Credit Card Fraud Detection

Sougata Ghosh

08/01/2019

Introduction

Billions of dollars of loss are caused every year due to fraudulent credit card transactions. The design of efficient fraud detection algorithms is key to reducing these losses, and more algorithms rely on advanced machine learning techniques to assist fraud investigators. The design of fraud detection algorithms is however particularly challenging due to non-stationary distribution of the data, highly imbalanced classes distributions and continuous streams of transactions. At the same time public data are scarcely available for confidentiality issues, leaving unanswered many questions about which is the best strategy to deal with them.

The dataset from Kaggle available at(https://www.kaggle.com/mlg-ulb/creditcardfraud) contains transactions made by credit cards in September 2013 by european cardholders.This dataset presents transactions that occurred in two days, where we have 492 frauds out of 284,807 transactions. The dataset is highly unbalanced, the positive class (frauds) account for 0.172% of all transactions.

It contains only numerical input variables which are the result of a PCA transformation. Unfortunately, due to confidentiality issues, we do not have access to the original features and more background information about the data. Features V1, V2, … V28 are the principal components obtained with PCA, the only features which have not been transformed with PCA are 'Time' and 'Amount'. Feature 'Time' contains the seconds elapsed between each transaction and the first transaction in the dataset. The feature 'Amount' is the transaction Amount, this feature can be used for example-dependant cost-senstive learning. Feature 'Class' is the response variable and it takes value 1 in case of fraud and 0 otherwise.

The objective of the project is to train a machine learning algorithm on the dataset to successfully predict fraudulent transactions.

Given the class imbalance ratio, we recommend measuring the accuracy using the Area Under the Precision-Recall Curve (AUCC). Confusion matrix accuracy is not meaningful for unbalanced classification.

We also recommend using different sampling techniques (detailed below) on the train dataset in order to address the issue of imbalanced classes while training our models.

The dataset has been collected and analysed during a research collaboration of Worldline and the Machine Learning Group (http://mlg.ulb.ac.be) of ULB (Université Libre de Bruxelles) on big data mining and fraud detection. More details on current and past projects on related topics are available on https://www.researchgate.net/project/Fraud-detection-5 and the page of the DefeatFraud project

Methods/ Analysis

```r
#Load Packages
if (!require(dplyr)) install.packages('dplyr')
library(dplyr) # for data manipulation
if (!require(stringr)) install.packages('stringr')
library(stringr) # for data manipulation
if (!require(caret)) install.packages('caret')
library(caret) # for sampling
if (!require(caTools)) install.packages('caTools')
library(caTools) # for train/test split
if (!require(ggplot2)) install.packages('ggplot2')
```

```r
library(ggplot2) # for data visualization
if (!require(corrplot)) install.packages('corrplot')
library(corrplot) # for correlations
if (!require(Rtsne)) install.packages('Rtsne')
library(Rtsne) # for tsne plotting
if (!require(DMwR)) install.packages('DMwR')
library(DMwR) # for smote implementation
if (!require(ROSE)) install.packages('ROSE')
library(ROSE)# for ROSE sampling
if (!require(rpart)) install.packages('rpart')
library(rpart)# for decision tree model
if (!require(Rborist)) install.packages('Rborist')
library(Rborist)# for random forest model
if (!require(xgboost)) install.packages('xgboost')
library(xgboost) # for xgboost model
```

```r
#Load data

df<- read.csv("creditcard.csv")
```

Basic Exploration

```r
head(df)
```

```
##   Time         V1          V2        V3         V4          V5          V6
## 1    0 -1.3598071 -0.07278117 2.5363467  1.3781552 -0.33832077  0.46238778
## 2    0  1.1918571  0.26615071 0.1664801  0.4481541  0.06001765 -0.08236081
## 3    1 -1.3583541 -1.34016307 1.7732093  0.3797796 -0.50319813  1.80049938
## 4    1 -0.9662717 -0.18522601 1.7929933 -0.8632913 -0.01030888  1.24720317
## 5    2 -1.1582331  0.87773675 1.5487178  0.4030339 -0.40719338  0.09592146
## 6    2 -0.4259659  0.96052304 1.1411093 -0.1682521  0.42098688 -0.02972755
##            V7          V8         V9         V10        V11         V12
## 1  0.23959855  0.09869790  0.3637870  0.09079417 -0.5515995 -0.61780086
## 2 -0.07880298  0.08510165 -0.2554251 -0.16697441  1.6127267  1.06523531
## 3  0.79146096  0.24767579 -1.5146543  0.20764287  0.6245015  0.06608369
## 4  0.23760894  0.37743587 -1.3870241 -0.05495192 -0.2264873  0.17822823
## 5  0.59294075 -0.27053268  0.8177393  0.75307443 -0.8228429  0.53819555
## 6  0.47620095  0.26031433 -0.5686714 -0.37140720  1.3412620  0.35989384
##          V13        V14        V15        V16         V17         V18
## 1 -0.9913898 -0.3111694  1.4681770 -0.4704005  0.20797124  0.02579058
## 2  0.4890950 -0.1437723  0.6355581  0.4639170 -0.11480466 -0.18336127
## 3  0.7172927 -0.1659459  2.3458649 -2.8900832  1.10996938 -0.12135931
## 4  0.5077569 -0.2879237 -0.6314181 -1.0596472 -0.68409279  1.96577500
## 5  1.3458516 -1.1196698  0.1751211 -0.4514492 -0.23703324 -0.03819479
## 6 -0.3580907 -0.1371337  0.5176168  0.4017259 -0.05813282  0.06865315
##           V19         V20          V21          V22         V23
## 1  0.40399296  0.25141210 -0.018306778  0.277837576 -0.11047391
## 2 -0.14578304 -0.06908314 -0.225775248 -0.638671953  0.10128802
## 3 -2.26185710  0.52497973  0.247998153  0.771679402  0.90941226
## 4 -1.23262197 -0.20803778 -0.108300452  0.005273597 -0.19032052
## 5  0.80348692  0.40854236 -0.009430697  0.798278495 -0.13745808
```

```
## 6 -0.03319379  0.08496767 -0.208253515 -0.559824796 -0.02639767
##           V24        V25        V26          V27         V28 Amount Class
## 1  0.06692807  0.1285394 -0.1891148  0.133558377 -0.02105305 149.62     0
## 2 -0.33984648  0.1671704  0.1258945 -0.008983099  0.01472417   2.69     0
## 3 -0.68928096 -0.3276418 -0.1390966 -0.055352794 -0.05975184 378.66     0
## 4 -1.17557533  0.6473760 -0.2219288  0.062722849  0.06145763 123.50     0
## 5  0.14126698 -0.2060096  0.5022922  0.219422230  0.21515315  69.99     0
## 6 -0.37142658 -0.2327938  0.1059148  0.253844225  0.08108026   3.67     0
```

```r
str(df)
```

```
## 'data.frame':    284807 obs. of  31 variables:
##  $ Time  : num  0 0 1 1 2 2 4 7 7 9 ...
##  $ V1    : num  -1.36 1.192 -1.358 -0.966 -1.158 ...
##  $ V2    : num  -0.0728 0.2662 -1.3402 -0.1852 0.8777 ...
##  $ V3    : num  2.536 0.166 1.773 1.793 1.549 ...
##  $ V4    : num  1.378 0.448 0.38 -0.863 0.403 ...
##  $ V5    : num  -0.3383 0.06 -0.5032 -0.0103 -0.4072 ...
##  $ V6    : num  0.4624 -0.0824 1.8005 1.2472 0.0959 ...
##  $ V7    : num  0.2396 -0.0788 0.7915 0.2376 0.5929 ...
##  $ V8    : num  0.0987 0.0851 0.2477 0.3774 -0.2705 ...
##  $ V9    : num  0.364 -0.255 -1.515 -1.387 0.818 ...
##  $ V10   : num  0.0908 -0.167 0.2076 -0.055 0.7531 ...
##  $ V11   : num  -0.552 1.613 0.625 -0.226 -0.823 ...
##  $ V12   : num  -0.6178 1.0652 0.0661 0.1782 0.5382 ...
##  $ V13   : num  -0.991 0.489 0.717 0.508 1.346 ...
##  $ V14   : num  -0.311 -0.144 -0.166 -0.288 -1.12 ...
##  $ V15   : num  1.468 0.636 2.346 -0.631 0.175 ...
##  $ V16   : num  -0.47 0.464 -2.89 -1.06 -0.451 ...
##  $ V17   : num  0.208 -0.115 1.11 -0.684 -0.237 ...
##  $ V18   : num  0.0258 -0.1834 -0.1214 1.9658 -0.0382 ...
##  $ V19   : num  0.404 -0.146 -2.262 -1.233 0.803 ...
##  $ V20   : num  0.2514 -0.0691 0.525 -0.208 0.4085 ...
##  $ V21   : num  -0.01831 -0.22578 0.248 -0.1083 -0.00943 ...
##  $ V22   : num  0.27784 -0.63867 0.77168 0.00527 0.79828 ...
##  $ V23   : num  -0.11 0.101 0.909 -0.19 -0.137 ...
##  $ V24   : num  0.0669 -0.3398 -0.6893 -1.1756 0.1413 ...
##  $ V25   : num  0.129 0.167 -0.328 0.647 -0.206 ...
##  $ V26   : num  -0.189 0.126 -0.139 -0.222 0.502 ...
##  $ V27   : num  0.13356 -0.00898 -0.05535 0.06272 0.21942 ...
##  $ V28   : num  -0.0211 0.0147 -0.0598 0.0615 0.2152 ...
##  $ Amount: num  149.62 2.69 378.66 123.5 69.99 ...
##  $ Class : int  0 0 0 0 0 0 0 0 0 0 ...
```

