ML Project Part 2

Mayank

6/18/2018

The main objective of this exercise was to fine tune any algorithm and use the trained model on our test data set such that the log loss value is less than 0.5 to obtain a 100% score on one of our assignments. The exact labels for the testing data was not available thus this was quite a rigorous task in order to attain the best model.

In this module I will show the final model I have used with and the accuracy score for certain hyper-parameter values as examples.

In order to obtain the final model or the best parameter more than 50 iterations were used and since this is a fairly large data set the training time involved was over weeks.

Further, in the last section of the algorithm I will enumerate a brute-force grid search algorithm taught to us in our lectures. More, on this in the last section.

Installing the required packages for this exercise and activating the libraries

```r
options(repos="https://cran.rstudio.com" )
install.packages("xgboost")
```

```
##
##   There is a binary version available but the source version is
##   later:
##         binary source needs_compilation
## xgboost 0.71.1 0.71.2              TRUE
```

```
## installing the source package 'xgboost'
```

```r
install.packages("NMOF")
```

```
##
##   There is a binary version available but the source version is
##   later:
##      binary source needs_compilation
## NMOF  1.4-1  1.4-3             FALSE
```

```
## installing the source package 'NMOF'
```

```r
library(xgboost)
library(data.table)
library(NMOF)
```

```r
d_path<-"/Users/mayank/Documents/College Documents/Q3 Courses/Machine Learning/Course Project/supervize
```

The two data sets provided are the actual train and testing data set, the training data set has the labels but the test data set doesnot have the labels

```r
train_data<-read.csv(file=paste(d_path,"train_sample.csv",sep="/"))
test_data<-read.csv(file=paste(d_path,"test_sample.csv",sep="/"))
head(train_data)
```

```
##   feat_1 feat_2 feat_3 feat_4 feat_5 feat_6 feat_7 feat_8 feat_9 feat_10
## 1      0      0      0      1      0      0      0      0      0       0
## 2      1      0      0      1      0      0      0      1      2       0
## 3      2      0      1      0      0      0      0      0      0       0
## 4      1      0      0      2      0      0      2      0      0       0
## 5      0      0      0      0      0      0      0      1      0       0
## 6      0      1      0      0      0      0      0      0      0       0
##   feat_11 feat_12 feat_13 feat_14 feat_15 feat_16 feat_17 feat_18 feat_19
## 1       0       0       0       0       0       0       0       0       1
## 2       0       0       0       0       0       0       0       0       5
## 3       0       0       0       0       0       0       0       0       0
## 4       0       0       1       0       0       0       2       0       3
## 5       0       0       0       0       0       1       0       0       0
## 6      52       0       0       0       0       2       0       0       0
##   feat_20 feat_21 feat_22 feat_23 feat_24 feat_25 feat_26 feat_27 feat_28
## 1       0       1       0       1       0       0       0       0       0
## 2       0       4       0       0       0       0       0       0       0
## 3       0       0       0       1       0       0       0       0       1
## 4       1       2       0       0       0       0       3       2       0
## 5       0       2       3       0       1       0      21       0       1
## 6       0       3       1       0       0       0       0       1       1
##   feat_29 feat_30 feat_31 feat_32 feat_33 feat_34 feat_35 feat_36 feat_37
## 1       2       3       0       0       0       0       0       0       1
## 2       1       0       0       3       0       0       0       2       0
## 3       0       0       0       0       0       0       0       0       4
## 4       0       0       2       0       0       0       0       3       6
## 5       2       0       0       0       0       0       0       1       0
## 6       1       0       0       0       0       0       0       4       0
##   feat_38 feat_39 feat_40 feat_41 feat_42 feat_43 feat_44 feat_45 feat_46
## 1       0       0       0       0       1       3       0       0       0
## 2       1       0       0       2       2       0       2       0       0
## 3       1       0       0       0       0       0       0       0       0
## 4       0       0       1       1       1       1       0       1       1
## 5       0       0       0       0       4       0       1       5       0
## 6       0       0       0       0       0       0       0       0       0
##   feat_47 feat_48 feat_49 feat_50 feat_51 feat_52 feat_53 feat_54 feat_55
## 1       0       0       0       0       0       0       0       0       0
## 2       0       0       0       2       1       0       2       2       1
## 3       0       0       0       0       0       0       1       0       0
## 4       0       0       2       0       0       0       1       0       0
## 5       0       0       0       0       0       0       3       0       0
## 6       0       0       0       0       0       0       0       0       0
##   feat_56 feat_57 feat_58 feat_59 feat_60 feat_61 feat_62 feat_63 feat_64
## 1       1       0       0       0       0       0       1       0       1
## 2       0       0       0       0       4       0       0       0       1
## 3       1       0       0       0       0       1       0       0       2
## 4       2       0       0       0       3       0       0       1       0
## 5       0       0       0       0       1       1       0      19       0
## 6       1       0       0       0       1       0       0       0       0
##   feat_65 feat_66 feat_67 feat_68 feat_69 feat_70 feat_71 feat_72 feat_73
## 1       0       0       0       0       0       0       0       0       6
## 2       0       1       0       0       0       0       0       0       0
## 3       0       0       0       1       0       0       0       1       0
## 4       0       8       0       0       0       0       3       2       0
## 5       0       0       0       0       0       0       0       1       0
## 6       0       0       0       0       0       0       0       6       0
##   feat_74 feat_75 feat_76 feat_77 feat_78 feat_79 feat_80 feat_81 feat_82
## 1       0       0       0       0       0       0       0       0       0
## 2       0       8       1       0       1       0       0       0       0
## 3       0       0       2       0       0       0       0       1       0
## 4       9       0       1       0       0       0       0       1       1
## 5       1       0       1       0       0       0       0       0       1
## 6       0       1       0       0       0       1       0       0       0
##   feat_83 feat_84 feat_85 feat_86 feat_87 feat_88 feat_89 feat_90 feat_91
## 1       0       0       0       0       0       1       0       0       0
## 2       2       0       0       1       0       1       0       0       0
## 3       2       0       0       0       0       0       0       1       0
## 4       1       0       1       1       0       1       4       1       1
## 5       0       0       0       1       0       0       1       0       0
## 6       0       0       0       2       0       0       0       0       0
##   feat_92 feat_93 target
## 1       0       0      1
## 2       2       0      0
## 3       0       0      1
## 4       3       1      3
## 5       1       1      7
## 6       0       1      6
```

