
xgboost

Release 0.4

xgboost developers

October 12, 2015

1	How to Get Started	3
2	Tutorials	5
3	Highlight Solutions	7
4	User Guide	9
4.1	Frequently Asked Questions	9
4.2	Introduction to Boosted Trees	10
4.3	Python Package Introduction	16
4.4	Using XGBoost External Memory Version(beta)	27
4.5	Text Input Format of DMatrix	28
4.6	Build XGBoost	29
4.7	XGBoost Parameters	31
4.8	Notes on Parameter Tuning	35
5	Developer Guide	37
5.1	Developer Guide	37
6	API Reference	39
6.1	Python API Reference	39
	Python Module Index	49

This is document of xgboost library. XGBoost is short for eXtreme gradient boosting. This is a library that is designed, and optimized for boosted (tree) algorithms. The goal of this library is to push the extreme of the computation limits of machines to provide a *scalable*, *portable* and *accurate* for large scale tree boosting.

This document is hosted at <http://xgboost.readthedocs.org/>. You can also browse most of the documents in github directly.

How to Get Started

The best way to get started to learn xgboost is by the examples. There are three types of examples you can find in xgboost.

- *Tutorials* are self-contained tutorials on a complete data science tasks.
- *XGBoost Code Examples* are collections of code and benchmarks of xgboost.
 - There is a walkthrough section in this to walk you through specific API features.
- *Highlight Solutions* are presentations using xgboost to solve real world problems.
 - These examples are usually more advanced. You can usually find state-of-art solutions to many problems and challenges in here.

After you gets familiar with the interface, checkout the following additional resources

- *Frequently Asked Questions*
- *Learning what is in Behind: Introduction to Boosted Trees*
- *User Guide* contains comprehensive list of documents of xgboost.
- *Developer Guide*

Tutorials

Tutorials are self contained materials that teaches you how to achieve a complete data science task with xgboost, these are great resources to learn xgboost by real examples. If you think you have something that belongs to here, send a pull request.

- [Binary classification using XGBoost Command Line \(CLI\)](#)
 - This tutorial introduces the basic usage of CLI version of xgboost
- [Introduction of XGBoost in Python \(python\)](#)
 - This tutorial introduces the python package of xgboost
- [Introduction to XGBoost in R \(R package\)](#)
 - This is a general presentation about xgboost in R.
- [Discover your data with XGBoost in R \(R package\)](#)
 - This tutorial explaining feature analysis in xgboost.
- [Understanding XGBoost Model on Otto Dataset \(R package\)](#)
 - This tutorial teaches you how to use xgboost to compete kaggle otto challenge.

Highlight Solutions

This section is about blogposts, presentation and videos discussing how to use xgboost to solve your interesting problem. If you think something belongs to here, send a pull request.

- [Kaggle CrowdFlower winner's solution by Chenglong Chen](#)
- [Kaggle Malware Prediction winner's solution](#)
- [Kaggle Tradeshift winning solution by daxiongshu](#)
- [Feature Importance Analysis with XGBoost in Tax audit](#)
- [Video tutorial: Better Optimization with Repeated Cross Validation and the XGBoost model](#)
- [Winning solution of Kaggle Higgs competition: what a single model can do](#)

4.1 Frequently Asked Questions

This document contains frequently asked questions about xgboost.

4.1.1 How to tune parameters

See [Parameter Tunning Guide](#)

4.1.2 Description on the model

See [Introduction to Boosted Trees](#)

4.1.3 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.

4.1.4 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.

4.1.5 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 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 API (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.

4.1.6 How can I port the 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.

4.1.7 Do you support LambdaMART

Yes, xgboost implements LambdaMART. Checkout the objective section in [parameters](#)

4.1.8 How to deal with Missing Value

xgboost support missing value by default

4.1.9 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.

4.2 Introduction to Boosted Trees

XGBoost is short for “Extreme Gradient Boosting”, where the term “Gradient Boosting” is proposed in the paper *Greedy Function Approximation: A Gradient Boosting Machine*, Friedman. Based on this original model. This is a tutorial on boosted trees, most of content are based on this [slide](#) by the author of xgboost.

The GBM(boosted trees) has been around for really a while, and there are a lot of materials on the topic. This tutorial tries to explain boosted trees in a self-contained and principled way of supervised learning. We think this explanation is cleaner, more formal, and motivates the variant used in xgboost.

4.2.1 Elements of Supervised Learning

XGBoost is used for supervised learning problems, where we use the training data x_i to predict a target variable y_i . Before we dive into trees, 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 on how to given the prediction y_i given x_i . For example, a common model is *linear model*, where the prediction is given by $\hat{y}_i = \sum_j w_j x_{ij}$, a linear combination of weighted input features. The prediction value can have different interpretations, depending on the task. For example, it can be logistic transformed to get the probability of positive class in logistic regression, and it can also be used as ranking score when we want to rank the outputs.

The **parameters** are the undermined part that we need to learn from data. In linear regression problem, the parameters are the co-efficients w . Usually we will use Θ to denote the parameters.

Objective Function : Training Loss + Regularization

Based on different understanding or assumption of y_i , we can have different problems as regression, classification, ordering, etc. We need to find a way to find the best parameters given the training data. In order to do so, we need to define a so called **objective function**, to measure the performance of the model under certain set of parameters.

A very important fact about objective functions, is they **must always** contains two parts: training loss and regularization.

$$Obj(\Theta) = L(\Theta) + \Omega(\Theta)$$

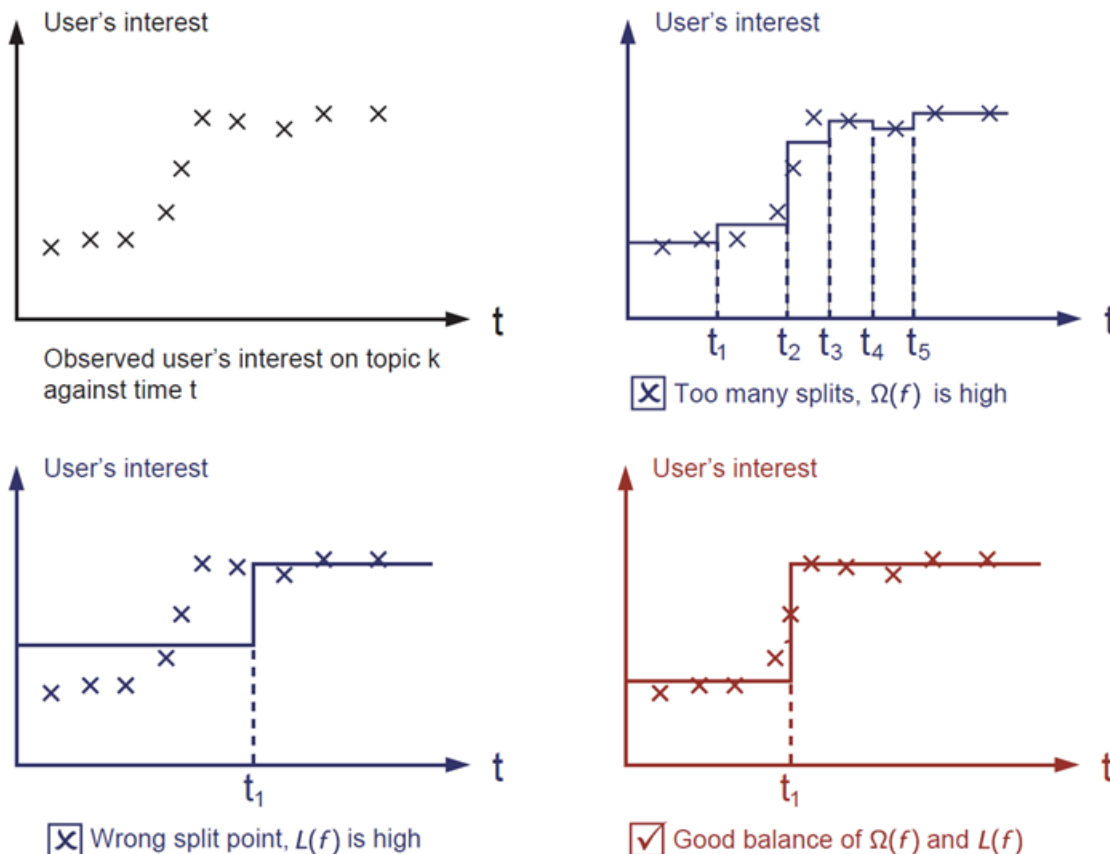
where L is the training loss function, and Ω is the regularization term. The training loss measures how *predictive* our model is on training data. For example, a commonly used training loss is mean squared error.

