgboost.callback.EvaluationMonitor(rank=0, period=1, show_stdv=False)
    Print the evaluation result at each iteration.
    New in version 1.3.0.
```

**Parameters**

- `metric (callable)` – Extra user defined metric.
- `rank (int)` – Which worker should be used for printing the result.
- `period (int)` – How many epochs between printing.
- `show_stdv (bool)` – Used in cv to show standard deviation. Users should not specify it.

**Return type:** None

```python
xgboost.callback.EarlyStopping(rounds, metric_name=None, data_name=None, maximize=None, save_best=False)
    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.

**Return type:** None

```python
xgboost.callback.LearningRateScheduler(learning_rates)
    Callback function for scheduling learning rate.
    New in version 1.3.0.
```

**Parameters**

- `learning_rates (callable/collections.Sequence)` – 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.

**Return type:** None

```python
xgboost.callback.TrainingCheckPoint(directory, name='model', as_pickle=False, iterations=100)
    Checkpointing operation.
    New in version 1.3.0.
```

**Parameters**

- `directory (os.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** *(boolean)* – 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.

### Dask API


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):

```python
xgboost.dask.DaskDMatrix(client, data, label=None, missing=None, weight=None, base_margin=None, label_lower_bound=None, label_upper_bound=None, feature_weights=None, feature_names=None, feature_types=None)
```

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`.

**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.


- **missing** *(float)* – Value in the input data (e.g. `numpy.ndarray`) 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]])* – Global bias for each instance.


• **feature_names** *(Optional[Union[List[str], str]])* – Set names for features.

• **feature_types** *(Optional[Union[List[Any], Any]])* – Set types for features

Return type None

```python
dask.DaskDeviceQuantileDMatrix(client, data, label=None, missing=None, weight=None, base_margin=None, label_lower_bound=None, label_upper_bound=None, feature_weights=None, feature_names=None, feature_types=None, max_bin=256)
```

Specialized data type for `gpu_hist` tree method. This class is used to reduce the memory usage by eliminating data copies. Internally the all partitions/chunks of data are merged by weighted GK sketching. So the number of partitions from dask may affect training accuracy as GK generates bounded error for each merge.

New in version 1.2.0.

Parameters

• **max_bin** *(Number of bins for histogram construction.)* –

• **client** *(distributed.Client)* –

• **data** *(Union[da.Array, dd.DataFrame, dd.Series])* –

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

• **missing** *(float)* –

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

• **base_margin** *(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]])* –

• **feature_names** *(Optional[Union[List[str], str]])* –

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

Return type None

```python
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)
```

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** (*xgboost.dask.DaskDMatrix*)–

• **num_boost_round** (*int*)–

• **evals** (*Optional[List[Tuple[xgboost.dask.DaskDMatrix, str]]]*)–

• **obj** (*Optional[Callable[[numpy.ndarray, xgboost.core.DMatrix], Tuple[numpy.ndarray, numpy.ndarray]]]*)–

• **feval** (*Optional[Callable[[numpy.ndarray, xgboost.core.DMatrix], Tuple[numpy.ndarray, numpy.ndarray]]]*)–

• **early_stopping_rounds** (*Optional[int]*)–

• **xgb_model** (*Optional[xgboost.core.Booster]*)–

• **verbose_eval** (*Union[int, bool]*)–

• **callbacks** (*Optional[List[xgboost.callback.TrainingCallback]]*)–

**Returns**

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

```python
dict{'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)

Run prediction with a trained booster.

**Note:** Only default prediction mode is supported right now.

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.

• **model** (*Union[Dict[str, Any], xgboost.core.Booster]*) – The trained model.

• **data** (*Union[xgboost.dask.DaskDMatrix, da.Array, dd.DataFrame, dd.Series]*) – 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*)–
xgboost, Release 1.4.0-SNAPSHOT

• `pred_contribs` (bool) –
• `approx_contribs` (bool) –
• `pred_interactions` (bool) –
• `validate_features` (bool) –

Returns prediction

Return type: `dask.array.Array/dask.dataframe.Series`

```python
xgboost.dask.inplace_predict(client, model, data, iteration_range=(0, 0), predict_type='value', missing=nan)
```

Inplace prediction.

New in version 1.1.0.

**Parameters**

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

• `model` (`Union[Dict[str, Any], xgboost.core.Booster]`) – The trained model.

• `iteration_range` (`Tuple[int, int]`) – Specify the range of trees used for prediction.

• `predict_type` (`str`) –
  – `'value'`: Normal prediction result.
  – `'margin'`: Output the raw untransformed margin value.

• `missing` (`float`) – Value in the input data which needs to be present as a missing value. If None, defaults to np.nan.

• `data` (`Union[da.Array, dd.DataFrame, dd.Series]`) –

**Returns**

Return type: prediction

