Credit Card Fraud Prediction

Credit card fraud is a serious and growing problem every day. With the increase of credit cards as a means of payment, due to the rapid increase in online sales and the change in people's payment behavior, there has been an increased exposure to fraud.

In addition, over the years and with the evolution of fraud detection methods, fraudsters have also evolved. Fraud perpetrators have also evolved their practices to avoid detection, so that fraud is dynamic. Fraud is dynamic, it is always changing. Consequently, fraud detection methods need to be constantly improved.

Within fraud behavior, you want to find invariants or patterns that allow you to predict fraud at the transactional level. to predict fraud at the transactional level, i.e., to find strange buying behavior based on the purchase history of the customer. This refers to finding strange buying behaviors based on the customer's transactional history and incoming transaction data.

Objective

Our aim is to predict fraudulent credit card transactions in the dataset using synthetic balancing techniques and machine learning. The three classifier algorithms we will train include:

- Decision Tree, which uses a tree-like model of decisions to arrive at a classification prediction.
- Naive Bayes, which uses Bayes' theorem to use probability to arrive at a classification prediction.
- Linear Discriminant Analysis, which finds a linear combination of features that is then used to separate the classes and arrive at a classification prediction.

We are going to focus on the classification performance results, hence we want to evaluate the model performance of each algorithm and balancing technique to get to know which combination works best for our purpose.

Data

The dataset we are going to use was gathered from a larger set that includes anonymized credit card transactions with around 50,000 entries with 31 variables. It is avaliable here.

```r
library(caret)
library(corrplot)
library(smotefamily)

df <- read.csv("Credit_data.csv")
df$class<-as.factor(df$class)
str(df)
```

```
'data.frame':   49692 obs. of  31 variables:
 $ Time  : int  406 472 4462 6986 7519 7526 7535 7543 7551 7610 ...
 $ V1    : num  -2.31 -3.04 -2.3 -4.4 1.23 ...
 $ V2    : num  1.95 -3.16 1.76 1.36 3.02 ...
 $ V3    : num  -1.61 1.09 -0.36 -2.59 -4.3 ...
 $ V4    : num  4 2.29 2.33 2.68 4.73 ...
 $ V5    : num  -0.522 1.36 -0.822 -1.128 3.624 ...
 $ V6    : num  -1.4265 -1.0648 -0.0758 -1.7065 -1.3577 ...
 $ V7    : num  -2.537 0.326 0.562 -3.496 1.713 ...
 $ V8    : num  1.3917 -0.0678 -0.3991 -0.2488 -0.4964 ...
 $ V9    : num  -2.77 -0.271 -0.238 -0.248 -1.283 ...
 $ V10   : num  -2.772 -0.839 -1.525 -4.802 -2.447 ...
 $ V11   : num  3.202 -0.415 2.033 4.896 2.101 ...
 $ V12   : num  -2.9 -0.503 -6.56 -10.913 -4.61 ...
 $ V13   : num  -0.5952 0.6765 0.0229 0.1844 1.4644 ...
 $ V14   : num  -4.29 -1.69 -1.47 -6.77 -6.08 ...
 $ V15   : num  0.38972 2.00063 -0.69883 -0.00733 -0.33924 ...
 $ V16   : num  -1.141 0.667 -2.282 -7.358 2.582 ...
 $ V17   : num  -2.83 0.6 -4.78 -12.6 6.74 ...
 $ V18   : num  -0.0168 1.7253 -2.6157 -5.1315 3.0425 ...
 $ V19   : num  0.417 0.283 -1.334 0.308 -2.722 ...
 $ V20   : num  0.12691 2.10234 -0.43002 -0.17161 0.00906 ...
 $ V21   : num  0.517 0.662 -0.294 0.574 -0.379 ...
 $ V22   : num  -0.035 0.435 -0.932 0.177 -0.704 ...
 $ V23   : num  -0.465 1.376 0.173 -0.436 -0.657 ...
 $ V24   : num  0.3202 -0.2938 -0.0873 -0.0535 -1.6327 ...
 $ V25   : num  0.0445 0.2798 -0.1561 0.2524 1.4889 ...
 $ V26   : num  0.178 -0.145 -0.543 -0.657 0.567 ...
 $ V27   : num  0.2611 -0.2528 0.0396 -0.8271 -0.01 ...
 $ V28   : num  -0.1433 0.0358 -0.153 0.8496 0.1468 ...
 $ Amount: num  0 529 240 59 1 ...
 $ class : Factor w/ 2 levels "no","yes": 2 2 2 2 2 2 2 2 2 2 ...
```

Within the data class tells us whether the transaction was a fraudulent transaction or not with a factor with a no/yes level. It seems that the dataset contains only one integer variable, one factor variable and 29 numerical input variables, which are the result of a previous principal component analysis transformation.