The dataframe has 284807 observations with 31 variables. The variable 'Class' indicates whether a transaction is fraudulent(1) or not (0).

```r
summary(df)
```

```
##       Time              V1                  V2
##  Min.   :     0   Min.   :-56.40751   Min.   :-72.71573
##  1st Qu.: 54202   1st Qu.: -0.92037   1st Qu.: -0.59855
##  Median : 84692   Median :  0.01811   Median :  0.06549
```

```
##  Mean   : 94814   Mean   :  0.00000   Mean   :  0.00000
##  3rd Qu.:139321   3rd Qu.:  1.31564   3rd Qu.:  0.80372
##  Max.   :172792   Max.   :  2.45493   Max.   : 22.05773
##        V3                  V4                  V5
##  Min.   :-48.3256   Min.   :-5.68317   Min.   :-113.74331
##  1st Qu.: -0.8904   1st Qu.:-0.84864   1st Qu.:  -0.69160
##  Median :  0.1799   Median :-0.01985   Median :  -0.05434
##  Mean   :  0.0000   Mean   : 0.00000   Mean   :   0.00000
##  3rd Qu.:  1.0272   3rd Qu.: 0.74334   3rd Qu.:   0.61193
##  Max.   :  9.3826   Max.   :16.87534   Max.   :  34.80167
##        V6                  V7                  V8
##  Min.   :-26.1605   Min.   :-43.5572   Min.   :-73.21672
##  1st Qu.: -0.7683   1st Qu.: -0.5541   1st Qu.: -0.20863
##  Median : -0.2742   Median :  0.0401   Median :  0.02236
##  Mean   :  0.0000   Mean   :  0.0000   Mean   :  0.00000
##  3rd Qu.:  0.3986   3rd Qu.:  0.5704   3rd Qu.:  0.32735
##  Max.   : 73.3016   Max.   :120.5895   Max.   : 20.00721
##        V9                  V10                 V11
##  Min.   :-13.43407   Min.   :-24.58826   Min.   :-4.79747
##  1st Qu.: -0.64310   1st Qu.: -0.53543   1st Qu.:-0.76249
##  Median : -0.05143   Median : -0.09292   Median :-0.03276
##  Mean   :  0.00000   Mean   :  0.00000   Mean   : 0.00000
##  3rd Qu.:  0.59714   3rd Qu.:  0.45392   3rd Qu.: 0.73959
##  Max.   : 15.59500   Max.   : 23.74514   Max.   :12.01891
##       V12                 V13                 V14
##  Min.   :-18.6837   Min.   :-5.79188   Min.   :-19.2143
##  1st Qu.: -0.4056   1st Qu.:-0.64854   1st Qu.: -0.4256
##  Median :  0.1400   Median :-0.01357   Median :  0.0506
##  Mean   :  0.0000   Mean   : 0.00000   Mean   :  0.0000
##  3rd Qu.:  0.6182   3rd Qu.: 0.66251   3rd Qu.:  0.4931
##  Max.   :  7.8484   Max.   : 7.12688   Max.   : 10.5268
##       V15                 V16                  V17
##  Min.   :-4.49894   Min.   :-14.12985   Min.   :-25.16280
##  1st Qu.:-0.58288   1st Qu.: -0.46804   1st Qu.: -0.48375
##  Median : 0.04807   Median :  0.06641   Median : -0.06568
##  Mean   : 0.00000   Mean   :  0.00000   Mean   :  0.00000
##  3rd Qu.: 0.64882   3rd Qu.:  0.52330   3rd Qu.:  0.39968
##  Max.   : 8.87774   Max.   : 17.31511   Max.   :  9.25353
##       V18                  V19                  V20
##  Min.   :-9.498746   Min.   :-7.213527   Min.   :-54.49772
##  1st Qu.:-0.498850   1st Qu.:-0.456299   1st Qu.: -0.21172
##  Median :-0.003636   Median : 0.003735   Median : -0.06248
##  Mean   : 0.000000   Mean   : 0.000000   Mean   :  0.00000
##  3rd Qu.: 0.500807   3rd Qu.: 0.458949   3rd Qu.:  0.13304
##  Max.   : 5.041069   Max.   : 5.591971   Max.   : 39.42090
##       V21                  V22                  V23
##  Min.   :-34.83038   Min.   :-10.933144   Min.   :-44.80774
##  1st Qu.: -0.22839   1st Qu.: -0.542350   1st Qu.: -0.16185
##  Median : -0.02945   Median :  0.006782   Median : -0.01119
##  Mean   :  0.00000   Mean   :  0.000000   Mean   :  0.00000
##  3rd Qu.:  0.18638   3rd Qu.:  0.528554   3rd Qu.:  0.14764
##  Max.   : 27.20284   Max.   : 10.503090   Max.   : 22.52841
##       V24                 V25                  V26
##  Min.   :-2.83663   Min.   :-10.29540   Min.   :-2.60455
```

```
##  1st Qu.:-0.35459   1st Qu.: -0.31715   1st Qu.:-0.32698
##  Median : 0.04098   Median :  0.01659   Median :-0.05214
##  Mean   : 0.00000   Mean   :  0.00000   Mean   : 0.00000
##  3rd Qu.: 0.43953   3rd Qu.:  0.35072   3rd Qu.: 0.24095
##  Max.   : 4.58455   Max.   :  7.51959   Max.   : 3.51735
##       V27                  V28                Amount
##  Min.   :-22.565679   Min.   :-15.43008   Min.   :    0.00
##  1st Qu.: -0.070840   1st Qu.: -0.05296   1st Qu.:    5.60
##  Median :  0.001342   Median :  0.01124   Median :   22.00
##  Mean   :  0.000000   Mean   :  0.00000   Mean   :   88.35
##  3rd Qu.:  0.091045   3rd Qu.:  0.07828   3rd Qu.:   77.17
##  Max.   : 31.612198   Max.   : 33.84781   Max.   :25691.16
##      Class
##  Min.   :0.000000
##  1st Qu.:0.000000
##  Median :0.000000
##  Mean   :0.001728
##  3rd Qu.:0.000000
##  Max.   :1.000000
```