The section of the code below is for random splitting of data in testing and training sets. This split is done for our initial training data set so that before uploading my solution I can check the accuracy of my model on a new data set.

However, in the section below I am working on my entire data set as the initial training and testing portion need not to be enumerated here.

This practice should always be employed to check the accuracy of our models.

```r
n<-nrow(train_data)

#splitting the training data into test and train again to check the accuracy of my predictions

trainIndex<-sample(1:n, size = round(0.99*n), replace=FALSE)

#unscaled data set to analyse the clusters

u_train_data<-train_data[trainIndex,]
u_test_data<-train_data[-trainIndex ,]
```

Creating the log loss function to test the train and test data log loss value

```r
MultiLogLoss <- function(act, pred)
{
  eps = 1e-15;
  nr <- nrow(pred)
  pred = matrix(sapply( pred, function(x) max(eps,x)), nrow = nr)
  pred = matrix(sapply( pred, function(x) min(1-eps,x)), nrow = nr)
  #normalize rows
  ll = sum(act*log(sweep(pred, 1, rowSums(pred), FUN="/")))
  ll = -ll/nrow(act)
  return(ll);
}
```

Since, we are going to use xgboost for this exercise we need to convert the categorical columns to numeric

```
xTrain<-u_train_data[,-which(names(u_train_data)=="target")]
xTest<-u_test_data[,-which(names(u_test_data)=="target")]

#creating a loop for transforming the train variable type to numeric from int

(n<-ncol(xTrain))
```

```
## [1] 93
```

```
for (i in 1:n){
  xTrain[,i]<-as.numeric(xTrain[,i])
}

#another for loop for test data set numeric conversion (which was created from the initial training dat

n2<-ncol(xTest)
for (i in 1:n2){
  xTest[,i]<-as.numeric(xTest[,i])
}

xgbTest<-data.matrix(xTest)
xgbTrain<-data.matrix(xTrain)
str(xgbTrain)
```

```
##  num [1:40832, 1:93] 0 0 1 0 0 1 0 0 1 2 ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : chr [1:40832] "12233" "20909" "20536" "21214" ...
##   ..$ : chr [1:93] "feat_1" "feat_2" "feat_3" "feat_4" ...
```

```
#creating the overall test data, for which we do not have the predictions, it is being created from
#test_data which has been imported above

n3<-ncol(test_data)
for (i in 1:n3){
  test_data[,i]<-as.numeric(test_data[,i])
}
xgb_out_test<-data.matrix(test_data)



yTrain = as.factor(u_train_data$target)
yTrain = as.integer(yTrain)-1
numClasses = max(yTrain) + 1
yTest<-as.factor(u_test_data$target)
yTest<-as.integer(yTest)-1
```

The below mentioned code contains the different hyper-parameters to tune the model. You need to try different combinations in-order to get the correct values. The best approach is to start with a couple of hyper-parameters and then test how do they behave by changing their values, then add on the other hyper-parameters one by one and decide on the basis of improvement in accuracy scores. Further, it is best advised to research the hyper-parameters and understand their use in gradient boosting algorithm to best select the parameters. Describing that will require an entire article on its own.

