Xgboost

Introduction to Boosted Trees:

Tree boosting is a highly effective and widely used machine learning method. Let us understand the concepts of Regression Tree, Ensemble and gradient boosting before we jump into the widely popular XGBoost.

Regression Tree is also known as classification and regression tree has the same decision rules as in decision tree but in addition it has one score in each leaf value.

Ensemble as we know use multiple learning algorithms to obtain better predictive performance that can be obtained from any of the constituent learning algorithms. Many of the data mining competitions are won by using some variants of tree ensemble methods.

Model: Assuming we have K trees

$\hat{y}_i = \sum_{k=1}^{K} f_k(x_i), \quad f_k \in \mathcal{F}$, where F is the space of all Regression trees

Objective:

$$Obj = \underbrace{\sum_{i=1}^{n} l(y_i, \hat{y}_i)}_{\text{Training loss}} + \underbrace{\sum_{k=1}^{K} \Omega(f_k)}_{\text{Complexity of the Trees}}$$

Gradient Boosting: As we are using trees rather than numerical vectors we cannot use SGD (Stochastic Gradient Descent). Hence we use Additive Training (Boosting)

- Start from constant prediction, add a new function each time

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

Our objective remains which f to add at each stage so that the Obj is minimized. Taking the Taylor expansion and removing the constants out new objective becomes

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

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

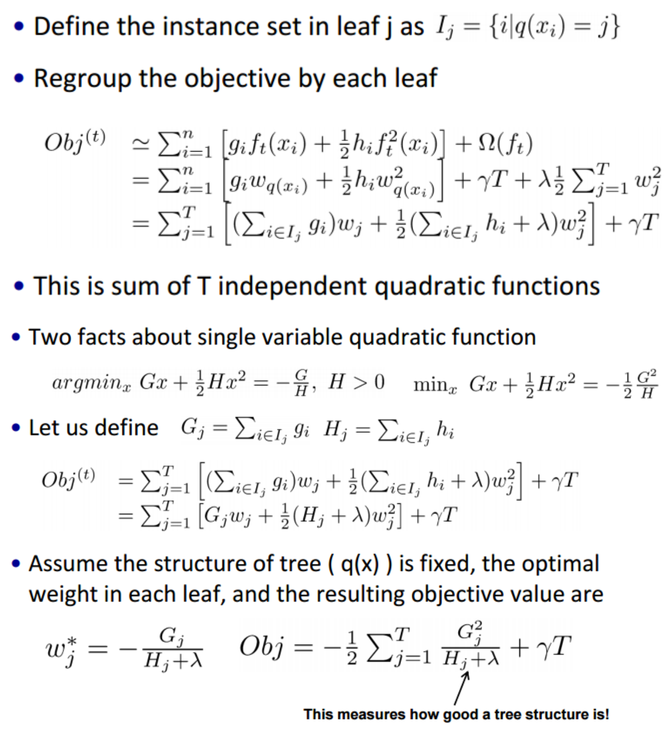
We define tree by a vector of scores in leafs, and a leaf index mapping function that maps an instance to a leaf

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

where w is the leaf weight of the tree and q is the structure of the tree.

Define complexity as (this is not the only possible definition)

$$\Omega(f_t) = \underbrace{\gamma T}_{\text{Number of leaves}} + \underbrace{\frac{1}{2} \lambda \sum_{j=1}^{T} w_j^2}_{\text{L2 norm of leaf scores}}$$



- Define the instance set in leaf j as $I_j = \{i|q(x_i) = j\}$
- Regroup the objective by each leaf

$$\begin{aligned} Obj^{(t)} &\simeq \sum_{i=1}^n \left[g_i f_t(x_i) + \tfrac{1}{2}h_i f_t^2(x_i)\right] + \Omega(f_t) \\ &= \sum_{i=1}^n \left[g_i w_{q(x_i)} + \tfrac{1}{2}h_i w_{q(x_i)}^2\right] + \gamma T + \lambda \tfrac{1}{2}\sum_{j=1}^T w_j^2 \\ &= \sum_{j=1}^T \left[(\sum_{i\in I_j} g_i)w_j + \tfrac{1}{2}(\sum_{i\in I_j} h_i + \lambda)w_j^2\right] + \gamma T \end{aligned}$$

- This is sum of T independent quadratic functions

- Two facts about single variable quadratic function

$$argmin_x\ Gx + \tfrac{1}{2}Hx^2 = -\tfrac{G}{H},\ H > 0 \qquad \min_x\ Gx + \tfrac{1}{2}Hx^2 = -\tfrac{1}{2}\tfrac{G^2}{H}$$