All the anonymised features seem to have been be normalised with mean 0. We will apply that transformation to the "Amount" column later on to facilitate training ML models.

```r
#Check for missing values
colSums(is.na(df))
```

```
##   Time     V1     V2     V3     V4     V5     V6     V7     V8     V9
##      0      0      0      0      0      0      0      0      0      0
##    V10    V11    V12    V13    V14    V15    V16    V17    V18    V19
##      0      0      0      0      0      0      0      0      0      0
##    V20    V21    V22    V23    V24    V25    V26    V27    V28 Amount
##      0      0      0      0      0      0      0      0      0      0
##  Class
##      0
```

None of the variables have missing values

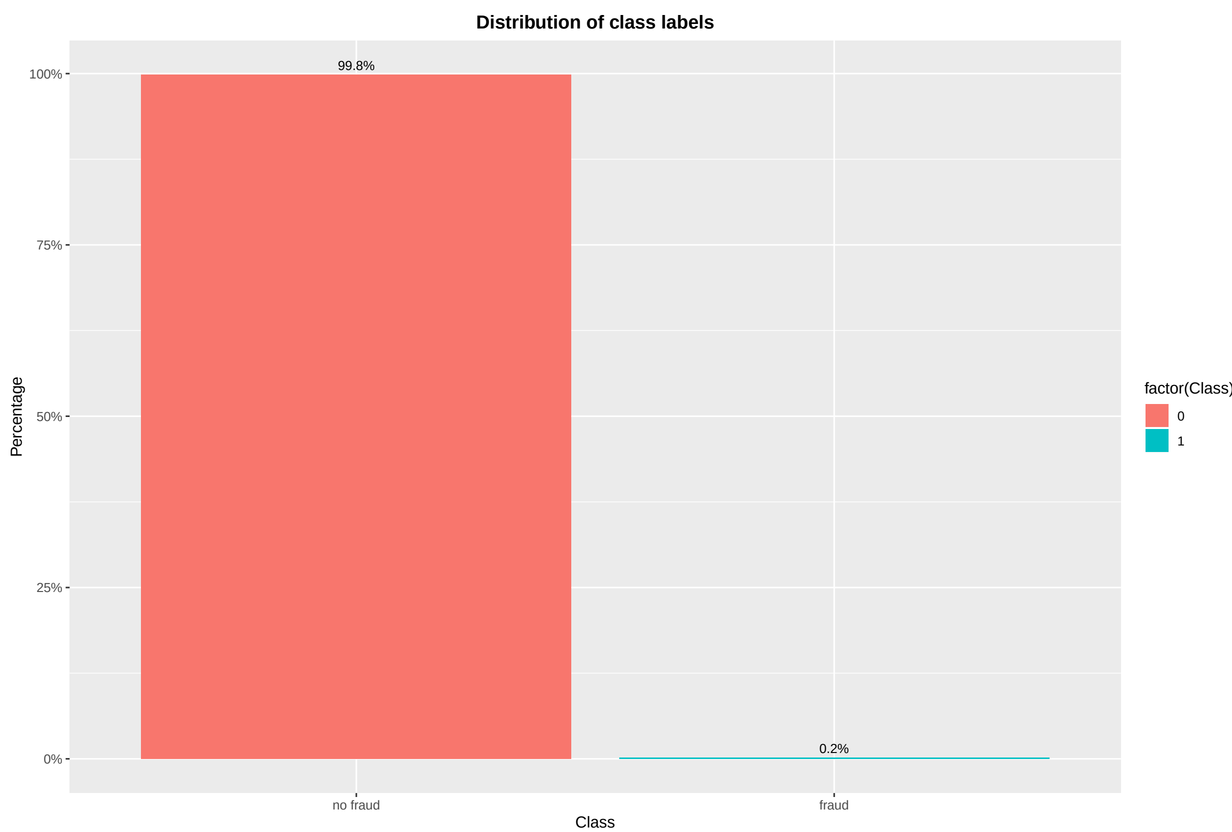
```r
#Check class imbalance
table(df$Class)
```

```
##
##      0      1
## 284315    492
```

```r
prop.table(table(df$Class))
```

```
##
##           0           1
## 0.998272514 0.001727486
```

```
common_theme <- theme(plot.title = element_text(hjust = 0.5, face = "bold"))

ggplot(data = df, aes(x = factor(Class),
                          y = prop.table(stat(count)), fill = factor(Class),
                          label = scales::percent(prop.table(stat(count))))) +
    geom_bar(position = "dodge") +
    geom_text(stat = 'count',
              position = position_dodge(.9),
              vjust = -0.5,
              size = 3) +
    scale_x_discrete(labels = c("no fraud", "fraud"))+
    scale_y_continuous(labels = scales::percent)+
    labs(x = 'Class', y = 'Percentage') +
    ggtitle("Distribution of class labels") +
    common_theme
```