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

Another commonly used loss function is logistic loss 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 tree you think is the best fit?



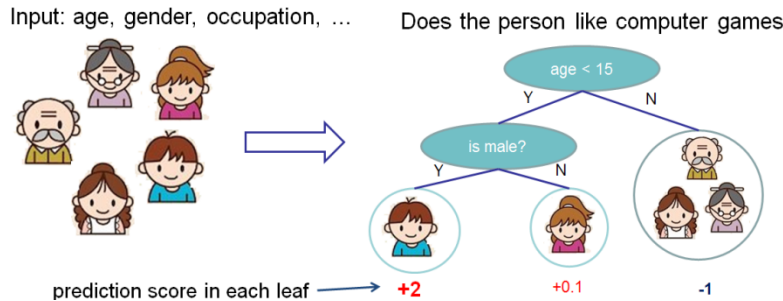
The answer is already marked as red. Please think if it is reasonable to you visually. The general principle is we want a **simple** and **predictive** model. The tradeoff between the two is also referred as bias-variance tradeoff in machine learning.

Why introduce the general principle

The elements introduced in above forms the basic elements of supervised learning, and they are naturally the building blocks of machine learning toolkits. For example, you should be able to answer what is the difference and common parts between boosted trees and random forest. 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.

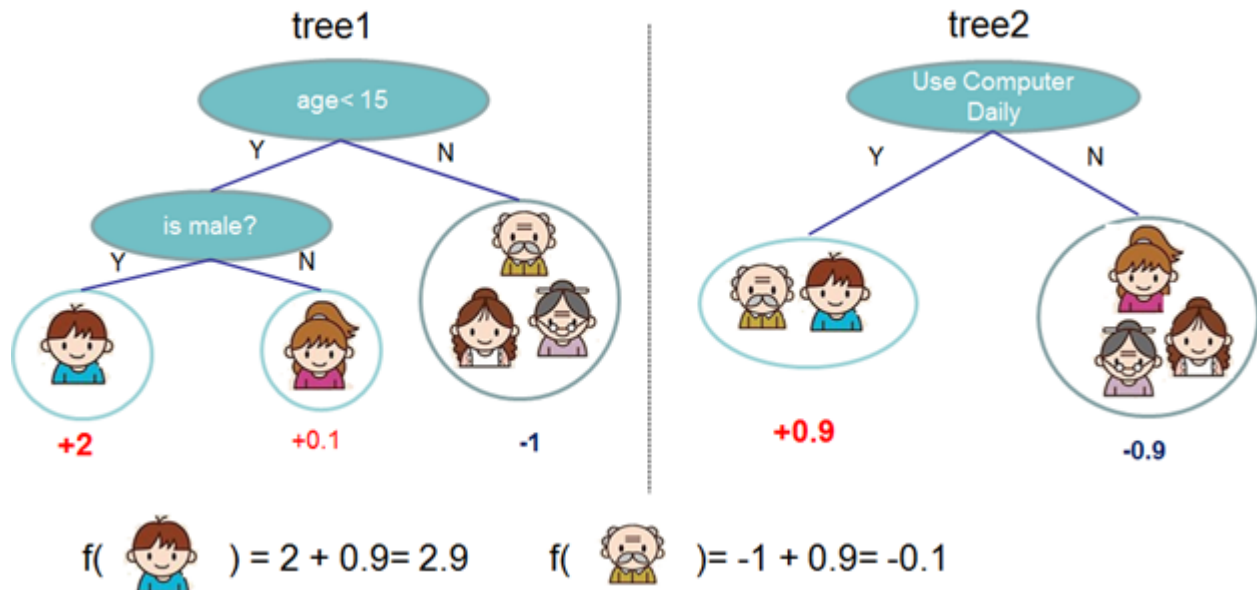
4.2.2 Tree Ensemble

Now that we have introduced the elements of supervised learning, let us get started with real trees. To begin with, let us first learn what is the **model** of xgboost: tree ensembles. The tree ensemble model is a set of classification and regression trees (CART). Here's a simple example of a CART that classifies is someone will like computer games.



We classify the members in this family into different leaves, and assign them the score on corresponding leaf. A CART is a bit different from decision trees, where the leaf only contain 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 makes the unified optimization step easier, as we will see in later part of this tutorial.

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



Here is an example of 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. Therefore our objective to optimize can be written as

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

Now here comes the question, what is the *model* of random forest? It is exactly tree ensembles! So random forest and boosted trees are not different in terms of model, the difference is how we train them. This means if you write a predictive service of tree ensembles, you only need to write one of them and they should directly work for both random forest and boosted trees. One example of elements of supervised learning rocks.