```python
xgboost.dask.DaskXGBClassifier(max_depth=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, 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='gain', gpu_id=None, validate_parameters=None, **kwargs)
```

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` (`int`) – Maximum tree depth for base learners.

• `learning_rate` (`float`) – Boosting learning rate (xgb’s “eta”)
• **verbosity** *(int)* – The degree of verbosity. Valid values are 0 (silent) - 3 (debug).

• **objective** *(string or callable)* – Specify the learning task and the corresponding learning objective or a custom objective function to be used (see note below).

• **booster** *(string)* – Specify which booster to use: gbtree, gblinear or dart.

• **tree_method** *(string)* – 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 parameters document.

• **n_jobs** *(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** *(float)* – Minimum loss reduction required to make a further partition on a leaf node of the tree.

• **min_child_weight** *(float)* – Minimum sum of instance weight(hessian) needed in a child.

• **max_delta_step** *(int)* – Maximum delta step we allow each tree’s weight estimation to be.

• **subsample** *(float)* – Subsample ratio of the training instance.

• **colsample_bytree** *(float)* – Subsample ratio of columns when constructing each tree.

• **colsample_bylevel** *(float)* – Subsample ratio of columns for each level.

• **colsample_bynode** *(float)* – Subsample ratio of columns for each split.

• **reg_alpha** *(float (xgb's alpha))* – L1 regularization term on weights

• **reg_lambda** *(float (xgb's lambda))* – L2 regularization term on weights

• **scale_pos_weight** *(float)* – Balancing of positive and negative weights.

• **base_score** – The initial prediction score of all instances, global bias.

• **random_state** *(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** *(int)* – Used for boosting random forest.

• **monotone_constraints** *(str)* – Constraint of variable monotonicity. See tutorial for more information.

• **interaction_constraints** *(str)* – 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 tutorial for more information

• **importance_type** *(string, default "gain")* – The feature importance type for the feature_importances_ property: either “gain”, “weight”, “cover”, “total_gain” or “total_cover".
xgboost, Release 1.4.0-SNAPSHOT

• **kwargs (dict, optional) – Keyword arguments for XGBoost Booster object. Full documentation of parameters can be found here: https://github.com/dmlc/xgboost/blob/master/doc/parameter.rst. 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.

xgboost.dask.DaskXGBRegressor (max_depth=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, 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='gain', gpu_id=None, validate_parameters=None, **kwargs)

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 (int)** – Maximum tree depth for base learners.

• **learning_rate (float)** – Boosting learning rate (xgb’s “eta”)

• **verbosity (int)** – The degree of verbosity. Valid values are 0 (silent) - 3 (debug).

• **objective (string or callable)** – Specify the learning task and the corresponding learning objective or a custom objective function to be used (see note below).

• **booster (string)** – Specify which booster to use: gbtree, gblinear or dart.

• **tree_method (string)** – 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 parameters document.

• **n_jobs (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 (float)** – Minimum loss reduction required to make a further partition on a leaf node of the tree.

• **min_child_weight (float)** – Minimum sum of instance weight(hessian) needed in a child.

• **max_delta_step (int)** – Maximum delta step we allow each tree’s weight estimation to be.

• **subsample (float)** – Subsample ratio of the training instance.

• **colsample_bytree (float)** – Subsample ratio of columns when constructing each tree.
• **colsample_bylevel** *(float)* – Subsample ratio of columns for each level.

• **colsample_bytree** *(float)* – Subsample ratio of columns for each split.

• **reg_alpha** *(float (xgb's alpha)) –* L1 regularization term on weights

• **reg_lambda** *(float (xgb's lambda)) –* L2 regularization term on weights

• **scale_pos_weight** *(float)* – Balancing of positive and negative weights.

• **base_score** – The initial prediction score of all instances, global bias.

• **random_state** *(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** *(int)* – Used for boosting random forest.

• **monotone_constraints** *(str)* – Constraint of variable monotonicity. See tutorial for more information.

• **interaction_constraints** *(str)* – 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 tutorial for more information.

• **importance_type** *(string, default "gain") –* The feature importance type for the feature_importances_ property: either “gain”, “weight”, “cover”, “total_gain” or “total_cover”.

• ****kwargs** *(dict, optional)* – Keyword arguments for XGBoost Booster object. Full documentation of parameters can be found here: https://github.com/dmlc/xgboost/blob/master/doc/parameter.rst. 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.

### Callback Functions

This document gives a basic walkthrough of callback function 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)

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

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: `xgboost.callback.TrainingCallback`, user defined callbacks should inherit this class and override corresponding methods. There’s a working example in `demo/guide-python/callbacks.py`

## 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.

# 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.

1.9 XGBoost R Package

You have found the XGBoost R Package!

1.9.1 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.

1.9.2 Tutorials

XGBoost R Tutorial

Introduction

Xgboost is short for eXtreme Gradient Boosting 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:

```r
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:

```r
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.

```r
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.

```r
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:

```r
str(train)
```

```r
## 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" ...
## .. ..@ 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.

```r
dim(train$data)
```

```r
## [1] 6513  126
```

```r
dim(test$data)
```
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\}):  

```r
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.

```r
bstSparse <- xgboost(data = train$data, label = train$label, max.depth = 2, eta = 1, nthread = 2, nrounds = 2, objective = "binary:logistic")
```

```r
## [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.

```r
bstDense <- xgboost(data = as.matrix(train$data), label = train$label, max.depth = 2, 
                   eta = 1, nthread = 2, nrounds = 2, objective = "binary:logistic")
```

```r
## [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.