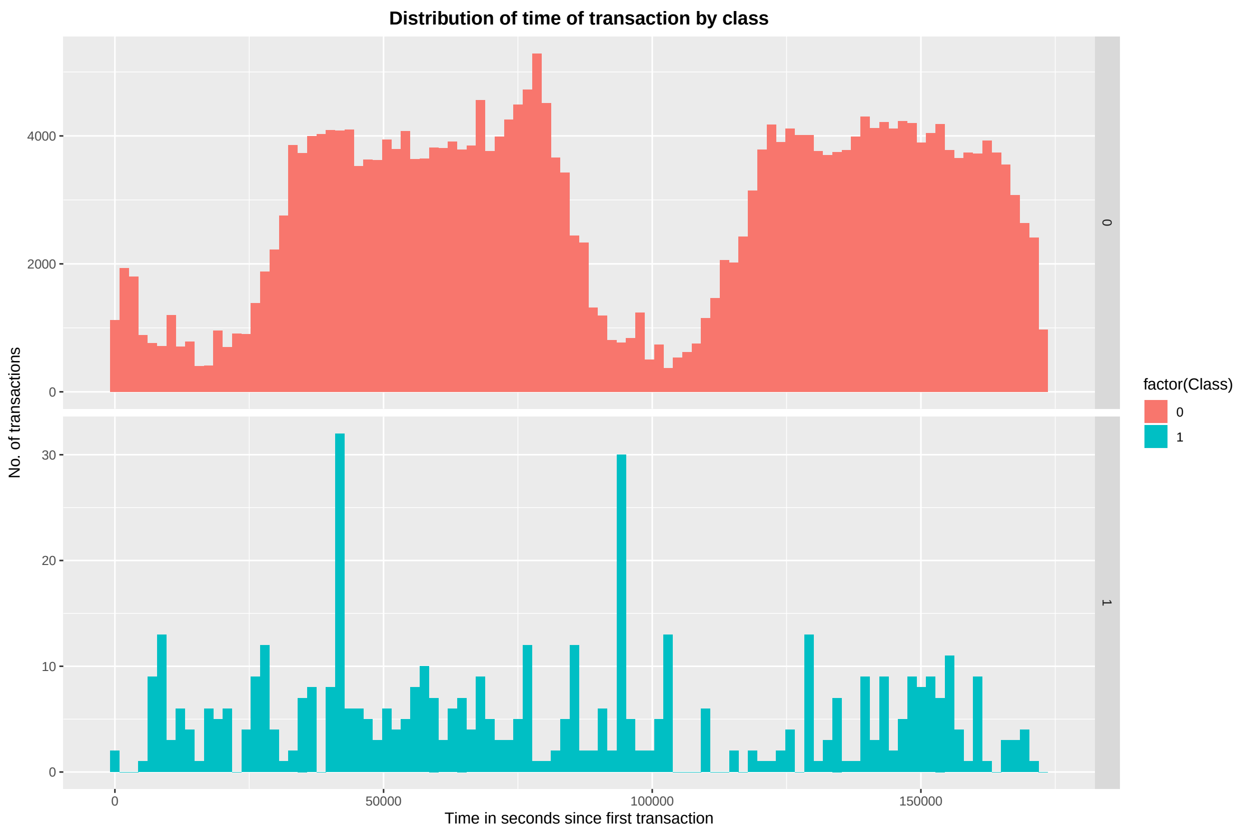


Clearly the dataset is very imbalanced with 99.8% of cases being non-fraudulent transactions. A simple measure like accuracy is not appropriate here as even a classifier which labels all transactions as non-fraudulent will have over 99% accuracy. An appropriate measure of model performance here would be AUC (Area Under the Precision-Recall Curve).

Data Visualization

Distribution of variable 'Time' by class

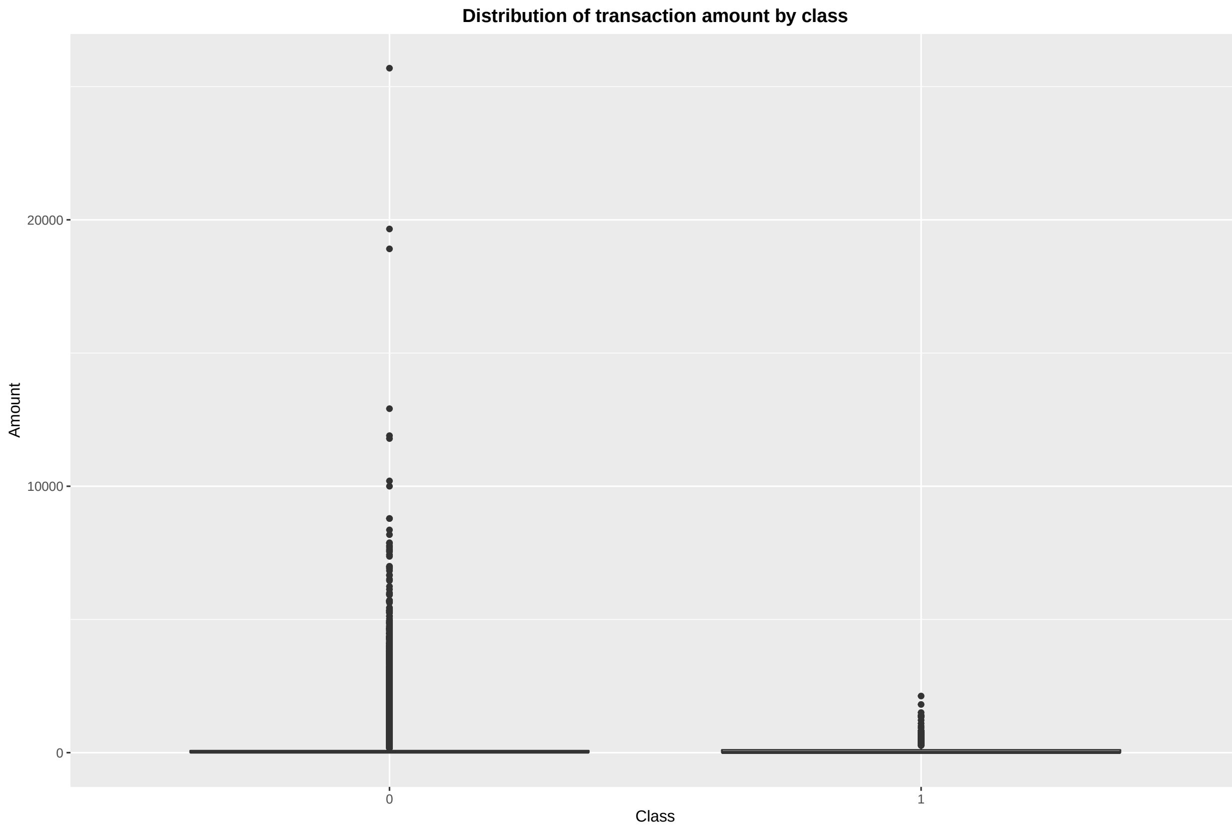
```
df %>%
  ggplot(aes(x = Time, fill = factor(Class))) + geom_histogram(bins = 100)+
  labs(x = 'Time in seconds since first transaction', y = 'No. of transactions') +
  ggtitle('Distribution of time of transaction by class') +
  facet_grid(Class ~ ., scales = 'free_y') + common_theme
```



The 'Time' feature looks pretty similar across both types of transactions. One could argue that fraudulent transactions are more uniformly distributed, while normal transactions have a cyclical distribution.

Distribution of variable 'Amount' by class

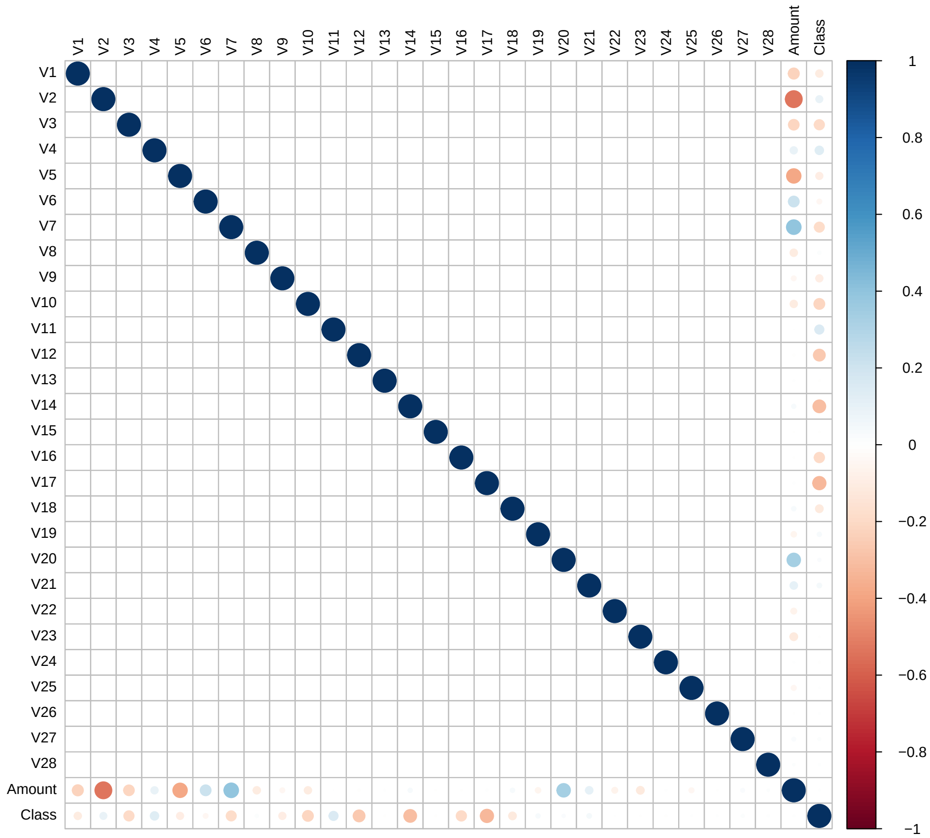
```
ggplot(df, aes(x = factor(Class), y = Amount)) + geom_boxplot() +
labs(x = 'Class', y = 'Amount') +
ggtitle("Distribution of transaction amount by class") + common_theme
```



There is clearly a lot more variability in the transaction values for non-fraudulent transactions.

