Defaults of Credit Card in Taiwan

Chin Hooi Yap

Contents

1 Overview

This Project is part of the "Choose Your Own Project" for PH125.9x, HarvardX Professional Certificate in Data Science program on edX. I have chosen this data set as I am interested in data set related to financial industry. The objective is to build a classification model to predict the defaults of Credit Card customers in Taiwan based on the data provided. I will be using the tools and methods learned throughout the course and also part of my learning from my work experience.

1.1 Introduction

Prediction of default of credit card customers is a good practical example of supervise learning to predict the likelihood of default for each customer so that the financial institution can take actions with a risk based approach on customers who are more likely to default. As for this project, we will use the data set provided on Kaggle.

Kaggle Link: Default of Credit Card Clients Dataset

Firstly, we will start by reading and doing a quick exploration on the data. We will then perform exploratory data analysis to better understand the data. After that, we will perform pre-processing and feature engineering to prepare the data for modeling.

Subsequently, we will train a few machine learning algorithms using the processed data and use the split test set to validate the performance. Lastly, we will optimize the performance and compare the performances of different algorithms to select the model with the best performance.

We will output the following files at the end of this project:

1. A report in PDF format (by publishing this RMD file as pdf).
2. This report in RMD file.
3. A detailed R script.
4. Various models files that I used to do the prediction because it takes a very long time to retrain the models. They are uploaded to Github.

1.2 Objective of Project

This project aims to train a machine learning algorithm to predict the probability of default of credit card clients in Taiwan given the information such as Demographics, Payment status, Bill Amount and etc.

The evaluation metric that we will be using is AUC-ROC (AUC), the area under curve for Receiver Operating Characteristic (ROC) curve, which is one of the most widely used evaluation metrics for classification problems in machine learning. AUC-ROC of a classifier is equal to the probability that the classifier will rank a randomly chosen positive example higher than a randomly chosen negative example. ROC is essentially the plot of True Positive Rate (TPR) and False Positive Rate (FPR), which can be thought of as a benefit vs cost plot. AUC-ROC has a range of [0, 1]. The greater the value, the better is the performance of our model.

Refer to Classification metrics for more details.

We will experiment a few models and select the model that has the best AUC-ROC (i.e. highest AUC-ROC).

2 Obtaining the Data

We will download the credit card data using the code as follows. The csv file is pre-downloaded from kaggle to make the process smoother.

```
UCI_Credit_Card <- read_csv("UCI_Credit_Card.csv",col_names = TRUE)
UCI_Credit_Card<-data.frame(UCI_Credit_Card)
```

3 Method and Analysis

3.1 Data Exploration

We will start off by checking the data structure and the columns.

```
str(UCI_Credit_Card)
```

```
## 'data.frame':    30000 obs. of  25 variables:
##  $ ID                        : num  1 2 3 4 5 6 7 8 9 10 ...
##  $ LIMIT_BAL                 : num  20000 120000 90000 50000 50000 50000 500000 100000 140000 20000
##  $ SEX                       : num  2 2 2 2 1 1 1 2 2 1 ...
##  $ EDUCATION                 : num  2 2 2 2 2 1 1 2 3 3 ...
##  $ MARRIAGE                  : num  1 2 2 1 1 2 2 2 1 2 ...
##  $ AGE                       : num  24 26 34 37 57 37 29 23 28 35 ...
##  $ PAY_0                     : num  2 -1 0 0 -1 0 0 0 0 -2 ...
##  $ PAY_2                     : num  2 2 0 0 0 0 0 -1 0 -2 ...
##  $ PAY_3                     : num  -1 0 0 0 -1 0 0 -1 2 -2 ...
##  $ PAY_4                     : num  -1 0 0 0 0 0 0 0 0 -2 ...
##  $ PAY_5                     : num  -2 0 0 0 0 0 0 0 0 -1 ...
##  $ PAY_6                     : num  -2 2 0 0 0 0 0 -1 0 -1 ...
##  $ BILL_AMT1                 : num  3913 2682 29239 46990 8617 ...
##  $ BILL_AMT2                 : num  3102 1725 14027 48233 5670 ...
##  $ BILL_AMT3                 : num  689 2682 13559 49291 35835 ...
##  $ BILL_AMT4                 : num  0 3272 14331 28314 20940 ...
##  $ BILL_AMT5                 : num  0 3455 14948 28959 19146 ...
##  $ BILL_AMT6                 : num  0 3261 15549 29547 19131 ...
##  $ PAY_AMT1                  : num  0 0 1518 2000 2000 ...
##  $ PAY_AMT2                  : num  689 1000 1500 2019 36681 ...
##  $ PAY_AMT3                  : num  0 1000 1000 1200 10000 657 38000 0 432 0 ...
##  $ PAY_AMT4                  : num  0 1000 1000 1100 9000 ...
##  $ PAY_AMT5                  : num  0 0 1000 1069 689 ...
##  $ PAY_AMT6                  : num  0 2000 5000 1000 679 ...
##  $ default.payment.next.month: num  1 1 0 0 0 0 0 0 0 0 ...
```

Next, we check on the missing values. We can see that there is no missing values in the data set.

```
colSums(sapply(UCI_Credit_Card, is.na))
```

```
##                         ID                  LIMIT_BAL
##                          0                          0
##                        SEX                  EDUCATION
```

```
##                         0                         0
##                  MARRIAGE                       AGE
##                         0                         0
##                     PAY_0                     PAY_2
##                         0                         0
##                     PAY_3                     PAY_4
##                         0                         0
##                     PAY_5                     PAY_6
##                         0                         0
##                  BILL_AMT1                 BILL_AMT2
##                         0                         0
##                  BILL_AMT3                 BILL_AMT4
##                         0                         0
##                  BILL_AMT5                 BILL_AMT6
##                         0                         0
##                   PAY_AMT1                  PAY_AMT2
##                         0                         0
##                   PAY_AMT3                  PAY_AMT4
##                         0                         0
##                   PAY_AMT5                  PAY_AMT6
##                         0                         0
## default.payment.next.month
##                         0
```

We also check whether this is an imbalance data set, in this case, it refers to the number of 1s in the target variable default.payment.next.month. There are 22% of labels with 1, which tells us that the data set is slightly imbalanced.

```
sum(UCI_Credit_Card$default.payment.next.month)/nrow(UCI_Credit_Card)
```

```
## [1] 0.2212
```

Besides that, we are removing the ID columns as it is not meaningful to our analysis.

```
UCI_Credit_Card <- UCI_Credit_Card%>%select(-ID)
```

