XGBoost

eXtreme Gradient Boosting

Tong He

Overview

- Introduction
- Basic Walkthrough
- Real World Application
- Model Specification
- Parameter Introduction
- Advanced Features
- Kaggle Winning Solution

Introduction

Introduction

Nowadays we have plenty of machine learning models. Those most well-knowns are

- Linear/Logistic Regression
- k-Nearest Neighbours
- Support Vector Machines
- Tree-based Model
    - Decision Tree
    - Random Forest
    - Gradient Boosting Machine
- Neural Networks

Introduction

XGBoost is short for eXtreme Gradient Boosting. It is

- An open-sourced tool
  - Computation in C++
  - R/python/Julia interface provided
- A variant of the gradient boosting machine
  - Tree-based model
- The winning model for several kaggle competitions

Introduction

XGBoost is currently host on github.

- The primary author of the model and the c++ implementation is Tianqi Chen.
- The author for the R-package is Tong He.

Introduction

XGBoost is widely used for kaggle competitions. The reason to choose XGBoost includes

- Easy to use
  - Easy to install.
  - Highly developed R/python interface for users.
- Efficiency
  - Automatic parallel computation on a single machine.
  - Can be run on a cluster.
- Accuracy
  - Good result for most data sets.
- Feasibility
  - Customized objective and evaluation
  - Tunable parameters

Basic Walkthrough

We introduce the R package for XGBoost. To install, please run

```
devtools::install_github('dmlc/xgboost',subdir='R-package')
```

This command downloads the package from github and compile it automatically on your machine. Therefore we need RTools installed on Windows.

Basic Walkthrough

XGBoost provides a data set to demonstrate its usages.

```
require(xgboost)
```

```
## Loading required package: xgboost
```

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

This data set includes the information for some kinds of mushrooms. The features are binary, indicate whether the mushroom has this characteristic. The target variable is whether they are poisonous.

Basic Walkthrough

Let's investigate the data first.

```
str(train$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=f
##   ..@ x       : num [1:143286] 1 1 1 1 1 1 1 1 1 1 ...
##   ..@ factors : list()
```

We can see that the data is a `dgCMatrix` class object. This is a sparse matrix class from the package `Matrix`. Sparse matrix is more memory efficient for some specific data.

Basic Walkthrough

To use XGBoost to classify poisonous mushrooms, the minimum information we need to provide is:

1. Input features
    - XGBoost allows dense and sparse matrix as the input.
2. Target variable
    - A numeric vector. Use integers starting from 0 for classification, or real values for regression
3. Objective
    - For regression use 'reg:linear'
    - For binary classification use 'binary:logistic'
4. Number of iteration
    - The number of trees added to the model

Basic Walkthrough

To run XGBoost, we can use the following command:

```
bst = xgboost(data = train$data, label = train$label,
              nround = 2, objective = "binary:logistic")
```

```
## [0]  train-error:0.000614
## [1]  train-error:0.001228
```

The output is the classification error on the training data set.

Basic Walkthrough

Sometimes we might want to measure the classification by 'Area Under the Curve':

```
bst = xgboost(data = train$data, label = train$label, nround = 2,
              objective = "binary:logistic", eval_metric = "auc")
```

```
## [0]  train-auc:0.999238
## [1]  train-auc:0.999238
```

Basic Walkthrough

To predict, you can simply write

```
pred = predict(bst, test$data)
head(pred)
```

```
## [1] 0.2582498 0.7433221 0.2582498 0.2582498 0.2576509 0.2750908
```

Basic Walkthrough

Cross validation is an important method to measure the model's predictive power, as well as the degree of overfitting. XGBoost provides a convenient function to do cross validation in a line of code.

```
cv.res = xgb.cv(data = train$data, nfold = 5, label = train$label, nround = 2,
                objective = "binary:logistic", eval_metric = "auc")
```

```
## [0]  train-auc:0.998780+0.000590 test-auc:0.998547+0.000854
## [1]  train-auc:0.999114+0.000728 test-auc:0.998736+0.001072
```

Notice the difference of the arguments between `xgb.cv` and `xgboost` is the additional `nfold` parameter. To perform cross validation on a certain set of parameters, we just need to copy them to the `xgb.cv` function and add the number of folds.

Basic Walkthrough

`xgb.cv` returns a `data.table` object containing the cross validation results. This is helpful for choosing the correct number of iterations.

```
cv.res
```

```
##    train.auc.mean train.auc.std test.auc.mean test.auc.std
## 1:       0.998780      0.000590      0.998547     0.000854
## 2:       0.999114      0.000728      0.998736     0.001072
```

Real World Experiment

Higgs Boson Competition

The debut of XGBoost was in the higgs boson competition.

Tianqi introduced the tool along with a benchmark code which achieved the top 10% at the beginning of the competition.

To the end of the competition, it was already the mostly used tool in that competition.

Higgs Boson Competition

XGBoost offers the script on github.

To run the script, prepare a `data` directory and download the competition data into this directory.

Higgs Boson Competition

Firstly we prepare the environment

```
require(xgboost)
require(methods)
testsize = 550000
```

Higgs Boson Competition

Then we can read in the data

```
dtrain = read.csv("data/training.csv", header=TRUE)
dtrain[33] = dtrain[33] == "s"
label = as.numeric(dtrain[[33]])
data = as.matrix(dtrain[2:31])
weight = as.numeric(dtrain[[32]]) * testsize / length(label)
```

Higgs Boson Competition

The data contains missing values and they are marked as `-999`. We can construct an `xgb.DMatrix` object containing the information of `weight` and `missing`.

