STAT542: Statistical Learning **Fall 2015**

Final Project: Springleaf Marketing Response

Yu Dai, Yilin Zhao *NetID: yudai3, zhao100*

1. Project Introduction

1.1. Background

The project comes from *Springleaf Marketing Response* competition on Kaggle website, which asks us to determine whether to send a direct mail piece to a customer or not. We are provided a high-dimensional dataset of anonymized customer information. Each row corresponds to one customer and the response variable is binary and labeled "targe". The goal of us is to predict the target variable for every row in the test set.

1.2. Data Description

The train data set contains 145,231 observations and 1,934 variables and the test data set contains 145,232 observations and 1,933 variables, the difference of variables between the train data set and the test data set is that the train data set has a response variable labeled *target*.

1.3. Challenge

- The features in both data sets have been anonymized to protect privacy and are comprised of a mix of continuous and categorical features. The meaning of the features, their values, and their types are provided "as-is" for this competition, so handling a huge number of messy features is part of the challenge here.
- The data set is of high dimension, both read in data and train models based on the data cost a lot of time. So how to deal with large data set is another challenge here.

1.4. Algorithms

Here we use two boosting methods to build the models. One is Extreme Gradient Boosting (XGB), the other is Adaboost. The main reasons that we choose boosting algorithms are:

- Easy to implement.
- Does feature selection resulting in relatively simple classifier.
- Fairly good generalization.
- Well handle high dimensional spaces as well as large number of training examples.

2. Data Preprocessing

2.1 **Remove Duplicated and Irrelevant Columns**

As described above, there are 1934 features in total in the train data set, some of them are duplicated with one another. So we first created a summary file containing for each feature the variable name, type (integer, string, date,?), min, max, na count. Then, using the previous file, we deleted duplicated columns in both train data set and test data set. Additionally, from the feature names we can know that the first feature is ID, which is useless for our analysis, so we also removed it.

2.2 **Remove Columns with small standard deviation**

If a feature has small variance, then it means that it is a constant and had nothing to do with the variation of the response variable, so we can simply remove it. In our project we remove 41 variables with $sd < 0.00005$.

2.3 **Feature Engineering**

We extracted time, year and month, and days from the date features. Also after doing some searches on the kaggle forum, we know that feature 241 is interpreted as postal code and feature 254 is interpreted as age, we encoded them along with other categorical features, with the mean target. We also notice that there are some typos in city names, such as different city names may represent the same place. After fixing these mistakes the level of city name is reduced from 12387 to 10511.

2.4 **Missing Values**

Replace all the missing values with -9999. In categorical variables -9999 is treated as "-9999".

After data processing our final training data set contains 145231 observations and 1918 variables.

3. **Extreme Gradient Boosting**

3.1 **Introduction of XGBoost**

We use package *XGBoost* in R to apply the extreme gradient boosting algorithm, which is an efficient implementation of gradient boosting framework. The advantage of XGBoost package is that it includes efficient linear model solver and tree learning algorithms, and the package can automatically do parallel computation on a single machine which could be more than 10 times faster than existing gradient boosting packages. Because the data set for this project is really large, so computation time is a very important element to be considered here. It also supports various objective functions, and we use classification for this project.

3.2 **Data Preparation for XGBoost**

- Many competitors on the kaggle forum said that the XGBoost package can only deal with numerical data, so besides all the pre-processing part of the data sets, we need to transform all the categorical features into numerical ones first.

- After the transformation, we split the train data set into two subsamples: one called x.train with 70% of the total observations and the second one called x.test with the remaining 30% of the total observations. The reason is that we use *xgb.train* function to train XGB models. *xgb.train* is able to follow the progress of the learning after each round and we provide x.test data set to it, therefore it can learn on the x.train data set and test its model on the x.test data set.

3.3 **Parameter Setting and Model Training**

We try to use five different sets of model parameters to train five different models using the same x.train data set and x.test data set. First, we write a function so that we can easily change the parameter list, shown in Table 3.1:

```
>param.change=function(paramlist, new.eta, new.subsample, new.max_depth){
  +num=length(paramlist) + 1
  +paramlist[[num]]=paramlist[[1]]
  +paramlist[[num]]$eta=new.eta
  +paramlist[[num]]$subsample=new.subsample
  +paramlist[[num]]$max_depth=new.max_depth
  +paramlist
+}
```

Table 3.1

The function can change the value of eta, subsample and max_depth parameter. Here is the R code for the setting of the tree booster parameters:

```
>xg.params=list(
  +"objective"  = "binary:logistic",
  +"eval_metric" = "auc",
  +"eta" = 0.03,
  +"subsample" = 0.7,
  +"colsample_bytree" = 0.8,
  +"max_depth" = 6
+)
>param.list=list(xg.params)
>param.list=param.change(param.list, 0.05, 0.5, 8)
>param.list=param.change(param.list, 0.2, 0.8, 4)
>param.list=param.change(param.list, 0.03, 0.7, 8)
>param.list=param.change(param.list, 0.3, 0.65, 2)
```