```r
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")
```

```r
## [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).

```r
# verbose = 0, no message
bst <- xgboost(data = dtrain, max.depth = 2, eta = 1, nthread = 2, nrounds = 2, 
               objective = "binary:logistic", verbose = 0)
```

```r
## [0] train-error:0.046522
## [1] train-error:0.022263
```

```r
# verbose = 1, print evaluation metric
bst <- xgboost(data = dtrain, max.depth = 2, eta = 1, nthread = 2, nrounds = 2, 
               objective = "binary:logistic", verbose = 1)
```

```r
## [0] train-error:0.046522
## [1] train-error:0.022263
```

```r
# 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)
```

```r
## [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
```
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.

```r
pred <- predict(bst, test$data)
# size of the prediction vector
print(length(pred))
```

```
## [1] 1611
```

```r
# 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 don’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).

```r
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.

```r
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.

```r
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 \texttt{watchlist} parameter. It is a list of \texttt{xgb.DMatrix}, each of them tagged with a name.

```r
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")
```

\texttt{XGBoost} has computed at each round the same average error metric seen above (we set \texttt{nrounds} to 2, that is why we have two lines). Obviously, the \texttt{train-error} number is related to the training dataset (the one the algorithm learns from) and the \texttt{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.

```r
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 test-error:0.042831
## [1] train-error:0.022263 test-error:0.021726
```

\texttt{eval.metric} allows us to monitor two new metrics for each round, \texttt{logloss} and \texttt{error}.

**Linear boosting**

Until now, all the learnings we have performed were based on boosting trees. \texttt{XGBoost} implements a second algorithm, based on linear boosting. The only difference with the previous command is \texttt{booster = "gblinear"} parameter (and removing \texttt{eta} parameter).

```r
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.
## [1] train-error:0.004146 train-logloss:0.069885 test-error:0.
```

In this specific case, \texttt{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.

```r
xgb.DMatrix.save(dtrain, "dtrain.buffer")
```

```r
## [1] TRUE
```

# to load it in, simply call xgb.DMatrix
dtrain2 <- xgb.DMatrix("dtrain.buffer")

```r
## [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")

```r
## [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.

```r
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))
```

```r
## [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).

```r
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.

```r
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```

```r
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.

```r
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:

```r
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.

```r
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.

```r
head(df[, AgeDiscret := as.factor(round(Age/10, 0))])
```

<table>
<thead>
<tr>
<th>ID</th>
<th>Treatment</th>
<th>Sex</th>
<th>Age</th>
<th>Improved</th>
<th>AgeDiscret</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Treated</td>
<td>Male</td>
<td>27</td>
<td>Some</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Treated</td>
<td>Male</td>
<td>29</td>
<td>None</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>Treated</td>
<td>Male</td>
<td>30</td>
<td>None</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>Treated</td>
<td>Male</td>
<td>32</td>
<td>Marked</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>Treated</td>
<td>Male</td>
<td>46</td>
<td>Marked</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>Treated</td>
<td>Male</td>
<td>58</td>
<td>Marked</td>
<td>6</td>
</tr>
</tbody>
</table>

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...).

```r
head(df[, AgeCat := as.factor(ifelse(Age > 30, "Old", "Young"))])
```

<table>
<thead>
<tr>
<th>ID</th>
<th>Treatment</th>
<th>Sex</th>
<th>Age</th>
<th>Improved</th>
<th>AgeCat</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Treated</td>
<td>Male</td>
<td>27</td>
<td>Some</td>
<td>Young</td>
</tr>
<tr>
<td>2</td>
<td>Treated</td>
<td>Male</td>
<td>29</td>
<td>None</td>
<td>Young</td>
</tr>
<tr>
<td>3</td>
<td>Treated</td>
<td>Male</td>
<td>30</td>
<td>None</td>
<td>Young</td>
</tr>
<tr>
<td>4</td>
<td>Treated</td>
<td>Male</td>
<td>32</td>
<td>Marked</td>
<td>Old</td>
</tr>
<tr>
<td>5</td>
<td>Treated</td>
<td>Male</td>
<td>46</td>
<td>Marked</td>
<td>Old</td>
</tr>
<tr>
<td>6</td>
<td>Treated</td>
<td>Male</td>
<td>58</td>
<td>Marked</td>
<td>Old</td>
</tr>
</tbody>
</table>

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 1 1 27 1 . . . 1
## [2] 1 . 1 1 1 29 1 . . . 1
## [3] 1 . 1 1 1 30 1 . . . 1
## [4] 1 . 1 1 1 32 1 . . . 1
## [5] 1 . 1 1 1 46 . . . .
## [6] 1 . 1 1 58 . . . .
```

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.
The code below is very usual. For more information, you can look at the documentation of xgboost function (or at the vignette Xgboost presentation).

```r
bst <- xgboost(data = sparse_matrix, label = output_vector, max.depth = 4,
                eta = 1, nthread = 2, nrounds = 10, objective = "binary:logistic")
```

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).

```r
importance <- xgb.importance(feature_names = sparse_matrix@Dimnames[[2]], model = bst)
head(importance)
```