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

Higgs Boson Competition

Next step is to set the basic parameters

```
param = list("objective" = "binary:logitraw",
             "scale_pos_weight" = sumwneg / sumwpos,
             "bst:eta" = 0.1,
             "bst:max_depth" = 6,
             "eval_metric" = "auc",
             "eval_metric" = "ams@0.15",
             "silent" = 1,
             "nthread" = 16)
```

Higgs Boson Competition

We then start the training step

```
bst = xgboost(params = param, data = xgmat, nround = 120)
```

Higgs Boson Competition

Then we read in the test data

```
dtest = read.csv("data/test.csv", header=TRUE)
data = as.matrix(dtest[2:31])
xgmat = xgb.DMatrix(data, missing = -999.0)
```

Higgs Boson Competition

We now can make prediction on the test data set.

```
ypred = predict(bst, xgmat)
```

Higgs Boson Competition

Finally we output the prediction according to the required format.

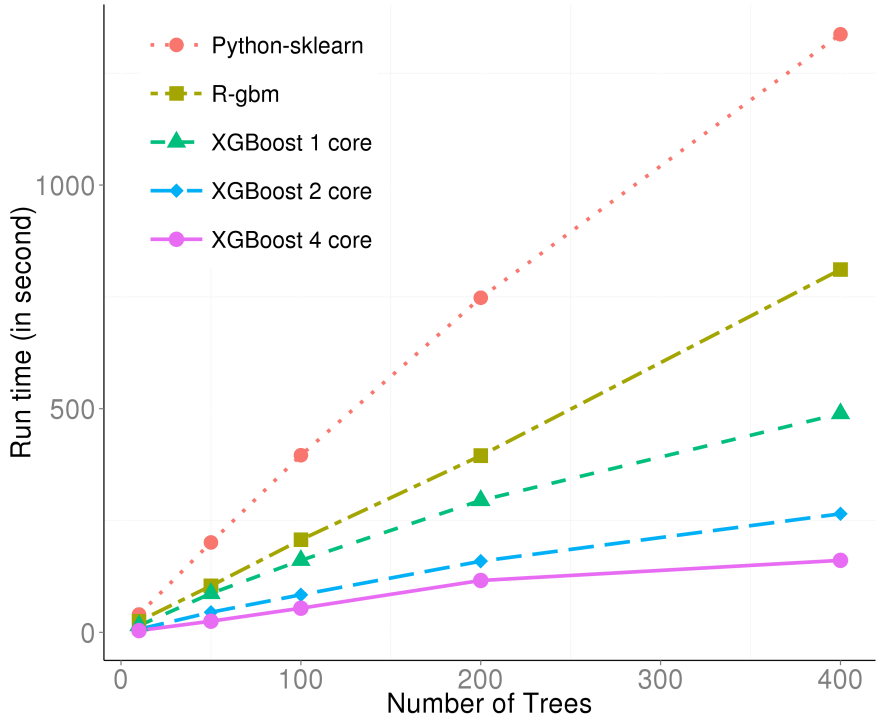
```
rorder = rank(ypred, ties.method="first")
threshold = 0.15
ntop = length(rorder) - as.integer(threshold*length(rorder))
plabel = ifelse(rorder > ntop, "s", "b")
outdata = list("EventId" = idx,
               "RankOrder" = rorder,
               "Class" = plabel)
write.csv(outdata, file = "submission.csv", quote=FALSE, row.names=FALSE)
```

Please submit the result to see your performance :)

Higgs Boson Competition

Besides the good performace, the efficiency is also a highlight of XGBoost.

The following plot shows the running time result on the Higgs boson data set.



Higgs Boson Competition

After some feature engineering and parameter tuning, one can achieve around 25th with a single model on the leaderboard. This is an article written by a former-physist introducing his solution with a single XGboost model.

On our post-competition attempts, we achieved 11th on the leaderboard with a single XGBoost model.

Model Specification

Training Objective

To understand other parameters, one need to have a basic understanding of the model behind.

Suppose we have K trees, the model is

$$\sum_{k=1}^{K} f_k$$

where each f_k is the prediction from a decision tree. The model is a collection of decision trees.

把每棵樹的預測結果加總

Training Objective

Having all the decision trees, we make prediction by

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

where x_i is the feature vector for the i-th data point.

Similarly, the prediction at the t-th step can be defined as

做到第t棵樹

$$\hat{y}_i^{(t)} = \sum_{k=1}^{t} f_k(x_i)$$

Training Objective

To train the model, we need to optimize a loss function.

Typically, we use

針對不同的y有不同的loss function

- Rooted Mean Squared Error for regression
  - $L = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$
- LogLoss for binary classification
  - $L = -\frac{1}{N}\sum_{i=1}^{N}(y_i \log(p_i) + (1 - y_i)\log(1 - p_i))$
- mlogloss for multi-classification
  - $L = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{i,j}\log(p_{i,j})$

Training Objective

Regularization is another important part of the model. A good regularization term controls the complexity of the model which prevents overfitting.

Define

omega項是regularization term

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

where T is the number of leaves, and w_j^2 is the score on the j-th leaf.

T 代表葉子數目，如果葉子太多代表模型太過複雜，所以要 T 不能太大否則將會違背極小化目標式的目的

Training Objective

Put loss function and regularization together, we have the objective of the model:

$$Obj = L + \Omega$$

where loss function controls the predictive power, and regularization controls the simplicity.

目標式會等於loss function+omega
目標就是要極小化目標式

Training Objective

In XGBoost, we use gradient descent to optimize the objective.

Given an objective $Obj(y, \hat{y})$ to optimize, gradient descent is an iterative technique which calculate

$$\partial_{\hat{y}} Obj(y, \hat{y})$$ 目標式對y hat微分

at each iteration. Then we improve $\hat{y}$ along the direction of the gradient to minimize the objective.