4.2.3 Tree Boosting

After introducing the model, let us begin with the real training part. How should we learn the trees? The answer is, as is always for all supervised learning models: *define an objective function, and optimize it!*

Assume we have the following objective function (remember it always need to contain training loss, and regularization)

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

Additive Training

First thing we want to ask is what are *parameters* of trees. You can find what we need to learn are those functions f_i , with each contains the structure of the tree, and the leaf score. This is much harder than traditional optimization problem where you can take the gradient and go. It is not easy to train all the trees at once. Instead, we use an additive strategy: fix what we have learned, add a new tree at a time. We note the prediction value at step t by $\hat{y}_i^{(t)}$, so we have

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

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

$$\begin{aligned} Obj^{(t)} &= \sum_{i=1}^n l(y_i, \hat{y}_i^{(t)}) + \sum_{i=1}^t \Omega(f_i) \\ &= \sum_{i=1}^n l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \Omega(f_t) + constant \end{aligned}$$

If we consider using MSE as our loss function, it becomes the following form.

$$\begin{aligned} Obj^{(t)} &= \sum_{i=1}^n (y_i - (\hat{y}_i^{(t-1)} + f_t(x_i)))^2 + \sum_{i=1}^t \Omega(f_i) \\ &= \sum_{i=1}^n [2(\hat{y}_i^{(t-1)} - y_i)f_t(x_i) + f_t(x_i)^2] + \Omega(f_t) + constant \end{aligned}$$

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

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

where the g_i and h_i are defined as

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

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

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

This becomes our optimization goal for the new tree. One important advantage of this definition, is that it only depends on g_i and h_i , this is how xgboost allows support of customization of loss functions. We can optimized every loss function, including logistic regression, weighted logistic regression, using the exactly the same solver that takes g_i and h_i as input!

Model Complexity

We have introduced the training step, but wait, there is one important thing, the **regularization**! 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 a tree $f(x)$ as

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

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 specific one works well in practice. The regularization is one part most tree packages takes less carefully, or simply ignore. This was due to the traditional treatment tree learning only emphasize improving impurity, while the complexity control part are more lies as part of heuristics. By defining it formally, we can get a better idea of what we are learning, and yes it works well in practice.

The Structure Score

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

$$\begin{aligned} Obj^{(t)} &\approx \sum_{i=1}^n [g_i w_q(x_i) + \frac{1}{2} h_i w_q^2(x_i)] + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^T w_j^2 \\ &= \sum_{j=1}^T [(\sum_{i \in I_j} g_i) w_j + \frac{1}{2} (\sum_{i \in I_j} h_i + \lambda) w_j^2] + \gamma T \end{aligned}$$

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 change 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 $H_j = \sum_{i \in I_j} h_i$:






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

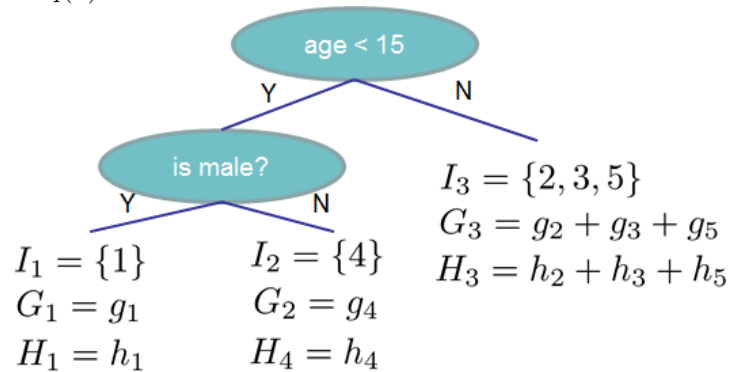
In this equation w_j are independent to each other, the form $G_j w_j + \frac{1}{2} (H_j + \lambda) w_j^2$ is quadratic and the best w_j for a given structure $q(x)$ and the best objective reduction we can get:

$$\begin{aligned} w_j^* &= -\frac{G_j}{H_j + \lambda} \\ Obj^* &= -\frac{1}{2} \sum_{j=1}^T \frac{G_j^2}{H_j + \lambda} + \gamma T \end{aligned}$$

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

Instance index gradient statistics

1		g_1, h_1
2		g_2, h_2
3		g_3, h_3
4		g_4, h_4
5		g_5, h_5



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

The smaller the score is, the better the structure is

If all these sounds a bit complicated. Let us take a look the 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 impurity measure in decision tree, except that it also takes the model complexity into account.

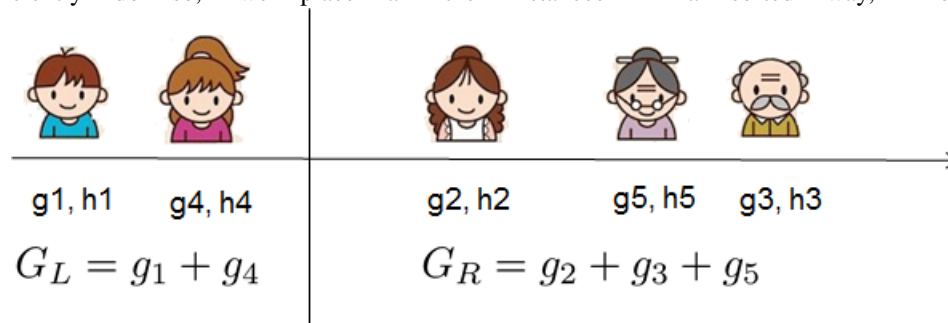
Learn the tree structure

Now we have a way to measure how good a tree is ideally we can enumerate all possible trees and pick the best one. In practice it is impossible, so we will try to 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 find an important fact here: if the gain is smaller than γ , we would 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 comes up with the reason these techniques :)

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



Then 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.

4.2.4 Final words on XGBoost

Now you have understand what is a boosted tree, you may ask, where is the introduction on **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!

4.3 Python Package Introduction

This document gives a basic walkthrough of xgboost python package.

List of other Helpful Links

4.3.1 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.

The document in this page is automatically generated by sphinx. The content do not render at github, you can view it at http://xgboost.readthedocs.org/en/latest/python/python_api.html

Core Data Structure

Core XGBoost Library.

```
class xgboost.DMatrix(data, label=None, missing=0.0, weight=None, silent=False, feature_names=None, feature_types=None)
```

Bases: object

Data Matrix used in XGBoost.

DMatrix is a internal data structure that used by XGBoost which is optimized for both memory efficiency and training speed. You can construct DMatrix from numpy.arrays

feature_names

Get feature names (column labels).

Returns **feature_names**

Return type list or None

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 float 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.

Parameters

- **fname** (*string*) – 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

set_group (*group*)

Set group size of DMatrix (used for ranking).

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

set_label (*label*)

Set label of dmatrix

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

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

set_weight (*weight*)

Set weight of each instance.

Parameters **weight** (*array like*) – Weight for each data point

slice (*rindex*)

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

Parameters **rindex** (*list*) – List of indices to be selected.

Returns **res** – A new DMatrix containing only selected indices.

Return type *DMatrix*

class `xgboost.Booster` (*params=None, cache=(), model_file=None*)

Bases: `object`

“A Booster of of XGBoost.

Booster is the model of xgboost, that contains low level routines for training, prediction and evaluation.

boost (*dtrain, grad, hess*)

Boost the booster for one iteration, with customized gradient statistics.

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 model into a text file.

Parameters

- **foout** (*string*) – Output file name.
- **fmap** (*string, optional*) – Name of the file containing feature map names.
- **with_stats** (*bool (optional)*) – Controls whether the split statistics are output.

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*)

Returns the dump the model as a list of strings.

get_fscore (*fmap=''*)

Get feature importance of each feature.

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

load_model (*fname*)

Load the model from a file.