- Let us define $G_j = \sum_{i\in I_j} g_i \quad H_j = \sum_{i\in I_j} h_i$

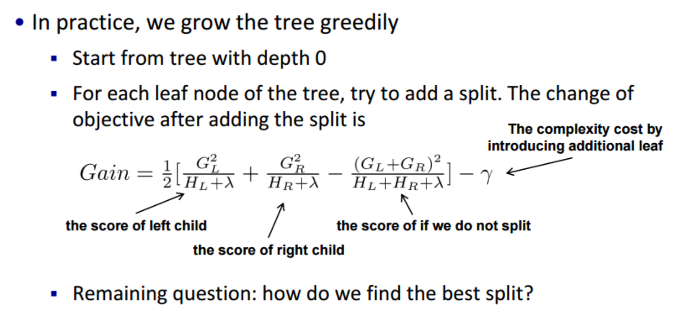
$$\begin{aligned} Obj^{(t)} &= \sum_{j=1}^T \left[(\sum_{i\in I_j} g_i)w_j + \tfrac{1}{2}(\sum_{i\in I_j} h_i + \lambda)w_j^2\right] + \gamma T \\ &= \sum_{j=1}^T \left[G_j w_j + \tfrac{1}{2}(H_j + \lambda)w_j^2\right] + \gamma T \end{aligned}$$

- Assume the structure of tree (q(x)) is fixed, the optimal weight in each leaf, and the resulting objective value are

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

This measures how good a tree structure is!

Though in algorithm we need to enumerate all the possible tree structures q it is not feasible because there can be infinitely many tree structures so,



- In practice, we grow the tree greedily
  - Start from tree with depth 0
  - For each leaf node of the tree, try to add a split. The change of objective after adding the split is

The complexity cost by introducing additional leaf

$$Gain = \frac{1}{2}\left[\frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{(G_L+G_R)^2}{H_L+H_R+\lambda}\right] - \gamma$$

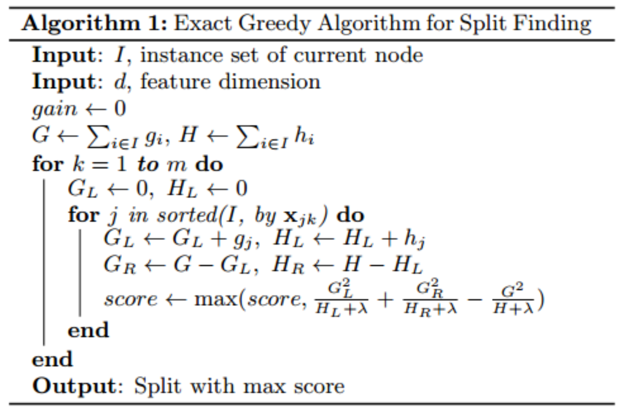
the score of left child

the score of if we do not split

the score of right child

  - Remaining question: how do we find the best split?

Ideally or the exact Greedy Algorithm for split finding is



```
Algorithm 1: Exact Greedy Algorithm for Split Finding
Input: $I$, instance set of current node
Input: $d$, feature dimension
$gain \leftarrow 0$
$G \leftarrow \sum_{i\in I} g_i$, $H \leftarrow \sum_{i\in I} h_i$
for $k = 1$ to $m$ do
    $G_L \leftarrow 0,\ H_L \leftarrow 0$
    for $j$ in sorted($I$, by $\mathbf{x}_{jk}$) do
        $G_L \leftarrow G_L + g_j,\ H_L \leftarrow H_L + h_j$
        $G_R \leftarrow G - G_L,\ H_R \leftarrow H - H_L$
        $score \leftarrow \max(score, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$
    end
end
Output: Split with max score
```

Though the exact algorithm is very powerful it becomes impossible to do it efficiently when the data doesn't fit entirely into memory. Therefore an approximation algorithm is used.

```
Algorithm 2: Approximate Algorithm for Split Finding
for $k = 1$ to $m$ do
    Propose $S_k = \{s_{k1}, s_{k2}, \cdots s_{kl}\}$ by percentiles on feature $k$.
    Proposal can be done per tree (global), or per split(local).
end
for $k = 1$ to $m$ do
    $G_{kv} \leftarrow= \sum_{j \in \{j | s_{k,v} \geq \mathbf{x}_{jk} > s_{k,v-1}\}} g_j$
    $H_{kv} \leftarrow= \sum_{j \in \{j | s_{k,v} \geq \mathbf{x}_{jk} > s_{k,v-1}\}} h_j$
end
Follow same step as in previous section to find max
score only among proposed splits.
```