Correlation of anonymised variables and 'Amount'

```
correlations <- cor(df[,-1],method="pearson")
corrplot(correlations, number.cex = .9, method = "circle", type = "full", tl.cex=0.8,
         tl.col = "black")
```



We observe that most of the data features are not correlated. This is because before publishing, most of the features were presented to a Principal Component Analysis (PCA) algorithm. The features V1 to V28 are most probably the Principal Components resulted after propagating the real features through PCA. We do not know if the numbering of the features reflects the importance of the Principal Components.

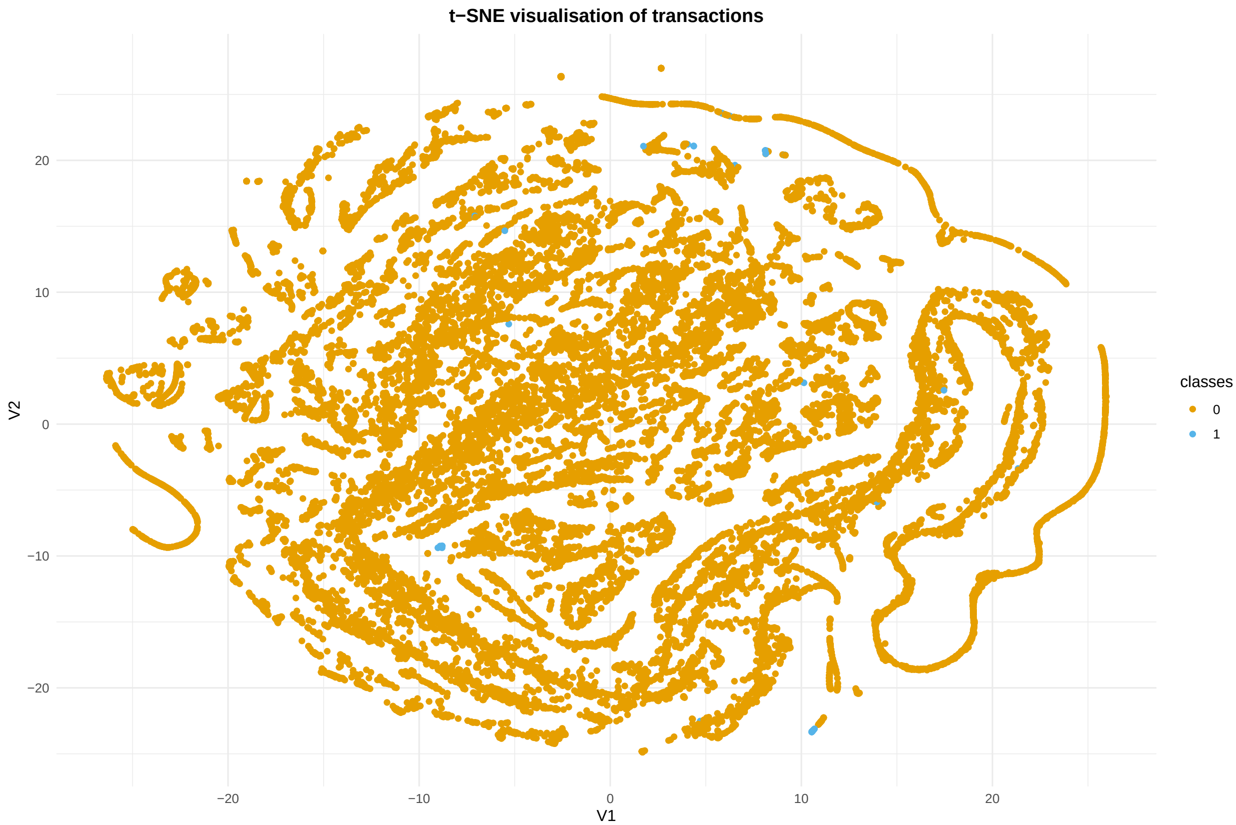
Visualization of transactions using t-SNE

To try to understand the data better, we will try visualizing the data using t-Distributed Stochastic Neighbour Embedding, a technique to reduce dimensionality using Barnes-Hut approximations.

To train the model, perplexity was set to 20.

The visualisation should give us a hint as to whether there exist any "discoverable" patterns in the data which the model could learn. If there is no obvious structure in the data, it is more likely that the model will perform poorly.

```
# Use 10% of data to compute t-SNE
tsne_subset <- 1:as.integer(0.1*nrow(df))
tsne <- Rtsne(df[tsne_subset,-c(1, 31)], perplexity = 20, theta = 0.5, pca = F,
              verbose = F, max_iter = 500, check_duplicates = F)

classes <- as.factor(df$Class[tsne_subset])
tsne_mat <- as.data.frame(tsne$Y)
ggplot(tsne_mat, aes(x = V1, y = V2)) + geom_point(aes(color = classes)) +
  theme_minimal() + common_theme +
  ggtitle("t-SNE visualisation of transactions") +
  scale_color_manual(values = c("#E69F00", "#56B4E9"))
```



There appears to be a separation between the two classes as most fraudulent transactions seem to lie near the edge of the blob of data.

Modeling Approach

Standard machine learning algorithms struggle with accuracy on imbalanced data for the following reasons:

1. ML algorithms struggle with accuracy because of the unequal distribution in dependent variable.This causes the performance of existing classifiers to get biased towards majority class.
2. The algorithms are accuracy driven i.e. they aim to minimize the overall error to which the minority class contributes very little.
3. ML algorithms assume that the data set has balanced class distributions.
4. They also assume that errors obtained from different classes have same cost

The methods to deal with this problem are widely known as 'Sampling Methods'. Generally, these methods aim to modify an imbalanced data into balanced distribution using some mechanism. The modification occurs by altering the size of original data set and provide the same proportion of balance.

These methods have acquired higher importance after many researches have proved that balanced data results in improved overall classification performance compared to an imbalanced data set. Hence, it's important to learn them.

Below are the methods used here to treat the imbalanced dataset:

- Undersampling
- Oversampling
- Synthetic Data Generation

Undersampling

This method reduces the number of observations from majority class to make the data set balanced. This method is best to use when the data set is huge and reducing the number of training samples helps to improve run time and storage troubles.

Undersampling methods are of 2 types: Random and Informative.

Random undersampling method randomly chooses observations from majority class which are eliminated until the data set gets balanced. Informative undersampling follows a pre-specified selection criterion to remove the observations from majority class.

A possible problem with this method is that removing observations may cause the training data to lose important information pertaining to majority class.

Oversampling

This method works with minority class. It replicates the observations from minority class to balance the data. It is also known as upsampling. Similar to undersampling, this method also can be divided into two types: Random Oversampling and Informative Oversampling.

Random oversampling balances the data by randomly oversampling the minority class. Informative oversampling uses a pre-specified criterion and synthetically generates minority class observations.

An advantage of using this method is that it leads to no information loss. The disadvantage of using this method is that, since oversampling simply adds replicated observations in original data set, it ends up adding multiple observations of several types, thus leading to overfitting.