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.

```r
importanceRaw <- xgb.importance(feature_names = sparse_matrix@Dimnames[[2]], model = bst, data = sparse_matrix, label = output_vector)

# Cleaning for better display
importanceClean <- importanceRaw[, c('Cover', 'Frequency')]

head(importanceClean)
```

<table>
<thead>
<tr>
<th>Feature</th>
<th>Split</th>
<th>Gain</th>
<th>RealCover</th>
<th>RealCover %</th>
</tr>
</thead>
<tbody>
<tr>
<td>TreatmentPlacebo</td>
<td>-1.00136e-05</td>
<td>0.28575</td>
<td>7</td>
<td>0.2500000</td>
</tr>
<tr>
<td>Age</td>
<td>61.5</td>
<td>0.16374</td>
<td>12</td>
<td>0.4285714</td>
</tr>
<tr>
<td>Age</td>
<td>39</td>
<td>0.08705</td>
<td>8</td>
<td>0.2857143</td>
</tr>
<tr>
<td>Age</td>
<td>57.5</td>
<td>0.06947</td>
<td>11</td>
<td>0.3928571</td>
</tr>
<tr>
<td>SexMale</td>
<td>-1.00136e-05</td>
<td>0.04874</td>
<td>4</td>
<td>0.1428571</td>
</tr>
<tr>
<td>Age</td>
<td>53.5</td>
<td>0.04620</td>
<td>10</td>
<td>0.3571429</td>
</tr>
</tbody>
</table>

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)
```

```r
## 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)
p = print(c2)
```

```r
## 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)
p = print(c2)
```

```r
## 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.
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 ($\Rightarrow$ 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:

```r
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.

### 1.10 XGBoost JVM Package

You have found the XGBoost JVM Package!

#### 1.10.1 Installation

- **Installation from Maven repository**
  - Access release version
  - Access SNAPSHOT version
- **Installation from source**
  - Enabling OpenMP for Mac OS
  - Building with GPU support
Installation from Maven repository

Access release version

You can use XGBoost4J in your Java/Scala application by adding XGBoost4J as a dependency:

Listing 15: 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 16: sbt

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

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

For the latest release version number, please check [here](#).

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:** Using Maven repository hosted by the XGBoost project

There may be some delay until a new release becomes available to Maven Central. If you would like to access the latest release immediately, add the Maven repository hosted by the XGBoost project:

Listing 17: Maven

```xml
<repository>
  <id>XGBoost4J Release Repo</id>
  <name>XGBoost4J Release Repo</name>
  <url>https://s3-us-west-2.amazonaws.com/xgboost-maven-repo/release/</url>
</repository>
```
Listing 18: sbt

```scala
resolvers += "XGBoost4J Release Repo" at "https://s3-us-west-2.amazonaws.com/xgboost-maven-repo/release/"
```

**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.

---

**Access SNAPSHOT version**

First add the following Maven repository hosted by the XGBoost project:

Listing 19: 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 20: sbt

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

Then add XGBoost4J as a dependency:

Listing 21: 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>
```
libraryDependencies ++= Seq(
   "ml.dmlc" % "xgboost4j" % "latest_version_num-SNAPSHOT",
   "ml.dmlc" % "xgboost4j-spark" % "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).

Installation from source

Building XGBoost4J using Maven requires Maven 3 or newer, Java 7+ and CMake 3.13+ for compiling the 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.

1.10. XGBoost JVM Package

163


Note: XGBoost4J-Spark requires Apache Spark 2.3+

XGBoost4J-Spark now requires **Apache Spark 2.3+**. Latest versions of XGBoost4J-Spark use 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
```

1.10.2 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:

  ```java
  import ml.dmlc.xgboost4j.java.DMatrix;
  ```

- Use DMatrix constructor to load data from a libsvm text format file:

  ```java
  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:

```java
import ml.dmlc.xgboost4j.java.Booster;
import ml.dmlc.xgboost4j.java.XGBoost;
```

- Training

```java
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.

```java
booster.saveModel("model.bin");
```

- Generating model dump with feature map

```java
// dump without feature map
String[] model_dump = booster.getModelDump(null, false);
// dump with feature map
String[] model_dump_with_feature_map = booster.getModelDump("featureMap.txt", false);
```

- Load a model

```java
Booster booster = XGBoost.loadModel("model.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 (n sample, n class); for predictLeaf(), the result would be of shape (n sample, n class*ntrees).

```java
DMatrix dtest = new DMatrix("test.svm.txt");
// predict
float[][] predicts = booster.predict(dtest);
// predict leaf
float[][] leafPredicts = booster.predictLeaf(dtest, 0);
```
XGBoost4J-Spark Tutorial (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

Note: SparkContext will be stopped by default when XGBoost training task fails.

XGBoost4J-Spark 1.2.0+ exposes a parameter kill_spark_context_on_worker_failure. Set kill_spark_context_on_worker_failure to false so that the SparkContext will not be stopping on training failure. Instead of stopping the SparkContext, XGBoost4J-Spark will throw an exception instead. Users who want to re-use the SparkContext should wrap the training code in a try-catch block.