Also in real world it is quite common that input x to be sparse (missing values too). To make the algorithm aware of the sparsity in the data a default direction is added at each tree node. The algorithm is shown below:

```
Algorithm 3: Sparsity-aware Split Finding
Input: $I$, instance set of current node
Input: $I_k = \{i \in I | x_{ik} \neq \text{missing}\}$
Input: $d$, feature dimension
Also applies to the approximate setting, only collect
statistics of non-missing entries into buckets
$gain \leftarrow 0$
$G \leftarrow \sum_{i \in I}, g_i, H \leftarrow \sum_{i \in I} h_i$
for $k = 1$ to $m$ do
    // enumerate missing value goto right
    $G_L \leftarrow 0,\ H_L \leftarrow 0$
    for $j$ in sorted($I_k$, ascent order by $\mathbf{x}_{jk}$) do
        $G_L \leftarrow G_L + g_j,\ H_L \leftarrow H_L + h_j$
        $G_R \leftarrow G - G_L,\ H_R \leftarrow H - H_L$
        $score \leftarrow \max(score, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$
    end
    // enumerate missing value goto left
    $G_R \leftarrow 0,\ H_R \leftarrow 0$
    for $j$ in sorted($I_k$, descent order by $\mathbf{x}_{jk}$) do
        $G_R \leftarrow G_R + g_j,\ H_R \leftarrow H_R + h_j$
        $G_L \leftarrow G - G_R,\ H_L \leftarrow H - H_R$
        $score \leftarrow \max(score, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$
    end
end
Output: Split and default directions with max gain
```

Column block for Parallel Learning:

The most time consuming part is to get the data in sorted order. In order to reduce the cost of sorting instead of storing the entire dataset in a single block and running the split search we store the data in inmemory units. In each such unit data is stored in the compressed column format (CSC) . Collecting statistics from each column can be parallelized, giving us a parallel algorithm for split finding. Apart from these XGBoost also has capabilities like **Cache-aware Access** (to prevent non-continuous memory access) and **Out-of-Core Computation** to fully utilize a machine's resources.

Xgboost in R

For a simple interface in R we use **xgboost()** and for more a advanced interface we use **xgboost.train()**

xgboost

Usage: xgboost(data = NULL, label = NULL, missing = NULL, params = list(), nrounds, verbose = 1, print.every.n = 1L, early.stop.round = NULL, maximize = NULL, . . .)

The list and description of some important arguments

- data takes matrix, dgCMatrix, local data file. Label is the response variable, missing to represent missing values (used only when input is dense).
- objective: "reg:linear" or "binary:logistic"
- eta: stepsize for boosting
- max.depth : maximum depth of the tree
- nthreads: number of threads used in training (if not set default uses all the threads)
- nrounds: number of iterations
- subsample: 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
- colsample_bytree: subsample ratio of columns when constructing each tree

---

We are going to use the Otto dataset from Kaggle to illustrate various parameters and their effect in trainig the model using xgboost.

Otto dataset

Here is a brief description of the dataset and the problem statement:

The Otto Group is one of the world's biggest e-commerce companies. Due to the diverse global infrastructure, many identical products get classified differently. The quality of product analysis depends heavily on the ability to accurately cluster similar products. The dataset has 93 features for more than 200,000 products. The objective is to build a predictive model which is able to distinguish between the main product categories.

So, here we train a multi-class xgboost model to classify each product into one of the nine categories that are provided.

R code documentation

Read the data

```r
data <- fread('train.csv', header = T, stringsAsFactors = F)
```

Dividing data into test and train

```
data[,rand:=runif(nrow(data))];
data[rand<=0.3,train:=F];
data[rand>0.3,train:=T];
data[,rand:=NULL];
train <- data[train==T];
train[,train:=NULL];
test <- data[train ==F];
test[,train:=NULL];
data[,train:=NULL];
```

Data Cleaning and preparation

Delete ID column in training dataset

```
train[, id := NULL]
```

Delete ID column in testing dataset

```
test[, id := NULL]
```

Save the name of the last column

```
nameLastCol <- names(train)[ncol(train)]
```

Convert from classes to numbers

```
y <- train[, nameLastCol, with = F][[1]] %>% gsub('Class_','',.) %>% {as.integer(.) -1}
```