Exploratory Data Analysis

Before starting building and training the models it is important to understand and explore our data as it helps us identify potential data quality issues. Data exploration provides us the needed context to develop an appropriate model, so let's briefly explore the data.

```
print(sum(is.na(df)))
```

```
[1] 0
```

```
summary(df$class);prop.table(table(df$class))
```

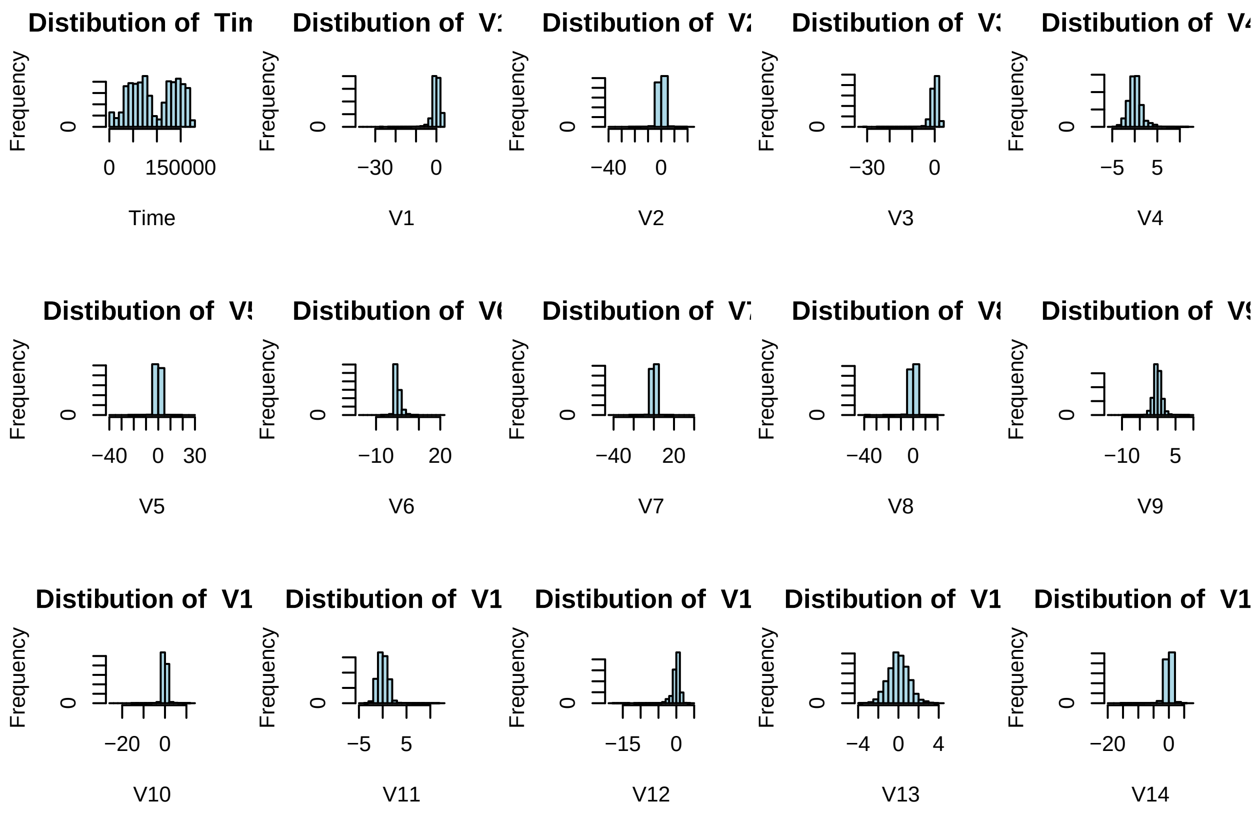
```
   no   yes 
49200   492 
```

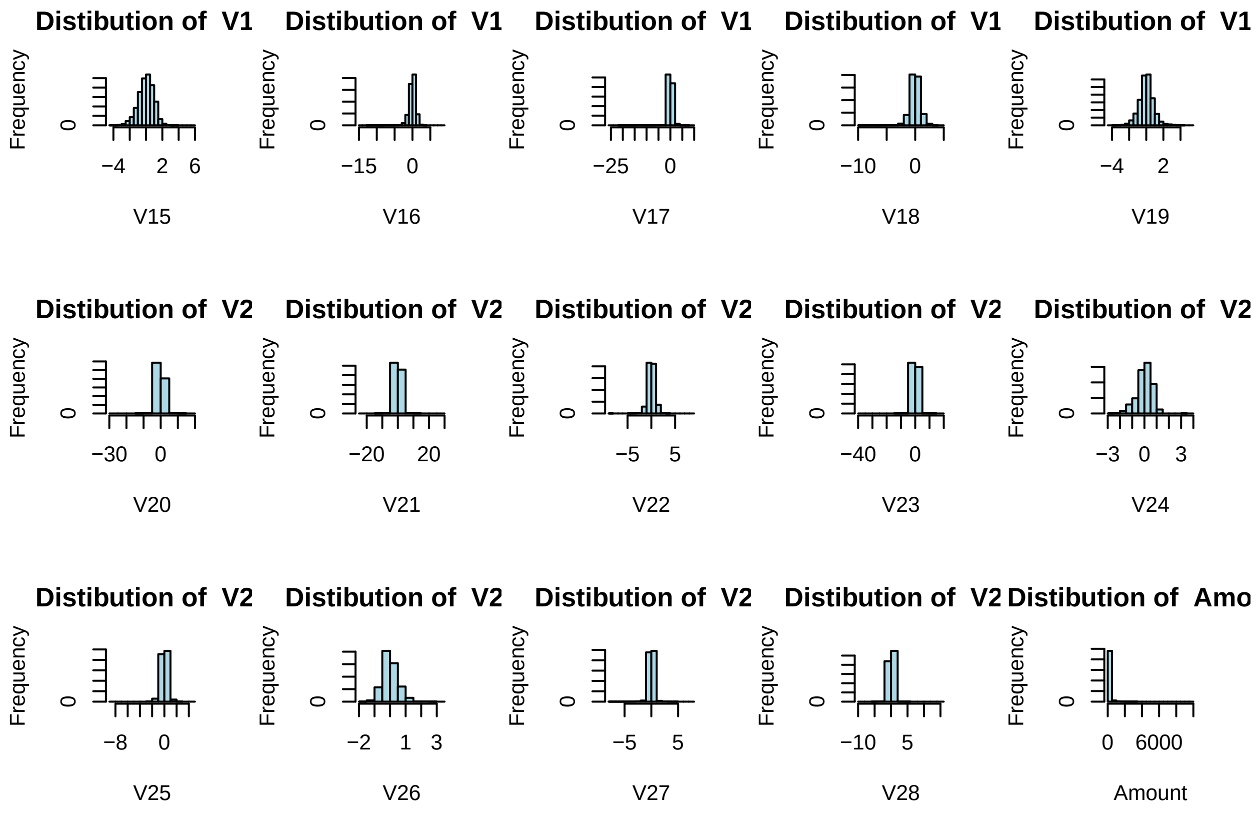
```
        no        yes 
0.99009901 0.00990099 
```