- Build an ML Application with XGBoost4J-Spark
  - Refer to XGBoost4J-Spark Dependency
  - Data Preparation
    * Read Dataset with Spark’s Built-In Reader
    * Transform Raw Iris Dataset
      * Dealing with missing values
  - Training
    * Early Stopping
    * Training with Evaluation Sets
  - Prediction
    * Batch Prediction
    * Single instance prediction
  - Model Persistence
    * Model and pipeline persistence
    * Interact with Other Bindings of XGBoost
- Building a ML Pipeline with XGBoost4J-Spark
### 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 `dmlc-core` to drive the training with XGBoost4J-Spark. It requires Python 2.7+. 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” columnm, which is essentially the label with three possible values: “Iris Setosa”, “Iris Versicolour” and “Iris Virginica”.

---

- Basic ML Pipeline
- Pipeline with Hyper-parameter Tuning
- Run XGBoost4J-Spark in Production
  - Parallel/Distributed Training
  - Gang Scheduling
  - Checkpoint During Training
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.

```java
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.types.
  DoubleType,
  StringType,
  StructField,
  StructType
val spark = SparkSession.builder().getOrCreate()
val schema = new StructType(
  Array(
    StructField("sepal length", DoubleType, true),
    StructField("sepal width", DoubleType, true),
    StructField("petal length", DoubleType, true),
    StructField("petal width", DoubleType, true),
    StructField("class", StringType, true)),
val rawInput = spark.read.schema(schema).csv("input_path")
```

At the first line, we create a instance of SparkSession which is the entry of any Spark program working with DataFrame. The schema variable defines the schema of DataFrame wrapping Iris data. With this explicitly set schema, we can define the columns’ name as well as their types; otherwise the column name would be the default ones derived by Spark, such as _col0, etc. Finally, we can use Spark’s built-in csv reader to load Iris csv file as a DataFrame named rawInput.

Spark also contains many built-in readers for other format. The latest version of Spark supports CSV, JSON, Parquet, and LIBSVM.

Transform Raw Iris Dataset

To make Iris dataset be recognizable to XGBoost, we need to

1. Transform String-typed label, i.e. “class”, to Double-typed label.
2. Assemble the feature columns as a vector to fit to the data interface of Spark ML framework.

To convert String-typed label to Double, we can use Spark’s built-in feature transformer StringIndexer.

```java
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.

### Dealing with missing values

XGBoost supports missing values by default (as described 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:

```scala
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:

```scala
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_value" -> "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 XGBoostClassifier:
```scala
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.

```scala
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 (`maximize_evaluation_metrics` with true), and `set num_early_stopping_rounds` with 5. The evaluation metric of 10th iteration is the maximum one until now. In the following iterations, if there is no evaluation metric greater than the 10th iteration’s (best one), the training would be early stopped at 15th iteration.

**Training with Evaluation 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

<table>
<thead>
<tr>
<th>features</th>
<th>classIndex</th>
<th>rawPrediction</th>
<th>probability</th>
<th>prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>[5.1, 3.5, 1.4, 0.2]</td>
<td>0.0</td>
<td>[3.4556984901428228222570]</td>
<td>[0.995796322822570]</td>
<td>0.0</td>
</tr>
<tr>
<td>[4.9, 3.0, 1.4, 0.2]</td>
<td>0.0</td>
<td>[3.4556984901428228222570]</td>
<td>[0.996180891990660]</td>
<td>0.0</td>
</tr>
<tr>
<td>[4.7, 3.2, 1.3, 0.2]</td>
<td>0.0</td>
<td>[3.4556984901428228222570]</td>
<td>[0.996433496475210]</td>
<td>0.0</td>
</tr>
<tr>
<td>[4.6, 3.1, 1.5, 0.2]</td>
<td>0.0</td>
<td>[3.4556984901428228222570]</td>
<td>[0.996360957622520]</td>
<td>0.0</td>
</tr>
<tr>
<td>[5.0, 3.6, 1.4, 0.2]</td>
<td>0.0</td>
<td>[3.4556984901428228222570]</td>
<td>[0.995796322822570]</td>
<td>0.0</td>
</tr>
<tr>
<td>[5.4, 3.9, 1.7, 0.4]</td>
<td>0.0</td>
<td>[3.4556984901428228222570]</td>
<td>[0.994285166263580]</td>
<td>0.0</td>
</tr>
<tr>
<td>[4.6, 3.4, 1.4, 0.3]</td>
<td>0.0</td>
<td>[3.4556984901428228222570]</td>
<td>[0.994334964752100]</td>
<td>0.0</td>
</tr>
<tr>
<td>[5.0, 3.4, 1.5, 0.2]</td>
<td>0.0</td>
<td>[3.4556984901428228222570]</td>
<td>[0.995796322822570]</td>
<td>0.0</td>
</tr>
<tr>
<td>[4.4, 2.9, 1.4, 0.2]</td>
<td>0.0</td>
<td>[3.4556984901428228222570]</td>
<td>[0.996180891990660]</td>
<td>0.0</td>
</tr>
<tr>
<td>[4.9, 3.1, 1.5, 0.1]</td>
<td>0.0</td>
<td>[3.4556984901428228222570]</td>
<td>[0.996360957622520]</td>
<td>0.0</td>
</tr>
<tr>
<td>[5.4, 3.7, 1.5, 0.2]</td>
<td>0.0</td>
<td>[3.4556984901428228222570]</td>
<td>[0.994285166263580]</td>
<td>0.0</td>
</tr>
<tr>
<td>[4.8, 3.4, 1.6, 0.2]</td>
<td>0.0</td>
<td>[3.4556984901428228222570]</td>
<td>[0.996433496475210]</td>
<td>0.0</td>
</tr>
<tr>
<td>[4.8, 3.0, 1.4, 0.1]</td>
<td>0.0</td>
<td>[3.4556984901428228222570]</td>
<td>[0.996180891990660]</td>
<td>0.0</td>
</tr>
<tr>
<td>[4.3, 3.0, 1.1, 0.1]</td>
<td>0.0</td>
<td>[3.4556984901428228222570]</td>
<td>[0.996180891990660]</td>
<td>0.0</td>
</tr>
<tr>
<td>[5.8, 4.0, 1.2, 0.2]</td>
<td>0.0</td>
<td>[3.4556984901428228222570]</td>
<td>[0.978092865556240]</td>
<td>0.0</td>
</tr>
<tr>
<td>[5.7, 4.4, 1.5, 0.4]</td>
<td>0.0</td>
<td>[3.4556984901428228222570]</td>
<td>[0.978092865556240]</td>
<td>0.0</td>
</tr>
<tr>
<td>[5.4, 3.9, 1.3, 0.4]</td>
<td>0.0</td>
<td>[3.4556984901428228222570]</td>
<td>[0.994285166263580]</td>
<td>0.0</td>
</tr>
<tr>
<td>[5.1, 3.5, 1.4, 0.3]</td>
<td>0.0</td>
<td>[3.4556984901428228222570]</td>
<td>[0.995796322822570]</td>
<td>0.0</td>
</tr>
<tr>
<td>[5.7, 3.8, 1.7, 0.3]</td>
<td>0.0</td>
<td>[3.4556984901428228222570]</td>
<td>[0.978092865556240]</td>
<td>0.0</td>
</tr>
<tr>
<td>[5.1, 3.8, 1.5, 0.3]</td>
<td>0.0</td>
<td>[3.4556984901428228222570]</td>
<td>[0.995796322822570]</td>
<td>0.0</td>
</tr>
</tbody>
</table>

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:

```scala
val xgbClassificationModelPath = "/tmp/xgbClassificationModel"
xgbClassificationModel.write.overwrite().save(xgbClassificationModelPath)
```

and then loading the model in another session:

```scala
import ml.dmlc.xgboost4j.scala.spark.XGBoostClassificationModel

val xgbClassificationModel2 = XGBoostClassificationModel.
  load(xgbClassificationModelPath)
xgbClassificationModel2.transform(xgbInput)
```

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. XGBoost4j-Spark supports export model to local by:

```scala
val nativeModelPath = "/tmp/nativeModel"
xgbClassificationModel.nativeBooster.saveModel(nativeModelPath)
```

Then we can load this model with single node Python XGBoost:

```python
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:

```python
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 IndexToString.
val labelConverter = new IndexToString()
  .setInputCol("prediction")
  .setOutputCol("realLabel")
  .setLabels(stringIndexer.labels)

We need to organize these steps as a Pipeline in Spark ML framework and evaluate the whole pipeline to get a PipelineModel:

```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:

```scala
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,
(continues on next page)```
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`:

```scala
xgbClassifier.setCheckpointInterval(2)
xgbClassifier.setCheckpointPath("/checkpoint_path")
```

An equivalent way is to pass in parameters in `XGBoostClassifier`’s constructor:

```scala
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 Java API

XGBoost4J Scala API

XGBoost4J-Spark Scala API

XGBoost4J-Flink Scala API

1.11 XGBoost.jl

See XGBoost.jl Project page.
1.12 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 demo/c-api/README.md for an overview and 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.

- C API documentation (latest master branch)
- C API documentation (last stable release)

1.13 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)

1.14 XGBoost Command Line version

See XGBoost Command Line walkthrough.

1.15 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:
1.15.1 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.

1.15.2 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 Travis and AppVeyor. 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 self-hosts a cloud server with Jenkins software installed: https://xgboost-ci.net/.

The self-hosted Jenkins CI server has recurring operating expenses. It utilizes a leading cloud provider (AWS) to
accommodate variable workload. The master node serving the web interface is available 24/7, to accommodate contrib-
utions from people around the globe. In addition, the master node launches slave nodes on demand, to run the test
suite on incoming contributions. To save cost, the slave nodes are terminated when they are no longer needed.

To help defray the hosting cost, the XGBoost project seeks donations from third parties.

Donations and Sponsorships

Donors may choose to make one-time donations or recurring donations on monthly or yearly basis. Donors who
commit to the Sponsor tier will have their logo displayed on the front page of the XGBoost project.
Fiscal host: Open Source Collective 501(c)(6)

The Project Management Committee (PMC) of the XGBoost project appointed Open Source Collective as their fiscal host. The platform is a 501(c)(6) registered entity and will manage the funds on the behalf of the PMC so that PMC members will not have to manage the funds directly. The platform currently hosts several well-known Javascript frameworks such as Babel, Vue, and Webpack.