Display the first 5 levels

```
y[1:5]
```

```
## [1] 0 0 0 0 0
```

We remove label column from training dataset, otherwise XGBoost would use it to guess the labels!

```
train[, nameLastCol:=NULL, with = F]
test[, nameLastCol:=NULL, with = F]
```

data.table is an awesome implementation of data.frame, unfortunately it is not a format supported natively by XGBoost. We need to convert both datasets (training and test) in numeric Matrix format.

r convertToNumericMatrix

```
trainMatrix <- train[,lapply(.SD,as.numeric)] %>% as.matrix
testMatrix <- test[,lapply(.SD,as.numeric)] %>% as.matrix
```

Number of Classes

```
numberOfClasses <- max(y) + 1
```

So, we train the model and try to explain different the different parameters and their effects on the model training.

Model Training

Subsample : 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. It makes computation shorter (because less data to analyse). It is advised to use this parameter with eta and increase nround.

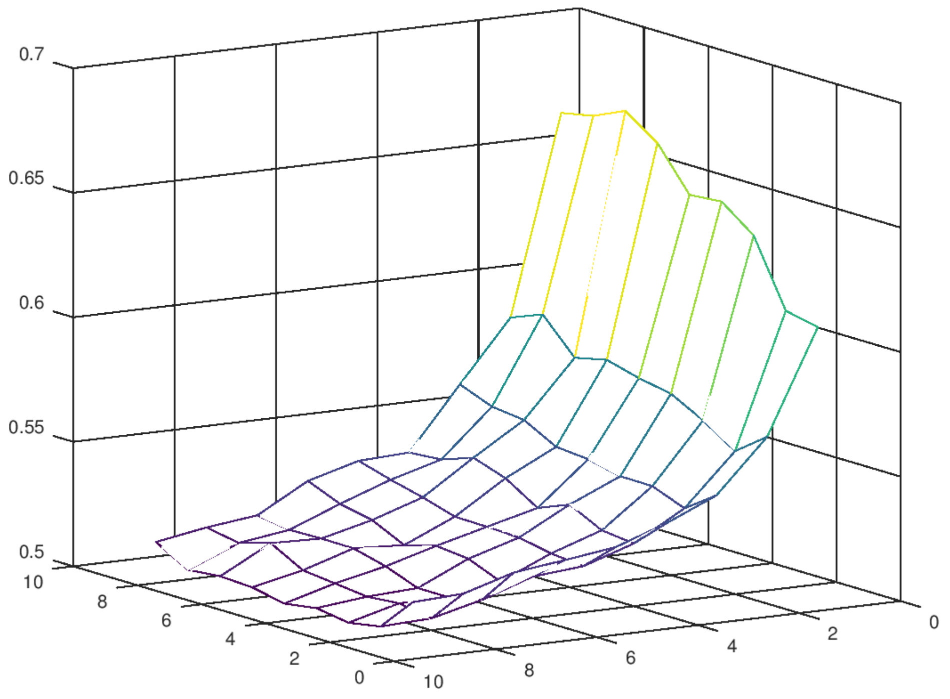
Colsample_bytree : subsample ratio of columns when constructing each tree

We create a 9X9 grid of **subsample** & **colsample_bytree** each varying from 0.1 to 0.9 in steps of 0.1 and find the optimal values that give the minimum loss.

```
sub_col_samplegrid <- data.frame(subsample=seq(0.1, 0.9, 0.1),colsample_bytree=seq(0.1, 0.9, 0.1),strin
mlogloss<- data.frame(values=seq(1,81,1),stringsAsFactors=FALSE)
mloglossgrid<-data.frame(v1=seq(1,9,1),v2=seq(1,9,1),v3=seq(1,9,1),v4=seq(1,9,1),v5=seq(1,9,1),v6=seq(1
```

```
for (ss in 1:9){
  for (cs in 1:9){
    param <- list("objective" = "multi:softprob",
                  "eval_metric" = "mlogloss",
                  "num_class" = numberOfClasses,subsample=sub_col_samplegrid[ss,1],colsample_bytree=sub
    #Finally, we are ready to train the real model!!!

    nround = 200
    bst = xgboost(param=param, data = trainMatrix, label = y, nrounds=nround)

    pred<-predict(bst,testMatrix)

    prob <- matrix(predict(bst, testMatrix), ncol = 9, byrow = T)
    prob=data.table(prob)
    prob[,target:=z]

    prob=data.frame(prob)

    ans=0
    #Evaluation
    for (i in (1:nrow(prob))){
      ans=ans-log(prob[,names(prob)[ncol(prob)-9+prob$target[i]]][i])
    }
    logloss=ans/nrow(prob)
    mlogloss[(ss-1)*9+cs,]=logloss
    mloglossgrid[ss,cs]=logloss
  }
}
```