Let's explore the distribution of the predictor variables by reviewing their histograms. Given that there are 30 predictor variables, it's too many to look at one by one. So let's generate the 30 histograms using a for loop, which will enable us to automatically create a histogram for each of the 30 variables in our data set.

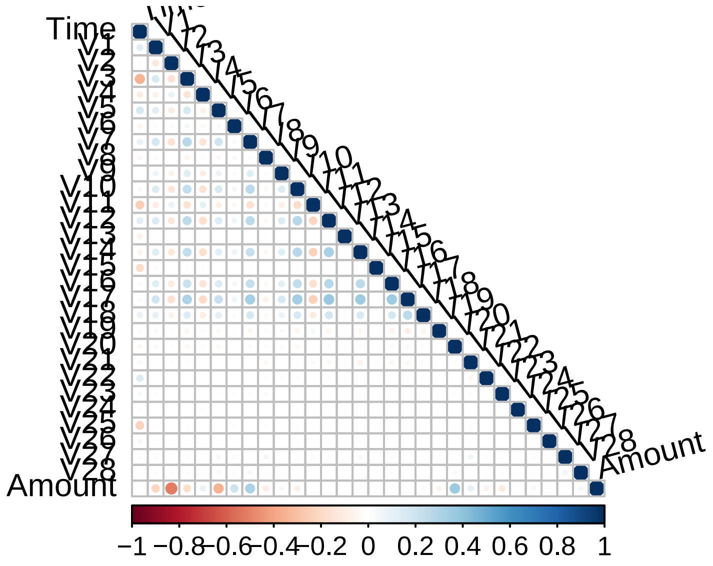
Let's also generate the correlations between our numerical variables and visualize them using the corrplot function.

```r
par(mfrow = c(3,5)) #Change setting to view 3x5 charts
i <- 1
for (i in 1:30)
{hist((df[,i]), main = paste("Distibution of ",
                             colnames(df[i])),
                xlab = colnames(df[i]),
                col = "light blue")
}
```





```
r <- cor(df[,1:30])
corrplot(r, type='lower', tl.col = 'black', tl.srt = 15)
```



As you may see most of the correlations are fairly low, which is good for our modeling purposes.

Data Spliting

To evaluate the performance of a model, we have to test the model with a not previously seen dataset. Therefore, we will split our dataset into a training dataset and a test dataset and to maintain the same level of imbalance as in the original dataset.

```
set.seed(1337)

train <- createDataPartition(df$class,
                             p=0.7,
                             times=1,
                             list=F)
train.org <- df[train,]
test <- df[-train,]

prop.table(table(train.org$class))
```

```
##
##          no         yes
## 0.990081932 0.009918068
```

```
prop.table(table(test$class))
```

```
##
##          no         yes
## 0.990138861 0.009861139
```

As seen, both training and test set are with class 99% as not fraudulent and approximately 1% as fraudulent.

Balancing Training Datasets

Now that we have split our dataset into a training and test dataset, lets create three new synthetically balanced datasets from the one imbalanced training dataset. To do this we will be using the "smotefamily" R package and we will be trying out three different techniques: SMOTE, ADASYN, and DB-SMOTE.