Parameters **fname** (*string or a memory buffer*) – Input file name or memory buffer(see also `save_raw`)

predict (*data*, *output_margin=False*, *ntree_limit=0*, *pred_leaf=False*)

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 `bst.copy()` to make copies of model object and then call predict

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 (nsample, ntrees) with each record indicating the predicted leaf index of each sample in each tree. Note that the leaf index of a tree is unique per tree, so you may find leaf 1 in both tree 1 and tree 0.

Returns prediction

Return type numpy array

save_model (*fname*)

Save the model to a file.

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

save_raw ()

Save the model to a in memory buffer representation

Returns

Return type a in memory buffer representation of the model

set_param (*params*, *value=None*)

Set parameters into the Booster.

Parameters

- **params** (*dict/list/str*) – list of key,value paris, dict of key to value or simply str key
- **value** (*optional*) – value of the specified parameter, when params is str key

update (*dtrain*, *iteration*, *fobj=None*)

Update for one iteration, with objective function calculated internally.

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*, *early_stopping_rounds=None*, *evals_result=None*, *verbose_eval=True*)
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)** (*watchlist*) – List of items to be evaluated during training, this allows user to watch performance on the validation set.
- **obj** (*function*) – Customized objective function.
- **feval** (*function*) – Customized evaluation function.
- **early_stopping_rounds** (*int*) – Activates early stopping. Validation error needs to decrease at least every <early_stopping_rounds> round(s) to continue training. Requires at least one item in evals. If there's more than one, will use the last. Returns the model from the last iteration (not the best one). If early stopping occurs, the model will have two additional fields: `bst.best_score` and `bst.best_iteration`.
- **evals_result** (*dict*) – This dictionary stores the evaluation results of all the items in watchlist
- **verbose_eval** (*bool*) – If *verbose_eval* then the evaluation metric on the validation set, if given, is printed at each boosting stage.

Returns booster

Return type a trained booster model

`xgboost.cv` (*params*, *dtrain*, *num_boost_round=10*, *nfold=3*, *metrics=()*, *obj=None*, *feval=None*, *fpreproc=None*, *as_pandas=True*, *show_progress=None*, *show_stdv=True*, *seed=0*)
Cross-validation with given parameters.

Parameters

- **params** (*dict*) – Booster params.
- **dtrain** (*DMatrix*) – Data to be trained.
- **num_boost_round** (*int*) – Number of boosting iterations.
- **nfold** (*int*) – Number of folds in CV.
- **metrics** (*list of strings*) – Evaluation metrics to be watched in CV.
- **obj** (*function*) – Custom objective function.
- **feval** (*function*) – Custom evaluation function.
- **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`
- **show_progress** (*bool or None, default None*) – Whether to display the progress. If None, progress will be displayed when `np.ndarray` is returned.
- **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`).

Returns evaluation history

Return type list(string)

Scikit-Learn API

Scikit-Learn Wrapper interface for XGBoost.

```
class xgboost.XGBRegressor(max_depth=3, learning_rate=0.1, n_estimators=100, silent=True,
                           objective='reg:linear', nthread=-1, gamma=0, min_child_weight=1,
                           max_delta_step=0, subsample=1, colsample_bytree=1, base_score=0.5,
                           seed=0, missing=None)
```

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

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

max_depth [int] Maximum tree depth for base learners.

learning_rate [float] Boosting learning rate (xgb's "eta")

n_estimators [int] Number of boosted trees to fit.

silent [boolean] Whether to print messages while running boosting.

objective [string] Specify the learning task and the corresponding learning objective.

nthread [int] Number of parallel threads used to run xgboost.

gamma [float] Minimum loss reduction required to make a further partition on a leaf node of the tree.

min_child_weight [int] 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.

base_score: The initial prediction score of all instances, global bias.

seed [int] Random number seed.

missing [float, optional] Value in the data which needs to be present as a missing value. If None, defaults to `np.nan`.

```
class xgboost.XGBClassifier(max_depth=3, learning_rate=0.1, n_estimators=100, silent=True,
                            objective='binary:logistic', nthread=-1, gamma=0, min_child_weight=1,
                            max_delta_step=0, subsample=1, colsample_bytree=1,
                            base_score=0.5, seed=0, missing=None)
```

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

Implementation of the scikit-learn API for XGBoost classification.

Parameters

max_depth [int] Maximum tree depth for base learners.

learning_rate [float] Boosting learning rate (xgb's "eta")

n_estimators [int] Number of boosted trees to fit.

silent [boolean] Whether to print messages while running boosting.

objective [string] Specify the learning task and the corresponding learning objective.

nthread [int] Number of parallel threads used to run xgboost.

gamma [float] Minimum loss reduction required to make a further partition on a leaf node of the tree.

min_child_weight [int] 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.

base_score: The initial prediction score of all instances, global bias.

seed [int] Random number seed.

missing [float, optional] Value in the data which needs to be present as a missing value. If None, defaults to np.nan.

fit (*X*, *y*, *sample_weight=None*, *eval_set=None*, *eval_metric=None*, *early_stopping_rounds=None*, *verbose=True*)
Fit gradient boosting classifier

Parameters

- **X** (*array_like*) – Feature matrix
- **y** (*array_like*) – Labels
- **sample_weight** (*array_like*) – Weight for each instance
- **eval_set** (*list, optional*) – A list of (X, y) pairs to use as a validation set for early-stopping
- **eval_metric** (*str, callable, optional*) – If a str, should be a built-in evaluation metric to use. See doc/parameter.md. 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. This objective is always minimized.
- **early_stopping_rounds** (*int, optional*) – Activates early stopping. Validation error needs to decrease at least every <early_stopping_rounds> round(s) to continue training. Requires at least one item in evals. If there's more than one, will use the last. Returns the model from the last iteration (not the best one). If early stopping occurs, the model will have two additional fields: bst.best_score and bst.best_iteration.
- **verbose** (*bool*) – If *verbose* and an evaluation set is used, writes the evaluation metric measured on the validation set to stderr.

Plotting API

Plotting Library.

`xgboost.plot_importance` (*booster*, *ax=None*, *height=0.2*, *xlim=None*, *title='Feature importance'*, *xlabel='F score'*, *ylabel='Features'*, *grid=True*, ***kwargs*)

Plot importance based on fitted trees.

Parameters

- **booster** (*Booster or dict*) – Booster 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.
- **height** (*float, default 0.2*) – Bar height, passed to `ax.barh()`
- **xlim** (*tuple, default None*) – Tuple passed to `axes.xlim()`
- **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.
- **kwargs** – Other keywords passed to `ax.barh()`

Returns ax

Return type matplotlib Axes

`xgboost.plot_tree` (*booster*, *num_trees=0*, *rankdir='UT'*, *ax=None*, ***kwargs*)

Plot specified tree.

Parameters

- **booster** ([Booster](#)) – Booster instance
- **num_trees** (*int, default 0*) – Specify the ordinal number of target tree
- **rankdir** (*str, default "UT"*) – 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*, *num_trees=0*, *rankdir='UT'*, *yes_color='#0000FF'*, *no_color='#FF0000'*, ***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](#)) – Booster instance
- **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.
- **kwargs** – Other keywords passed to `graphviz graph_attr`

Returns ax

Return type matplotlib Axes

4.3.2 Install XGBoost

To install XGBoost, do the following steps.