This is the plot of the grid



Find out the minimum from the grid and this will give us the optimal values for colsample_bytree and subsample

So, here we basically find the optimal number of data instances and ratio of columns while constructing the trees.

```
index_of_min=match(min(mlogloss),mlogloss[,])
cs=mod(index_of_min,9)
if(cs==0){cs=9}
ss=(index_of_min-cs)/9+1

sub_sample=sub_col_samplegrid[ss,1]
col_sample=sub_col_samplegrid[cs,1]

sub_sample
col_sample
```

eta : Learning rate. scale the contribution of each tree by a factor of $0 < \text{eta} < 1$ when it is added to the current approximation. Used to prevent overfitting by making the boosting process more conservative. Lower value for eta implies larger value for nrounds: low eta value means model more robust to overfitting but slower to compute

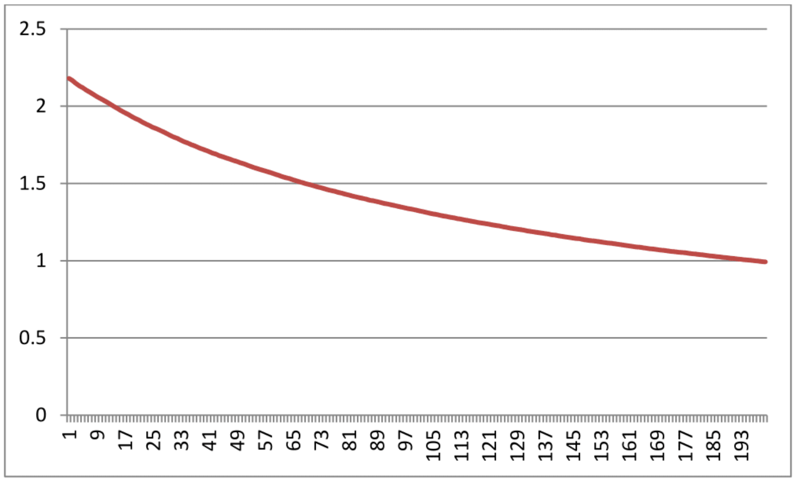
Now we vary different eta (the learning rate) and see how the loss on test data varies

```
ETA=data.frame(0.01,0.02,0.03,0.04,0.05,0.08,0.1,0.15,0.2,0.25,0.3,0.35,0.4)

mloglosstest<- data.frame(values=seq(1,13,1),stringsAsFactors=FALSE)
```

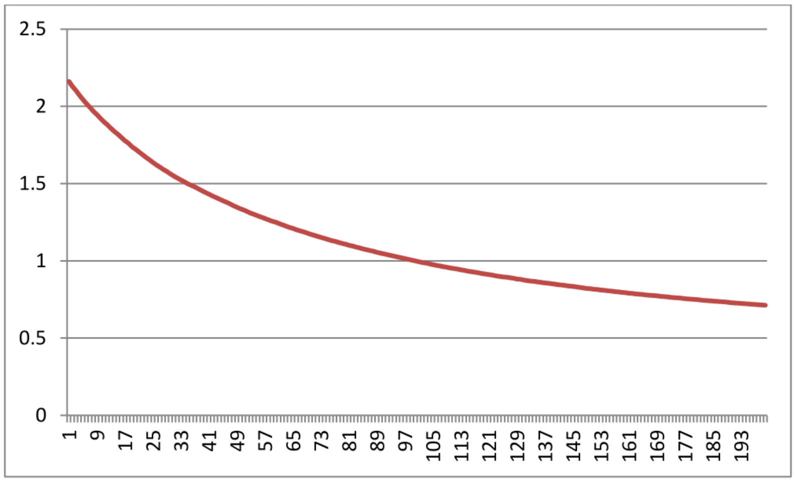
```r
#ETA=data.frame(0.01,0.08,0.1,0.15,0.2)

for (eta in ETA){
  print("this is")
  print(eta)
  param <- list("objective" = "multi:softprob",
                "eval_metric" = "mlogloss",
                "num_class" = numberOfClasses,subsample=sub_sample,colsample_bytree=col_sample,eta=eta)

  nrounds=200

  bst = xgboost(param=param, data = trainMatrix, label = y, nrounds=nrounds)

  prob <- matrix(predict(bst, testMatrix), ncol = 9, byrow = T)
  prob=data.table(prob)
  prob[,target:=z]
  prob=data.frame(prob)

  ans=0
  #Evaluation
  for (i in (1:nrow(prob))){
    ans=ans-log(prob[,names(prob)[ncol(prob)-9+prob$target[i]]][i])
  }
  logloss=ans/nrow(prob)
  logloss
  mloglosstest[match(eta,ETA)]=logloss
}
```

