**XGBoost R Tutorial**

**Introduction**

**Xgboost** is short for e**X**treme **G**radient **Boost**ing package.

The purpose of this Vignette is to show you how to use **Xgboost** to build a model and make predictions.

It is an efficient and scalable implementation of gradient boosting framework by @friedman2000additive and @friedman2001greedy. Two solvers are included:

* *linear* model ;
* *tree learning* algorithm.

It supports various objective functions, including *regression*, *classification* and*ranking*. The package is made to be extendible, so that users are also allowed to define their own objective functions easily.

It has been [used](https://github.com/dmlc/xgboost) to win several [Kaggle](http://www.kaggle.com/) competitions.

It has several features:

* Speed: it can automatically do parallel computation on *Windows* and *Linux*, with *OpenMP*. It is generally over 10 times faster than the classical gbm.
* Input Type: it takes several types of input data:
  + *Dense* Matrix: *R*'s *dense* matrix, i.e. matrix ;
  + *Sparse* Matrix: *R*'s *sparse* matrix, i.e. Matrix::dgCMatrix ;
  + Data File: local data files ;
  + xgb.DMatrix: its own class (recommended).
* Sparsity: it accepts *sparse* input for both *tree booster* and *linear booster*, and is optimized for *sparse* input ;
* Customization: it supports customized objective functions and evaluation functions.

**Installation**

**Github version**

For weekly updated version (highly recommended), install from *Github*:

install.packages("drat", repos="https://cran.rstudio.com")

drat:::addRepo("dmlc")

install.packages("xgboost", repos="http://dmlc.ml/drat/", type = "source")

*Windows* user will need to install [Rtools](https://cran.r-project.org/bin/windows/Rtools/" \t "_blank) first.

**CRAN version**

The version 0.4-2 is on CRAN, and you can install it by:

install.packages("xgboost")

Formerly available versions can be obtained from the CRAN [archive](https://cran.r-project.org/src/contrib/Archive/xgboost)

**Learning**

For the purpose of this tutorial we will load **XGBoost** package.

**require**(xgboost)

**Dataset presentation**

In this example, we are aiming to predict whether a mushroom can be eaten or not (like in many tutorials, example data are the the same as you will use on in your every day life :-).

Mushroom data is cited from UCI Machine Learning Repository. @Bache+Lichman:2013.

**Dataset loading**

We will load the agaricus datasets embedded with the package and will link them to variables.

The datasets are already split in:

* train: will be used to build the model ;
* test: will be used to assess the quality of our model.

Why *split* the dataset in two parts?

In the first part we will build our model. In the second part we will want to test it and assess its quality. Without dividing the dataset we would test the model on the data which the algorithm have already seen.

data(agaricus.train, package='xgboost')

data(agaricus.test, package='xgboost')

train <- agaricus.train

test <- agaricus.test

In the real world, it would be up to you to make this division between train andtest data. The way to do it is out of the purpose of this article, however caretpackage may [help](http://topepo.github.io/caret/data-splitting.html).

Each variable is a list containing two things, label and data:

str(train)

## List of 2

## $ data :Formal class 'dgCMatrix' [package "Matrix"] with 6 slots

## .. ..@ i : int [1:143286] 2 6 8 11 18 20 21 24 28 32 ...

## .. ..@ p : int [1:127] 0 369 372 3306 5845 6489 6513 8380 8384 10991 ...

## .. ..@ Dim : int [1:2] 6513 126

## .. ..@ Dimnames:List of 2

## .. .. ..$ : NULL

## .. .. ..$ : chr [1:126] "cap-shape=bell" "cap-shape=conical" "cap-shape=convex" "cap-shape=flat" ...

## .. ..@ x : num [1:143286] 1 1 1 1 1 1 1 1 1 1 ...

## .. ..@ factors : list()

## $ label: num [1:6513] 1 0 0 1 0 0 0 1 0 0 ...

label is the outcome of our dataset meaning it is the binary *classification* we will try to predict.

Let's discover the dimensionality of our datasets.

dim(train$data)

## [1] 6513 126

dim(test$data)

## [1] 1611 126

This dataset is very small to not make the **R** package too heavy, however **XGBoost**is built to manage huge dataset very efficiently.

As seen below, the data are stored in a dgCMatrix which is a *sparse* matrix andlabel vector is a numeric vector ({0,1}):

class(train$data)[1]

## [1] "dgCMatrix"

class(train$label)

## [1] "numeric"

**Basic Training using XGBoost**

This step is the most critical part of the process for the quality of our model.

**Basic training**

We are using the train data. As explained above, both data and label are stored in a list.

In a *sparse* matrix, cells containing 0 are not stored in memory. Therefore, in a dataset mainly made of 0, memory size is reduced. It is very usual to have such dataset.