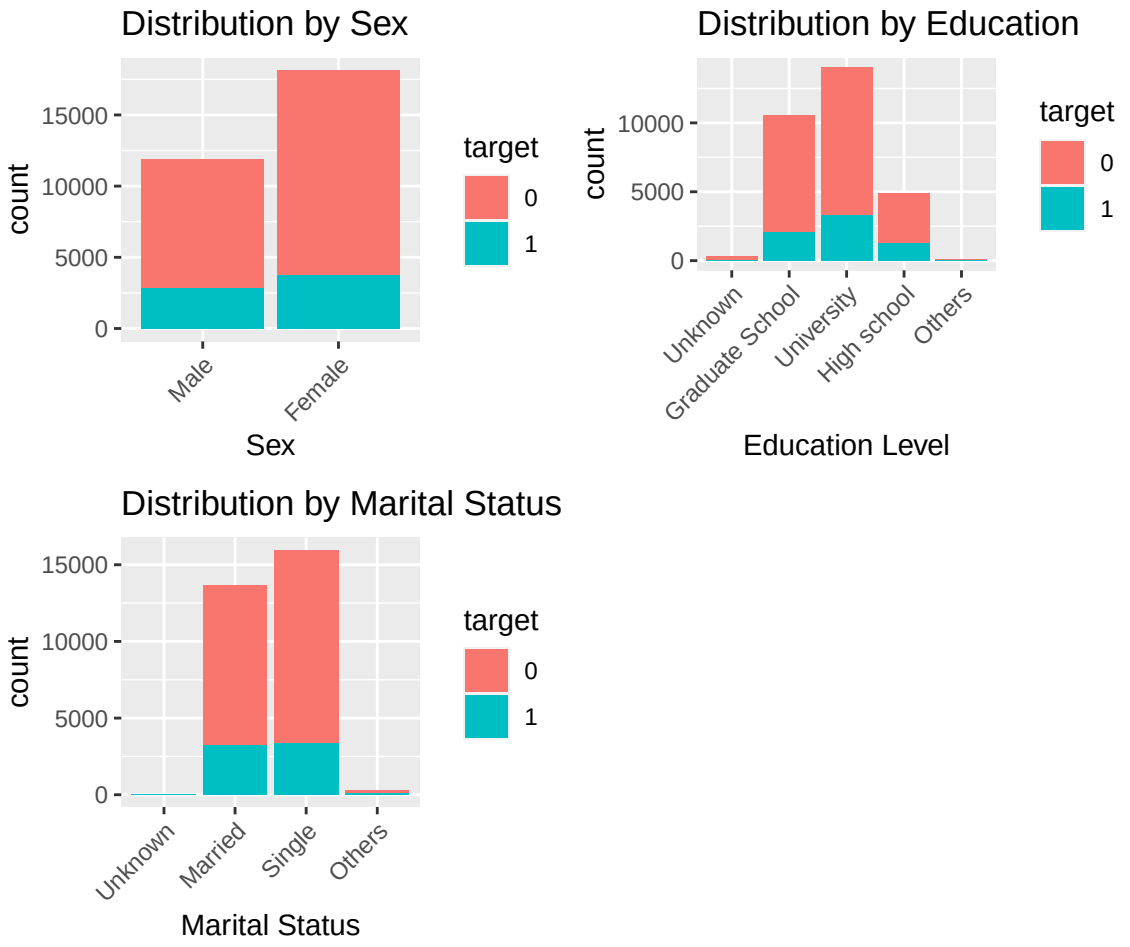
3.2 Pre-processing and Exploratory Data Analysis

Next, we will now proceed to exploratory data analysis. We will start by plotting a few visualizations to gain more insights of the data. Do note that before we plot certain graphs, we will be doing some simple pre-processing to improve the readability of the visualizations. Besides, values that are not listed in the data dictionary are put to "Unknown" for Education and Marriage.

The following plots show that there is not much relationship between Sex, Education and Marriage to Default.Payment.Next.Month (target).
We can see that the default payments is distributed almost evenly across genders, Marital Status and Education Level.

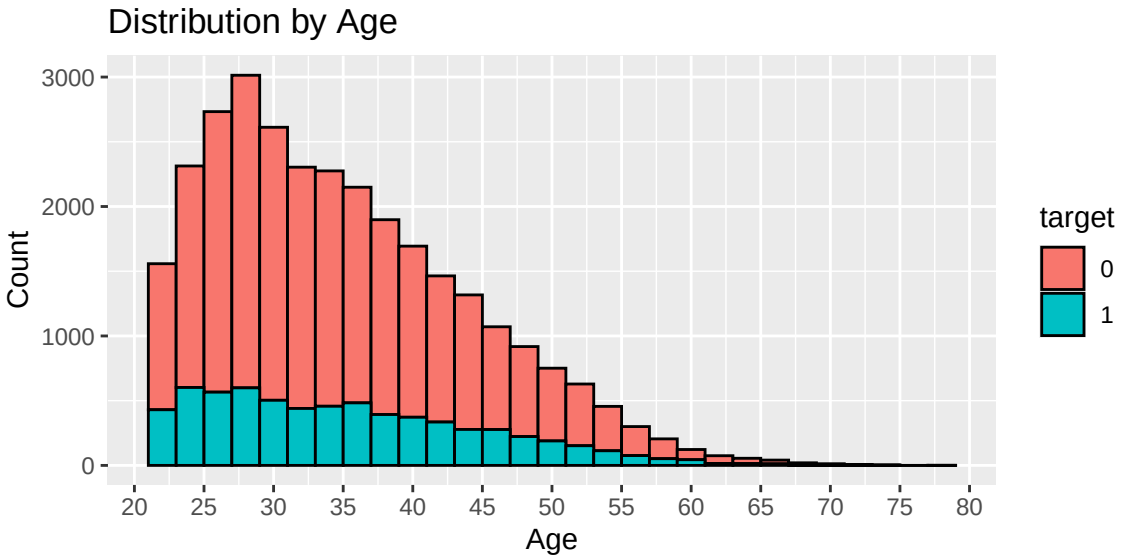
```
graph1<-UCI_Credit_Card%>%ggplot(aes(x=SEX,fill = target))+ geom_bar()+
  labs(title = "Distribution by Sex", x ="Sex",fill = "target") +
  theme(axis.text.x = element_text(angle = 45,hjust=1))
```

```
graph2<-UCI_Credit_Card%>%ggplot(aes(x=EDUCATION,fill = target))+ geom_bar()+
  labs(title = "Distribution by Education", x ="Education Level",fill = "target") +
  theme(axis.text.x = element_text(angle = 45,hjust=1))

graph3<-UCI_Credit_Card%>%ggplot(aes(x=MARRIAGE,fill = target))+ geom_bar()+
  labs(title = "Distribution by Marital Status", x ="Marital Status",fill = "target") +
  theme(axis.text.x = element_text(angle = 45,hjust=1))

grid.arrange(graph1,graph2,graph3,ncol=2)
```