ETA values [0.01,0.02,0.03,0.04,0.05,0.08,0.1,0.15,0.2]

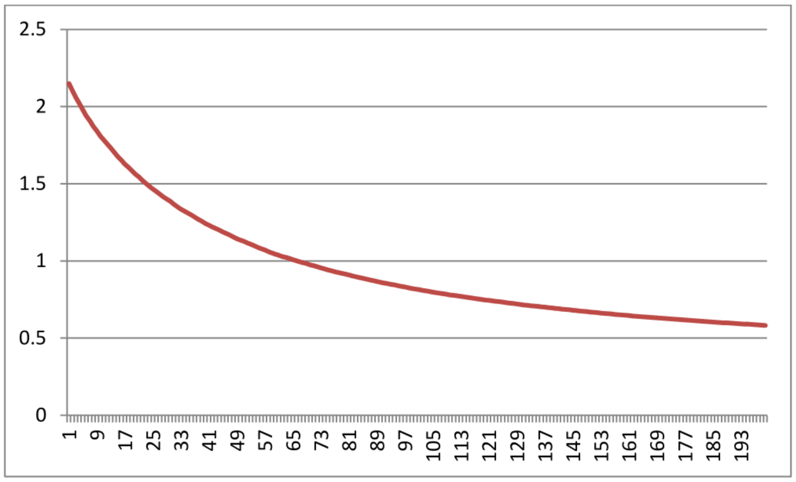
0.01



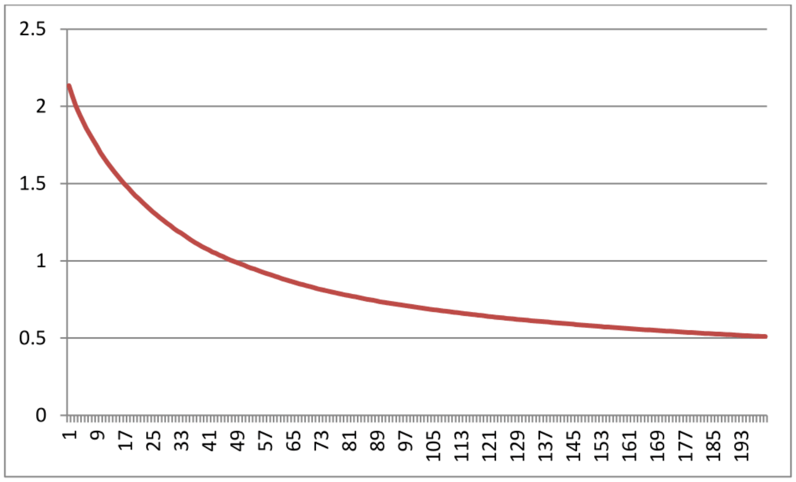
0.02



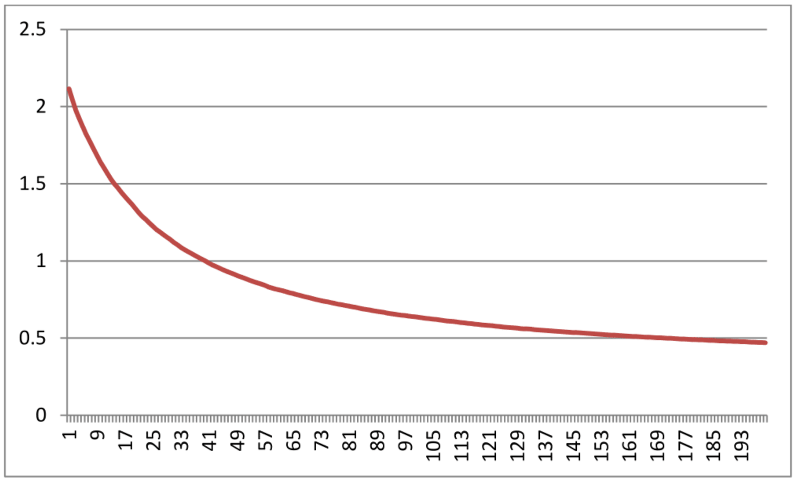
0.03


2.5
2
1.5
1
0.5
0
1
9
17
25
33
41
49
57
65
73
81
89
97
105
113
121
129
137
145
153
161
169
177
185
193