- You need to run `make` in the root directory of the project
- In the `python-package` directory run

```
python setup.py install
```

```
import xgboost as xgb
```

4.3.3 Data Interface

XGBoost python module is able to loading from libsvm txt format file, Numpy 2D array and xgboost binary buffer file. The data will be store in `DMatrix` object.

- To load libsvm text format file and XGBoost binary file into `DMatrix`, the usage is like

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

- To load numpy array into `DMatrix`, the usage is like

```
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)
```

- Build `DMatrix` from `scipy.sparse`

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

- Saving `DMatrix` into XGBoost binary file will make loading faster in next time. The usage is like:

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

- To handle missing value in `DMatrix`, you can initialize the `DMatrix` like:

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

- Weight can be set when needed, like

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

4.3.4 Setting Parameters

XGBoost use list of pair to save `parameters`. Eg

- Booster parameters

```
param = {'bst:max_depth':2, 'bst:eta':1, 'silent':1, 'objective':'binary:logistic' }
param['nthread'] = 4
plst = param.items()
plst += [('eval_metric', 'auc')] # Multiple evals can be handled in this way
plst += [('eval_metric', 'ams@0')]
```

- Specify validations set to watch performance

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

4.3.5 Training

With parameter list and data, you are able to train a model.

- Training

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

- Saving model After training, you can save model and dump it out.

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

- Dump Model and Feature Map You can dump the model to txt and review the meaning of model

```
# dump model
bst.dump_model('dump.raw.txt')
# dump model with feature map
bst.dump_model('dump.raw.txt', 'featmap.txt')
```

- Loading model After you save your model, you can load model file at anytime by using

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

4.3.6 Early Stopping

If you have a validation set, you can use early stopping to find the optimal number of boosting rounds. Early stopping requires at least one set in evals. If there's more than one, it will use the last.

```
train(..., evals=evals, early_stopping_rounds=10)
```

The model will train until the validation score stops improving. Validation error needs to decrease at least every `early_stopping_rounds` to continue training.

If early stopping occurs, the model will have two additional fields: `bst.best_score` and `bst.best_iteration`. Note that `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).

4.3.7 Prediction

After you training/loading a model and preparing the data, you can start to do prediction.

```
# 7 entities, each contains 10 features
data = np.random.rand(7, 10)
dtest = xgb.DMatrix(data)
ypred = bst.predict(xgmat)
```

If early stopping is enabled during training, you can predict with the best iteration.

```
ypred = bst.predict(xgmat, ntree_limit=bst.best_iteration)
```

4.3.8 Plotting

You can use plotting module to plot importance and output tree.

To plot importance, use `plot_importance`. This function requires `matplotlib` to be installed.

```
xgb.plot_importance(bst)
```

To output tree via `matplotlib`, use `plot_tree` specifying ordinal number of the target tree. This function requires `graphviz` and `matplotlib`.

```
xgb.plot_tree(bst, num_trees=2)
```

When you use IPython, you can use `to_graphviz` function which converts the target tree to `graphviz` instance. `graphviz` instance is automatically rendered on IPython.

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

4.4 Using XGBoost External Memory Version(beta)

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 filename format

filename#cacheprefix

The `filename` is the normal path to libsvm file you want to load in, `cacheprefix` is a path to a cache file that xgboost will use for external memory cache.

The following code was extracted from `../demo/guide-python/external_memory.py`

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

You can find that there is additional `#dtrain.cache` following the libsvm file, this is the name of cache file. For CLI version, simply use `"../data/agaricus.txt.train#dtrain.cache"` in filename.

4.4.1 Performance Note

- the parameter `nthread` should be set to number of *real* cores
 - Most modern CPU offer hyperthreading, which means you can have a 4 core cpu with 8 threads
 - Set `nthread` to be 4 for maximum performance in such case

4.4.2 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 temporal so that you can directly use `dtrain.cache` to cache to current folder.

4.4.3 Usage Note

- This is a experimental version
 - If you like to try and test it, report results to <https://github.com/dmlc/xgboost/issues/244>
- Currently only importing from libsvm format is supported
 - Contribution of ingestion from other common external memory data source is welcomed

4.5 Text Input Format of DMatrix

4.5.1 Basic Input Format

As we have mentioned, XGBoost takes LibSVM format. For training or predicting, XGBoost takes an instance file with the format as below:

train.txt

```
1 101:1.2 102:0.03
0 1:2.1 10001:300 10002:400
0 0:1.3 1:0.3
1 0:0.01 1:0.3
0 0:0.2 1:0.3
```

Each line represent a single instance, and in the first line ‘1’ is the instance label, ‘101’ and ‘102’ are feature indices, ‘1.2’ and ‘0.03’ are feature values. In the binary classification case, ‘1’ is used to indicate positive samples, and ‘0’ is used to indicate negative samples. We also support probability values in [0,1] as label, to indicate the probability of the instance being positive.

4.5.2 Group Input Format

As XGBoost supports accomplishing [ranking task](#), we support the group input format. In ranking task, instances are categorized into different groups in real world scenarios, for example, in the learning to rank web pages scenario, the web page instances are grouped by their queries. Except the instance file mentioned in the group input format, XGBoost need an file indicating the group information. For example, if the instance file is the “train.txt” shown above, and the group file is as below:

train.txt.group

```
2
3
```

This means that, the data set contains 5 instances, and the first two instances are in a group and the other three are in another group. The numbers in the group file are actually indicating the number of instances in each group in the instance file in order. While 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 and decides whether to read the data as group input format.

4.5.3 Instance Weight File

XGBoost supports providing each instance an weight to differentiate the importance of instances. For example, if we provide an instance weight file for the “train.txt” file in the example as below:

train.txt.weight


```
1
0.5
0.5
1
0.5
```

It means that XGBoost will emphasize more on the first and fourth instance that is to say positive instances while training. The configuration is similar to configuring the group information. If the instance file name is “xxx”, XGBoost will check whether there is a file named “xxx.weight” in the same directory and if there is, will use the weights while training models. Weights will be included into an “xxx.buffer” file that is created by XGBoost automatically. If you want to update the weights, you need to delete the “xxx.buffer” file prior to launching XGBoost.

4.5.4 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:

```
train.txt.base_margin
-0.4
1.0
3.4
```

XGBoost will take these values as initial margin prediction and boost from that. An important note about `base_margin` is that it should be margin prediction before transformation, so if you are doing logistic loss, you will need to put in value before logistic transformation. If you are using XGBoost predictor, use `pred_margin=1` to output margin values.

4.6 Build XGBoost

- Run `bash build.sh` (you can also type `make`)
- If you have C++11 compiler, it is recommended to type `make cxx11=1`
 - C++11 is not used by default
- If your compiler does not come with OpenMP support, it will fire an warning telling you that the code will compile into single thread mode, and you will get single thread xgboost
- You may get a error: `-lgomp` is not found
 - You can type `make no_omp=1`, this will get you single thread xgboost
 - Alternatively, you can upgrade your compiler to compile multi-thread version
- Windows(VS 2010): see [../windows](#) folder
 - In principle, you put all the cpp files in the Makefile to the project, and build
- OS X with multi-threading support: see *next section*

4.6.1 Build XGBoost in OS X with OpenMP

Here is the complete solution to use OpenMp-enabled compilers to install XGBoost.