Training Objective

Recall the definition of objective $Obj = L + \Omega$. For a iterative algorithm we can re-define the objective function as

$$Obj^{(t)} = \sum_{i=1}^{N} L(y_i, \hat{y}_i^{(t)}) + \sum_{i=1}^{t} \Omega(f_i) = \sum_{i=1}^{N} L(y_i, \hat{y}_i{}^{(t-1)} + f_t(x_i)) + \sum_{i=1}^{t} \Omega(f_i)$$

To optimize it by gradient descent, we need to calculate the gradient. The performance can also be improved by considering both the first and the second order gradient.

$$\partial_{\hat{y}_i{}^{(t)}} Obj^{(t)}$$

$$\partial^2_{\hat{y}_i{}^{(t)}} Obj^{(t)}$$

Training Objective

Since we don't have derivative for every objective function, we calculate the second order taylor approximation of it

$$Obj^{(t)} \simeq \sum_{i=1}^{N} [L(y_i, \hat{y}^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i {f_t}^2(x_i)] + \sum_{i=1}^{t} \Omega(f_i)$$

where

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

Training Objective

Remove the constant terms, we get

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

This is the objective at the t-th step. Our goal is to find a f_t to optimize it.

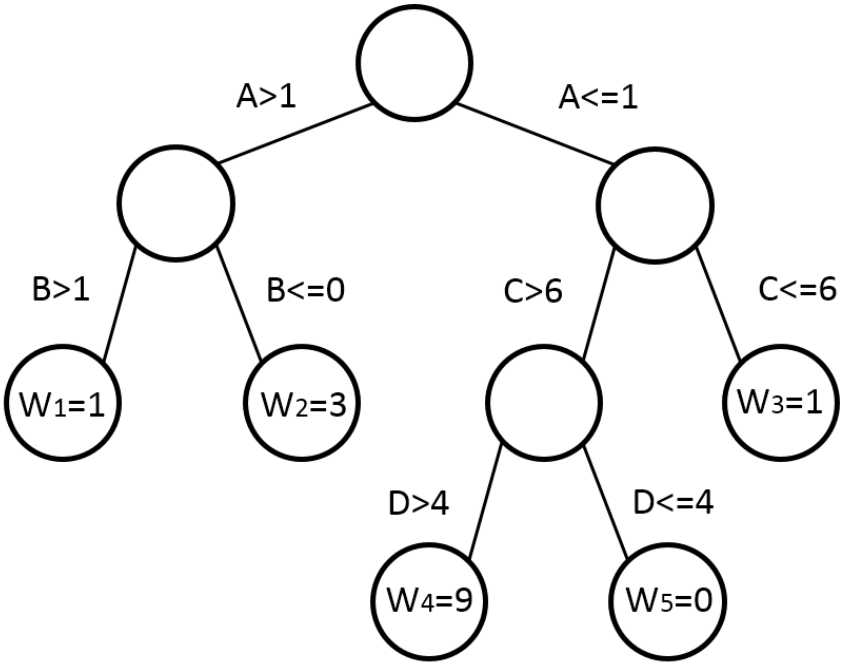
Tree Building Algorithm

The tree structures in XGBoost leads to the core problem:

how can we find a tree that improves the prediction along the gradient?

Tree Building Algorithm

Every decision tree looks like this



Each data point flows to one of the leaves following the direction on each node.

Tree Building Algorithm

The core concepts are:

- Internal Nodes
  - Each internal node split the flow of data points by one of the features.
  - The condition on the edge specifies what data can flow through.
- Leaves
  - Data points reach to a leaf will be assigned a weight.
  - The weight is the prediction.

Tree Building Algorithm

Two key questions for building a decision tree are

1. How to find a good structure?
2. How to assign prediction score?

We want to solve these two problems with the idea of gradient descent.

Tree Building Algorithm

Let us assume that we already have the solution to question 1.

We can mathematically define a tree as

$$f_t(x) = w_{q(x)}$$

where $q(x)$ is a "directing" function which assign every data point to the $q(x)$-th leaf.

This definition describes the prediction process on a tree as

- Assign the data point x to a leaf by q
- Assign the corresponding score $w_{q(x)}$ on the $q(x)$-th leaf to the data point.

Tree Building Algorithm

Define the index set

$$I_j = \{i | q(x_i) = j\}$$

This set contains the indices of data points that are assigned to the j-th leaf.

Tree Building Algorithm

Then we rewrite the objective as

$$Obj^{(t)} = \sum_{i=1}^{n} [g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^{T} w_j^2$$

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

Since all the data points on the same leaf share the same prediction, this form sums the prediction by leaves.

Tree Building Algorithm

It is a quadratic problem of w_j, so it is easy to find the best w_j to optimize Obj.

$$w_j^* = -\frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda}$$

The corresponding value of Obj is

$$Obj^{(t)} = -\frac{1}{2} \sum_{j=1}^{T} \frac{(\sum_{i \in I_j} g_i)^2}{\sum_{i \in I_j} h_i + \lambda} + \gamma T$$

Tree Building Algorithm

The leaf score

$$w_j = -\frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda}$$

relates to

- The first and second order of the loss function g and h
- The regularization parameter λ

Tree Building Algorithm

Now we come back to the first question: How to find a good structure?

We can further split it into two sub-questions:

1. How to choose the feature to split?
2. When to stop the split?

Tree Building Algorithm

In each split, we want to greedily find the best splitting point that can optimize the objective.

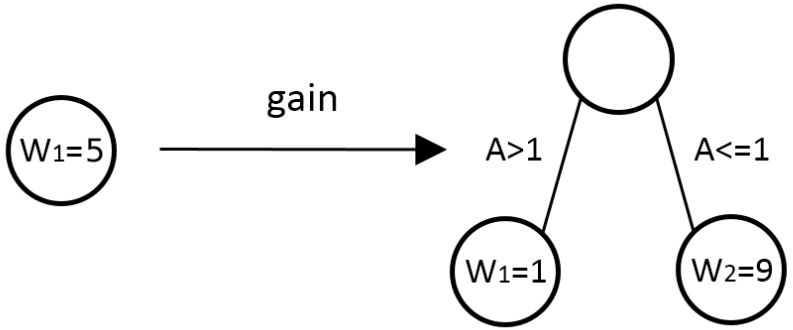
For each feature

1. Sort the numbers
2. Scan the best splitting point.
3. Choose the best feature.

Tree Building Algorithm

Now we give a definition to "the best split" by the objective.

Everytime we do a split, we are changing a leaf into a internal node.