- eta - it is the learning rate for the algorithm

- gamma - gamma controls the regularization or in other words prevents overfitting. Loss reduction required to make a further partition on a leaf node of the tree. the larger
- lambda - L2 regularization term on weights, increase this value will make model more conservative. default is 1
- alpha - L1 regularization term on weights, increase this value will make model more conservative. defualt is 0
- max_depth - for larger sample size the algorithm needs to create dense trees. In-order to evaluate complex rules inherent to the data set. Larger the depth, more complex the model; higher chances of overfitting. Its default value is 6
- min_child_weight - it is the minimum number of instances required in the child node, less the number of data points in the child node, higher the chances of overfitting the dataset
-

In the snippet below I am only showing one iteration which helped me get the logloss below 0.5. Further, the result was obtained by using more than 50 iterations. I could have also employed GridSearch but it was brute-force grid search which created more than 300 combinations for evaluation and given the computation limitation of my laptop it did not seem a good idea. Thus I changed the values manually by understanding their effect on the algorithm and then iterating the values intuitively. Anyhow, in the later part of the script I will also enumerate the method to apply grid search.

Fruther, the below mentioned combination worked a variety of data sets which my classmates got.

```
#Setting the parameters to run xgboost
xg_eta<-0.05
cv.nround <- 2000
cv.nfold <- 4
xg_gamma<-0
xg_lambda<-6
xg_alpha<-7
xg_max_depth<-8
xg_min_child_weight<-20
xg_subsample<-0.7
xg_colsample_bytree<-0.5
xg_rowsample_bytree<-0.9

param <- list("objective" = "multi:softprob",
              "eval_metric" = "mlogloss",
              "num_class" = numClasses,
              "eta"=xg_eta,
            "alpha"=xg_alpha,
             "lambda"=xg_lambda,
              "max_depth"=xg_max_depth,
              "gamma"=xg_gamma,
            #  "colsample_bytree"=xg_colsample_bytree,
            #"rowsample_bytree"=xg_rowsample_bytree,
              "subsample"=xg_subsample
            )
```

Now, we will be running cross validation to obtain the optimal number of iterations required. In my case it was 1031. Also note in the output that if the CV error keeps on decreasing and does not change its behavior then mostly it is not over-fittig.

```
#ALWAYS set seed before you run this algorithm to be able to reproduce the results
set.seed(1)
```

```r
#run cross validation to fin the best optimal number of iterations required
cvOtto<-xgb.cv(param=param, data = xgbTrain, label = yTrain,
                 nfold = cv.nfold, nrounds = cv.nround,verbose=F)
cvOtto[1031]
```

```
## $<NA>
## NULL
```

```r
names(cvOtto)
```

```
## [1] "call"            "params"          "callbacks"       "evaluation_log"
## [5] "niter"           "nfeatures"       "folds"
```

