HarvardX PH125.9x Credit Card Fraud Prediction Project

Rogelio Montemayor

June 25, 2021

1 Introduction and overview

1.1 Introduction

The goal of this project is to create a model that identifies fraudulent credit card transactions.

The dataset contains transactions made over two days in September 2013 by European cardholders. The main problem with this type of problem is that the data is highly unbalanced. Only 0.173% of the transactions in this dataset are fraudulent. There are 284,807 transactions of which 492 are fraudulent transactions.

Due to privacy and confidentiality issues, features ***V1*** to ***V28*** are the principal components of the original features after PCA transformation. There are only two features that are not transformed: ***Time*** and ***Amount***. The target variable is ***Class***, and it takes the value of 1 in case of fraud and 0 otherwise.

This is a classification project.

It is recommended to use Area Under the Precision-Recall Curve (AUPRC) to measure accuracy because the classes are so unbalanced. Even though Area Under Receiver Operating Characteristic (AUROC) is more common, it is not recommended for highly unbalanced classification. AUPRC was used as the metric to measure performance in the cross-validation our models.

The best results were obtained using an Adaptive Boosting Trees algorithm (adaboost). The F1 score of the best model was **0.8506** as measured on the validation set. F1 is the harmonic mean of Precision and Recall.

Credit card fraud increases costs for everyone and machine learning techniques can help flag fraudulent transactions and lower costs for banks, merchants and their customers.

1.1.1 Acknowledgements

My version of the dataset was downloaded from Kaggle: https://www.kaggle.com/mlg-ulb/creditcardfraud.

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.

I want to thank Max Kuhn: https://topepo.github.io/caret/ and DataCamp: https://www.datacamp.com for their help.

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.

Please refer to the following papers:
Andrea Dal Pozzolo, Olivier Caelen, Reid A. Johnson and Gianluca Bontempi. ***Calibrating Probability with Undersampling for Unbalanced Classification***. In Symposium on Computational Intelligence and Data Mining (CIDM), IEEE, 2015
Dal Pozzolo, Andrea; Caelen, Olivier; Le Borgne, Yann-Ael; Waterschoot, Serge; Bontempi, Gianluca. ***Learned lessons in credit card fraud detection from a practitioner perspective***, Expert systems with applications, 41,10,4915-4928,2014, Pergamon
Dal Pozzolo, Andrea; Boracchi, Giacomo; Caelen, Olivier; Alippi, Cesare; Bontempi, Gianluca. ***Credit card fraud detection: a realistic modeling and a novel learning strategy***, IEEE transactions on neural networks and learning systems, 29,8,3784-3797,2018,IEEE
Dal Pozzolo, Andrea ***Adaptive Machine learning for credit card fraud detection*** ULB MLG PhD thesis (supervised by G. Bontempi)
Carcillo, Fabrizio; Dal Pozzolo, Andrea; Le Borgne, Yann-Aël; Caelen, Olivier; Mazzer, Yannis; Bontempi, Gianluca. ***Scarff: a scalable framework for streaming credit card fraud detection with Spark***, Information fusion,41, 182-194,2018,Elsevier
Carcillo, Fabrizio; Le Borgne, Yann-Aël; Caelen, Olivier; Bontempi, Gianluca. ***Streaming active learning strategies for real-life credit card fraud detection: assessment and visualization***, International Journal of Data Science and Analytics, 5,4,285-300,2018,Springer International Publishing
Bertrand Lebichot, Yann-Aël Le Borgne, Liyun He, Frederic Oblé, Gianluca Bontempi ***Deep-Learning Domain Adaptation Techniques for Credit Cards Fraud Detection***, INNSBDDL 2019: Recent Advances in Big Data and Deep Learning, pp 78-88, 2019
Fabrizio Carcillo, Yann-Aël Le Borgne, Olivier Caelen, Frederic Oblé, Gianluca Bontempi ***Combining Unsupervised and Supervised Learning in Credit Card Fraud Detection*** Information Sciences, 2019
Yann-Aël Le Borgne, Gianluca Bontempi ***Machine Learning for Credit Card Fraud Detection*** - Practical Handbook

1.2 Overview

These are the steps I followed to go from raw dataset to model and insights:

- Decompress, Read, and Build the Dataset
- Analysis
    - Initial Exploratory Analysis
    - Visual Analysis
    - Data Cleaning and Feature Engineering

    - Modeling Approach
        - Split the Dataset
        - Pre-process and Setup
        - Train Models
        - Algorithm Selection
- Results and Final Model
- Conclusion and Insights

1.3 Read in the compressed file

The file with the credit card data is in the same directory (the current working directory). To comply with Github limitations, the file was compressed (creditcard.csv.zip). We can read it and unzip it in one step:

```r
# Note: this process takes 1-2 minutes
df <- read_csv("creditcard.csv.zip")
```

The readr function assigned the *numeric* class to all our columns. We need to change the target variable *Class* to integer:

```r
df$Class <- as.integer(df$Class)
```

2 Analysis

Our first step is to explore the data and get a sense of the way the information is presented, to get a better understanding of the features and to get ideas for data cleaning or feature engineering.

2.1 Initial Exploratory Analysis

We know that features *V1* to *V28* are the PCA components of the original features. We can also see the *Time* and *Amount* features.

```r
head(df)
```

```
# A tibble: 6 x 31
   Time     V1      V2    V3     V4      V5      V6      V7      V8     V9
  <dbl>  <dbl>   <dbl> <dbl>  <dbl>   <dbl>   <dbl>   <dbl>   <dbl>  <dbl>
1     0 -1.36  -0.0728 2.54   1.38  -0.338   0.462   0.240   0.0987  0.364
2     0  1.19   0.266  0.166  0.448  0.0600 -0.0824 -0.0788  0.0851 -0.255
3     1 -1.36  -1.34   1.77   0.380 -0.503   1.80    0.791   0.248  -1.51
4     1 -0.966 -0.185  1.79  -0.863 -0.0103  1.25    0.238   0.377  -1.39
5     2 -1.16   0.878  1.55   0.403 -0.407   0.0959  0.593  -0.271   0.818
6     2 -0.426  0.961  1.14  -0.168  0.421  -0.0297  0.476   0.260  -0.569
# ... with 21 more variables: V10 <dbl>, V11 <dbl>, V12 <dbl>, V13 <dbl>,
#   V14 <dbl>, V15 <dbl>, V16 <dbl>, V17 <dbl>, V18 <dbl>, V19 <dbl>,
#   V20 <dbl>, V21 <dbl>, V22 <dbl>, V23 <dbl>, V24 <dbl>, V25 <dbl>,
#   V26 <dbl>, V27 <dbl>, V28 <dbl>, Amount <dbl>, Class <int>
```

```
glimpse(df)
```

```
Rows: 284,807
Columns: 31
$ Time   <dbl> 0, 0, 1, 1, 2, 2, 4, 7, 7, 9, 10, 10, 10, 11, 12, 12, 12, 13, 1~
$ V1     <dbl> -1.3598071, 1.1918571, -1.3583541, -0.9662717, -1.1582331, -0.4~
$ V2     <dbl> -0.07278117, 0.26615071, -1.34016307, -0.18522601, 0.87773675, ~
$ V3     <dbl> 2.53634674, 0.16648011, 1.77320934, 1.79299334, 1.54871785, 1.1~
$ V4     <dbl> 1.37815522, 0.44815408, 0.37977959, -0.86329128, 0.40303393, -0~
$ V5     <dbl> -0.33832077, 0.06001765, -0.50319813, -0.01030888, -0.40719338,~
$ V6     <dbl> 0.46238778, -0.08236081, 1.80049938, 1.24720317, 0.09592146, -0~
$ V7     <dbl> 0.239598554, -0.078802983, 0.791460956, 0.237608940, 0.59294074~
$ V8     <dbl> 0.098697901, 0.085101655, 0.247675787, 0.377435875, -0.27053267~
$ V9     <dbl> 0.3637870, -0.2554251, -1.5146543, -1.3870241, 0.8177393, -0.56~
$ V10    <dbl> 0.09079417, -0.16697441, 0.20764287, -0.05495192, 0.75307443, -~
$ V11    <dbl> -0.55159953, 1.61272666, 0.62450146, -0.22648726, -0.82284288, ~
$ V12    <dbl> -0.61780086, 1.06523531, 0.06608369, 0.17822823, 0.53819555, 0.~
$ V13    <dbl> -0.99138985, 0.48909502, 0.71729273, 0.50775687, 1.34585159, -0~
$ V14    <dbl> -0.31116935, -0.14377230, -0.16594592, -0.28792375, -1.11966983~
$ V15    <dbl> 1.468176972, 0.635558093, 2.345864949, -0.631418118, 0.17512113~
$ V16    <dbl> -0.47040053, 0.46391704, -2.89008319, -1.05964725, -0.45144918,~
$ V17    <dbl> 0.207971242, -0.114804663, 1.109969379, -0.684092786, -0.237033~
$ V18    <dbl> 0.02579058, -0.18336127, -0.12135931, 1.96577500, -0.03819479, ~
$ V19    <dbl> 0.40399296, -0.14578304, -2.26185710, -1.23262197, 0.80348692, ~
$ V20    <dbl> 0.25141210, -0.06908314, 0.52497973, -0.20803778, 0.40854236, 0~
$ V21    <dbl> -0.018306778, -0.225775248, 0.247998153, -0.108300452, -0.00943~
$ V22    <dbl> 0.277837576, -0.638671953, 0.771679402, 0.005273597, 0.79827849~
$ V23    <dbl> -0.110473910, 0.101288021, 0.909412262, -0.190320519, -0.137458~
$ V24    <dbl> 0.06692807, -0.33984648, -0.68928096, -1.17557533, 0.14126698, ~
$ V25    <dbl> 0.12853936, 0.16717040, -0.32764183, 0.64737603, -0.20600959, -~
$ V26    <dbl> -0.18911484, 0.12589453, -0.13909657, -0.22192884, 0.50229222, ~
$ V27    <dbl> 0.133558377, -0.008983099, -0.055352794, 0.062722849, 0.2194222~
$ V28    <dbl> -0.021053053, 0.014724169, -0.059751841, 0.061457629, 0.2151531~
$ Amount <dbl> 149.62, 2.69, 378.66, 123.50, 69.99, 3.67, 4.99, 40.80, 93.20, ~
$ Class  <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ~
```