1. Obtain gcc with openmp support by `brew install gcc --without-multilib` **or** clang with openmp by `brew install clang-omp`. The clang one is recommended because the first method requires us compiling gcc inside the machine (more than an hour in mine)! (BTW, brew is the de facto standard of apt-get on OS X. So installing [HPC](#) separately is not recommended, but it should work.)

2. if you are planing to use clang-omp - in step 3 and/or 4, change line 9 in `xgboost/src/utils/omp.h` to

```
#include <libiomp/omp.h> /* instead of #include <omp.h> */`
```

to make it work, otherwise you might get this error

```
src/tree/../../utils/omp.h:9:10: error: 'omp.h' file not found...
```

1. Set the Makefile correctly for compiling cpp version xgboost then python version xgboost.

```
export CC = gcc-4.9
export CXX = g++-4.9
```

Or

```
export CC = clang-omp
export CXX = clang-omp++
```

Remember to change header (mentioned in step 2) if using clang-omp.

Then `cd xgboost` then `bash build.sh` to compile XGBoost. And go to wrapper sub-folder to install python version.

1. Set the Makevars file in highest priority for R.

The point is, there are three Makevars : `~/R/Makevars`, `xgboost/R-package/src/Makevars`, and `/usr/local/Cellar/r/3.2.0/R.framework/Resources/etc/Makeconf` (the last one obtained by running `file.path(R.home("etc"), "Makeconf")` in R), and `SHLIB_OPENMP_CXXFLAGS` is not set by default!! After trying, it seems that the first one has highest priority (surprise!).

So, add or change `~/R/Makevars` to the following lines:

```
CC=gcc-4.9
CXX=g++-4.9
SHLIB_OPENMP_CFLAGS = -fopenmp
SHLIB_OPENMP_CXXFLAGS = -fopenmp
SHLIB_OPENMP_FCFLAGS = -fopenmp
SHLIB_OPENMP_FFLAGS = -fopenmp
```

Or

```
CC=clang-omp
CXX=clang-omp++
SHLIB_OPENMP_CFLAGS = -fopenmp
SHLIB_OPENMP_CXXFLAGS = -fopenmp
SHLIB_OPENMP_FCFLAGS = -fopenmp
SHLIB_OPENMP_FFLAGS = -fopenmp
```

Again, remember to change header if using clang-omp.

Then inside R, run

```
install.packages('xgboost/R-package/', repos=NULL, type='source')
```

Or

```
devtools::install_local('xgboost/', subdir = 'R-package') # you may use devtools
```

4.6.2 Build with HDFS and S3 Support

- To build xgboost use with HDFS/S3 support and distributed learnig. It is recommended to build with dmlc, with the following steps

- `git clone https://github.com/dmlc/dmlc-core`
- Follow instruction in `dmlc-core/make/config.mk` to compile `libdmlc.a`
- In root folder of `xgboost`, type `make dmlc=dmlc-core`
- This will allow `xgboost` to directly load data and save model from/to `hdfs` and `s3`
 - Simply replace the filename with prefix `s3://` or `hdfs://`
- This `xgboost` that can be used for distributed learning

4.7 XGBoost Parameters

Before running XGboost, we must set three types of parameters, general parameters, booster parameters and task parameters:

- General parameters relates to which booster we are using to do boosting, commonly tree or linear model
- Booster parameters depends on which booster you have chosen
- Learning Task parameters that decides on the learning scenario, for example, regression tasks may use different parameters with ranking tasks.
- Command line parameters that relates to behavior of CLI version of `xgboost`.

4.7.1 Parameters in R Package

In R-package, you can use `.`(dot) to replace under score in the parameters, for example, you can use `max.depth` as `max_depth`. The underscore parameters are also valid in R.

4.7.2 General Parameters

- `booster` [default=`gbtree`]
 - which booster to use, can be `gbtree` or `gblinear`. `gbtree` uses tree based model while `gblinear` uses linear function.
- `silent` [default=`0`]
 - `0` means printing running messages, `1` means silent mode.
- `nthread` [default to maximum number of threads available if not set]
 - number of parallel threads used to run `xgboost`
- `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

4.7.3 Parameters for Tree Booster

- eta [default=0.3]
 - step size shrinkage used in update to prevents overfitting. After each boosting step, we can directly get the weights of new features. and eta actually shrinks the feature weights to make the boosting process more conservative.
 - range: [0,1]
- gamma [default=0]
 - minimum loss reduction required to make a further partition on a leaf node of the tree. the larger, the more conservative the algorithm will be.
 - range: [0,∞]
- max_depth [default=6]
 - maximum depth of a tree
 - range: [1,∞]
- 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 mode, this simply corresponds to minimum number of instances needed to be in each node. The larger, the more conservative the algorithm will be.
 - range: [0,∞]
- max_delta_step [default=0]
 - Maximum delta step we allow each tree's weight estimation 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 instance. Setting it to 0.5 means that XGBoost randomly collected half of the data instances to grow trees and this will prevent overfitting.
 - range: (0,1]
- colsample_bytree [default=1]
 - subsample ratio of columns when constructing each tree.
 - range: (0,1]
- lambda [default=1]
 - L2 regularization term on weights
- alpha [default=0]
 - L1 regularization term on weights

4.7.4 Parameters for Linear Booster

- `lambda` [default=0]
 - L2 regularization term on weights
- `alpha` [default=0]
 - L1 regularization term on weights
- `lambda_bias`
 - L2 regularization term on bias, default 0 (no L1 reg on bias because it is not important)

4.7.5 Learning Task Parameters

- `objective` [default=reg:linear]
- specify the learning task and the corresponding learning objective, and the objective options are below:
- “reg:linear” –linear regression
- “reg:logistic” –logistic regression
- “binary:logistic” –logistic regression for binary classification, output probability
- “binary:logitraw” –logistic regression for binary classification, output score before logistic transformation
- “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)
- “multi:softmax” –set XGBoost to do multiclass classification using the softmax objective, you also need to set `num_class`(number of classes)
- “multi:softprob” –same as softmax, but output a vector of `ndata * nclass`, which can be further reshaped to `ndata, nclass` matrix. The result contains predicted probability of each data point belonging to each class.
- “rank:pairwise” –set XGBoost to do ranking task by minimizing the pairwise loss
- `base_score` [default=0.5]
 - the initial prediction score of all instances, global bias
- `eval_metric` [default according to objective]
 - evaluation metrics for validation data, a default metric will be assigned according to objective(`rmse` for regression, and error for classification, mean average precision for ranking)
 - User can add multiple evaluation metrics, for python user, 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
 - “logloss”: negative log-likelihood
 - “error”: Binary classification error rate. It is calculated as $\#(\text{wrong cases})/\#(\text{all cases})$. 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.
 - “merror”: Multiclass classification error rate. It is calculated as $\#(\text{wrong cases})/\#(\text{all cases})$.
 - “mlogloss”: Multiclass logloss

- “auc”: Area under the curve for ranking evaluation.
 - “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. training repeatedly
- seed [default=0]
 - random number seed.