Table 3.2

- Here the objective function is *binary:logistic* because our response variable is a 0/1 binary factor. The evaluation metrix is set to be "auc", which is short for "area under the ROC curve".
- *eta* controls the learning rate: scale the contribution of each tree by a factor of $0 < eta < 1$ wen it is added to the current approximation. We use 4 different low eta values: (0.03, 0.05, 0.2, 0.3), to prevent overfitting by making the boosting process more conservative.
- Subsample is the subsample ratio of the training instance, we use 4 different values for subsample: 0.5, 0.65, 0.7, 0.8, in order to prevent overfitting problem and also to make the computation shorter. Our idea is to lower the eta and subsample as well as increase n rounds, which is the max number of iterations.
- Max_depth is the maximum depth of a tree, we still use 4 different values: 2, 4, 6, 8.

As you can see from Table 3.2, using different combinations of the parameter values, we construct five groups of parameter settings.
Table 3.3 shows the R code for model training using *xgb.train* function:

```
clf=xgb.train(params=param.list[[idx]],
              data=x.train,
              nrounds=180,
              watchlist=list(validation=x.test),
              print.every.n=5,
              maximize=TRUE,
              early.stop.round=10)
```

Table 3.3

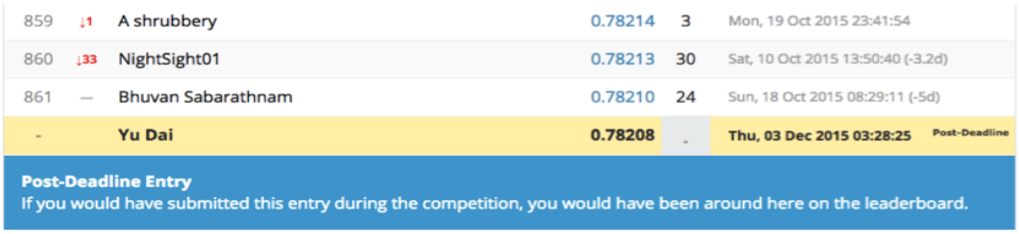
- The function in Table 3.3 is inside a loop to run the 5 different parameter settings and predict the probabilities for both the x.test data set and the test data set.
- The data set used to train the model is x.train, which is the 70% of the total observations of the original train data set, and the data set used for model validation is x.test, which is the remaining 30% of the original train data set.
- We set nrounds to be 180, not to large, in order to save some computation time, and early.stop.round is set to be 10, which means that the training process will stop of the performance keeps getting worse consecutively for 10 rounds.

3.4 **Fusing Models**

After the model training part, we save the prediction results of the five different XGBoost models of both x.test data set and test data set in two data frames. We use GAM (generalized additive model) to combine the five models. The idea is that we use the true target value of the x.test data set, y.test, as our new response, and the five columns of predictions of x.test data set as our new five features, and simply predict y.test using these five variables. Finally, we can use the combined model on the five groups of prediction of test data set to predict the target.

3.5 **Final Result**

The score of each XGBoost model is in the range of (0.7, 0.75), none of them exceeds 0.75. However, after fusing the five models, our final score is 0.782 on Kaggle, shown in Plot 3.1.



Plot 3.1. Submission of XGB

4. **Adaboost**

4.1 **Algorithm Introduction**

Adaboost is a good way to combine many other types of learning algorithms to improve their performance. The output of the other learning algorithms (*weak learners*) is combined into a weighted sum that represents the final output of the boosted classifier. The individual learners can be weak, but the final model can be proven to converge to a strong learner. Here we use *gbm* package in R with $distribution =' Adaboost'$. This package will improve the boosting methods using control of the learning rate, sub-sampling, and a decomposition for interpretation.

4.2 **Data Processing**

- Split the training data into 4 subsets because the original data set is quite large. Here each subset contains 36308 observations.
- The *gbm* package can only handle factors with level less than 1024. Here for those variables with more than 1024 levels we keep the top 1023 levels and merge the remain levels as 'other'.

4.3 **Algorithm Implement**

There are 5 important parameters in *gbm* function.