We will train decision tree model using the following parameters:

* objective = "binary:logistic": we will train a binary classification model ;
* max\_depth = 2: the trees won't be deep, because our case is very simple ;
* nthread = 2: the number of cpu threads we are going to use;
* nrounds = 2: there will be two passes on the data, the second one will enhance the model by further reducing the difference between ground truth and prediction.

bstSparse <- xgboost(data = train$data, label = train$label, max\_depth = 2, eta = 1, nthread = 2, nrounds = 2, objective = "binary:logistic")

## [1] train-error:0.046522

## [2] train-error:0.022263

More complex the relationship between your features and your label is, more passes you need.

**Parameter variations**

**Dense matrix**

Alternatively, you can put your dataset in a *dense* matrix, i.e. a basic **R** matrix.

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

## [1] train-error:0.046522

## [2] 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. It will be useful for the most advanced features we will discover later.

dtrain <- xgb.DMatrix(data = train$data, label = train$label)

bstDMatrix <- xgboost(data = dtrain, max\_depth = 2, eta = 1, nthread = 2, nrounds = 2, objective = "binary:logistic")

## [1] train-error:0.046522

## [2] train-error:0.022263

**Verbose option**

**XGBoost** has several features to help you to view how 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 technics).

*# verbose = 0, no message*

bst <- xgboost(data = dtrain, max\_depth = 2, eta = 1, nthread = 2, nrounds = 2, objective = "binary:logistic", verbose = 0)

*# verbose = 1, print evaluation metric*

bst <- xgboost(data = dtrain, max\_depth = 2, eta = 1, nthread = 2, nrounds = 2, objective = "binary:logistic", verbose = 1)

## [1] train-error:0.046522

## [2] train-error:0.022263

*# verbose = 2, also print information about tree*

bst <- xgboost(data = dtrain, max\_depth = 2, eta = 1, nthread = 2, nrounds = 2, objective = "binary:logistic", verbose = 2)

## [11:17:30] amalgamation/../src/tree/updater\_prune.cc:74: tree pruning end, 1 roots, 6 extra nodes, 0 pruned nodes, max\_depth=2

## [1] train-error:0.046522

## [11:17:30] amalgamation/../src/tree/updater\_prune.cc:74: tree pruning end, 1 roots, 4 extra nodes, 0 pruned nodes, max\_depth=2

## [2] train-error:0.022263

**Basic prediction using XGBoost**

**Perform the prediction**

The purpose of the model we have built is to classify new data. As explained before, we will use the test dataset for this step.

pred <- predict(bst, test$data)

*# size of the prediction vector*

print(length(pred))

## [1] 1611

*# limit display of predictions to the first 10*

print(head(pred))

## [1] 0.28583017 0.92392391 0.28583017 0.28583017 0.05169873 0.92392391

These numbers doesn't look like *binary classification* {0,1}. We need to perform a simple transformation before being able to use these results.

**Transform the regression in a binary classification**

The only thing that **XGBoost** does is a *regression*. **XGBoost** is using label vector to build its *regression* model.

How can we use a *regression* model to perform a binary classification?

If we think about the meaning of a regression applied to our data, the numbers we get are probabilities that a datum will be classified as 1. Therefore, we will set the rule that if this probability for a specific datum is > 0.5 then the observation is classified as 1 (or 0 otherwise).

prediction <- as.numeric(pred > 0.5)

print(head(prediction))

## [1] 0 1 0 0 0 1

**Measuring model performance**

To measure the model performance, we will compute a simple metric, the *average error*.

err <- mean(as.numeric(pred > 0.5) != test$label)

print(paste("test-error=", err))

## [1] "test-error= 0.0217256362507759"

Note that the algorithm has not seen the test data during the model construction.

Steps explanation:

1. as.numeric(pred > 0.5) applies our rule that when the probability (<=> regression <=> prediction) is > 0.5 the observation is classified as 1 and 0otherwise ;
2. probabilityVectorPreviouslyComputed != test$label computes the vector of error between true data and computed probabilities ;
3. mean(vectorOfErrors) computes the *average error* itself.

The most important thing to remember is that **to do a classification, you just do a regression to the** label **and then apply a threshold**.

*Multiclass* classification works in a similar way.

This metric is **0.02** and is pretty low: our yummly mushroom model works well!

**Advanced features**

Most of the features below have been implemented to help you to improve your model by offering a better understanding of its content.

**Dataset preparation**

For the following advanced features, we need to put data in xgb.DMatrix as explained above.

dtrain <- xgb.DMatrix(data = train$data, label=train$label)

dtest <- xgb.DMatrix(data = test$data, label=test$label)

**Measure learning progress with xgb.train**

Both xgboost (simple) and xgb.train (advanced) functions train models.