4.7.6 Command Line Parameters

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

- use_buffer [default=1]
- whether create binary buffer for text input, this normally will speedup loading when do
- num_round
- the number of round 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 save any model during 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(preliminary)
- model_in [default=NULL]
 - path to input model, needed for test, eval, dump, 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, will output like 0003.model where 0003 is number of rounds to do boosting.
- model_dir [default=models]
 - The output directory of the saved models during training
- fmap
 - feature map, used for dump model

- `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

4.8 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.

4.8.1 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.

4.8.2 Control Overfitting

When you observe high training accuracy, but low tests accuracy, it is likely that you encounter 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 include `max_depth`, `min_child_weight` and `gamma`
- The second way is to add randomness to make training robust to noise
 - This include `subsample`, `colsample_bytree`
 - You can also reduce stepsize `eta`, but needs to remember to increase `num_round` when you do so.

4.8.3 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 ranking order (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
- In such a case, set parameter `max_delta_step` to a finite number (say 1) will help convergence

Developer Guide

5.1 Developer Guide

This page contains guide for developers of xgboost. XGBoost has been developed and used by a group of active community. Everyone is more than welcomed to is a great way to make the project better. The project is maintained by a committee of [committers](#) who will review and merge pull requests from contributors.

5.1.1 Contributing Code

- The C++ code follows Google C++ style
- We follow numpy style to document our python module
- Tools to precheck codestyle
 - clone <https://github.com/dmlc/dmlc-core> into root directory
 - type `make lint` and fix possible errors.

API Reference

6.1 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.

The document in this page is automatically generated by sphinx. The content do not render at github, you can view it at http://xgboost.readthedocs.org/en/latest/python/python_api.html

6.1.1 Core Data Structure

Core XGBoost Library.

class xgboost.DMatrix(*data*, *label=None*, *missing=0.0*, *weight=None*, *silent=False*, *feature_names=None*, *feature_types=None*)

Bases: object

Data Matrix used in XGBoost.

DMatrix is a internal data structure that used by XGBoost which is optimized for both memory efficiency and training speed. You can construct DMatrix from numpy.arrays

feature_names

Get feature names (column labels).

Returns feature_names

Return type list or None

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 float 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.

Parameters

- **fname** (*string*) – 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

set_group (*group*)
Set group size of DMatrix (used for ranking).

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

set_label (*label*)
Set label of dmatrix

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

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

set_weight (*weight*)
Set weight of each instance.

Parameters **weight** (*array like*) – Weight for each data point

slice (*rindex*)
Slice the DMatrix and return a new DMatrix that only contains *rindex*.

Parameters **rindex** (*list*) – List of indices to be selected.

Returns **res** – A new DMatrix containing only selected indices.

Return type *DMatrix*

class `xgboost.Booster` (*params=None, cache=(), model_file=None*)
Bases: `object`

“A Booster of of XGBoost.

Booster is the model of xgboost, that contains low level routines for training, prediction and evaluation.

boost (*dtrain, grad, hess*)
Boost the booster for one iteration, with customized gradient statistics.

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 model into a text file.

Parameters

- **fout** (*string*) – Output file name.
- **fmap** (*string, optional*) – Name of the file containing feature map names.

- **with_stats** (*bool (optional)*) – Controls whether the split statistics are output.

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*)

Returns the dump the model as a list of strings.

get_fscore (*fmap=''*)

Get feature importance of each feature.

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

load_model (*fname*)

Load the model from a file.

Parameters **fname** (*string or a memory buffer*) – Input file name or memory buffer(see also `save_raw`)

predict (*data, output_margin=False, ntree_limit=0, pred_leaf=False*)

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 `bst.copy()` to make copies of model object and then call predict

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 (nsample, ntrees) with each record indicating the predicted leaf index of each sample in each tree. Note that the leaf index of a tree is unique per tree, so you may find leaf 1 in both tree 1 and tree 0.

Returns **prediction**

Return type numpy array

save_model (*fname*)

Save the model to a file.

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

save_raw ()

Save the model to a in memory buffer representation

Returns

Return type a in memory buffer representation of the model

set_param (*params, value=None*)

Set parameters into the Booster.

Parameters

- **params** (*dict/list/str*) – list of key,value paris, dict of key to value or simply str key
- **value** (*optional*) – value of the specified parameter, when params is str key

update (*dtrain, iteration, fobj=None*)

Update for one iteration, with objective function calculated internally.

Parameters

- **dtrain** ([DMatrix](#)) – Training data.
- **iteration** (*int*) – Current iteration number.
- **fobj** (*function*) – Customized objective function.

6.1.2 Learning API

Training Library containing training routines.

`xgboost.train` (*params, dtrain, num_boost_round=10, evals=(), obj=None, feval=None, early_stopping_rounds=None, evals_result=None, verbose_eval=True*)

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)** (*watchlist*) – List of items to be evaluated during training, this allows user to watch performance on the validation set.
- **obj** (*function*) – Customized objective function.
- **feval** (*function*) – Customized evaluation function.
- **early_stopping_rounds** (*int*) – Activates early stopping. Validation error needs to decrease at least every <early_stopping_rounds> round(s) to continue training. Requires at least one item in evals. If there's more than one, will use the last. Returns the model from the last iteration (not the best one). If early stopping occurs, the model will have two additional fields: `bst.best_score` and `bst.best_iteration`.
- **evals_result** (*dict*) – This dictionary stores the evaluation results of all the items in watchlist

- **verbose_eval** (*bool*) – If *verbose_eval* then the evaluation metric on the validation set, if given, is printed at each boosting stage.

Returns booster

Return type a trained booster model

`xgboost.cv(params, dtrain, num_boost_round=10, nfold=3, metrics=(), obj=None, feval=None, fpreproc=None, as_pandas=True, show_progress=None, show_stdv=True, seed=0)`
Cross-validation with given paramaters.

Parameters

- **params** (*dict*) – Booster params.
- **dtrain** (*DMatrix*) – Data to be trained.
- **num_boost_round** (*int*) – Number of boosting iterations.
- **nfold** (*int*) – Number of folds in CV.
- **metrics** (*list of strings*) – Evaluation metrics to be watched in CV.
- **obj** (*function*) – Custom objective function.
- **feval** (*function*) – Custom evaluation function.
- **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`
- **show_progress** (*bool or None, default None*) – Whether to display the progress. If None, progress will be displayed when `np.ndarray` is returned.
- **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()`).