- Synthetic Minority Oversampling Technique (SMOTE)

A subset of data is taken from the minority class as an example. New synthetic similar examples are generated from the "feature space" rather than the "data space."

```
train.smote <- SMOTE(train.org[,-31], train.org[,31], K=5)
names(train.smote)
```

[1] "data" "syn_data" "orig_N" "orig_P" "K" "K_all"
[7] "dup_size" "outcast" "eps" "method"

```
train.smote <- train.smote$data
train.smote$class<- as.factor(train.smote$class)
prop.table(table(train.smote$class))
```

```
   no       yes
```

0.5020774 0.4979226

- Adaptive Synthetic Sampling (ADASYN)

A weighted distribution is used depending on each minority class according to their degree of learning difficulty. More synthetic observations are generated for some minority class instances that are more difficult to learn as compared to others

```
train.adas <- ADAS(train.org[,-31], train.org[,31], K=5)
train.adas <- train.adas$data
train.adas$class <- as.factor(train.adas$class)
prop.table(table(train.adas$class))
```

```
##
##        no       yes
## 0.4993041 0.5006959
```

- Density Based SMOTE (DB-SMOTE)

This over-samples the minority class at the decision boundary and over-examines the region to maintain the majority class detection rate. These are more likely to be misclassified than those far from the border.

```
train.dbsmote <- DBSMOTE(train.org[,-31], train.org[,31])
train.dbsmote <- train.dbsmote$data
train.dbsmote$class <- as.factor(train.dbsmote$class)
```

```
prop.table(table(train.dbsmote$class))
```

```
       no       yes
0.5184483 0.4815517
```

Training Models with Original Dataset

So far we have 4 training datasets:

- Original imbalanced
- SMOTE balanced
- ADASYN balanced
- DB-SMOTE balanced

We want to compare results with over theses 4 datasets using three different classifier models: Decision tree, Naive Bayes and Linear discriminant analysis.

```
ctrl <- trainControl(method = "cv",
                     number = 10,
                     classProbs = TRUE,
                     summaryFunction = twoClassSummary)
```

```r
tree_org <- train(class~.,
                  data=train.org,
                  method='rpart',
                  trControl=ctrl,
                  metric='ROC')

naive_org <- train(class~.,
                  data=train.org,
                  method='naive_bayes',
                  trControl=ctrl,
                  metric='ROC')

ld_org <- train(class~.,
                  data=train.org,
                  method='lda',
                  trControl=ctrl,
                  metric='ROC')
```

Test Predictions on Original Dataset

With the models we have built, next we will use generate predictions on the test set using the original imbalanced training set.

- We will then compile three measures of performance, which we will use to compare the performance of the models across all of our trained models:
  - Precision = TP / (TP+FP) - measures proportion of positive cases that are truly positive
  - Recall = TP / (TP+FN) - measures how complete the results are. This is often also called the senSitivity
  - F1 measure = (2xPrecision*Recall)/(Recall+Precision) - this combines the precision and recall into a single number

```r
tree_org_pred <- predict(tree_org, test, type='prob')
tree_org_test <- factor(ifelse(tree_org_pred$yes>0.5, 'yes', 'no'))
tree_precision_org <- posPredValue(tree_org_test, test$class, positive='yes')
tree_recall_org <- sensitivity(tree_org_test, test$class, positive='yes')
tree_F1_org <- (2*tree_precision_org*tree_recall_org)/
  (tree_precision_org+tree_recall_org)

naive_org_pred <- predict(naive_org, test, type='prob')
naive_org_test <- factor(ifelse(naive_org_pred$yes>0.5, 'yes', 'no'))
naive_precision_org <- posPredValue(naive_org_test, test$class, positive='yes')
naive_recall_org <- sensitivity(naive_org_test, test$class, positive='yes')
naive_F1_org <- (2*naive_precision_org*naive_recall_org)/
  (naive_precision_org+naive_recall_org)

ld_org_pred <- predict(ld_org, test, type='prob')
ld_org_test <- factor(ifelse(ld_org_pred$yes>0.5, 'yes', 'no'))
ld_precision_org <- posPredValue(ld_org_test, test$class, positive='yes')
ld_recall_org <- sensitivity(ld_org_test, test$class, positive='yes')
ld_F1_org <- (2*ld_precision_org*ld_recall_org)/
  (ld_precision_org+ld_recall_org)
```

Training Models with SMOTE Balanced Data

Next, We will train the three classifier models using the SMOTE balanced training dataset. To train the models, we can simply copy and paste the code we used to train the models in task 5, create new names for the models and change the data we are using to train our models using from 'train.orig' to the 'train.smote' dataset.

```
tree_smote <- train(class~.,
                  data=train.smote,
                  method='rpart',
                  trControl=ctrl,
                  metric='ROC')

naive_smote <- train(class~.,
                  data=train.smote,
                  method='naive_bayes',
                  trControl=ctrl,
                  metric='ROC')

ld_smote <- train(class~.,
                  data=train.smote,
                  method='lda',
                  trControl=ctrl,
                  metric='ROC')
```