Synthetic Data Generation (SMOTE and ROSE)

In simple words, instead of replicating and adding the observations from the minority class, it overcome imbalances by generates artificial data. It is also a type of oversampling technique.

In regards to synthetic data generation, synthetic minority oversampling technique (SMOTE) is a powerful and widely used method. SMOTE algorithm draws artificial samples by choosing points that lie on the line connecting the rare observation to one of its nearest neighbors in the feature space. ROSE (random over-sampling examples) uses smoothed bootstrapping to draw artificial samples from the feature space neighbourhood around the minority class.

It is important to note that sampling techniques should only be applied to the training set and not the testing set.

Our modeling approach will involve training a single classifier on the train set with class imbalance suitably altered using each of the techniques above. Depending on which technique yields the best roc-auc score on a holdout test set. we will build subsequent models using that chosen technique.

Data Preparation

'Time' feature does not indicate the actual time of the transaction and is more of listing the data in chronological order. Based on the data visualization above we assume that 'Time' feature has little or no significance in correctly classifying a fraud transaction and hence eliminate this column from further analysis.

```r
#Remove 'Time' variable
df <- df[,-1]

#Change 'Class' variable to factor
df$Class <- as.factor(df$Class)
levels(df$Class) <- c("Not_Fraud", "Fraud")

#Scale numeric variables
```

```
df[,-30] <- scale(df[,-30])
```

Split data into train and test sets

```
set.seed(123)
split <- sample.split(df$Class, SplitRatio = 0.7)
train <-  subset(df, split == TRUE)
test <- subset(df, split == FALSE)
```

Choosing sampling technique

Let us create different versions of the training set as per sampling technique

```
table(train$Class)
```

```
##
## Not_Fraud     Fraud
##    199020       344
```

```
set.seed(9560)
down_train <- downSample(x = train[, -ncol(train)],
                         y = train$Class)
table(down_train$Class)
```

```
##
## Not_Fraud     Fraud
##       344       344
```

```
set.seed(9560)
up_train <- upSample(x = train[, -ncol(train)],
                         y = train$Class)
table(up_train$Class)
```

```
##
## Not_Fraud     Fraud
##    199020    199020
```

```
set.seed(9560)
smote_train <- SMOTE(Class ~ ., data  = train)

table(smote_train$Class)
```

```
##
## Not_Fraud     Fraud
##      1376      1032
```

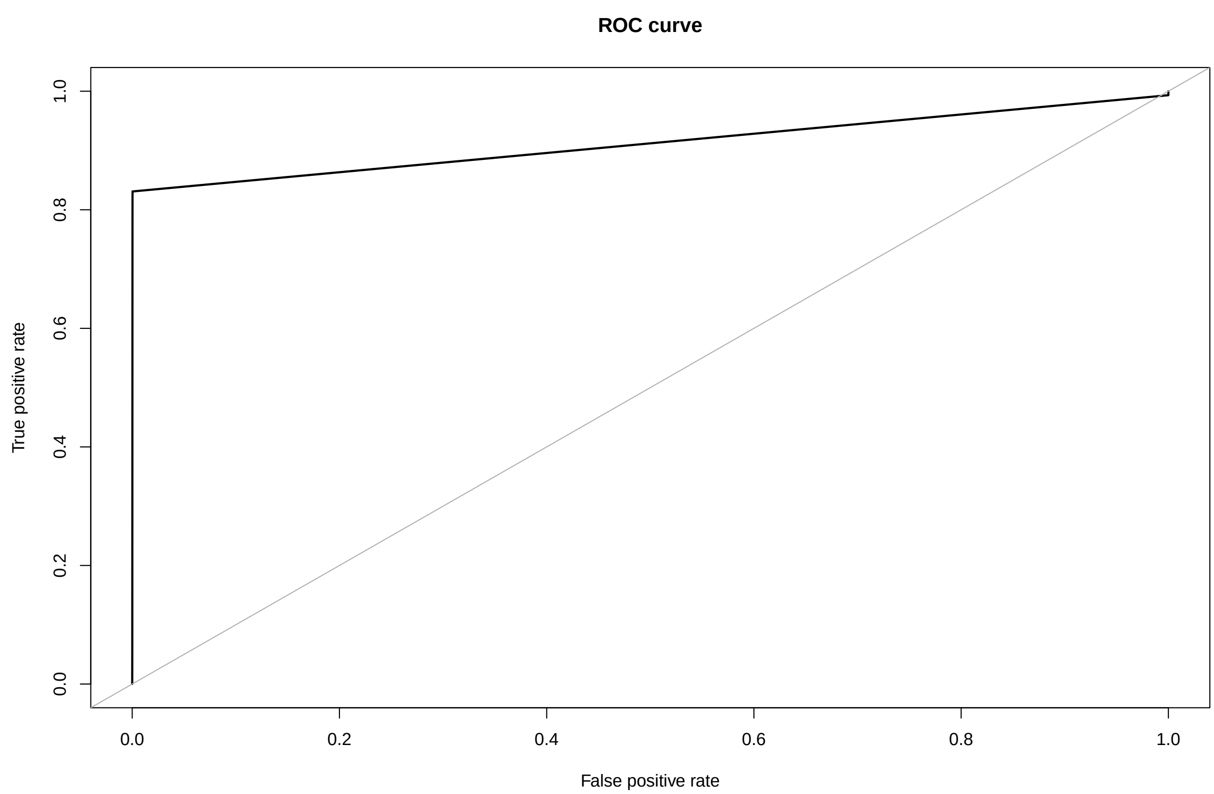
```
set.seed(9560)
rose_train <- ROSE(Class ~ ., data  = train)$data

table(rose_train$Class)
```

```
##
## Not_Fraud      Fraud
##     99844      99520
```

We choose CART(classification and regression tree) as first model.

Before we start using sampling let us first look at how CART performs with imbalanced data. We use the function *roc.curve* available in the ROSE package to gauge model performance on the test set.

```
#CART Model Performance on imbalanced data
set.seed(5627)

orig_fit <- rpart(Class ~ ., data = train)

#Evaluate model performance on test set
pred_orig <- predict(orig_fit, newdata = test, method = "class")

roc.curve(test$Class, pred_orig[,2], plotit = TRUE)
```




```
## Area under the curve (AUC): 0.912
```

We evaluate the model performance on test data by finding the roc auc score

We see that the auc score on the original dataset is 0.912 . We will now apply various sampling techniques to the data and see the performance on the test set.

```r
set.seed(5627)
# Build down-sampled model


down_fit <- rpart(Class ~ ., data = down_train)


set.seed(5627)
# Build up-sampled model


up_fit <- rpart(Class ~ ., data = up_train)


set.seed(5627)
# Build smote model


smote_fit <- rpart(Class ~ ., data = smote_train)

set.seed(5627)
# Build rose model


rose_fit <- rpart(Class ~ ., data = rose_train)
```

```r
pred_down <- predict(down_fit, newdata = test)

print('Fitting downsampled model to test data')
```

```
## [1] "Fitting downsampled model to test data"
```

```r
roc.curve(test$Class, pred_down[,2], plotit = FALSE)
```

```
## Area under the curve (AUC): 0.942
```

```r
pred_up <- predict(up_fit, newdata = test)

print('Fitting upsampled model to test data')
```

```
## [1] "Fitting upsampled model to test data"
```

```r
roc.curve(test$Class, pred_up[,2], plotit = FALSE)
```

```
## Area under the curve (AUC): 0.943
```

```r
pred_smote <- predict(smote_fit, newdata = test)

print('Fitting smote model to test data')
```

```
## [1] "Fitting smote model to test data"
```

```r
roc.curve(test$Class, pred_smote[,2], plotit = FALSE)
```

```
## Area under the curve (AUC): 0.934
```

```r
pred_rose <- predict(rose_fit, newdata = test)

print('Fitting rose model to test data')
```

```
## [1] "Fitting rose model to test data"
```

```r
roc.curve(test$Class, pred_rose[,2], plotit = FALSE)
```

```
## Area under the curve (AUC): 0.942
```

We see that all the sampling techniques have yielded better auc scores than the simple imbalanced dataset. We will test different models now using the **up sampling technique** as that has given the highest auc score.