Let’s take a look at the last few lines of our data as well

```
tail(df)
```

```
# A tibble: 6 x 31
    Time      V1      V2     V3     V4      V5     V6      V7     V8     V9
   <dbl>   <dbl>   <dbl>  <dbl>  <dbl>   <dbl>  <dbl>   <dbl>  <dbl>   <dbl>
1 172785   0.120  0.931  -0.546 -0.745  1.13   -0.236  0.813   0.115 -0.204
```

```
2 172786 -11.9   10.1    -9.83  -2.07  -5.36   -2.61  -4.92    7.31   1.91
3 172787  -0.733 -0.0551  2.04  -0.739  0.868   1.06   0.0243  0.295  0.585
4 172788   1.92  -0.301  -3.25  -0.558  2.63    3.03  -0.297   0.708  0.432
5 172788  -0.240  0.530   0.703  0.690 -0.378   0.624 -0.686   0.679  0.392
6 172792  -0.533 -0.190   0.703 -0.506 -0.0125 -0.650  1.58   -0.415  0.486
# ... with 21 more variables: V10 <dbl>, V11 <dbl>, V12 <dbl>, V13 <dbl>,
#   V14 <dbl>, V15 <dbl>, V16 <dbl>, V17 <dbl>, V18 <dbl>, V19 <dbl>,
#   V20 <dbl>, V21 <dbl>, V22 <dbl>, V23 <dbl>, V24 <dbl>, V25 <dbl>,
#   V26 <dbl>, V27 <dbl>, V28 <dbl>, Amount <dbl>, Class <int>
```

Time is the time in seconds between transactions and *Amount* is the amount of the transaction. We can readily see that the values of *Amount* are on a different scale than the other values.