All expenses incurred for hosting CI will be submitted to the fiscal host with receipts. Only the expenses in the following categories will be approved for reimbursement:

- Cloud expenses for the Jenkins CI server (https://xgboost-ci.net)
- Cost of domain https://xgboost-ci.net
- Meetup.com account for XGBoost project
- Hosting cost of the User Forum (https://discuss.xgboost.ai)

Administration of Jenkins CI server

The PMC shall appoint committer(s) to administer the Jenkins CI server 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.

1.15.3 Coding Guideline

Contents

- C++ Coding Guideline
- Python Coding Guideline
- R Coding Guideline
  - Code Style
  - Rmarkdown Vignettes
  - R package versioning
  - Registering native routines in R
- Running Formatting Checks Locally
  - Linter
  - Clang-tidy
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.
- You can check the style of the code by typing the following command at root folder.

```
make rcpplint
```

- 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 vignette:

- Add the original markdown 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 the 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.

**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 occurred 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:

```bash
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:

```bash
cd /path/to/xgboost/
python3 tests/ci_build/tidy.py --cuda=0
```

Similarly, if you want to exclude C++ source from linting:

```bash
cd /path/to/xgboost/
python3 tests/ci_build/tidy.py --cpp=0
```

1.15.4 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

1.15. Contribute to XGBoost
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:

```shell
# Tell Python where to find XGBoost module
export PYTHONPATH=./python-package
pytest -v -s --fulltrace tests/python
```

In addition, to test CUDA code, run:

```shell
# 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

Running XGBoost on CUDA with address sanitizer (asan) will raise memory error. To use asan with CUDA correctly, you need to configure asan via ASAN_OPTIONS environment variable:

```
ASAN_OPTIONS=protect_shadow_gap=0 ${{BUILD_DIR}}/testxgboost
```

Other sanitizer runtime options

By default undefined sanitizer doesn’t print out the backtrace. You can enable it by exporting environment variable:

```
UBSAN_OPTIONS=print_stacktrace=1 ${{BUILD_DIR}}/testxgboost
```

For details, please consult official documentation for sanitizers.
1.15.5 Documentation and Examples

Contents

- Documents
- Examples

Documents

- 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.

Examples

- Use cases and examples will be in demo.
- We are super excited to hear about your story, if you have blogposts, tutorials code solutions using XGBoost, please tell us and we will add a link in the example pages.

1.15.6 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.

1.15.7 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.
1.15.8 Automated testing in XGBoost project

This document collects tips for using the Continuous Integration (CI) service of the XGBoost project.