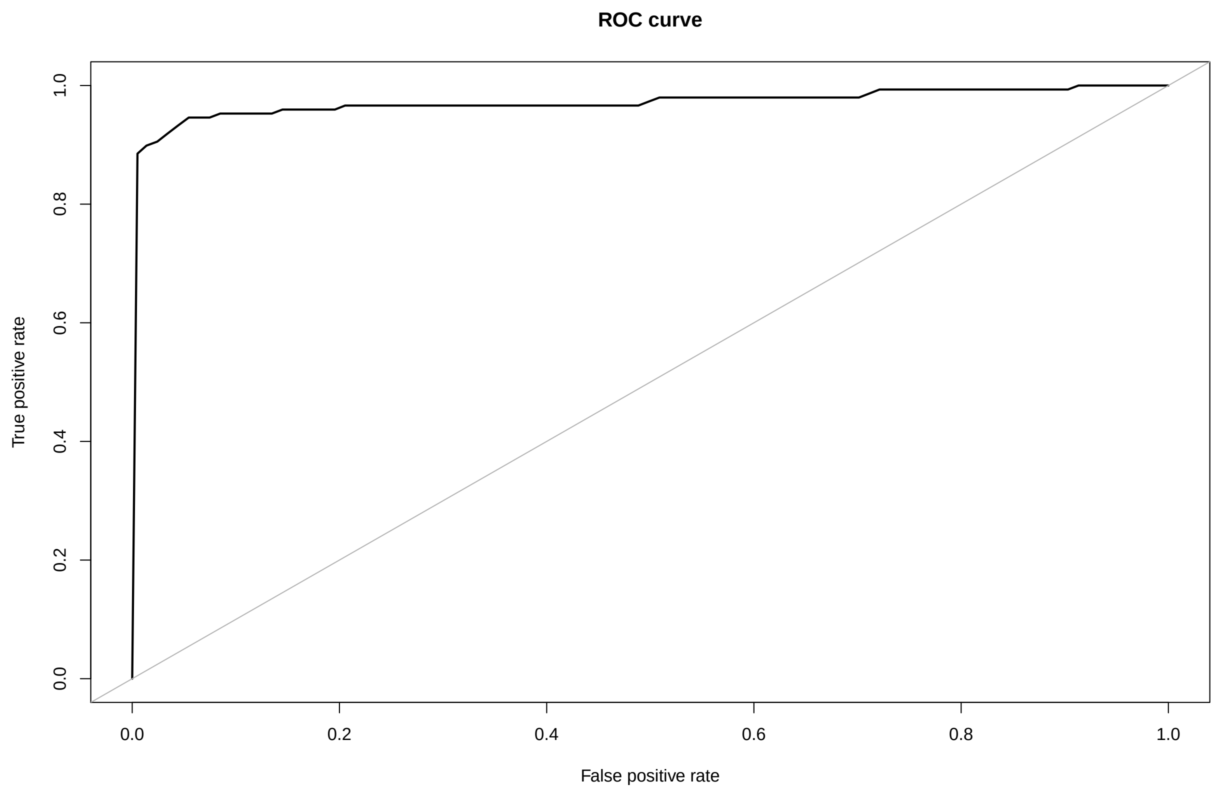
Results

Specifically the following models will be tested:

- logistic regression (GLM)
- random forest (RF)
- xgboost (XGB) 1

GLM Fit

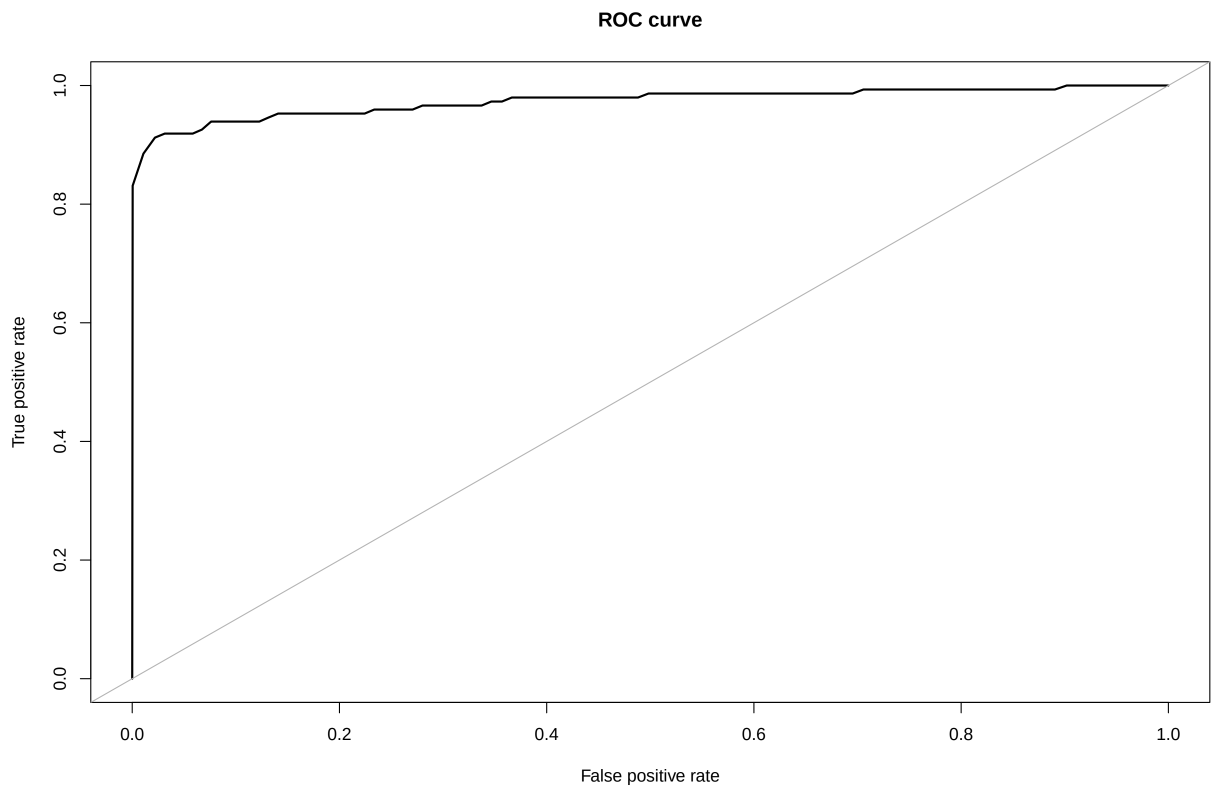
```r
glm_fit <- glm(Class ~ ., data = up_train, family = 'binomial')

pred_glm <- predict(glm_fit, newdata = test, type = 'response')

roc.curve(test$Class, pred_glm, plotit = TRUE)
```




```
## Area under the curve (AUC): 0.971
```

RF Fit (we use the Rborist package)

```r
x = up_train[, -30]
y = up_train[,30]

rf_fit <- Rborist(x, y, ntree = 500, minNode = 20, maxLeaf = 13)


rf_pred <- predict(rf_fit, test[,-30], ctgCensus = "prob")
prob <- rf_pred$prob

roc.curve(test$Class, prob[,2], plotit = TRUE)
```