One of the special feature 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 an overfitting. You can see this feature as a cousin of 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 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 below it was after building the model, and now it is during the construction that we measure errors.

For the purpose of this example, we use watchlist parameter. It is a list ofxgb.DMatrix, each of them tagged with a name.

watchlist <- list(train=dtrain, test=dtest)

bst <- xgb.train(data=dtrain, max\_depth=2, eta=1, nthread = 2, nrounds=2, watchlist=watchlist, objective = "binary:logistic")

## [1] train-error:0.046522 test-error:0.042831

## [2] train-error:0.022263 test-error:0.021726

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

Both training and test error related metrics are very similar, and in some way, it makes sense: what we have learned from the training dataset matches the observations from the test dataset.

If with your own dataset you have not 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](http://topepo.github.io/caret/data-splitting.html).

For a better understanding of the learning progression, you may want to have some specific metric or even use multiple evaluation metrics.

bst <- xgb.train(data=dtrain, max\_depth=2, eta=1, nthread = 2, nrounds=2, watchlist=watchlist, eval\_metric = "error", eval\_metric = "logloss", objective = "binary:logistic")

## [1] train-error:0.046522 train-logloss:0.233376 test-error:0.042831 test-logloss:0.226686

## [2] train-error:0.022263 train-logloss:0.136658 test-error:0.021726 test-logloss:0.137874

eval\_metric allows us to monitor two new metrics for each round, logloss anderror.

**Linear boosting**

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

bst <- xgb.train(data=dtrain, booster = "gblinear", max\_depth=2, nthread = 2, nrounds=2, watchlist=watchlist, eval\_metric = "error", eval\_metric = "logloss", objective = "binary:logistic")

## [1] train-error:0.023491 train-logloss:0.182193 test-error:0.018001 test-logloss:0.180818

## [2] train-error:0.003992 train-logloss:0.068945 test-error:0.003724 test-logloss:0.066951

In this specific case, *linear boosting* gets sligtly better performance metrics than decision trees based algorithm.

In simple cases, it will happen because there is nothing better than a linear algorithm to catch a linear link. However, decision trees are much better to catch a non linear link between predictors and outcome. Because there is no silver bullet, we advise you to check both algorithms with your own datasets to have an idea of what to use.

**Manipulating xgb.DMatrix**

**Save / Load**

Like saving models, xgb.DMatrix object (which groups both dataset and outcome) can also be saved using xgb.DMatrix.save function.

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

## [1] TRUE

*# to load it in, simply call xgb.DMatrix*

dtrain2 <- xgb.DMatrix("dtrain.buffer")

## [11:17:30] 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")

## [1] train-error:0.046522 test-error:0.042831

## [2] train-error:0.022263 test-error:0.021726

**Information extraction**

Information can be extracted from xgb.DMatrix using getinfo function. Hereafter we will extract label data.

label = getinfo(dtest, "label")

pred <- predict(bst, dtest)

err <- as.numeric(sum(as.integer(pred > 0.5) != label))/length(label)

print(paste("test-error=", err))

## [1] "test-error= 0.0217256362507759"

**View feature importance/influence from the learnt model**

Feature importance is similar to R gbm package's relative influence (rel.inf).

importance\_matrix <- xgb.importance(model = bst)

print(importance\_matrix)

xgb.plot.importance(importance\_matrix = importance\_matrix)

**View the trees from a model**

You can dump the tree you learned using xgb.dump into a text file.

xgb.dump(bst, with\_stats = T)

## [1] "booster[0]"

## [2] "0:[f28<-9.53674e-07] yes=1,no=2,missing=1,gain=4000.53,cover=1628.25"

## [3] "1:[f55<-9.53674e-07] 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<-9.53674e-07] 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<-9.53674e-07] yes=1,no=2,missing=1,gain=832.545,cover=788.852"

## [11] "1:[f28<-9.53674e-07] yes=3,no=4,missing=3,gain=569.725,cover=768.39"

## [12] "3:leaf=0.784718,cover=458.937"

## [13] "4:leaf=-0.96853,cover=309.453"

## [14] "2:leaf=-6.23624,cover=20.4624"

You can plot the trees from your model using xgb.plot.tree

xgb.plot.tree(model = bst)

if you provide a path to fname parameter you can save the trees to your hard drive.

**Save and load models**

Maybe your dataset is big, and it takes time to train a model on it? May be you are not a big fan of losing time in redoing the same task again and again? In these very rare cases, you will want to save your model and load it when required.

Hopefully 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 caretpackage, 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)

*# pred2 should be identical to pred*

print(paste("sum(abs(pred3-pred))=", sum(abs(pred2-pred))))

## [1] "sum(abs(pred3-pred))= 0"

Again 0? It seems that XGBoost works pretty well!

**References**