The dataset has 284,807 transactions and 31 columns (30 features and our target variable):

```
[1] 284807     31
```

We can see that *Time* is 7 seconds short of 48 hours (172,800 seconds):

```
      Time              V1                  V2                  V3
 Min.   :     0   Min.   :-56.40751   Min.   :-72.71573   Min.   :-48.3256
 1st Qu.: 54202   1st Qu.: -0.92037   1st Qu.: -0.59855   1st Qu.: -0.8904
 Median : 84692   Median :  0.01811   Median :  0.06549   Median :  0.1799
 Mean   : 94814   Mean   :  0.00000   Mean   :  0.00000   Mean   :  0.0000
 3rd Qu.:139320   3rd Qu.:  1.31564   3rd Qu.:  0.80372   3rd Qu.:  1.0272
 Max.   :172792   Max.   :  2.45493   Max.   : 22.05773   Max.   :  9.3826
       V4                 V5                   V6                 V7
 Min.   :-5.68317   Min.   :-113.74331   Min.   :-26.1605   Min.   :-43.5572
 1st Qu.:-0.84864   1st Qu.:  -0.69160   1st Qu.: -0.7683   1st Qu.: -0.5541
 Median :-0.01985   Median :  -0.05434   Median : -0.2742   Median :  0.0401
 Mean   : 0.00000   Mean   :   0.00000   Mean   :  0.0000   Mean   :  0.0000
 3rd Qu.: 0.74334   3rd Qu.:   0.61193   3rd Qu.:  0.3986   3rd Qu.:  0.5704
 Max.   :16.87534   Max.   :  34.80167   Max.   : 73.3016   Max.   :120.5895
       V8                  V9                 V10                 V11
 Min.   :-73.21672   Min.   :-13.43407   Min.   :-24.58826   Min.   :-4.79747
 1st Qu.: -0.20863   1st Qu.: -0.64310   1st Qu.: -0.53543   1st Qu.:-0.76249
 Median :  0.02236   Median : -0.05143   Median : -0.09292   Median :-0.03276
 Mean   :  0.00000   Mean   :  0.00000   Mean   :  0.00000   Mean   : 0.00000
 3rd Qu.:  0.32735   3rd Qu.:  0.59714   3rd Qu.:  0.45392   3rd Qu.: 0.73959
 Max.   : 20.00721   Max.   : 15.59500   Max.   : 23.74514   Max.   :12.01891
      V12                V13                V14                V15
 Min.   :-18.6837   Min.   :-5.79188   Min.   :-19.2143   Min.   :-4.49894
 1st Qu.: -0.4056   1st Qu.:-0.64854   1st Qu.: -0.4256   1st Qu.:-0.58288
 Median :  0.1400   Median :-0.01357   Median :  0.0506   Median : 0.04807
 Mean   :  0.0000   Mean   : 0.00000   Mean   :  0.0000   Mean   : 0.00000
 3rd Qu.:  0.6182   3rd Qu.: 0.66251   3rd Qu.:  0.4931   3rd Qu.: 0.64882
 Max.   :  7.8484   Max.   : 7.12688   Max.   : 10.5268   Max.   : 8.87774
```

```
     V16                  V17                  V18           
 Min.   :-14.12985   Min.   :-25.16280   Min.   :-9.498746  
 1st Qu.: -0.46804   1st Qu.: -0.48375   1st Qu.:-0.498850  
 Median :  0.06641   Median : -0.06568   Median :-0.003636  
 Mean   :  0.00000   Mean   :  0.00000   Mean   : 0.000000  
 3rd Qu.:  0.52330   3rd Qu.:  0.39968   3rd Qu.: 0.500807  
 Max.   : 17.31511   Max.   :  9.25353   Max.   : 5.041069  
     V19                  V20                  V21           
 Min.   :-7.213527   Min.   :-54.49772   Min.   :-34.83038  
 1st Qu.:-0.456299   1st Qu.: -0.21172   1st Qu.: -0.22839  
 Median : 0.003735   Median : -0.06248   Median : -0.02945  
 Mean   : 0.000000   Mean   :  0.00000   Mean   :  0.00000  
 3rd Qu.: 0.458949   3rd Qu.:  0.13304   3rd Qu.:  0.18638  
 Max.   : 5.591971   Max.   : 39.42090   Max.   : 27.20284  
     V22                   V23                  V24          
 Min.   :-10.933144   Min.   :-44.80774   Min.   :-2.83663  
 1st Qu.: -0.542350   1st Qu.: -0.16185   1st Qu.:-0.35459  
 Median :  0.006782   Median : -0.01119   Median : 0.04098  
 Mean   :  0.000000   Mean   :  0.00000   Mean   : 0.00000  
 3rd Qu.:  0.528554   3rd Qu.:  0.14764   3rd Qu.: 0.43953  
 Max.   : 10.503090   Max.   : 22.52841   Max.   : 4.58455  
     V25                  V26                 V27            
 Min.   :-10.29540   Min.   :-2.60455   Min.   :-22.565679  
 1st Qu.: -0.31715   1st Qu.:-0.32698   1st Qu.: -0.070840  
 Median :  0.01659   Median :-0.05214   Median :  0.001342  
 Mean   :  0.00000   Mean   : 0.00000   Mean   :  0.000000  
 3rd Qu.:  0.35072   3rd Qu.: 0.24095   3rd Qu.:  0.091045  
 Max.   :  7.51959   Max.   : 3.51735   Max.   : 31.612198  
     V28                Amount              Class         
 Min.   :-15.43008   Min.   :    0.00   Min.   :0.000000  
 1st Qu.: -0.05296   1st Qu.:    5.60   1st Qu.:0.000000  
 Median :  0.01124   Median :   22.00   Median :0.000000  
 Mean   :  0.00000   Mean   :   88.35   Mean   :0.001728  
 3rd Qu.:  0.07828   3rd Qu.:   77.17   3rd Qu.:0.000000  
 Max.   : 33.84781   Max.   :25691.16   Max.   :1.000000  
```