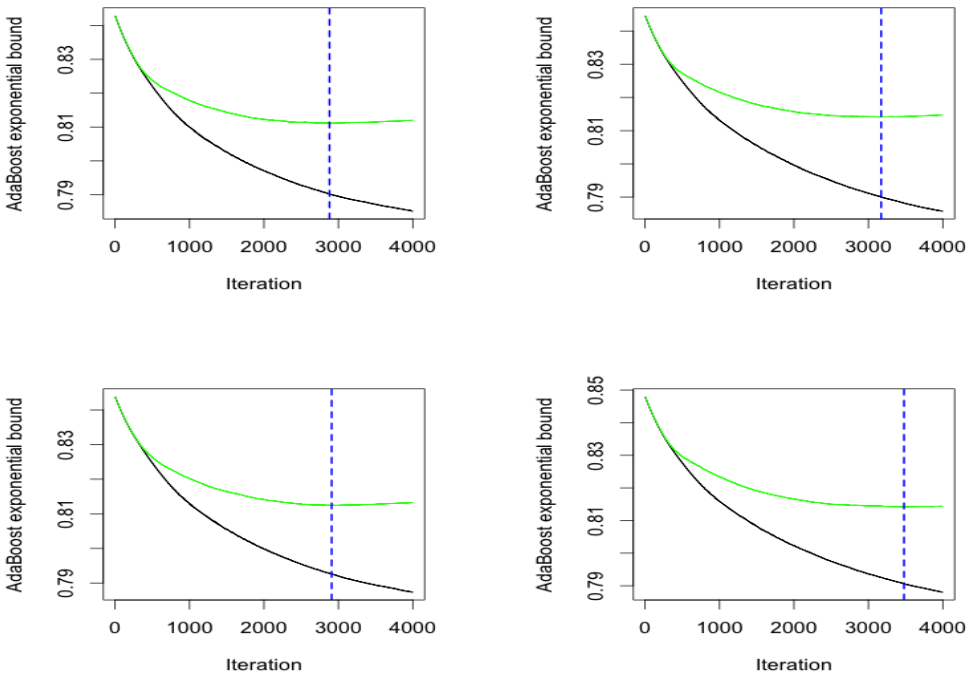
- The number of iterations, T (n.trees). We use 3 different values: 3000, 4000, 5000 and later find the exponential bound only converges after 3000 iterations so we set our $n.trees = 4000$.
- The depth of each tree, K (interaction.depth). Here we use the default value 1.
- The shrinkage (or learning rate) parameter, λ(shrinkage). We use 3 different values: 0.005, 0.001, 0.0001. Lower learning rate will increase the running time to converge. Consider the trade-off between accuracy and time complexity, we set $shrinkage = 0.001$.
- The subsampling rate, p (bag.fraction). We use the default value 0.5.
- The cross validation folder. We set $cv.folds = 3$,

In order to record the prediction results using different parameters we write a function, which is shown in Table 4.1:

```
> FunctionGBM = function(train, newtrain, test, test_pred, train_pred, best, cv, iter, LR){
+       boosting = gbm(target~.,distribution='adaboost', cv.folds=cv, data=train, n.trees=iter,bag.fraction = 0.5,
    shrinkage=LR)
+       n = gbm.perf(boosting, method="cv")
+       mytrain1 = predict(boosting,newtrain,type='response',n.trees = n)
+       train_pred = cbind(train_pred,mytrain1)
+       mytest1 = predict(boosting,test,type='response',n.trees = n)
+       test_pred = cbind(test_pred,mytest1)
+       best = c(best,n)
+       return(list(test_pred,train_pred,best))
+ }
```

Table 4.1

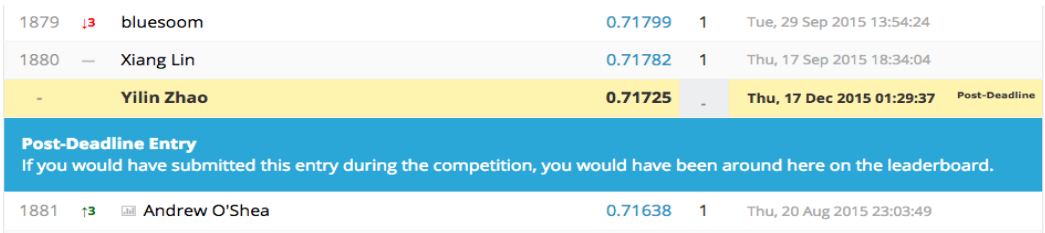
We generate 4 models using different training subsets and calculate the mean of 4 lists of prediction as the final result. Here are their error curve versus number of iterations, respectively.



Plot 4.1. Train Error and Valid Error v.s Number of iterations.

4.4 **Final Result**

The final score of Adaboost is 0.717 on Kaggle, shown in Plot 4.2.



Plot 4.2. Submission of Adaboost

5. **Summary**

Comparing these two algorithms we used, XGB gets less running time and higher accuracy. Adaboost will cost 7 hours to build a model with a single pair of parameters, which increases the difficulty to tune parameters. Hence, XGB might be a better classification algorithm for large data set.