Tree Building Algorithm

Let

- I be the set of indices of data points assigned to this node
- I_L and I_R be the sets of indices of data points assigned to two new leaves.

Recall the best value of objective on the j-th leaf is

$$Obj^{(t)} = -\frac{1}{2}\frac{(\sum_{i\in I_j} g_i)^2}{\sum_{i\in I_j} h_i + \lambda} + \gamma$$

Tree Building Algorithm

The gain of the split is

$$gain = \frac{1}{2}\left[\frac{(\sum_{i\in I_L} g_i)^2}{\sum_{i\in I_L} h_i + \lambda} + \frac{(\sum_{i\in I_R} g_i)^2}{\sum_{i\in I_R} h_i + \lambda} - \frac{(\sum_{i\in I} g_i)^2}{\sum_{i\in I} h_i + \lambda}\right] - \gamma$$

Tree Building Algorithm

To build a tree, we find the best splitting point recursively until we reach to the maximum depth.

Then we prune out the nodes with a negative gain in a bottom-up order.

Tree Building Algorithm

XGBoost can handle missing values in the data.

For each node, we guide all the data points with a missing value

- to the left subnode, and calculate the maximum gain
- to the right subnode, and calculate the maximum gain
- Choose the direction with a larger gain

Finally every node has a "default direction" for missing values.

Tree Building Algorithm

To sum up, the outline of the algorithm is

- Iterate for `nround` times
  - Grow the tree to the maximun depth
    - Find the best splitting point
    - Assign weight to the two new leaves
  - Prune the tree to delete nodes with negative gain

Parameters

Parameter Introduction

XGBoost has plenty of parameters. We can group them into

1. General parameters
    - Number of threads
2. Booster parameters
    - Stepsize
    - Regularization
3. Task parameters
    - Objective
    - Evaluation metric

Parameter Introduction

After the introduction of the model, we can understand the parameters provided in XGBoost.

To check the parameter list, one can look into

- The documentation of `xgb.train`.
- The documentation in the repository.

Parameter Introduction

General parameters:

- `nthread`
  - Number of parallel threads.
- `booster`
  - `gbtree`: tree-based model.
  - `gblinear`: linear function.

Parameter Introduction

Parameter for Tree Booster

- `eta`
  - Step size shrinkage used in update to prevents overfitting.
  - Range in [0,1], default 0.3
- `gamma`
  - Minimum loss reduction required to make a split.
  - Range [0, ∞], default 0

Parameter Introduction

Parameter for Tree Booster

- `max_depth`
  - Maximum depth of a tree.
  - Range [1, ∞], default 6
- `min_child_weight`
  - Minimum sum of instance weight needed in a child.
  - Range [0, ∞], default 1
- `max_delta_step`
  - Maximum delta step we allow each tree's weight estimation to be.
  - Range [0, ∞], default 0

Parameter Introduction

Parameter for Tree Booster

- `subsample`
  - Subsample ratio of the training instance.
  - Range (0, 1], default 1
- `colsample_bytree`
  - Subsample ratio of columns when constructing each tree.
  - Range (0, 1], default 1

Parameter Introduction

Parameter for Linear Booster

- `lambda`
  - L2 regularization term on weights
  - default 0
- `alpha`
  - L1 regularization term on weights
  - default 0
- `lambda_bias`
  - L2 regularization term on bias
  - default 0

Parameter Introduction

- objectives
  - "reg:linear": linear regression, default option.
  - "binary:logistic": logistic regression for binary classification, output probability
  - "multi:softmax": multiclass classification using the softmax objective, need to specify `num_class`
  - User specified objective

Parameter Introduction

- eval_metric
  - "rmse"
  - "logloss"
  - "error"
  - "auc"
  - "merror"
  - "mlogloss"
  - User specified evaluation metric

Guide on Parameter Tuning

It is nearly impossible to give a set of universal optimal parameters, or a global algorithm achieving it.

The key pointsof parameter tuning are

- Control Overfitting
- Deal with Imbalanced data
- Trust the cross validation

Guide on Parameter Tuning

The "Bias-Variance Tradeoff", or the "Accuracy-Simplicity Tradeoff" is the main idea for controlling overfitting.

For the booster specific parameters, we can group them as

- Controlling the model complexity
  - `max_depth`, `min_child_weight` and `gamma`
- Robust to noise
  - `subsample`, `colsample_bytree`

Guide on Parameter Tuning

Sometimes the data is imbalanced among classes.

- Only care about the ranking order
  - Balance the positive and negative weights, by `scale_pos_weight`
  - Use "auc" as the evaluation metric
- Care about predicting the right probability
  - Cannot re-balance the dataset
  - Set parameter `max_delta_step` to a finite number (say 1) will help convergence

Guide on Parameter Tuning

To select ideal parameters, use the result from `xgb.cv`.

- Trust the score for the test
- Use `early.stop.round` to detect continuously being worse on test set.
- If overfitting observed, reduce stepsize `eta` and increase `nround` at the same time.

Advanced Features

Advanced Features

There are plenty of highlights in XGBoost:

- Customized objective and evaluation metric
- Prediction from cross validation
- Continue training on existing model
- Calculate and plot the variable importance

Customization

According to the algorithm, we can define our own loss function, as long as we can calculate the first and second order gradient of the loss function.

Define $grad = \partial_{y^{t-1}} l$ and $hess = \partial^2_{y^{t-1}} l$. We can optimize the loss function if we can calculate these two values.

Customization

We can rewrite logloss for the i-th data point as

$$L = y_i \log(p_i) + (1 - y_i) \log(1 - p_i)$$

The p_i is calculated by applying logistic transformation on our prediction $\hat{y}_i$.

Then the logloss is

$$L = y_i \log \frac{1}{1 + e^{-\hat{y}_i}} + (1 - y_i) \log \frac{e^{-\hat{y}_i}}{1 + e^{-\hat{y}_i}}$$

Customization

We can see that

- $grad = \frac{1}{1+e^{-\hat{y}_i}} - y_i = p_i - y_i$
- $hess = \frac{1+e^{-\hat{y}_i}}{(1+e^{-\hat{y}_i})^2} = p_i(1 - p_i)$

Next we translate them into the code.