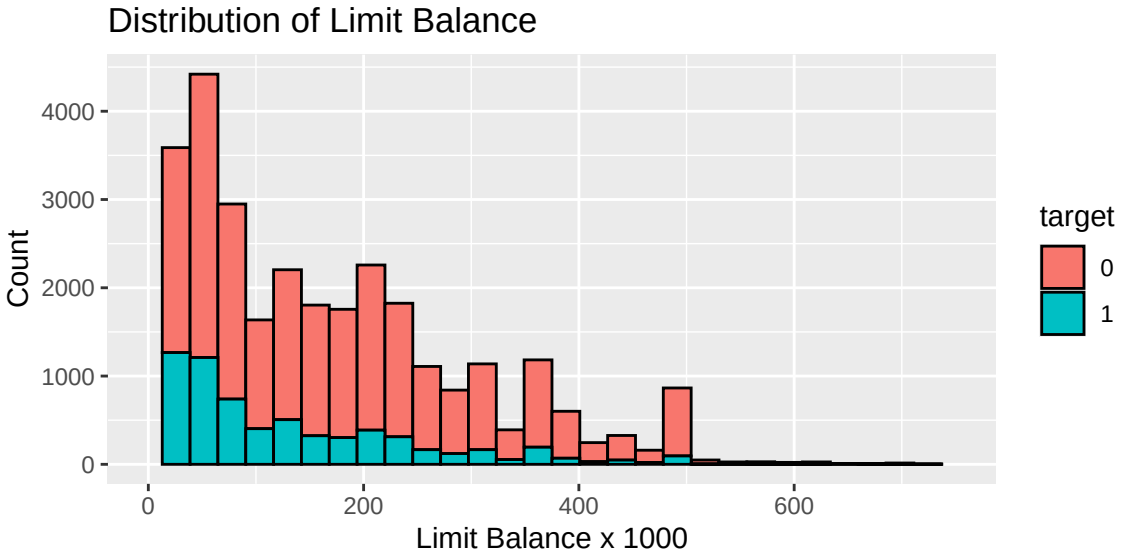
Next, we will study the distribution of Age to default payments. We can also see that the distribution of default group is similar to that of the non-default group. Hence, it shows that there is no direct relationship between Age and Default payment.

```
UCI_Credit_Card%>% ggplot(aes(x=AGE,fill = target))+
  geom_histogram(bins=30, color = "black")+
  scale_x_continuous(breaks = seq(min(0), max(90), by = 5), na.value = TRUE)+
labs(title = "Distribution by Age", x ="Age", y ="Count")
```



We will now study the relationship between limit balance and default payment. The graph shown that the ratio of default clients to the non-default is about the same and distribution is similar.

```
ggplot(aes(x = UCI_Credit_Card$LIMIT_BAL/1000), data = UCI_Credit_Card) +
  geom_histogram(aes(fill = UCI_Credit_Card$target),col="black") +
  labs(title = "Distribution of Limit Balance", x ="Limit Balance x 1000",
       y ="Count",fill = "target")+
  xlim(c(0,750))
```



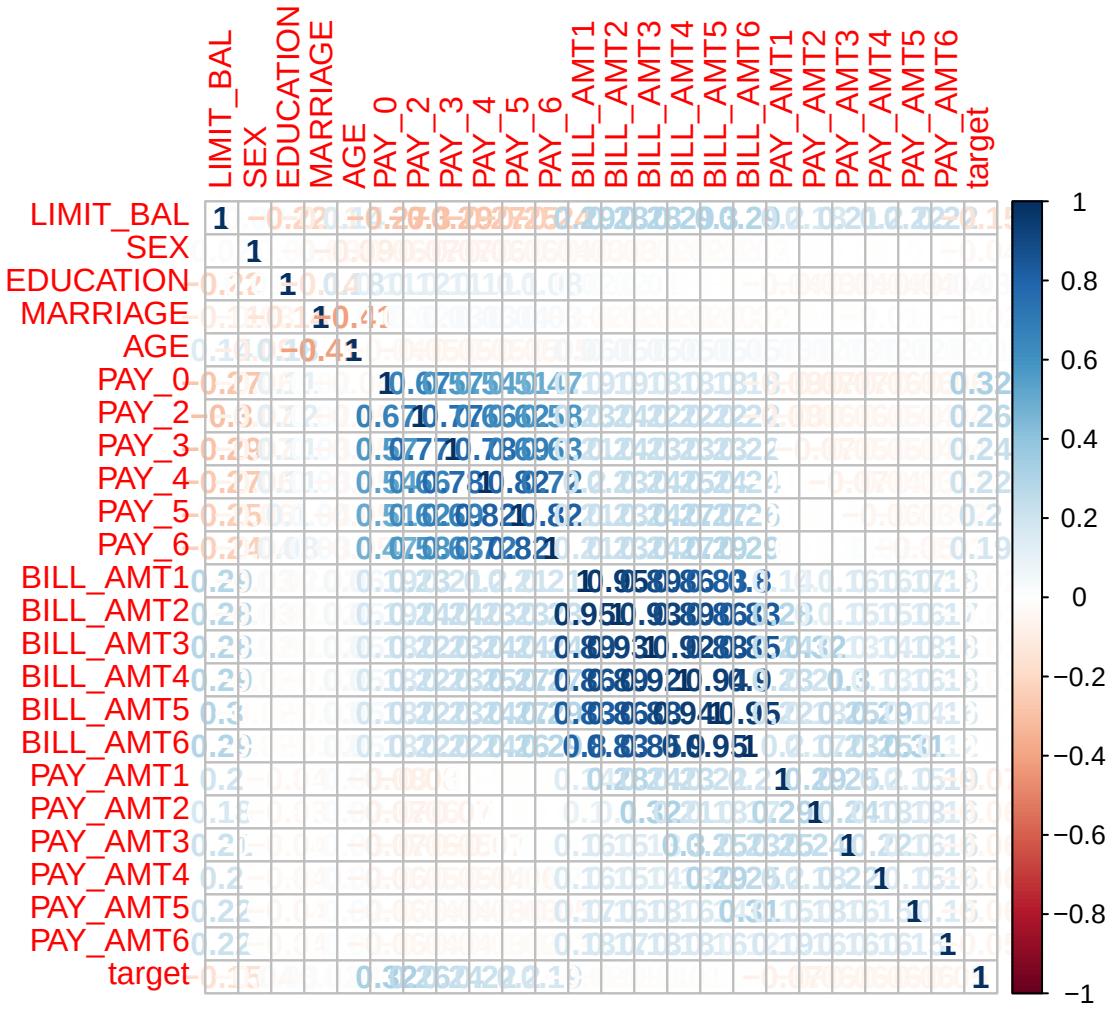
Next, we will make a correlation plot to study the correlation between features.

```
#reload data without preprocessing to ensure that they are numeric for correlation plot
UCI_Credit_Card_cor <- read_csv("UCI_Credit_Card.csv",col_names = TRUE)
UCI_Credit_Card_cor<-data.frame(UCI_Credit_Card_cor)%>%select(-ID)
```