Test Predictions on SMOTE balanced data

Next, we will use the models we have trained using the SMOTE balanced training dataset to generate predictions on the test dataset, and we will compute our three performance measures. To complete this, we can copy the code from the earlier task and change the names of the output and models to reference the models trained using the SMOTE balanced training dataset.

```
tree_smote_pred <- predict(tree_smote, test, type='prob')
tree_smote_test <- factor(ifelse(tree_smote_pred$yes>0.5, 'yes', 'no'))
tree_precision_smote <- posPredValue(tree_smote_test, test$class, positive='yes')
tree_recall_smote <- sensitivity(tree_smote_test, test$class, positive='yes')
tree_F1_smote <- (2*tree_precision_smote*tree_recall_smote)/
  (tree_precision_smote+tree_recall_smote)

naive_smote_pred <- predict(naive_smote, test, type='prob')
naive_smote_test <- factor(ifelse(naive_smote_pred$yes>0.5, 'yes', 'no'))
naive_precision_smote <- posPredValue(naive_smote_test, test$class, positive='yes')
naive_recall_smote <- sensitivity(naive_smote_test, test$class, positive='yes')
naive_F1_smote <- (2*naive_precision_smote*naive_recall_smote)/
  (naive_precision_smote+naive_recall_smote)

ld_smote_pred <- predict(ld_smote, test, type='prob')
ld_smote_test <- factor(ifelse(ld_smote_pred$yes>0.5, 'yes', 'no'))
ld_precision_smote <- posPredValue(ld_smote_test, test$class, positive='yes')
ld_recall_smote <- sensitivity(ld_smote_test, test$class, positive='yes')
ld_F1_smote <- (2*ld_precision_smote*ld_recall_smote)/
  (ld_precision_smote+ld_recall_smote)
```

Training Models with ADASYN Balanced Data

In task 7, we will train the three classifier models using the ADASYN balanced training dataset. Again, to train the models, we can simply copy and paste the code we used to train the models in task 6, create new names for the model and change the data we are using to train our model to 'train.adas'

```
tree_adas <- train(class~.,
                  data=train.adas,
                  method='rpart',
                  trControl=ctrl,
                  metric='ROC')

naive_adas <- train(class~.,
                  data=train.adas,
                  method='naive_bayes',
                  trControl=ctrl,
                  metric='ROC')

ld_adas <- train(class~.,
                  data=train.adas,
                  method='lda',
                  trControl=ctrl,
                  metric='ROC')
```

Test Predictions on ADASYN balanced data

Next, we will use the models we have trained using the ADASYN balanced training dataset to generate predictions on the test dataset, and we will compute our three performance measures. To complete this, we can copy the code from the earlier task and change the names of the output and models to reference the models trained using the SMOTE balanced training dataset.

```
tree_adas_pred <- predict(tree_adas, test, type='prob')
tree_adas_test <- factor(ifelse(tree_adas_pred$yes>0.5, 'yes', 'no'))
tree_precision_adas <- posPredValue(tree_adas_test, test$class, positive='yes')
tree_recall_adas <- sensitivity(tree_adas_test, test$class, positive='yes')
tree_F1_adas <- (2*tree_precision_adas*tree_recall_adas)/
  (tree_precision_adas+tree_recall_adas)

naive_adas_pred <- predict(naive_adas, test, type='prob')
naive_adas_test <- factor(ifelse(naive_adas_pred$yes>0.5, 'yes', 'no'))
naive_precision_adas <- posPredValue(naive_adas_test, test$class, positive='yes')
naive_recall_adas <- sensitivity(naive_adas_test, test$class, positive='yes')
naive_F1_adas <- (2*naive_precision_adas*naive_recall_adas)/
  (naive_precision_adas+naive_recall_adas)

ld_adas_pred <- predict(ld_adas, test, type='prob')
ld_adas_test <- factor(ifelse(ld_adas_pred$yes>0.5, 'yes', 'no'))
ld_precision_adas <- posPredValue(ld_adas_test, test$class, positive='yes')
ld_recall_adas <- sensitivity(ld_adas_test, test$class, positive='yes')
ld_F1_adas <- (2*ld_precision_adas*ld_recall_adas)/
  (ld_precision_adas+ld_recall_adas)
```

Training Models with DB-SMOTE Balanced Data

In task 8, we will train the three classifier models using the DB-SMOTE balanced training dataset. To train the models, we can simply copy and paste the code we used to train the models in task 7, create new names for the model and change the data we are using to train our model to 'train.dbsmote'

```
tree_dbsmote <- train(class~.,
                  data=train.dbsmote,
                  method='rpart',
                  trControl=ctrl,
                  metric='ROC')

naive_dbsmote <- train(class~.,
                  data=train.dbsmote,
                  method='naive_bayes',
                  trControl=ctrl,
                  metric='ROC')

ld_dbsmote <- train(class~.,
                  data=train.dbsmote,
                  method='lda',
                  trControl=ctrl,
                  metric='ROC')
```