Customization

The complete code:

```
logregobj = function(preds, dtrain) {

  labels = getinfo(dtrain, "label")

  preds = 1/(1 + exp(-preds))

  grad = preds - labels

  hess = preds * (1 - preds)

  return(list(grad = grad, hess = hess))
}
```

Customization

The complete code:

```
logregobj = function(preds, dtrain) {
  # Extract the true label from the second argument
  labels = getinfo(dtrain, "label")

  preds = 1/(1 + exp(-preds))

  grad = preds - labels

  hess = preds * (1 - preds)

  return(list(grad = grad, hess = hess))
}
```

Customization

The complete code:

```
logregobj = function(preds, dtrain) {
  # Extract the true label from the second argument
  labels = getinfo(dtrain, "label")
  # apply logistic transformation to the output
  preds = 1/(1 + exp(-preds))

  grad = preds - labels

  hess = preds * (1 - preds)

  return(list(grad = grad, hess = hess))
}
```

Customization

The complete code:

```
logregobj = function(preds, dtrain) {
  # Extract the true label from the second argument
  labels = getinfo(dtrain, "label")
  # apply logistic transformation to the output
  preds = 1/(1 + exp(-preds))
  # Calculate the 1st gradient
  grad = preds - labels

  hess = preds * (1 - preds)

  return(list(grad = grad, hess = hess))
}
```

Customization

The complete code:

```
logregobj = function(preds, dtrain) {
  # Extract the true label from the second argument
  labels = getinfo(dtrain, "label")
  # apply logistic transformation to the output
  preds = 1/(1 + exp(-preds))
  # Calculate the 1st gradient
  grad = preds - labels
  # Calculate the 2nd gradient
  hess = preds * (1 - preds)

  return(list(grad = grad, hess = hess))
}
```

Customization

The complete code:

```
logregobj = function(preds, dtrain) {
  # Extract the true label from the second argument
  labels = getinfo(dtrain, "label")
  # apply logistic transformation to the output
  preds = 1/(1 + exp(-preds))
  # Calculate the 1st gradient
  grad = preds - labels
  # Calculate the 2nd gradient
  hess = preds * (1 - preds)
  # Return the result
  return(list(grad = grad, hess = hess))
}
```

Customization

We can also customize the evaluation metric.

```
evalerror = function(preds, dtrain) {

  labels = getinfo(dtrain, "label")

  err = as.numeric(sum(labels != (preds > 0)))/length(labels)

  return(list(metric = "error", value = err))
}
```

Customization

We can also customize the evaluation metric.

```
evalerror = function(preds, dtrain) {
  # Extract the true label from the second argument
  labels = getinfo(dtrain, "label")

  err = as.numeric(sum(labels != (preds > 0)))/length(labels)

  return(list(metric = "error", value = err))
}
```

Customization

We can also customize the evaluation metric.

```
evalerror = function(preds, dtrain) {
  # Extract the true label from the second argument
  labels = getinfo(dtrain, "label")
  # Calculate the error
  err = as.numeric(sum(labels != (preds > 0)))/length(labels)

  return(list(metric = "error", value = err))
}
```

Customization

We can also customize the evaluation metric.

```
evalerror = function(preds, dtrain) {
  # Extract the true label from the second argument
  labels = getinfo(dtrain, "label")
  # Calculate the error
  err = as.numeric(sum(labels != (preds > 0)))/length(labels)
  # Return the name of this metric and the value
  return(list(metric = "error", value = err))
}
```

Customization

To utilize the customized objective and evaluation, we simply pass them to the arguments:

```
param = list(max.depth=2,eta=1,nthread = 2, silent=1,
             objective=logregobj, eval_metric=evalerror)
bst = xgboost(params = param, data = train$data, label = train$label, nround = 2)
```

```
## [0]  train-error:0.0465223399355136
## [1]  train-error:0.0222631659757408
```

Prediction in Cross Validation

"Stacking" is an ensemble learning technique which takes the prediction from several models. It is widely used in many scenarios.

One of the main concern is avoid overfitting. The common way is use the prediction value from cross validation.

XGBoost provides a convenient argument to calculate the prediction during the cross validation.

Prediction in Cross Validation

```
res = xgb.cv(params = param, data = train$data, label = train$label, nround = 2,
             nfold=5, prediction = TRUE)
```

```
## [0]  train-error:0.046522+0.001041   test-error:0.046523+0.004164
## [1]  train-error:0.022263+0.001221   test-error:0.022263+0.004885
```

```
str(res)
```

```
## List of 2
##  $ dt  :Classes 'data.table' and 'data.frame':   2 obs. of  4 variables:
##   ..$ train.error.mean: num [1:2] 0.0465 0.0223
##   ..$ train.error.std : num [1:2] 0.00104 0.00122
##   ..$ test.error.mean : num [1:2] 0.0465 0.0223
##   ..$ test.error.std  : num [1:2] 0.00416 0.00488
##   ..- attr(*, ".internal.selfref")=<externalptr>
##  $ pred: num [1:6513] 2.58 -1.04 -1.12 2.57 -3.04 ...
```

xgb.DMatrix

XGBoost has its own class of input data `xgb.DMatrix`. One can convert the usual data set into it by

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

It is the data structure used by XGBoost algorithm. XGBoost preprocess the input `data` and `label` into an `xgb.DMatrix` object before feed it to the training algorithm.

If one need to repeat training process on the same big data set, it is good to use the `xgb.DMatrix` object to save preprocessing time.

xgb.DMatrix

An `xgb.DMatrix` object contains

1. Preprocessed training data
2. Several features
    - Missing values
    - data weight

Continue Training

Train the model for 5000 rounds is sometimes useful, but we are also taking the risk of overfitting.

A better strategy is to train the model with fewer rounds and repeat that for many times. This enable us to observe the outcome after each step.