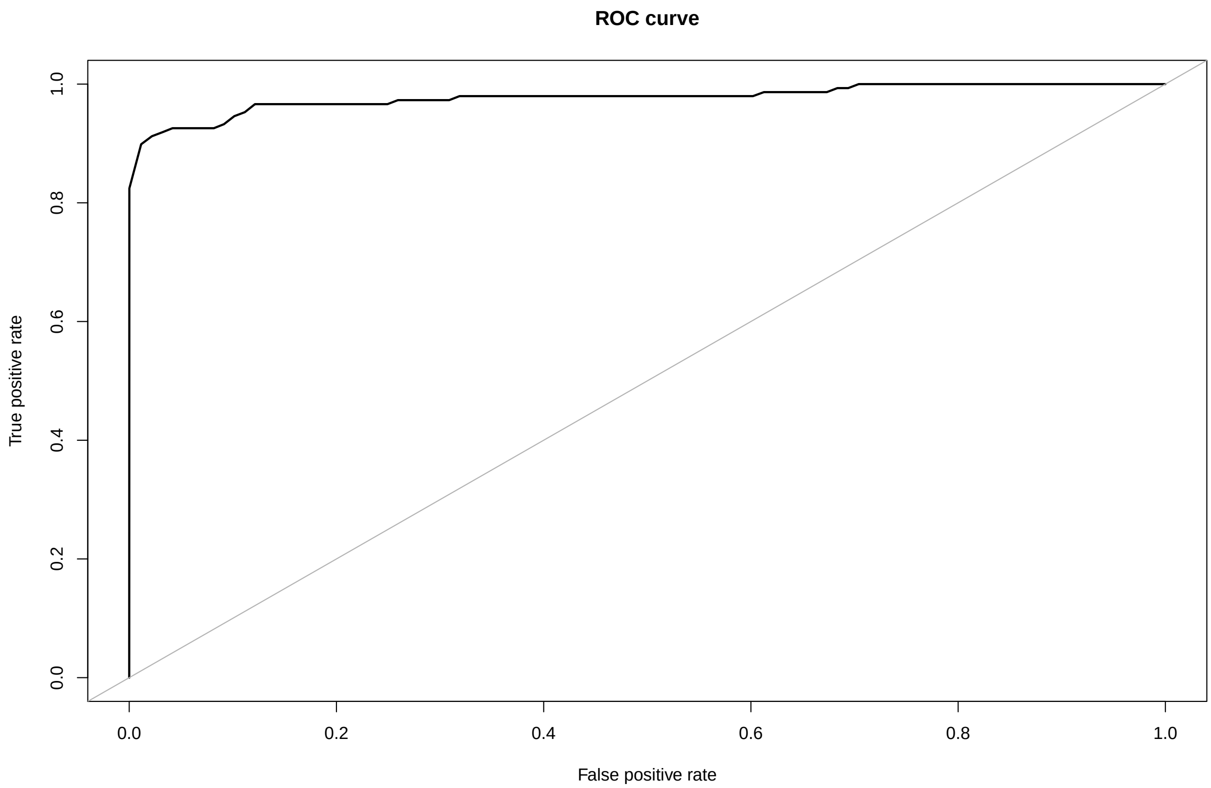



```
## Area under the curve (AUC): 0.973
```

XGB Fit

```r
#COnvert class labels from factor to numeric

labels <- up_train$Class

y <- recode(labels, 'Not_Fraud' = 0, "Fraud" = 1)


xgb <- xgboost(data = data.matrix(up_train[,-30]),
 label = y,
 eta = 0.1,
 gamma = 0.1,
 max_depth = 10,
 nrounds = 300,
 objective = "binary:logistic",
 colsample_bytree = 0.6,
 verbose = 0,
 nthread = 7,
 seed = 42
)
```

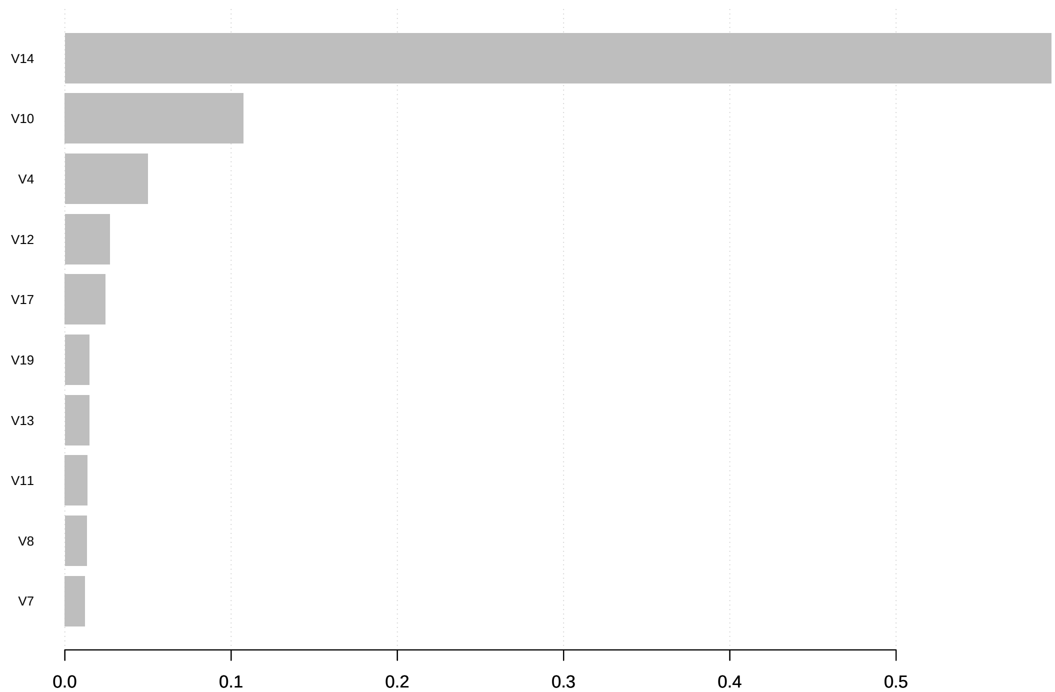
```r
xgb_pred <- predict(xgb, data.matrix(test[,-30]))

roc.curve(test$Class, xgb_pred, plotit = TRUE)
```




```
## Area under the curve (AUC): 0.977
```

We can also take a look at the important features here.

```r
names <- dimnames(data.matrix(up_train[,-30]))[[2]]

# Compute feature importance matrix
importance_matrix <- xgb.importance(names, model = xgb)
# Nice graph
xgb.plot.importance(importance_matrix[1:10,])
```



With an auc score of 0.977 the XGBOOST model has performed the best though both the random forest and logistic regression models have shown reasonable performance.

Conclusion

In this project we have tried to show different methods of dealing with unbalanced datasets like the fraud credit card transaction dataset where the instances of fraudulent cases is few compared to the instances of normal transactions. We have argued why accuracy is not a appropriate measure of model performance here and used the metric AREA UNDER ROC CURVE to evaluate how different methods of oversampling or undersampling the response variable can lead to better model training. We concluded that the oversampling technique works best on the dataset and achieved significant improvement in model performance over the imbalanced data. The best score of 0.977 was achieved using an XGBOOST model though both random forest and logistic regression models performed well too. It is likely that by further tuning the XGBOOST model parameters we can achieve even better performance. However this exercise has demonstrated the importance of sampling in effectively modelling and predicting with an imbalanced dataset.