Test Predictions on DB SMOTE balanced data

Next, we will use the models we have trained using the DB-SMOTE balanced training dataset to generate predictions on the test dataset, and we will compute our three performance measures. To complete this, we can copy the code from the earlier task and change the names of the output and models to reference the models trained using the DB-SMOTE balanced training dataset.

```
tree_dbsmote_pred <- predict(tree_dbsmote, test, type='prob')
tree_dbsmote_test <- factor(ifelse(tree_dbsmote_pred$yes>0.5, 'yes', 'no'))
tree_precision_dbsmote <- posPredValue(tree_dbsmote_test, test$class, positive='yes')
tree_recall_dbsmote <- sensitivity(tree_dbsmote_test, test$class, positive='yes')
tree_F1_dbsmote <- (2*tree_precision_dbsmote*tree_recall_dbsmote)/
  (tree_precision_dbsmote+tree_recall_dbsmote)

naive_dbsmote_pred <- predict(naive_dbsmote, test, type='prob')
naive_dbsmote_test <- factor(ifelse(naive_dbsmote_pred$yes>0.5, 'yes', 'no'))
naive_precision_dbsmote <- posPredValue(naive_dbsmote_test, test$class, positive='yes')
naive_recall_dbsmote <- sensitivity(naive_dbsmote_test, test$class, positive='yes')
naive_F1_dbsmote <- (2*naive_precision_dbsmote*naive_recall_dbsmote)/
  (naive_precision_dbsmote+naive_recall_dbsmote)

ld_dbsmote_pred <- predict(ld_dbsmote, test, type='prob')
ld_dbsmote_test <- factor(ifelse(ld_dbsmote_pred$yes>0.5, 'yes', 'no'))
ld_precision_dbsmote <- posPredValue(ld_dbsmote_test, test$class, positive='yes')
ld_recall_dbsmote <- sensitivity(ld_dbsmote_test, test$class, positive='yes')
ld_F1_dbsmote <- (2*ld_precision_dbsmote*ld_recall_dbsmote)/
  (ld_precision_dbsmote+ld_recall_dbsmote)
```

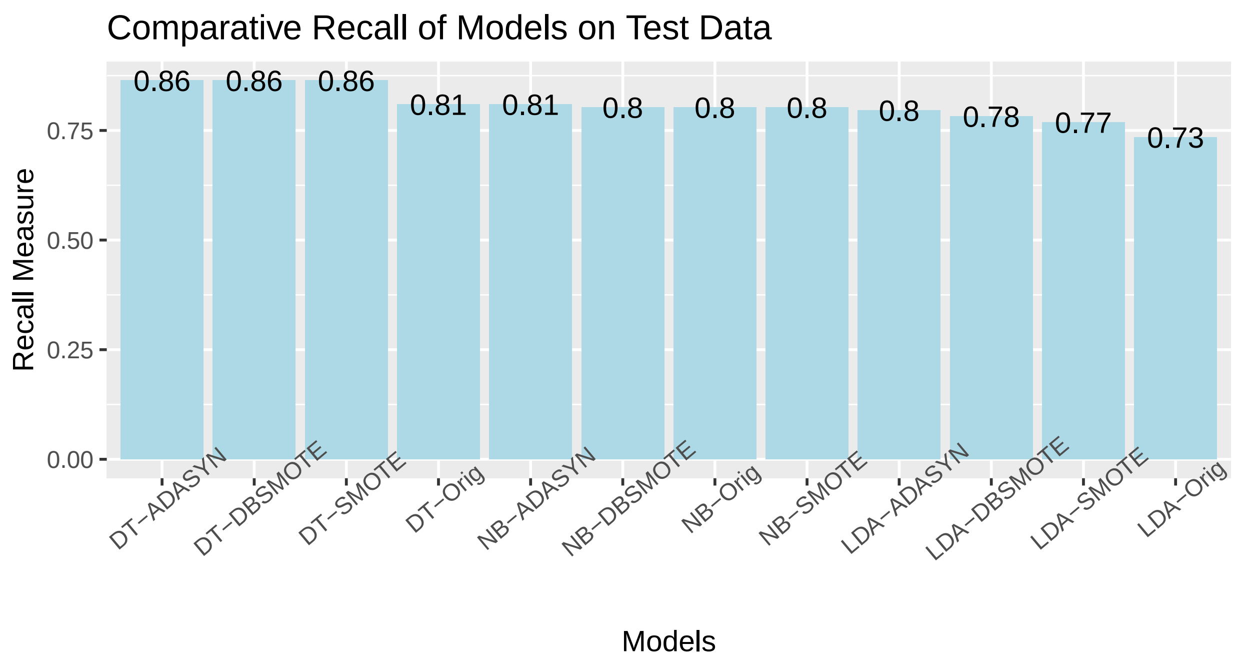
Model Performance Comparison

We will compare the recall, precision, and F1 performance measures for each of the three models we trained using the four training datasets: original imbalanced, SMOTE balanced, ADASYN balanced, and DB-SMOTE balanced.

Recall that the most important performance measure for the fraud problem is the recall, which measures how complete our results are indicating the model captures more of the fraudulent transactions. Now we'll combine our results into a dataframe and plot it to have a visual and understand how each compares to each other.