Continue Training

```
bst = xgboost(params = param, data = dtrain, nround = 1)
```

```
## [0]  train-error:0.0465223399355136
```

```
ptrain = predict(bst, dtrain, outputmargin = TRUE)

setinfo(dtrain, "base_margin", ptrain)
```

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

```
bst = xgboost(params = param, data = dtrain, nround = 1)
```

```
## [0]  train-error:0.0222631659757408
```

Continue Training

```
# Train with only one round
bst = xgboost(params = param, data = dtrain, nround = 1)
```

```
## [0]  train-error:0.0222631659757408
```

```
ptrain = predict(bst, dtrain, outputmargin = TRUE)

setinfo(dtrain, "base_margin", ptrain)
```

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

```
bst = xgboost(params = param, data = dtrain, nround = 1)
```

```
## [0]  train-error:0.00706279748195916
```

Continue Training

```
# Train with only one round
bst = xgboost(params = param, data = dtrain, nround = 1)
```

```
## [0]  train-error:0.007062797481959l6
```

```
# margin means the baseline of the prediction
ptrain = predict(bst, dtrain, outputmargin = TRUE)

setinfo(dtrain, "base_margin", ptrain)
```

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

```
bst = xgboost(params = param, data = dtrain, nround = 1)
```

```
## [0]  train-error:0.0152003684937817
```

Continue Training

```
# Train with only one round
bst = xgboost(params = param, data = dtrain, nround = 1)
```

```
## [0]  train-error:0.0152003684937817
```

```
# margin means the baseline of the prediction
ptrain = predict(bst, dtrain, outputmargin = TRUE)
# Set the margin information to the xgb.DMatrix object
setinfo(dtrain, "base_margin", ptrain)
```

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

```
bst = xgboost(params = param, data = dtrain, nround = 1)
```

```
## [0]  train-error:0.00706279748195916
```

Continue Training

```
# Train with only one round
bst = xgboost(params = param, data = dtrain, nround = 1)
```

```
## [0]  train-error:0.007062797481959l6
```

```
# margin means the baseline of the prediction
ptrain = predict(bst, dtrain, outputmargin = TRUE)
# Set the margin information to the xgb.DMatrix object
setinfo(dtrain, "base_margin", ptrain)
```

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

```
# Train based on the previous result
bst = xgboost(params = param, data = dtrain, nround = 1)
```

```
## [0]  train-error:0.00122831260555811
```

Importance and Tree plotting

The result of XGBoost contains many trees. We can count the number of appearance of each variable in all the trees, and use this number as the importance score.

```
bst = xgboost(data = train$data, label = train$label, max.depth = 2, verbose = FALSE,
              eta = 1, nthread = 2, nround = 2,objective = "binary:logistic")
xgb.importance(train$dataDimnames[[2]], model = bst)
```

```
##    Feature       Gain     Cover Frequence
## 1:      28 0.67615484 0.4978746       0.4
## 2:      55 0.17135352 0.1920543       0.2
## 3:      59 0.12317241 0.1638750       0.2
## 4:     108 0.02931922 0.1461960       0.2
```

Importance and Tree plotting

We can also plot the trees in the model by `xgb.plot.tree`.

```
xgb.plot.tree(agaricus.train$data@Dimnames[[2]], model = bst)
```

Tree plotting

Early Stopping

When doing cross validation, it is usual to encounter overfitting at a early stage of iteration. Sometimes the prediction gets worse consistantly from round 300 while the total number of iteration is 1000. To stop the cross validation process, one can use the `early.stop.round` argumetn in `xgb.cv`.

```
bst = xgb.cv(params = param, data = train$data, label = train$label,
             nround = 20, nfold = 5,
             maximize = FALSE, early.stop.round = 3)
```

Early Stopping

```
## [0]  train-error:0.046522+0.001280   test-error:0.046523+0.005119
## [1]  train-error:0.022263+0.001463   test-error:0.022264+0.005852
## [2]  train-error:0.007063+0.000905   test-error:0.007064+0.003619
## [3]  train-error:0.015200+0.001163   test-error:0.015201+0.004653
## [4]  train-error:0.004414+0.002811   test-error:0.005989+0.004839
## [5]  train-error:0.001689+0.001114   test-error:0.002304+0.002304
## [6]  train-error:0.001228+0.000219   test-error:0.001228+0.000876
## [7]  train-error:0.001228+0.000219   test-error:0.001228+0.000876
## [8]  train-error:0.000921+0.000533   test-error:0.001228+0.000876
## [9]  train-error:0.000653+0.000601   test-error:0.001075+0.001030
## [10] train-error:0.000422+0.000582   test-error:0.000768+0.001086
## [11] train-error:0.000000+0.000000   test-error:0.000000+0.000000
## [12] train-error:0.000000+0.000000   test-error:0.000000+0.000000
## [13] train-error:0.000000+0.000000   test-error:0.000000+0.000000
## [14] train-error:0.000000+0.000000   test-error:0.000000+0.000000
## Stopping. Best iteration: 12
```

Kaggle Winning Solution

Kaggle Winning Solution

To get a higher rank, one need to push the limit of

1. Feature Engineering
2. Parameter Tuning
3. Model Ensemble

The winning solution in the recent Otto Competition is an excellent example.

Kaggle Winning Solution

Then they used a 3-layer ensemble learning model, including

- 33 models on top of the original data
- XGBoost, neural network and adaboost on 33 predictions from the models and 8 engineered features
- Weighted average of the 3 prediction from the second step

Kaggle Winning Solution

The data for this competition is special: the meanings of the featuers are hidden.