We can see that both Payment status and Bill amount are highly correlated to each other, this is understandable as the bill amount are cumulated if it is not paid off. It is also interesting to note that our target

variable (default.payment.next.month) is somewhat correlated to payment status (PAY_0 to PAY_6)

```
#rename default.payment.next.month to "target" to improve the readibility
colnames(UCI_Credit_Card_cor)[colnames((UCI_Credit_Card_cor)) == "default.payment.next.month"] = "target
M <- cor(subset(UCI_Credit_Card_cor, select = colnames(UCI_Credit_Card_cor)))
corrplot(M, method="number")
```



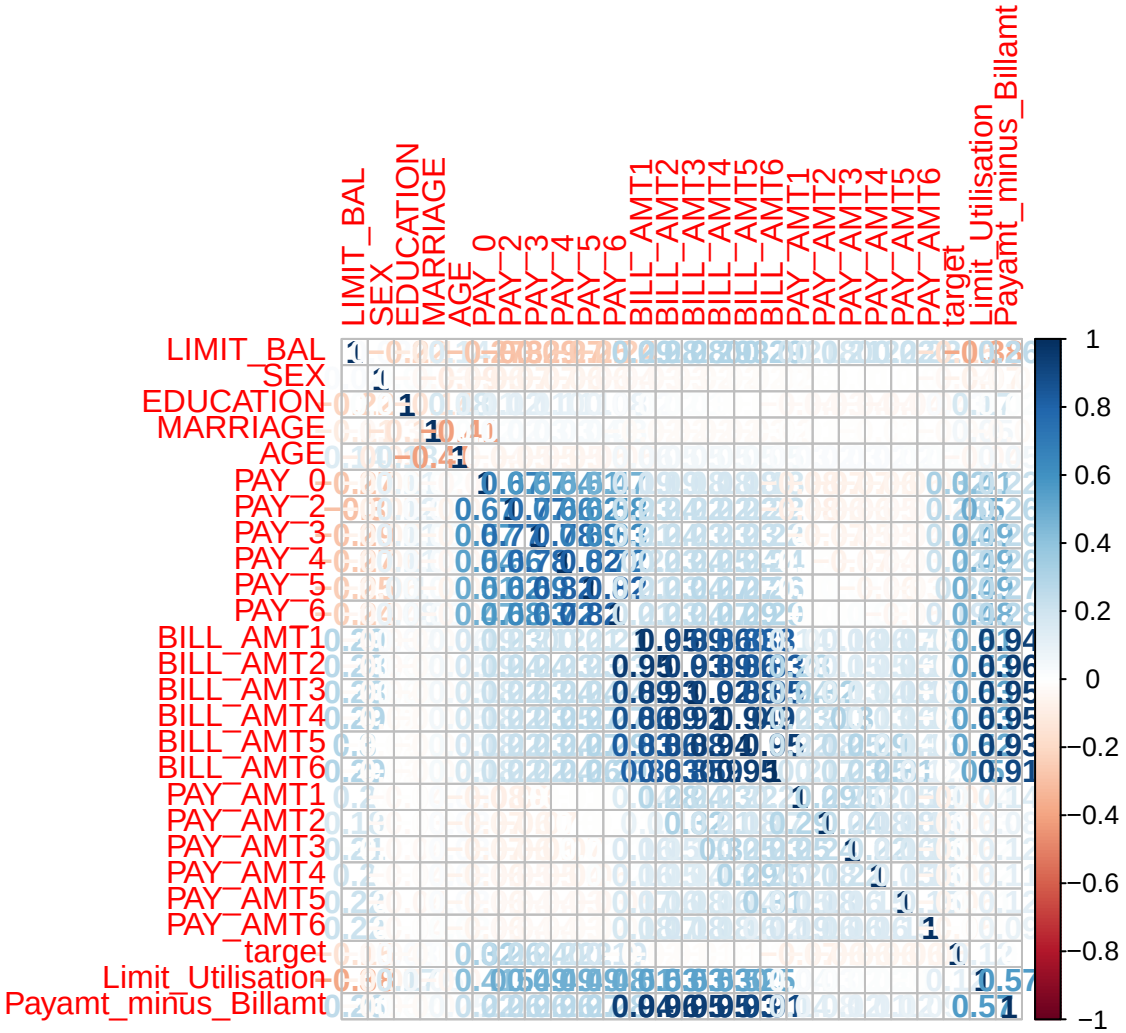
3.3 Feature Engineering

We will now perform feature engineering to create more useful features for our models. To fully utilize the features provided in the data set, the following new features have been generated,