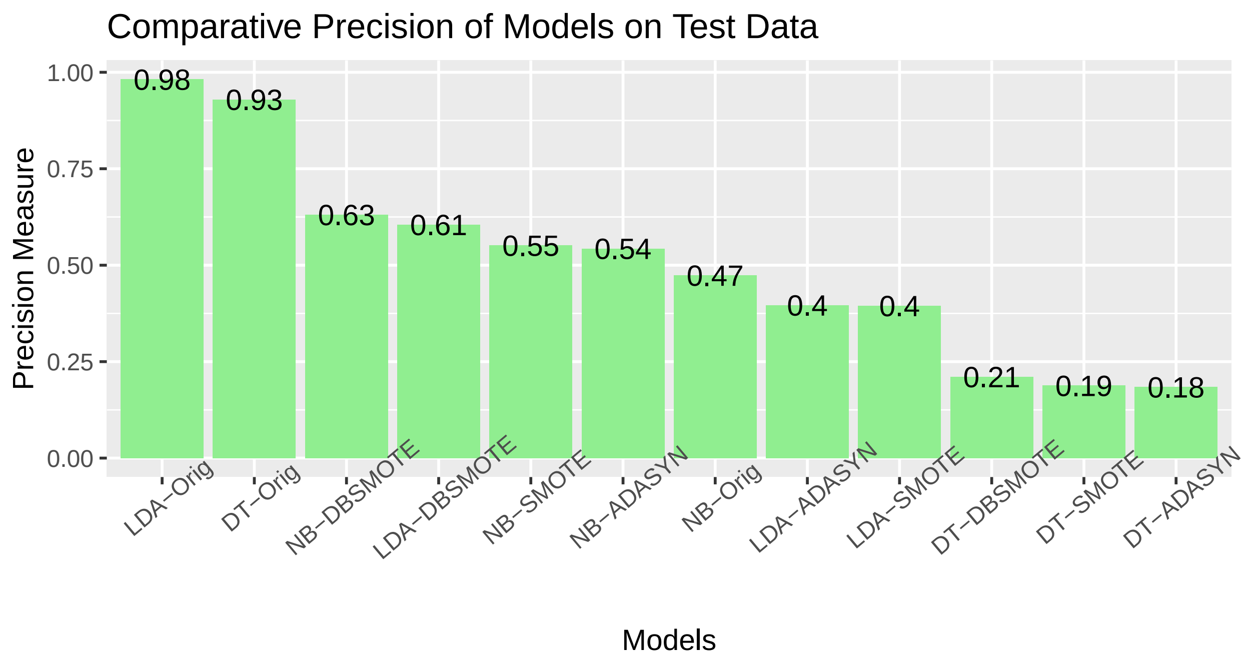
```r
par(mfrow = c(1,1))
model_compare_recall <- data.frame(Model = c('DT-Orig', 'NB-Orig', 'LDA-Orig', 'DT-SMOTE',
                                             'NB-SMOTE', 'LDA-SMOTE', 'DT-ADASYN',
                                             'NB-ADASYN', 'LDA-ADASYN', 'DT-DBSMOTE',
                                             'NB-DBSMOTE', 'LDA-DBSMOTE' ),
                                   Recall = c(tree_recall_org, naive_recall_org, ld_recall_org,
                                              tree_recall_smote, naive_recall_smote,
                                              ld_recall_smote, tree_recall_adas,
                                              naive_recall_adas, ld_recall_adas,
                                              tree_recall_dbsmote, naive_recall_dbsmote,
                                              ld_recall_dbsmote))

ggplot(aes(x=reorder(Model,-Recall) , y=Recall), data=model_compare_recall) +
  geom_bar(stat='identity', fill = 'light blue') +
  ggtitle('Comparative Recall of Models on Test Data') +
  xlab('Models')  +
  ylab('Recall Measure')+
  geom_text(aes(label=round(Recall,2)))+
  theme(axis.text.x = element_text(angle = 40))
```