0.04

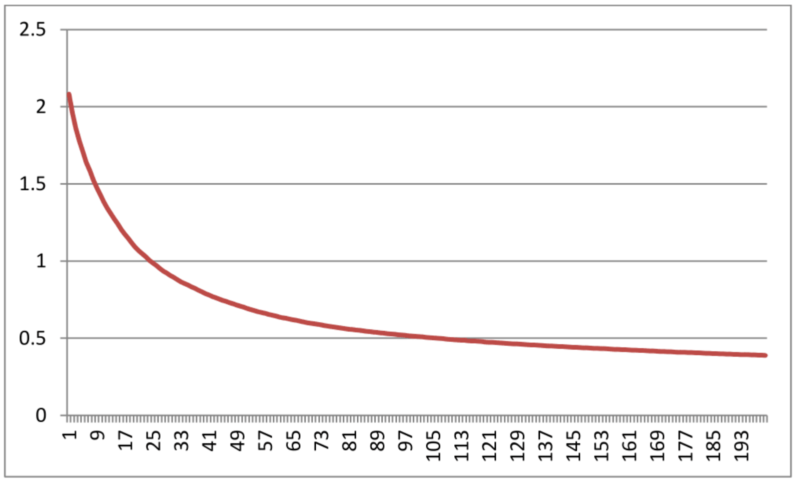

2.5
2
1.5
1
0.5
0
1
9
17
25
33
41
49
57
65
73
81
89
97
105
113
121
129
137
145
153
161
169
177
185
193

0.05



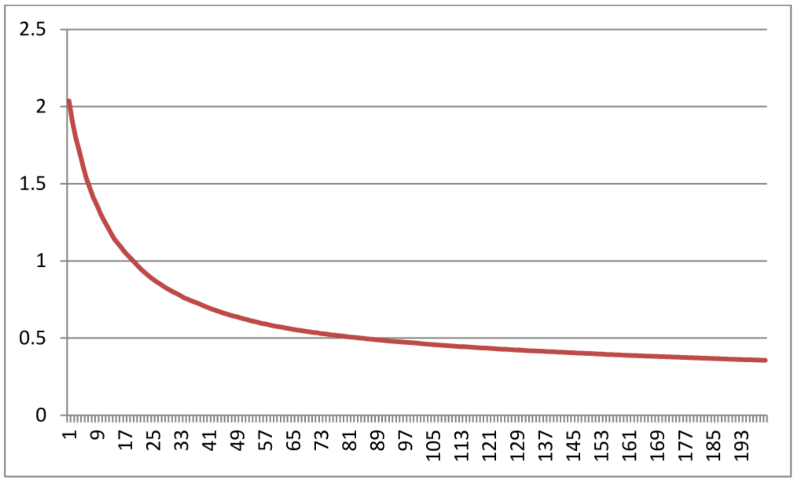
2.5
2
1.5
1
0.5
0
1
9
17
25
33
41
49
57
65
73
81
89
97
105
113
121
129
137
145
153
161
169
177
185
193

0.08

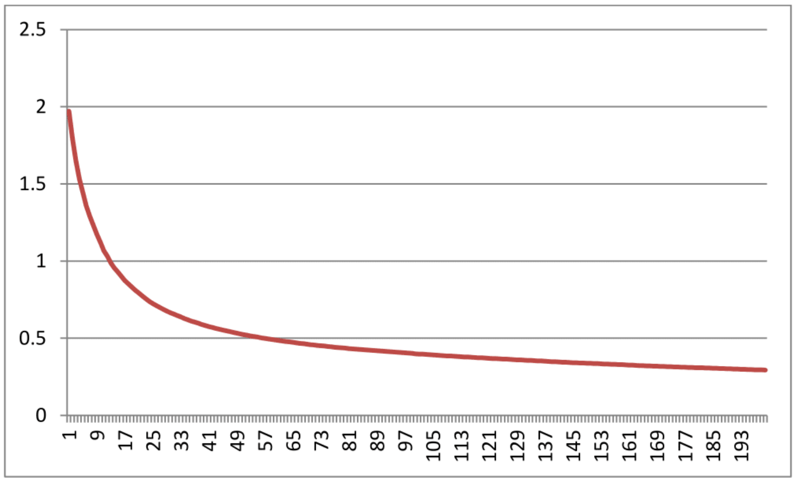


2.5
2
1.5
1
0.5
0
1
9
17
25
33
41
49
57
65
73
81
89
97
105
113
121
129
137
145
153
161
169
177
185
193