We can also make a mental note that *Amount* is skewed to the right: 75% of transactions are below € 77 and the maximum transaction is for € 25,691. This will need further exploration.

There seems to be no missing values:

```r
colSums(is.na(df))
```

```
  Time     V1     V2     V3     V4     V5     V6     V7     V8     V9    V10 
     0      0      0      0      0      0      0      0      0      0      0 
   V11    V12    V13    V14    V15    V16    V17    V18    V19    V20    V21 
```

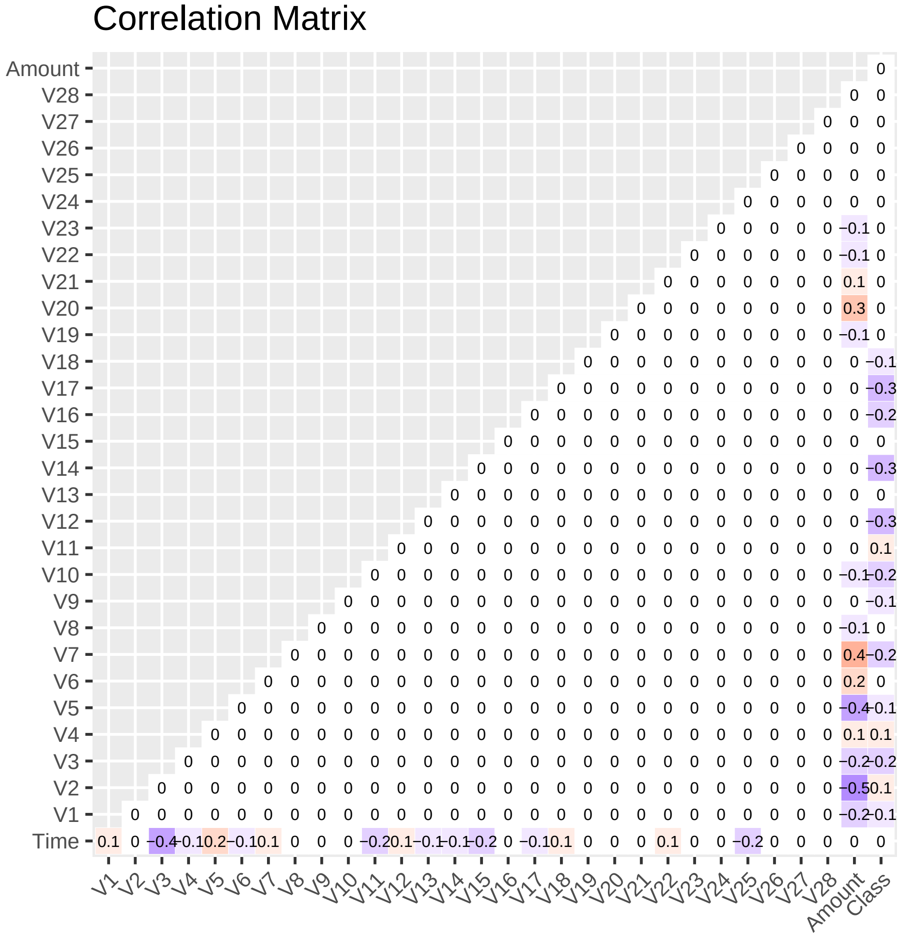
```
       0        0        0        0        0        0        0        0        0        0        0
     V22      V23      V24      V25      V26      V27      V28   Amount    Class
       0        0        0        0        0        0        0        0        0
```