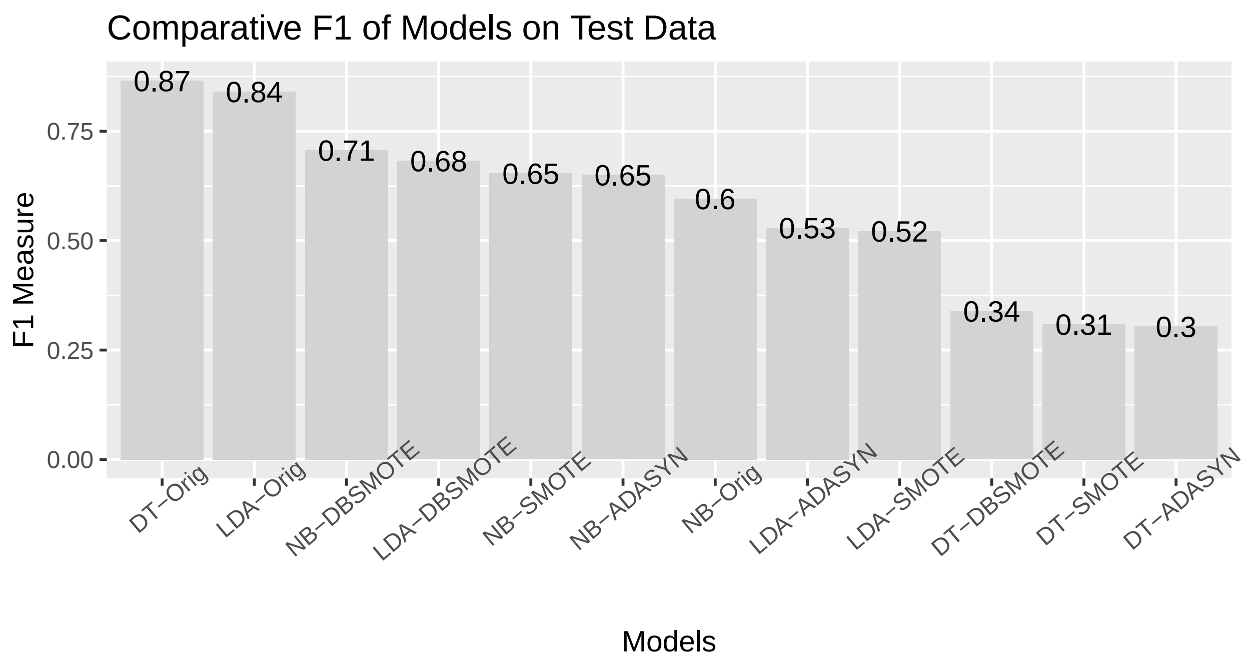
```
model_compare_precision <- data.frame(Model = c('DT-Orig', 'NB-Orig', 'LDA-Orig', 'DT-SMOTE',
                                                'NB-SMOTE', 'LDA-SMOTE', 'DT-ADASYN',
                                                'NB-ADASYN', 'LDA-ADASYN', 'DT-DBSMOTE',
                                                'NB-DBSMOTE', 'LDA-DBSMOTE' ),
                                      Precision = c(tree_precision_org, naive_precision_org,
                                                    ld_precision_org, tree_precision_smote,
                                                    naive_precision_smote, ld_precision_smote,
                                                    tree_precision_adas,  naive_precision_adas,
                                                    ld_precision_adas, tree_precision_dbsmote,
                                                    naive_precision_dbsmote, ld_precision_dbsmote))

ggplot(aes(x=reorder(Model,-Precision) , y=Precision), data=model_compare_precision) +
  geom_bar(stat='identity', fill = 'light green') +
  ggtitle('Comparative Precision of Models on Test Data') +
  xlab('Models')  +
  ylab('Precision Measure')+
  geom_text(aes(label=round(Precision,2)))+
  theme(axis.text.x = element_text(angle = 40))
```



```
model_compare_f1 <- data.frame(Model = c('DT-Orig', 'NB-Orig', 'LDA-Orig', 'DT-SMOTE',
                                         'NB-SMOTE', 'LDA-SMOTE', 'DT-ADASYN', 'NB-ADASYN',
                                         'LDA-ADASYN', 'DT-DBSMOTE', 'NB-DBSMOTE', 'LDA-DBSMOTE'),
                               F1 = c(tree_F1_org, naive_F1_org, ld_F1_org, tree_F1_smote,
                                      naive_F1_smote, ld_F1_smote, tree_F1_adas, naive_F1_adas,
                                      ld_F1_adas, tree_F1_dbsmote, naive_F1_dbsmote, ld_F1_dbsmote))

ggplot(aes(x=reorder(Model,-F1) , y=F1), data=model_compare_f1) +
  geom_bar(stat='identity', fill = 'light grey') +
  ggtitle('Comparative F1 of Models on Test Data') +
  xlab('Models')  +
  ylab('F1 Measure')+
  geom_text(aes(label=round(F1,2)))+
  theme(axis.text.x = element_text(angle = 40))
```



We can see that the top three performing models were the decision trees trained on the synthetically balanced data. Specifically, the Decision Tree model trained using the ADASYN balanced dataset, performed best followed by DB SMOTE and then smote. Similarly, the naive Bayes and LDA models trained using the ADASYN performed better than the models trained using the imbalanced original dataset.

within the precision results are more mixed, whereby the LDA and Decision Tree models using the original data set performed better. But the naive Bayes model trained using the DB smote balanced dataset performed better than the naive Bayes model trained on the original imbalance dataset.

However, in aggregate the cost of not identifying fraudulent transactions can be very costly. Therefore, the recall measure is the most important performance measure, and the results indicate that models trained using synthetically balanced training data have a superior recall performance relative to models trained using the original imbalance dataset.

In our Recall plot he Decision Tree and LDA Models for the original imbalanced dataset has a higher F one, and only the naive Bayes trained on the DB smote performed better. However, this is being driven by the higher performance in precision which is driving the F one measure higher. But as we discussed earlier, the recall measure is the most important for this problem and, as we saw earlier, the recall performance measure for the models trained on the synthetically balanced datasets performed significantly better than those trained on the original imbalanced dataset.

Conclusions

Credit card fraud detection using Machine Learning is the next evolution in credit card fraud prevention. Machine Learning technology can analyze massive amounts of transaction data and identify anomalies in the data that otherwise humans may miss. It can help reduce the amount of fraudulent credit card transactions and protect financial service providers and their customers.

In order to improve the performance of classification models, we have to try with different algorithms and balance techniques if necesary, to get to know what is the best way to go forward when we want to solve a problem using machine learning. Creating synthetically balanced training datasets has an impact when we are working with classification algorithms and may improve their performance, as well as using different machine learning algorithms. It will depend of the users criteria to select the best approach.