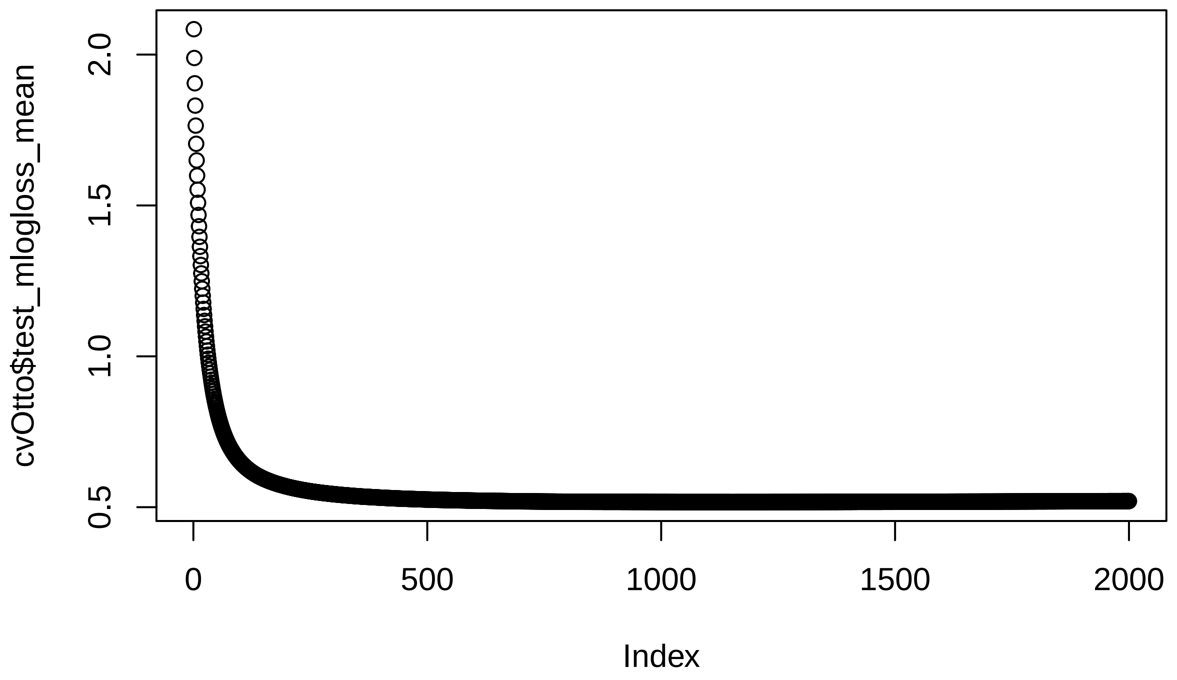
```r
#check the output to see the best set of iteration number

(cvOtto<-cvOtto$evaluation_log)
```

```
##       iter train_mlogloss_mean train_mlogloss_std test_mlogloss_mean
##    1:    1           2.0818717       0.0006851063          2.0841522
##    2:    2           1.9838655       0.0011081856          1.9886175
##    3:    3           1.8981045       0.0013917846          1.9048947
##    4:    4           1.8221022       0.0017414460          1.8308710
##    5:    5           1.7540457       0.0019137182          1.7645960
##   ---
## 1996: 1996           0.1556887       0.0007692407          0.5200858
## 1997: 1997           0.1556505       0.0007648269          0.5200947
## 1998: 1998           0.1556160       0.0007628752          0.5201075
## 1999: 1999           0.1555758       0.0007632340          0.5201310
## 2000: 2000           0.1555410       0.0007667082          0.5201398
##       test_mlogloss_std
##    1:      0.0008150075
##    2:      0.0016480652
##    3:      0.0019983965
##    4:      0.0025578429
##    5:      0.0031011818
##   ---
## 1996:      0.0116419054
## 1997:      0.0116668688
## 1998:      0.0116673023
## 1999:      0.0116726947
## 2000:      0.0116860319
```

```r
#best number of iterations
(bestNR = which.min(cvOtto$test_mlogloss_mean))
```

```
## [1] 1152
```

```r
#checking the charts to see what the optimal number of iterations should be
plot(cvOtto$test_mlogloss_mean)
```



Now, we are set to run the final model.

```
best_xg_model<-xgboost(param=param, data = xgbTrain, label = yTrain,
                                nrounds=1031,verbose=F,save_period=NULL)
```

We, can check the log loss of this model, since my test it reallt small it can throw off a non-conclusive value but it was highly helpful while I was trying to build the model.

```
#testing the train and test log loss values
xgbPred <- matrix(predict(best_xg_model, xgbTest), ncol = numClasses, byrow = TRUE)
xg_log_loss<-MultiLogLoss(yTest, xgbPred)
xg_log_loss
```

```
## numeric(0)
```

After this we had to create the output values in terms of probability and upload it to the course website to obtain our final score and grade. The below mentioned code, creates an output file in the format required for submission.

```
#storing the ID's in a separate variable
ID<-xgb_out_test[,1]

xgbPred <- matrix(predict(best_xg_model, xgb_out_test[,-1]), ncol = numClasses, byrow = TRUE)

head(xgbPred)
```

```
##              [,1]       [,2]        [,3]        [,4]         [,5]
## [1,] 0.0001263453 0.98912084 0.006995653 0.002658012 1.908621e-04
## [2,] 0.0002557819 0.26096672 0.568227947 0.164219290 1.116269e-04
## [3,] 0.0003524730 0.96493328 0.013663891 0.006741181 4.330680e-04
## [4,] 0.0008500041 0.92370576 0.053656187 0.014984104 3.921586e-04
## [5,] 0.0000551559 0.36654902 0.609212220 0.008603560 2.523673e-05
## [6,] 0.1928757280 0.07126565 0.120095491 0.036404308 1.417680e-02
##              [,6]         [,7]         [,8]         [,9]
## [1,] 0.0001534185 0.0001586089 0.0001722016 4.241279e-04
## [2,] 0.0002792451 0.0056450372 0.0001719113 1.224813e-04
## [3,] 0.0005324255 0.0118671786 0.0006365117 8.400111e-04
## [4,] 0.0005941805 0.0035871146 0.0003518111 1.878728e-03
## [5,] 0.0008453481 0.0145752337 0.0000833066 5.089653e-05
## [6,] 0.1847053617 0.1666623801 0.0681454465 1.456688e-01
```

```
#storing the IDs and prediction in a single data frame
output<-data.frame(ID, xgbPred)

#writing this file to the directory to upload to check the accuracy and all
write.csv(output,
          '/Users/mayank/Documents/College Documents/Q3 Courses/Machine Learning/Course Project/course_
          quote = F, row.names = F)
```