Contents

- GitHub Actions

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.)
X

xgboost.core, 90
xgboost.dask, 134
xgboost.plotting, 131
xgboost.sklearn, 102
xgboost.training, 100
<table>
<thead>
<tr>
<th>Class/Method/Property</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>apply() (xgboost.XGBClassifier method)</td>
<td>109</td>
</tr>
<tr>
<td>apply() (xgboost.XGBRanker method)</td>
<td>115</td>
</tr>
<tr>
<td>apply() (xgboost.XGBRegressor method)</td>
<td>104</td>
</tr>
<tr>
<td>apply() (xgboost.XGBRFClassifier method)</td>
<td>126</td>
</tr>
<tr>
<td>apply() (xgboost.XGBRFRegressor method)</td>
<td>121</td>
</tr>
<tr>
<td>attr() (xgboost.Booster method)</td>
<td>94</td>
</tr>
<tr>
<td>attributes() (xgboost.Booster method)</td>
<td>95</td>
</tr>
<tr>
<td>boost() (xgboost.Booster method)</td>
<td>95</td>
</tr>
<tr>
<td>Booster (class in xgboost)</td>
<td>94</td>
</tr>
<tr>
<td>coef_() (xgboost.XGBClassifier property)</td>
<td>109</td>
</tr>
<tr>
<td>coef_() (xgboost.XGBRanker property)</td>
<td>115</td>
</tr>
<tr>
<td>coef_() (xgboost.XGBRegressor property)</td>
<td>104</td>
</tr>
<tr>
<td>coef_() (xgboost.XGBRFClassifier property)</td>
<td>126</td>
</tr>
<tr>
<td>coef_() (xgboost.XGBRFRegressor property)</td>
<td>121</td>
</tr>
<tr>
<td>config_context() (in module xgboost)</td>
<td>88</td>
</tr>
<tr>
<td>copy() (xgboost.Booster method)</td>
<td>95</td>
</tr>
<tr>
<td>cv() (in module xgboost)</td>
<td>101</td>
</tr>
<tr>
<td>DaskDeviceQuantileDMatrix() (in module xgboost.dask)</td>
<td>135</td>
</tr>
<tr>
<td>DaskDMatrix() (in module xgboost.dask)</td>
<td>134</td>
</tr>
<tr>
<td>DaskXGBClassifier() (in module xgboost.dask)</td>
<td>137</td>
</tr>
<tr>
<td>DaskXGBRegressor() (in module xgboost.dask)</td>
<td>139</td>
</tr>
<tr>
<td>DeviceQuantileDMatrix (class in xgboost)</td>
<td>93</td>
</tr>
<tr>
<td>DMATRIX (class in xgboost)</td>
<td>90</td>
</tr>
<tr>
<td>dump_model() (xgboost.Booster method)</td>
<td>95</td>
</tr>
<tr>
<td>EarlyStopping() (in module xgboost.callback)</td>
<td>133</td>
</tr>
<tr>
<td>eval() (xgboost.Booster method)</td>
<td>95</td>
</tr>
<tr>
<td>eval_set() (xgboost.Booster method)</td>
<td>95</td>
</tr>
<tr>
<td>evals_result() (xgboost.XGBClassifier method)</td>
<td>110</td>
</tr>
<tr>
<td>evals_result() (xgboost.XGBRanker method)</td>
<td>116</td>
</tr>
<tr>
<td>evals_result() (xgboost.XGBRegressor method)</td>
<td>104</td>
</tr>
<tr>
<td>evals_result() (xgboost.XGBRFClassifier method)</td>
<td>127</td>
</tr>
<tr>
<td>evals_result() (xgboost.XGBRFRegressor method)</td>
<td>121</td>
</tr>
<tr>
<td>EvaluationMonitor() (in module xgboost.callback)</td>
<td>133</td>
</tr>
<tr>
<td>feature_importances_() (xgboost.XGBClassifier property)</td>
<td>110</td>
</tr>
<tr>
<td>feature_importances_() (xgboost.XGBRanker property)</td>
<td>116</td>
</tr>
<tr>
<td>feature_importances_() (xgboost.XGBRegressor property)</td>
<td>105</td>
</tr>
<tr>
<td>feature_importances_() (xgboost.XGBRFClassifier property)</td>
<td>127</td>
</tr>
<tr>
<td>feature_importances_() (xgboost.XGBRFRegressor property)</td>
<td>122</td>
</tr>
<tr>
<td>feature_names() (xgboost.DMatrix property)</td>
<td>91</td>
</tr>
<tr>
<td>feature_types() (xgboost.DMatrix property)</td>
<td>91</td>
</tr>
<tr>
<td>fit() (xgboost.XGBClassifier method)</td>
<td>110</td>
</tr>
<tr>
<td>fit() (xgboost.XGBRanker method)</td>
<td>116</td>
</tr>
<tr>
<td>fit() (xgboost.XGBRegressor method)</td>
<td>105</td>
</tr>
<tr>
<td>fit() (xgboost.XGBRFClassifier method)</td>
<td>127</td>
</tr>
<tr>
<td>fit() (xgboost.XGBRFRegressor method)</td>
<td>122</td>
</tr>
<tr>
<td>get_base_margin() (xgboost.DMatrix method)</td>
<td>91</td>
</tr>
<tr>
<td>get_booster() (xgboost.XGBClassifier method)</td>
<td>111</td>
</tr>
<tr>
<td>get_booster() (xgboost.XGBRanker method)</td>
<td>118</td>
</tr>
<tr>
<td>get_booster() (xgboost.XGBRegressor method)</td>
<td>106</td>
</tr>
<tr>
<td>get_booster() (xgboost.XGBRFClassifier method)</td>
<td>128</td>
</tr>
<tr>
<td>get_booster() (xgboost.XGBRFRegressor method)</td>
<td>123</td>
</tr>
<tr>
<td>get_config() (in module xgboost)</td>
<td>90</td>
</tr>
<tr>
<td>get_dump() (xgboost.Booster method)</td>
<td>96</td>
</tr>
<tr>
<td>get_float_info() (xgboost.DMatrix method)</td>
<td>91</td>
</tr>
<tr>
<td>get_fscore() (xgboost.Booster method)</td>
<td>96</td>
</tr>
<tr>
<td>get_dump() (xgboost.Booster method)</td>
<td>96</td>
</tr>
<tr>
<td>get_config() (in module xgboost)</td>
<td>90</td>
</tr>
<tr>
<td>get_fscore() (xgboost.Booster method)</td>
<td>96</td>
</tr>
</tbody>
</table>

**INDEX**
set_label() (xgboost.DMatrix method), 93
set_param() (xgboost.Booster method), 99
set_params() (xgboost.XGBClassifier method), 113
set_params() (xgboost.XGBRanker method), 119
set_params() (xgboost.XGBRegressor method), 107
set_params() (xgboost.XGBRFClassifier method), 130
set_params() (xgboost.XGBRFRegressor method), 124
set_uint_info() (xgboost.DMatrix method), 93
set_weight() (xgboost.DMatrix method), 93
slice() (xgboost.DMatrix method), 93

to_graphviz() (in module xgboost), 132
train() (in module xgboost), 100
train() (in module xgboost.dask), 135
TrainingCallback() (in module xgboost.callback), 133
TrainingCheckpoint() (in module xgboost.callback), 133
trees_to_dataframe() (xgboost.Booster method), 99

update() (xgboost.Booster method), 100

XGBClassifier (class in xgboost), 107
xgboost.core module, 90
xgboost.dask module, 134
xgboost.plotting module, 131
xgboost.sklearn module, 102
xgboost.training module, 100
XGBRanker (class in xgboost), 113
XGBRegressor (class in xgboost), 102
XGBRFClassifier (class in xgboost), 124
XGBRFRegressor (class in xgboost), 119