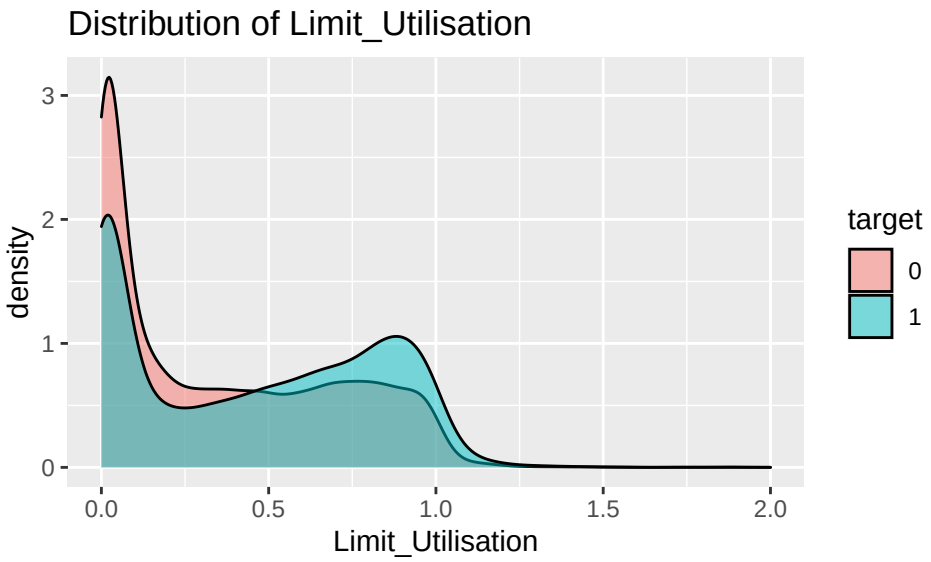
- Payamt_minus_Billamt - to calculate the difference between Payment amount (rowsum) and Bill amount (rowsum) to obtain the outstanding debt.
- Limit_Utilisation - to calculate the ratio of Bill amount (rowsum) to the credit balance.

```r
#creating intermediate columns Billamt_rowsum and Payamt_rowsum to generate the features
UCI_Credit_Card$Billamt_rowsum<-rowSums(UCI_Credit_Card[grep("BILL_AMT", names(UCI_Credit_Card))])
UCI_Credit_Card$Payamt_rowsum<-rowSums(UCI_Credit_Card[grep("PAY_AMT", names(UCI_Credit_Card))])

#new features
UCI_Credit_Card$Limit_Utilisation<-UCI_Credit_Card$Billamt_rowsum/(UCI_Credit_Card$LIMIT_BAL*6)
UCI_Credit_Card$Payamt_minus_Billamt<-UCI_Credit_Card$Billamt_rowsum-UCI_Credit_Card$Payamt_rowsum

#remove intermediate columns before next step modeling
UCI_Credit_Card<-UCI_Credit_Card%>%select(-Payamt_rowsum,-Billamt_rowsum)
```

Besides, we also perform a sanity check on the correlation of the new features with the others. It can be seen that Limit Utilitsation does have certain correlation with target.



A density plot below further demonstrated the relationship between the Limit Utilisation with the target. The profile of Limit Utilisation for both groups are similar for Limit Utilisation <0.25 but clients with Limit Utilisation>0.5 seems to have higher tendency to default.



3.4 Modelling

3.4.1 Modelling techniques

To train a good model, we will need to go through the following steps:

1. Pre-processing - pre-process the data to ensure a clean and consistent data frame are input to the model
2. Feature Engineering - Adding more useful features and do experiments to validate the performance.
3. Feature Selection - Select the top features input to the models and remove the remaining to reduce the noise to the model.
4. Hyperparameters tuning - Tune the hyperparameters of the respective models to ensure that our models are optimized.

In this project, we have selected Logistic Regression, Random Forest and XGBoost as our algorithms to study. These three models are known to be very powerful for classification problem in supervised machine learning. We will proceed to train three mentioned algorithms with their default parameters to obtain their respective baseline model.

3.4.2 Baseline Model - Logistic Regression

Let us start off by training a Logistic Regression model with the default parameters to obtain a baseline model.

```
glm1 <- glm(y_train_first ~ ., data=X_train_first, family=binomial)
```

Subsequently, we will validate the performance by calculating various metrics such as confusion matrix (using threshold of 0.5), best accuracy, AUC by using the previously split test data.

```
#To solve the PAY_2 and PAY_5 have new levels 8 error (due to splitting)
glm1$xlevels[["PAY_2"]] <- union(glm1$xlevels[["PAY_2"]], levels(X_test_first$PAY_2))
glm1$xlevels[["PAY_5"]] <- union(glm1$xlevels[["PAY_5"]], levels(X_test_first$PAY_5))
```

```
#prediction with model
glm1_pred<-predict(glm1, X_test_first, type="response")
glm_ROCRpred <- prediction(glm1_pred, y_test_first)

#confusion matrix
# use caret and compute a confusion matrix
pred_0.5_glm1 <- as.factor(as.numeric(glm1_pred>0.5))
cm_glm1<-confusionMatrix(data = pred_0.5_glm1, reference = y_test_first,positive='1')

#AUC curve and accuracy
glm1_perf <- performance(glm_ROCRpred, "tpr", "fpr")
glm1_perf_auc <- performance(glm_ROCRpred, measure = "auc")
glm1_perf_acc <- performance(glm_ROCRpred, measure = "acc")
glm1_auc<-glm1_perf_auc@y.values[[1]]

# Get best accuracy and cutoff
ind <- which.max( slot(glm1_perf_acc, "y.values")[[1]] )
glm1_acc <- slot(glm1_perf_acc, "y.values")[[1]][ind]
glm1_cutoff <- slot(glm1_perf_acc, "x.values")[[1]][ind]

Model_performance <- data.frame(Model= "Logistic Regression Baseline Model",
                                Best_Accuracy=glm1_acc,
                                #Cut_off=glm1_cutoff,
                                AUC=glm1_perf_auc@y.values[[1]],
                                Sensitivity=cm_glm1$byClass['Sensitivity'],
                                Specificity=cm_glm1$byClass['Specificity']
)
```

It can be seen that Logistic Regression's AUC performance is 0.465, not a very good model considering it is worse than random guessing (AUC=0.5).

```
Model_performance %>% knitr::kable()
```

	Model	Best_Accuracy	AUC	Sensitivity	Specificity
Sensitivity	Logistic Regression Baseline Model	0.7787965	0.4646265	0.8274303	0.1581764

3.4.3 Baseline Model - Random Forest

Next we will be training Random Forest baseline model.

```
rf1 <- randomForest(y_train_first~.,data=X_train_first, importance=T,
                            ntree=500, keep.forest=T)
```