For feature engineering, they generated 8 new features:

- Distances to nearest neighbours of each classes
- Sum of distances of 2 nearest neighbours of each classes
- Sum of distances of 4 nearest neighbours of each classes
- Distances to nearest neighbours of each classes in TFIDF space
- Distances to nearest neighbours of each classed in T-SNE space (3 dimensions)
- Clustering features of original dataset
- Number of non-zeros elements in each row
- X (That feature was used only in NN 2nd level training)

Kaggle Winning Solution

This means a lot of work. However this also implies they need to try a lot of other models, although some of them turned out to be not helpful in this competition. Their attempts include:

- A lot of training algorithms in first level as
  - Vowpal Wabbit(many configurations)
  - R glm, glmnet, scikit SVC, SVR, Ridge, SGD, etc...
- Some preprocessing like PCA, ICA and FFT
- Feature Selection
- Semi-supervised learning

Influencers in Social Networks

Let's learn to use a single XGBoost model to achieve a high rank in an old competition!

The competition we choose is the Influencers in Social Networks competition.

It was a hackathon in 2013, therefore the size of data is small enough so that we can train the model in seconds.

Influencers in Social Networks

First let's download the data, and load them into R

```
train = read.csv('train.csv',header = TRUE)
test = read.csv('test.csv',header = TRUE)
y = train[,1]
train = as.matrix(train[,-1])
test = as.matrix(test)
```

Influencers in Social Networks

Observe the data:

```
colnames(train)
```

```
##  [1] "A_follower_count"    "A_following_count"   "A_listed_count"
##  [4] "A_mentions_received" "A_retweets_received" "A_mentions_sent"
##  [7] "A_retweets_sent"     "A_posts"             "A_network_feature_1"
## [10] "A_network_feature_2" "A_network_feature_3" "B_follower_count"
## [13] "B_following_count"   "B_listed_count"      "B_mentions_received"
## [16] "B_retweets_received" "B_mentions_sent"     "B_retweets_sent"
## [19] "B_posts"             "B_network_feature_1" "B_network_feature_2"
## [22] "B_network_feature_3"
```

Influencers in Social Networks

Observe the data:

```
train[1,]
```

```
##    A_follower_count   A_following_count      A_listed_count
##        2.280000e+02        3.020000e+02        3.000000e+00
## A_mentions_received A_retweets_received     A_mentions_sent
##        5.839794e-01        1.005034e-01        1.005034e-01
##     A_retweets_sent             A_posts A_network_feature_1
##        1.005034e-01        3.621501e-01        2.000000e+00
## A_network_feature_2 A_network_feature_3    B_follower_count
##        1.665000e+02        1.135500e+04        3.446300e+04
##   B_following_count      B_listed_count B_mentions_received
##        2.980800e+04        1.689000e+03        1.543050e+01
## B_retweets_received     B_mentions_sent     B_retweets_sent
##        3.984029e+00        8.204331e+00        3.324230e-01
##             B_posts B_network_feature_1 B_network_feature_2
##        6.988815e+00        6.600000e+01        7.553030e+01
## B_network_feature_3
##        1.916894e+03
```

Influencers in Social Networks

The data contains information from two users in a social network service. Our mission is to determine who is more influencial than the other one.

This type of data gives us some room for feature engineering.

Influencers in Social Networks

The first trick is to increase the information in the data.

Every data point can be expressed as . Actually it indicates <1-y, B, A> as well. We can simply use extract this part of information from the training set.

```
new.train = cbind(train[,12:22],train[,1:11])
train = rbind(train,new.train)
y = c(y,1-y)
```

Influencers in Social Networks

The following feature engineering steps are done on both training and test set. Therefore we combine them together.

```
x = rbind(train,test)
```

Influencers in Social Networks

The next step could be calculating the ratio between features of A and B seperately:

- followers/following
- mentions received/sent
- retweets received/sent
- followers/posts
- retweets received/posts
- mentions received/posts

Influencers in Social Networks

Considering there might be zeroes, we need to smooth the ratio by a constant.

```
calcRatio = function(dat,i,j,lambda = 1) (dat[,i]+lambda)/(dat[,j]+lambda)
```

Next we can calculate the ratio with this helper function.

Influencers in Social Networks

```
A.follow.ratio = calcRatio(x,1,2)
A.mention.ratio = calcRatio(x,4,6)
A.retweet.ratio = calcRatio(x,5,7)
A.follow.post = calcRatio(x,1,8)
A.mention.post = calcRatio(x,4,8)
A.retweet.post = calcRatio(x,5,8)
B.follow.ratio = calcRatio(x,12,13)
B.mention.ratio = calcRatio(x,15,17)
B.retweet.ratio = calcRatio(x,16,18)
B.follow.post = calcRatio(x,12,19)
B.mention.post = calcRatio(x,15,19)
B.retweet.post = calcRatio(x,16,19)
```

Influencers in Social Networks

Combine the features into the data set.

```
x = cbind(x[,1:11],
          A.follow.ratio,A.mention.ratio,A.retweet.ratio,
          A.follow.post,A.mention.post,A.retweet.post,
          x[,12:22],
          B.follow.ratio,B.mention.ratio,B.retweet.ratio,
          B.follow.post,B.mention.post,B.retweet.post)
```

Influencers in Social Networks

Then we can compare the difference between A and B. Because XGBoost is scale invariant, therefore minus and division are the essentially same.

```
AB.diff = x[,1:17]-x[,18:34]

x = cbind(x,AB.diff)
train = x[1:nrow(train),]
test = x[-(1:nrow(train)),]
```

Influencers in Social Networks

Now comes to the modeling part. We first investigate how far can we gowith a single model.

The parameter tuning step is very important in this step. We can see the performance from cross validation.