And we can confirm that the frauds are 0.173% of the transactions (492 out of 284,807):

```
     0      1
284315    492
```

```
[1] 0.001727486
```

Now, let's take a look at the correlations between variables:



The PCA components, per design, have no correlation with one another. Correlations between *Amount*, *Time*, and *Class* and with the rest of the features are small.

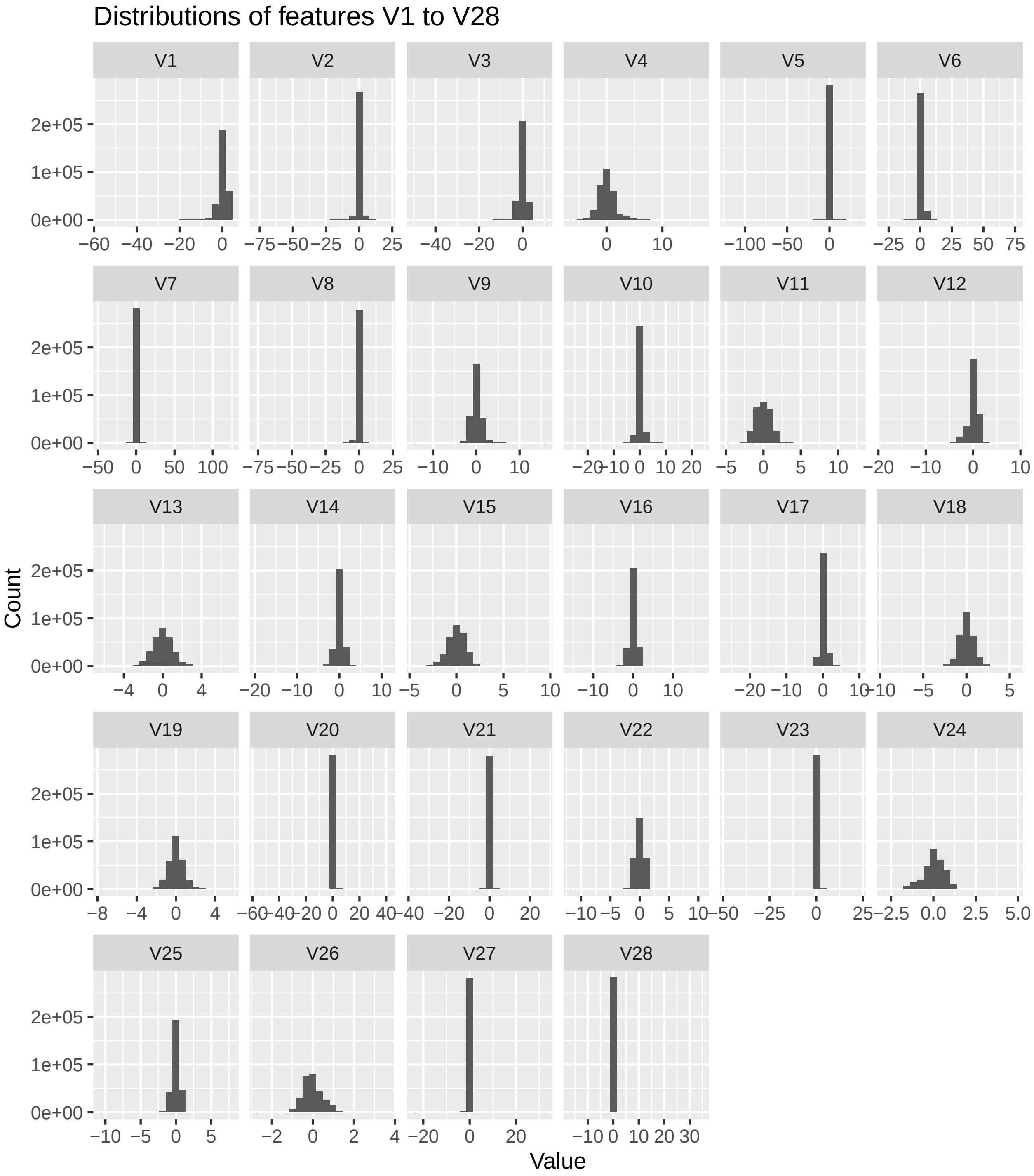
Once the correlations have been calculated, I will convert *Class* to a factor with levels **Yes** and **No**. This will help with a couple of our plots and will be useful as well at the modeling stage when we use the caret package.

```r
df$Class = as.factor(ifelse(df$Class == 1, "Yes", "No"))
```