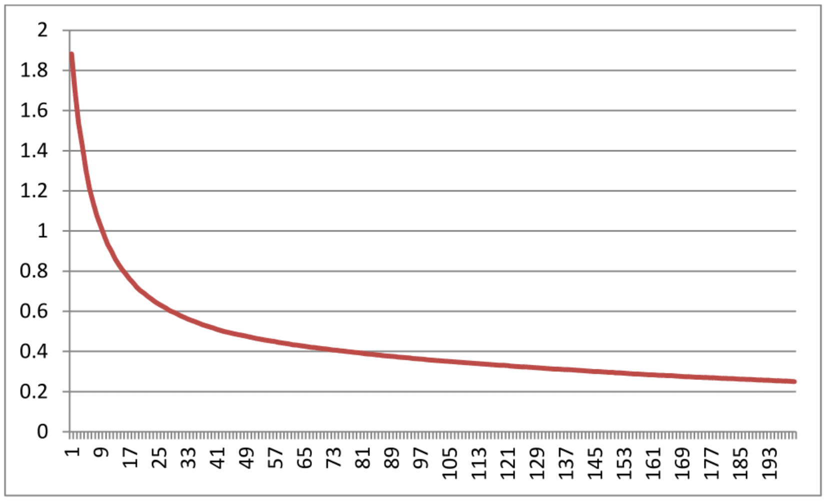
0.1


2.5
2
1.5
1
0.5
0
1
9
17
25
33
41
49
57
65
73
81
89
97
105
113
121
129
137
145
153
161
169
177
185
193

0.15


2.5
2
1.5
1
0.5
0
1
9
17
25
33
41
49
57
65
73
81
89
97
105
113
121
129
137
145
153
161
169
177
185
193

0.2



The mlogloss on the testdata is as follows:

mlogloss	eta
1.0322113	0.01
0.7683885	0.02
0.66323	0.03
0.6030883	0.04
0.5743571	0.05
0.5285042	0.08
0.5179335	0.10
0.502922	0.15
0.497319	0.20
0.4961615	0.25
0.504498	0.30
0.5054173	0.35
0.5153798	0.40

Cross Validation

We are going to use the best parameters and do cross-validation

```
eta=0.25
param <- list("objective" = "multi:softprob",
              "eval_metric" = "mlogloss",
              "num_class" = numberOfClasses,subsample=0.9,colsample_bytree=0.3,eta=eta)
```

Create five-folds cross validation

```
require(caret)
flds <- createFolds(y, k = 5, list = TRUE, returnTrain = FALSE)
```

Cross validation - implementation

```
prob_matrix=matrix(nrow=nrow(testMatrix_kaggle),ncol=5)
validationloss=data.frame(values=1:5)

for (fold in 1:5){

  validation_fold1=trainMatrix[ flds[[fold]], ]
  y_fold1=y[ flds[[fold]] ]

  train_fold1=trainMatrix[-(flds[[fold]]),]
  y_trainfold1=y[-(flds[[fold]])]

  nrounds=200
  bst_1 = xgboost(param=param, data = train_fold1, label = y_trainfold1,          nrounds=nrounds)

  prob <- matrix(predict(bst_1, testMatrix_kaggle), ncol = 9, byrow = T)
  prob=data.frame(prob)
  assign(paste("prob_",fold,sep=""), prob)

}

# Get the feature real names
names <- dimnames(trainMatrix)[[2]]

# Compute feature importance matrix
importance_matrix <- xgb.importance(names, model = bst_1)

# Nice graph
xgb.plot.importance(importance_matrix[1:10,])
```

We combine the output all the five models and create a final table giving the probabilities for all the entries falling into each category.

```
prob_matrix_kaggle=(prob_1+prob_2+prob_3+prob_4+prob_5)
prob_matrix_kaggle=prob_matrix_kaggle/5
names(prob_matrix_kaggle)=c("Class_1","Class_2","Class_3","Class_4","Class_5","Class_6","Class_7","Clas
# write.csv(prob_matrix_kaggle,file="answer_kaggle.csv")
# 0.471 when submitted in Kaggle
```