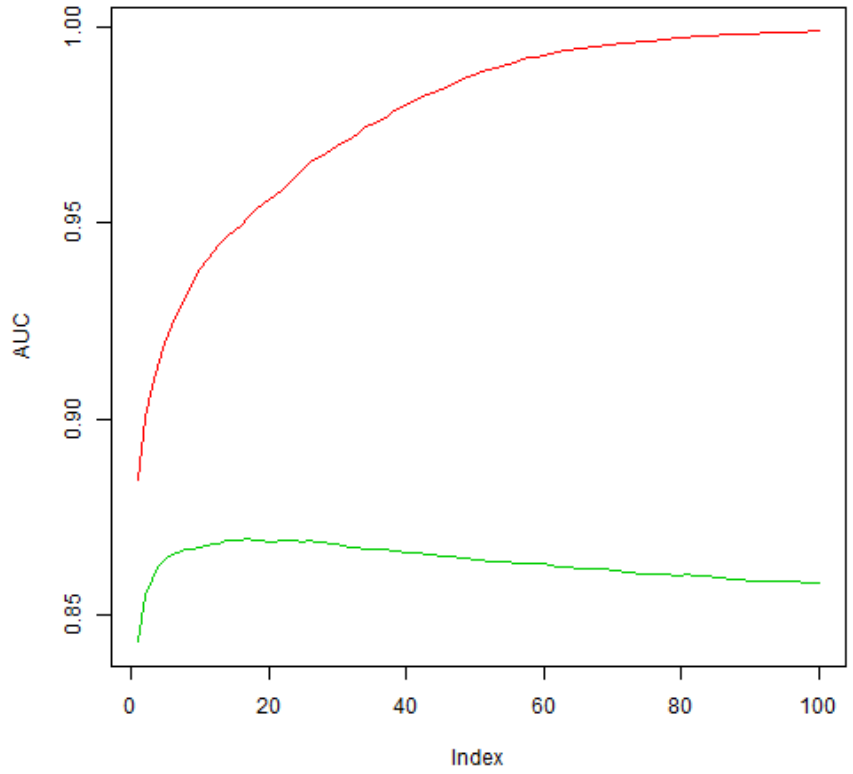
Influencers in Social Networks

Here's the `xgb.cv` with default parameters.

```
set.seed(1024)
cv.res = xgb.cv(data = train, nfold = 3, label = y, nrounds = 100, verbose = FALSE,
                objective='binary:logistic', eval_metric = 'auc')
```

Influencers in Social Networks

We can see the trend of AUC on training and test sets.



Influencers in Social Networks

It is obvious our model severly overfits. The direct reason is simple: the default value of `eta` is 0.3, which is too large for this mission.

Recall the parameter tuning guide, we need to decrease `eta` and inccrease `nrounds` based on the result of cross validation.

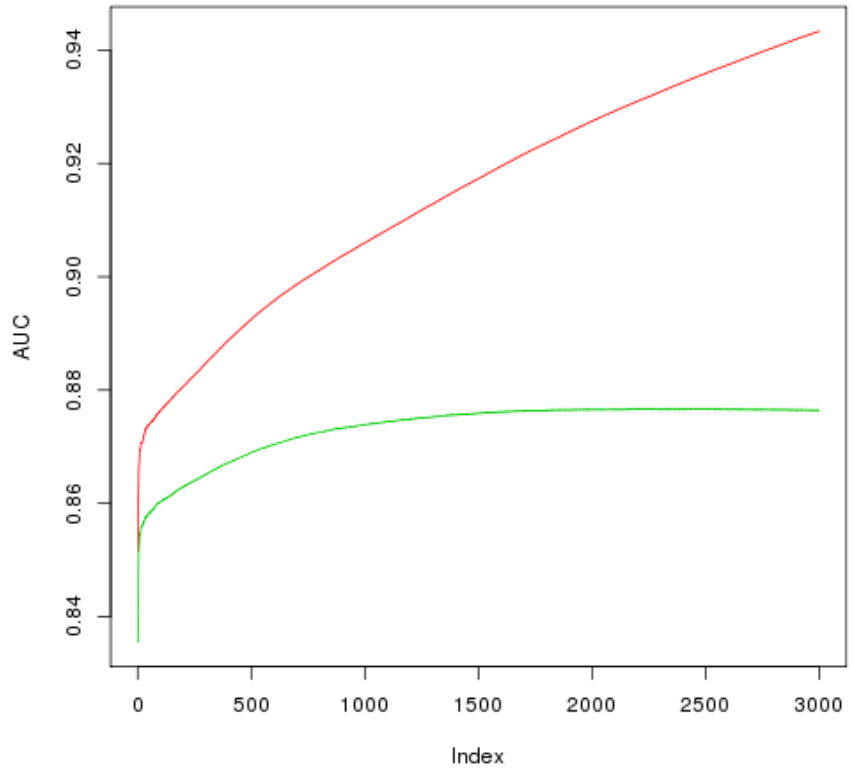
Influencers in Social Networks

After some trials, we get the following set of parameters:

```
set.seed(1024)
cv.res = xgb.cv(data = train, nfold = 3, label = y, nrounds = 3000,
                objective='binary:logistic', eval_metric = 'auc',
                eta = 0.005, gamma = 1,lambda = 3, nthread = 8,
                max_depth = 4, min_child_weight = 1, verbose = F,
                subsample = 0.8,colsample_bytree = 0.8)
```

Influencers in Social Networks

We can see the trend of AUC on training and test sets.



Influencers in Social Networks

Next we extract the best number of iterations.

We calculate the AUC minus the standard deviation, and choose the iteration with the largest value.

```
bestRound = which.max(as.matrix(cv.res)[,3]-as.matrix(cv.res)[,4])
bestRound
```

```
## [1] 2442
```

```
cv.res[bestRound,]
```

```
##    train.auc.mean train.auc.std test.auc.mean test.auc.std
## 1:       0.934967       0.00125      0.876629     0.002073
```

Influencers in Social Networks

Then we train the model with the same set of parameters:

```
set.seed(1024)
bst = xgboost(data = train, label = y, nrounds = 3000,
              objective='binary:logistic', eval_metric = 'auc',
              eta = 0.005, gamma = 1,lambda = 3, nthread = 8,
              max_depth = 5, min_child_weight = 1,
              subsample = 0.8,colsample_bytree = 0.8)
preds = predict(bst,test,ntreelimit = bestRound)
```

Influencers in Social Networks

Finally we submit our solution

```
result = data.frame(Id = 1:nrow(test),
                    Choice = preds)
write.csv(result,'submission.csv',quote=FALSE,row.names=FALSE)
```

This wins us into top 10 on the leaderboard!

FAQ