Returns evaluation history

Return type `list(string)`

6.1.3 Scikit-Learn API

Scikit-Learn Wrapper interface for XGBoost.

```
class xgboost.XGBRegressor(max_depth=3, learning_rate=0.1, n_estimators=100, silent=True,
                           objective='reg:linear', nthread=-1, gamma=0, min_child_weight=1,
                           max_delta_step=0, subsample=1, colsample_bytree=1, base_score=0.5,
                           seed=0, missing=None)
```

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

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

max_depth [int] Maximum tree depth for base learners.

learning_rate [float] Boosting learning rate (xgb's "eta")

n_estimators [int] Number of boosted trees to fit.

silent [boolean] Whether to print messages while running boosting.

objective [string] Specify the learning task and the corresponding learning objective.

nthread [int] Number of parallel threads used to run xgboost.

gamma [float] Minimum loss reduction required to make a further partition on a leaf node of the tree.

min_child_weight [int] 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.

base_score: The initial prediction score of all instances, global bias.

seed [int] Random number seed.

missing [float, optional] Value in the data which needs to be present as a missing value. If None, defaults to np.nan.

```
class xgboost.XGBClassifier(max_depth=3, learning_rate=0.1, n_estimators=100, silent=True, ob-
                           jective='binary:logistic', nthread=-1, gamma=0, min_child_weight=1,
                           max_delta_step=0, subsample=1, colsample_bytree=1,
                           base_score=0.5, seed=0, missing=None)
```

Bases: xgboost.sklearn.XGBModel, object

Implementation of the scikit-learn API for XGBoost classification.

Parameters

max_depth [int] Maximum tree depth for base learners.

learning_rate [float] Boosting learning rate (xgb's "eta")

n_estimators [int] Number of boosted trees to fit.

silent [boolean] Whether to print messages while running boosting.

objective [string] Specify the learning task and the corresponding learning objective.

nthread [int] Number of parallel threads used to run xgboost.

gamma [float] Minimum loss reduction required to make a further partition on a leaf node of the tree.

min_child_weight [int] 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.

base_score: The initial prediction score of all instances, global bias.

seed [int] Random number seed.

missing [float, optional] Value in the data which needs to be present as a missing value. If None, defaults to np.nan.

```
fit (X, y, sample_weight=None, eval_set=None, eval_metric=None, early_stopping_rounds=None, ver-
     bose=True)
Fit gradient boosting classifier
```

Parameters

- **X** (*array_like*) – Feature matrix

- **y** (*array_like*) – Labels
- **sample_weight** (*array_like*) – Weight for each instance
- **eval_set** (*list, optional*) – A list of (X, y) pairs to use as a validation set for early-stopping
- **eval_metric** (*str, callable, optional*) – If a str, should be a built-in evaluation metric to use. See doc/parameter.md. 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. This objective is always minimized.
- **early_stopping_rounds** (*int, optional*) – Activates early stopping. Validation error needs to decrease at least every <early_stopping_rounds> round(s) to continue training. Requires at least one item in evals. If there's more than one, will use the last. Returns the model from the last iteration (not the best one). If early stopping occurs, the model will have two additional fields: bst.best_score and bst.best_iteration.
- **verbose** (*bool*) – If *verbose* and an evaluation set is used, writes the evaluation metric measured on the validation set to stderr.

6.1.4 Plotting API

Plotting Library.

```
xgboost.plot_importance(booster, ax=None, height=0.2, xlim=None, title='Feature importance',
                        xlabel='F score', ylabel='Features', grid=True, **kwargs)
```

Plot importance based on fitted trees.

Parameters

- **booster** (*Booster or dict*) – Booster 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.
- **height** (*float, default 0.2*) – Bar height, passed to ax.barh()
- **xlim** (*tuple, default None*) – Tuple passed to axes.xlim()
- **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.
- **kwargs** – Other keywords passed to ax.barh()

Returns ax

Return type matplotlib Axes

```
xgboost.plot_tree(booster, num_trees=0, rankdir='UT', ax=None, **kwargs)
```

Plot specified tree.

Parameters

- **booster** (*Booster*) – Booster instance
- **num_trees** (*int, default 0*) – Specify the ordinal number of target tree
- **rankdir** (*str, default "UT"*) – 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, num_trees=0, rankdir='UT', yes_color='#0000FF', no_color='#FF0000', **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`) – Booster instance
- **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 condigion.
- **no_color** (*str, default '#FF0000'*) – Edge color when doesn't meet the node condigion.
- **kwargs** – Other keywords passed to graphviz `graph_attr`

Returns **ax**

Return type `matplotlib Axes`

X

`xgboost.core`, [39](#)
`xgboost.plotting`, [46](#)
`xgboost.sklearn`, [44](#)
`xgboost.training`, [43](#)

B

`boost()` (xgboost.Booster method), 19, 41
`Booster` (class in xgboost), 19, 41

C

`copy()` (xgboost.Booster method), 19, 41
`cv()` (in module xgboost), 21, 44

D

`DMatrix` (class in xgboost), 17, 39
`dump_model()` (xgboost.Booster method), 19, 41

E

`eval()` (xgboost.Booster method), 19, 42
`eval_set()` (xgboost.Booster method), 19, 42

F

`feature_names` (xgboost.DMatrix attribute), 17, 39
`feature_types` (xgboost.DMatrix attribute), 17, 39
`fit()` (xgboost.XGBClassifier method), 23, 45

G

`get_base_margin()` (xgboost.DMatrix method), 17, 39
`get_dump()` (xgboost.Booster method), 19, 42
`get_float_info()` (xgboost.DMatrix method), 17, 39
`get_fscore()` (xgboost.Booster method), 20, 42
`get_label()` (xgboost.DMatrix method), 17, 40
`get_uint_info()` (xgboost.DMatrix method), 17, 40
`get_weight()` (xgboost.DMatrix method), 17, 40

L

`load_model()` (xgboost.Booster method), 20, 42

N

`num_col()` (xgboost.DMatrix method), 17, 40
`num_row()` (xgboost.DMatrix method), 18, 40

P

`plot_importance()` (in module xgboost), 23, 46

`plot_tree()` (in module xgboost), 24, 46
`predict()` (xgboost.Booster method), 20, 42

S

`save_binary()` (xgboost.DMatrix method), 18, 40
`save_model()` (xgboost.Booster method), 20, 43
`save_raw()` (xgboost.Booster method), 20, 43
`set_base_margin()` (xgboost.DMatrix method), 18, 40
`set_float_info()` (xgboost.DMatrix method), 18, 40
`set_group()` (xgboost.DMatrix method), 18, 41
`set_label()` (xgboost.DMatrix method), 18, 41
`set_param()` (xgboost.Booster method), 20, 43
`set_uint_info()` (xgboost.DMatrix method), 18, 41
`set_weight()` (xgboost.DMatrix method), 18, 41
`slice()` (xgboost.DMatrix method), 18, 41

T

`to_graphviz()` (in module xgboost), 24, 47
`train()` (in module xgboost), 21, 43

U

`update()` (xgboost.Booster method), 20, 43

X

`XGBClassifier` (class in xgboost), 22, 45
`xgboost.core` (module), 17, 39
`xgboost.plotting` (module), 23, 46
`xgboost.sklearn` (module), 22, 44
`xgboost.training` (module), 21, 43
`XGBRegressor` (class in xgboost), 22, 44