Obtaining the various performance metrics by using the previously split test data.

```
#predict with test data, obtain the probability
rf1_prob <- predict(rf1, X_test_first, type='prob')[,2]
pred_0.5_rf1 <- as.factor(as.numeric(rf1_prob>0.5))

#COnfusion matrix
```

```
cm_rf1<-confusionMatrix(pred_0.5_rf1, y_test_first, positive='1')

#AUC and accuracy
rf1_ROCRpred <- prediction(rf1_prob,y_test_first)
rf1_perf <- performance(rf1_ROCRpred,"tpr","fpr")
rf1_perf_auc <- performance(rf1_ROCRpred, measure = "auc")
rf1_perf_acc <- performance(rf1_ROCRpred, measure = "acc")

# Get best accuracy and cutoff
ind <- which.max( slot(rf1_perf_acc, "y.values")[[1]] )
rf1_acc <- slot(rf1_perf_acc, "y.values")[[1]][ind]
rf1_cutoff <- slot(rf1_perf_acc, "x.values")[[1]][ind]

Model_performance <- bind_rows(Model_performance,
                               data.frame(Model= "Random Forest Baseline Model",
                                          Best_Accuracy=rf1_acc,
                                          #Cut_off=rf1_cutoff,
                                          AUC=rf1_perf_auc@y.values[[1]],
                                          Sensitivity=cm_rf1$byClass['Sensitivity'],
                                          Specificity=cm_rf1$byClass['Specificity']
                               ))
```

It can be seen that Random Forest has a much better baseline AUC performance.

```
Model_performance %>% knitr::kable()
```

Model	Best_Accuracy	AUC	Sensitivity	Specificity
Logistic Regression Baseline Model	0.7787965	0.4646265	0.8274303	0.1581764
Random Forest Baseline Model	0.8221370	0.7723476	0.3798041	0.9441353

3.4.4 Baseline Model - XGBoost

We will now train our last baseline model with XGBoost.

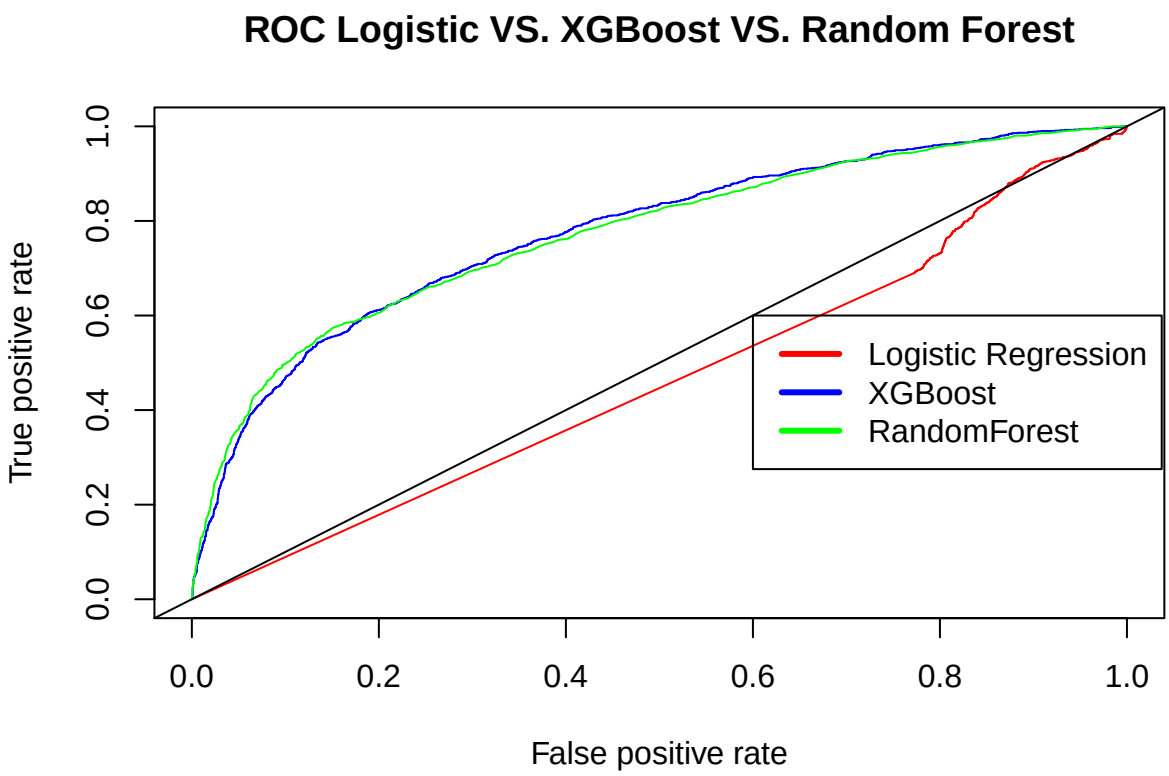
```
params <- list(booster = "gbtree", objective = "binary:logistic",
               nrounds = 100,
               eta=0.3,
               gamma=0,
               max_depth=6,
               min_child_weight=1,
               subsample=1,
               colsample_bytree=1)

Mat1 <- data.matrix(X_train_first)

dtrain <- xgb.DMatrix(Mat1, label = (as.numeric(y_train_first)-1))
xgb1 <- xgb.train (params = params,
                   data = dtrain,
                   nrounds = 100,
                   print_every_n = 10, early_stop_round = 10,
                   maximize = F , eval_metric = "auc")
```

Obtaining the various performance metrics by using the previously split test data.

```
Mat2<-data.matrix(X_test_first)

dtest <- xgb.DMatrix(Mat2, label = y_test_first)
xgb1_pred <- predict (xgb1,dtest)
xgb1_pred_confusion <- ifelse (xgb1_pred > 0.5,1,0)
#confusion matrix
library(caret)
xgb1_pred_confusion<-as.factor(xgb1_pred_confusion)
cm_xgb1<- confusionMatrix (xgb1_pred_confusion, y_test_first, positive='1')

#AUC and Accuracy
xgb1_ROCRpred <- prediction(xgb1_pred,y_test_first)
xgb1_perf <- performance(xgb1_ROCRpred,"tpr","fpr")
xgb1_perf_auc <- performance(xgb1_ROCRpred, measure = "auc")
xgb1_perf_acc <- performance(xgb1_ROCRpred, measure = "acc")

# Get best accuracy and cutoff
ind <- which.max( slot(xgb1_perf_acc, "y.values")[[1]] )
xgb1_acc <- slot(xgb1_perf_acc, "y.values")[[1]][ind]
xgb1_cutoff <- slot(xgb1_perf_acc, "x.values")[[1]][ind]

Model_performance <- bind_rows(Model_performance,
                               data.frame(Model= "XGBoost Baseline Model",
                                          Best_Accuracy=xgb1_acc,
                                          #Cut_off=xgb1_cutoff,
                                          AUC=xgb1_perf_auc@y.values[[1]],
                                          Sensitivity=cm_xgb1$byClass['Sensitivity'],
                                          Specificity=cm_xgb1$byClass['Specificity']
                               ))
```

Comparing the performance of these 3 baseline models, we can see that Logistic Regression is not performing well for this kind of slightly imbalanced data set. Hence, for the subsequent study, we will focus on Random Forest and XGBoost.