GridSearch

In this part we will employ grid search in order to automate this task of finding the optimal mix of hyper-parameters, in the script below I will run only a model for enumeration, however, we can pass different values and even sequences in the vectors of the hyper-parameters.

Grid search will basically accept a range of values of all the hyper parameterrs, and then loop it across them, then depending upon the result, it will get compared with the previous result and then it will be deicede whether to increase or decrease the value of that particular hyper-parameter. So in a nutshell a you are essentially creating a grid (imagine a 2 dimensional space where the x axis is para 1 and the y being parameter 2), thus we will be constricting our movement in this grid depending upon the best result.

Setting the parameters for grid search, here I will use the same parameters as before.

```
# xgboost task parameters
folds <- 5
eval_metric = list("mlogloss")  # evaluation metric for validation

cv.nfold <- 4
xg_gamma<-0

# Parameters grid to search
xg_min_child_weight<-20
xg_subsample<-0.7
xg_colsample_bytree<-0.5
xg_rowsample_bytree<-0.9
eta = c(.03,0.05)
max_depth<-8
#you can also set it up as
#max_depth = c(4,5,6,10)
nrounds <- 2000
lambda_grid<-6
#you can also set it up as
#lambda_grid<-c(seq(from=5,to=10,by=1))

alpha_grid<-7
#you can also set it up as
#alpha_grid<-c(seq(from=5,to=10,by=1))



# Table to track performance from each worker node
```

```r
res <- data.frame(Value=numeric(),Eta=numeric(),Max_Depth=numeric(),Nrounds=numeric(),
                  Lambda_grid=numeric(), alpha_grid=numeric())
```

Fitting the above mentioned parameters to the xgboost and then running a grid search

```r
# Fitting function for iterations
xgbCV <- function (Inputs) {
  myEta<-Inputs$eta
  myMax_depth<-Inputs$max_depth
  myNRounds<-Inputs$n_Rounds
  mylambda_grid<-Inputs$lambda_grid
  myalpha_grid=Inputs$alpha_grid
  #print the vals
  set.seed(0)
  fit <- xgb.cv(
    params =list(eta=myEta,max_depth=myMax_depth, lambda=mylambda_grid, alpha=myalpha_grid,
                 num_class = numClasses),
    data = xgbTrain,
    label = yTrain,
    metrics=eval_metric,
    objective = "multi:softprob",
    nfold = folds,
    nrounds = myNRounds,
    verbose=F
  )
  mybestNR = which.min(fit$evaluation_log$test_mlogloss_mean)
  val <- fit$evaluation_log$test_mlogloss_mean[mybestNR]
  res <<- rbind(res,c(val,myEta,myMax_depth,mybestNR, mylambda_grid,myalpha_grid))

  return(val)
}


#running grid search on through the above function
#the below command is running the grid search
sol <- gridSearch(
  fun = xgbCV,
  levels = list(eta=eta,max_depth=max_depth, lambda_grid=lambda_grid, alpha_grid=alpha_grid,
                n_Rounds=nrounds),
  method = 'loop',
  keepNames = TRUE,
  asList = TRUE
)
```

```
## 5 variables with 2, 1, 1, 1, ... levels: 2 function evaluations required.
```

Checking the output of the grid search

```r
levels<-list(eta=eta,max_depth=max_depth, lambda_grid=lambda_grid, alpha_grid=alpha_grid)

#optimal level of the parameters
sol$minlevels
```

```
## $eta
## [1] 0.03
##
```

```
## $max_depth
## [1] 8
##
## $lambda_grid
## [1] 6
##
## $alpha_grid
## [1] 7
##
## $n_Rounds
## [1] 2000
```

The output from the grid search algorithm can now be used to extract the optimal mix and values of the parameters and then finally the final model can be tuned.