Model	Best_Accuracy	AUC	Sensitivity	Specificity
Logistic Regression Baseline Model	0.7787965	0.4646265	0.8274303	0.1581764
Random Forest Baseline Model	0.8221370	0.7723476	0.3798041	0.9441353
XGBoost Baseline Model	0.8168028	0.7746467	0.3843255	0.9389983



3.4.5 Random Forest - Hyperparameter Tuning

We will now tune the hyperparameters of the Random Forest model to obtain the most optimal performance of the model.

```
#Create control function for training with 5 folds, search method is grid.
control <- trainControl(method='cv',
                        number=5,
                        #repeats=3,
                        search='grid')
#create tunegrid with 15 values from 7:20 for mtry to tunning model. Our train function will change num
tune_grid <- expand.grid(#ntree = c(500,800,1000),
                         mtry = (7:20)
                         )

rf_tune1 <- train(X_train_first,y_train_first,
                        method = 'rf',
                        metric = 'Accuracy',
                        tuneGrid = tune_grid)
```

It can be seen that the performance only improve very slightly after tuning the hyperparameters.

Model	Best_Accuracy	AUC	Sensitivity	Specificity
Logistic Regression Baseline Model	0.7787965	0.4646265	0.8274303	0.1581764
Random Forest Baseline Model	0.8221370	0.7723476	0.3798041	0.9441353
XGBoost Baseline Model	0.8168028	0.7746467	0.3843255	0.9389983
Random Forest Tuned Model	0.8219703	0.7761422	0.3896006	0.9434932

3.4.6 XGBoost - Hyperparameter Tuning

We will now tune the hyperparameters of the XGBoost model to obtain the most optimal performance of the model.

```
tune_grid <- expand.grid(nrounds = c(300,500,600),
                         max_depth = c(3,6,8),
                         eta = c(0.1,0.3),
                         gamma = c(0,3),
                         colsample_bytree = c(0.6,0.8,1),
                         min_child_weight = c(1,3),
                         subsample = c(0.6,0.8,1))


 tune_grid <- expand.grid(nrounds = c(100),
                          max_depth = c(6),
                          eta = c(0.3),
                          gamma = c(0),
                          colsample_bytree = c(1),
                          min_child_weight = c(1),
                          subsample = c(1))

trctrl <- trainControl(method = "cv", number = 5)
```

We can see a good improvement of AUC from 0.77 to 0.79 after tuning the hyperparameter of XGBoost.

Model	Best_Accuracy	AUC	Sensitivity	Specificity
Logistic Regression Baseline Model	0.7787965	0.4646265	0.8274303	0.1581764
Random Forest Baseline Model	0.8221370	0.7723476	0.3798041	0.9441353
XGBoost Baseline Model	0.8168028	0.7746467	0.3843255	0.9389983
Random Forest Tuned Model	0.8219703	0.7761422	0.3896006	0.9434932
XGBoost Tuned Model	0.8243041	0.7902259	0.3820648	0.9492723

3.4.7 Random Forest (Tuned) - Feature Selection 15

Next, we will select the top 15 features base on the random forest tuned model and retrain it. This is to also test out the performance after removing the less important features which could cause noise to the model.

```
importance_rf_tune1 <- varImp(rf_tune1, scale=FALSE)
#select top 15 features
features_selected<- rownames(importance_rf_tune1$importance)[1:15]
X_train_fselect<-X_train_first%>%select(features_selected)
X_test_fselect<-X_test_first%>%select(features_selected)
```

```
control <- trainControl(method='cv',
                        number=5,
                        #repeats=3,
                        search='grid')
#create tunegrid with values from 7:12 for mtry to tunning model. Our train function will change number
tune_grid <- expand.grid(
  mtry = (7:12)
)

rf_tune1_fselect <- train(X_train_fselect,y_train_first,
                  method = 'rf',
                  metric = 'Accuracy',
                  tuneGrid = tune_grid)
```

The performance is slightly worse off than the original tuned model.

Model	Best_Accuracy	AUC	Sensitivity	Specificity
Logistic Regression Baseline Model	0.7787965	0.4646265	0.8274303	0.1581764
Random Forest Baseline Model	0.8221370	0.7723476	0.3798041	0.9441353
XGBoost Baseline Model	0.8168028	0.7746467	0.3843255	0.9389983
Random Forest Tuned Model	0.8219703	0.7761422	0.3896006	0.9434932
XGBoost Tuned Model	0.8243041	0.7902259	0.3820648	0.9492723
Random Forest Tune_fselect15	0.8179697	0.7614389	0.3873399	0.9366438

3.4.8 XGBoost (Tuned) - Feature Selection 15

Lastly, we will select the top 15 features base on the XGB tuned model and retrain it. This is to also test out the performance after removing the less important features which could cause noise to the model.

```
importance_xgb_tune1 <- varImp(xgb_tune1, scale=FALSE)
features_selected_xgb_tune1<- rownames(importance_xgb_tune1$importance)[1:15]
X_train_fselect<-X_train_first%>%select(features_selected_xgb_tune1)
X_test_fselect<-X_test_first%>%select(features_selected_xgb_tune1)
```

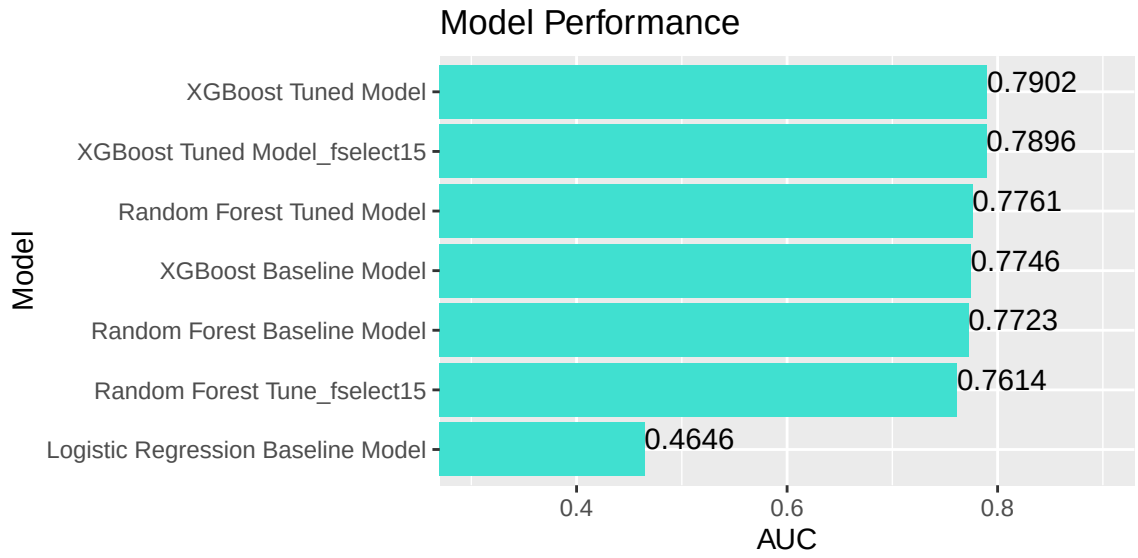
```
tune_grid <- expand.grid(nrounds = c(300,500,600),
                         max_depth = c(3,6,8),
                         eta = c(0.1,0.3),
                         gamma = 3,
                         colsample_bytree = c(0.8,1),
                         min_child_weight = c(1,3),
                         subsample = 1)



trctrl <- trainControl(method = "cv", number = 5)

Mat_X_train_fselect <- data.matrix(X_train_fselect)
xgb_tune1_fselect <- caret::train(Mat_X_train_fselect,y_train_first, method = "xgbTree",
                          metric="Accuracy",
                          trControl=trctrl,
                          tuneGrid = tune_grid)
```

The performance is very much comparable with the original tuned model.

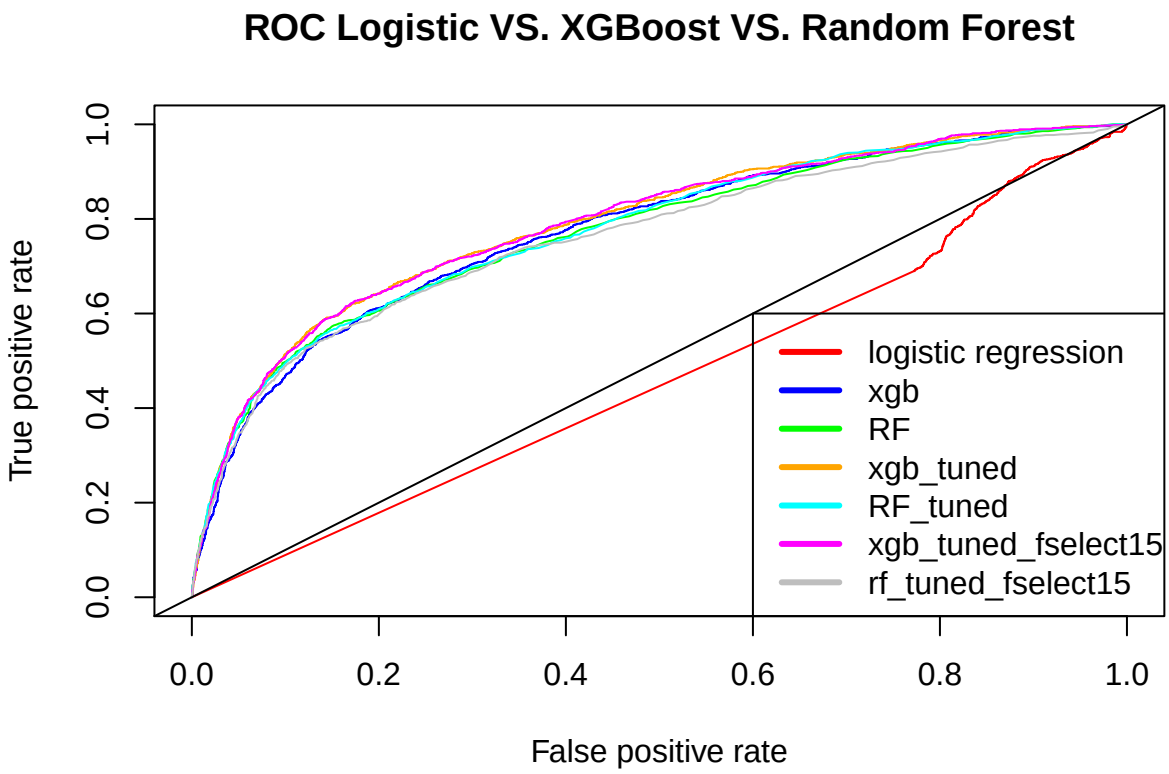
Model	Best_Accuracy	AUC	Sensitivity	Specificity
Logistic Regression Baseline Model	0.7787965	0.4646265	0.8274303	0.1581764
Random Forest Baseline Model	0.8221370	0.7723476	0.3798041	0.9441353
XGBoost Baseline Model	0.8168028	0.7746467	0.3843255	0.9389983
Random Forest Tuned Model	0.8219703	0.7761422	0.3896006	0.9434932
XGBoost Tuned Model	0.8243041	0.7902259	0.3820648	0.9492723
Random Forest Tune_fselect15	0.8179697	0.7614389	0.3873399	0.9366438
XGBoost Tuned Model_fselect15	0.8244707	0.7895887	0.3828184	0.9484161

3.5 Model Selection

Judging from the performances of the Models, both XGBoost and Random Forest have a much better performance than logistic regression. Among which, XGBoost slightly outperforms Random Forest and have a higher AUC. In this case, feature selection does not help in improving the performance, i.e. almost comparable for XGBoost and even worse off performance for Random Forest.



A closer look at the AUC plot comparison for all the models built,



4 Conclusion

We have built a series of useful machine learning models to predict the probability of default of the credit card clients in Taiwan. It can be seen that after doing various experiments with algorithms such as Logistic Regression, Random Forest and XGBoost, XGBoost comes up to be the best performer, closely followed by Random Forest. The final model we select is the XGboost tuned model, as it has the best AUC (0.7902) which is acceptable as a useful model as the AUC is close to 0.8. Last but not least, for future improvement, we can consider using techniques such as under-sampling and over-sampling to tackle the slightly imbalance data set or testing other machine learning models which could also improve the results further.

5 Acknowledgement

Lichman, M. (2013). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.

Original Dateset from UCI Machine Learning Repository: UCI Archive