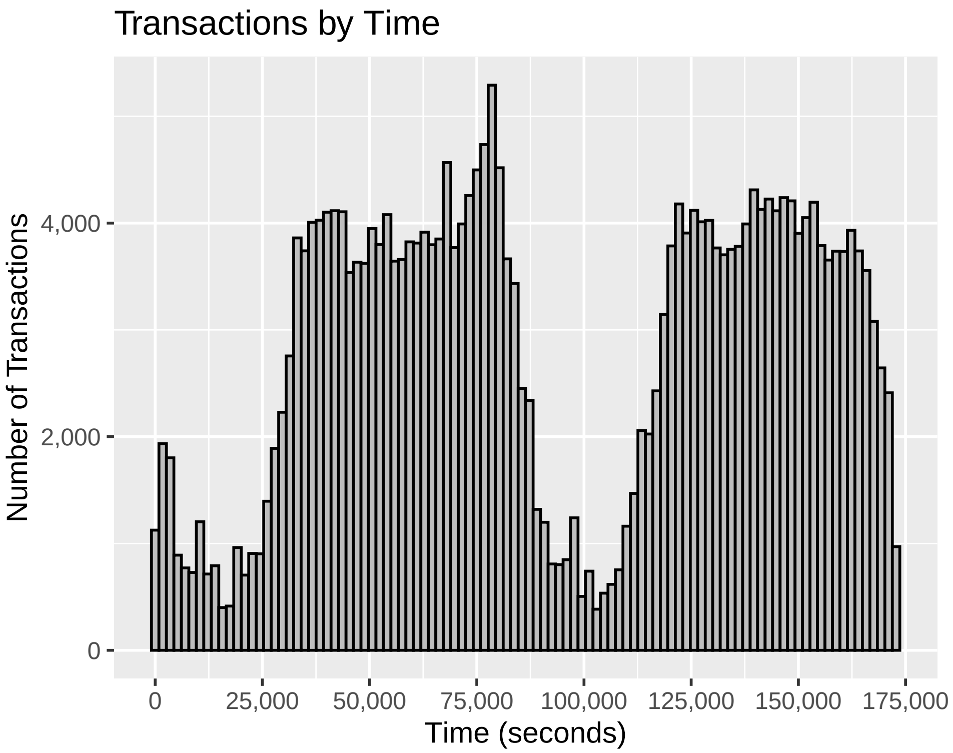
2.2 Visual Analysis

Let’s now take a visual tour of the features.

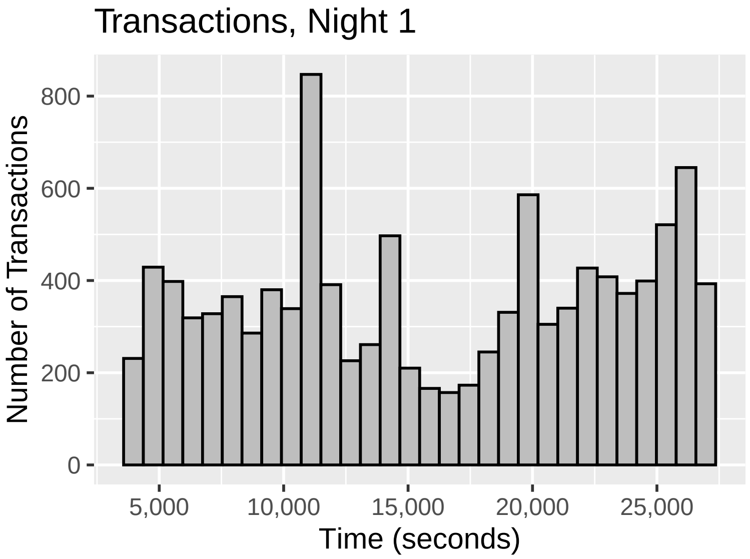
We can start by taking a look at the distributions of the PCA components of the original features, columns *V1* to *V28*:

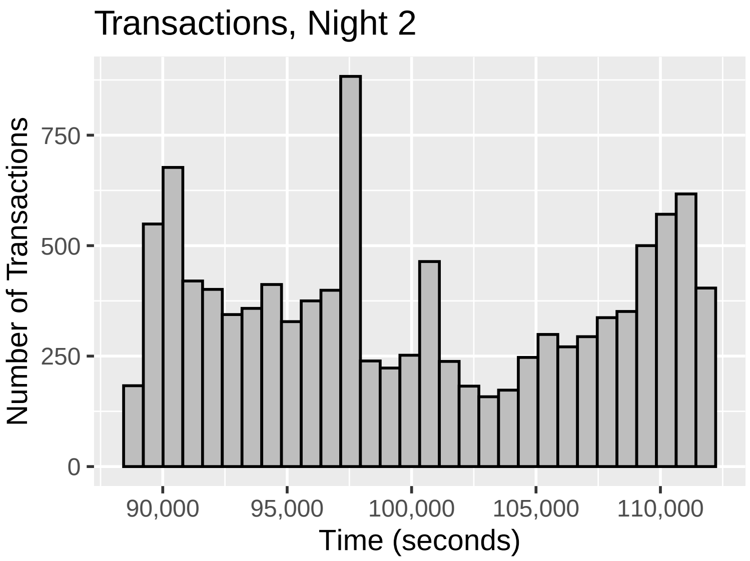


V1 to *V28* are centered around zero as expected. We can now take a closer look at the distribution of *Time*:

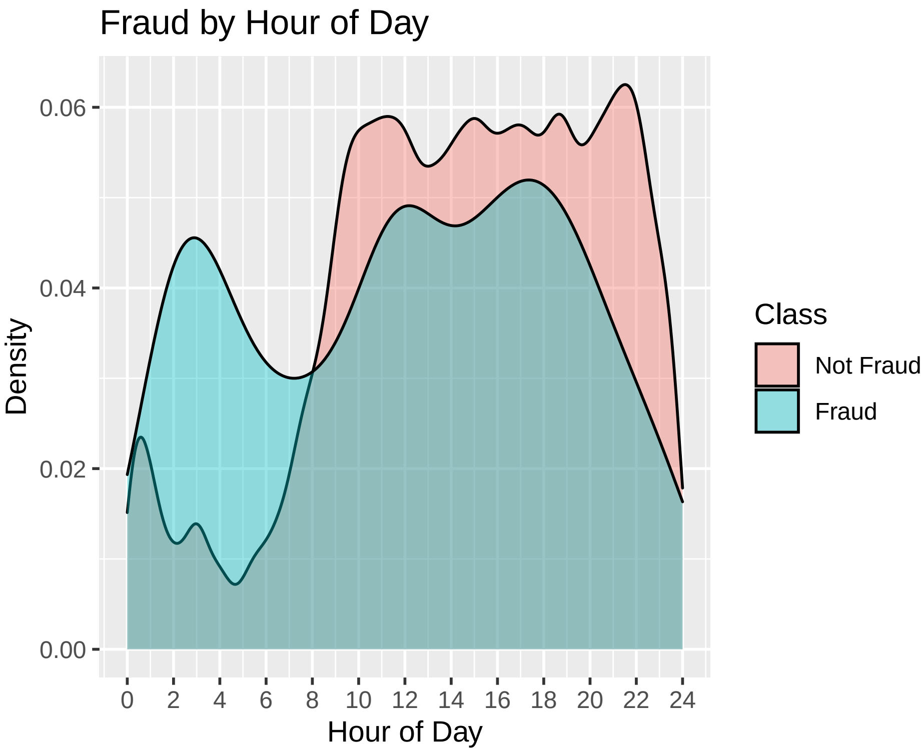


Since we know the data represents transactions over 2 days, we can see the low points of both nights at around values 15,000 and 100,000. Let's look at those two slots in greater detail:



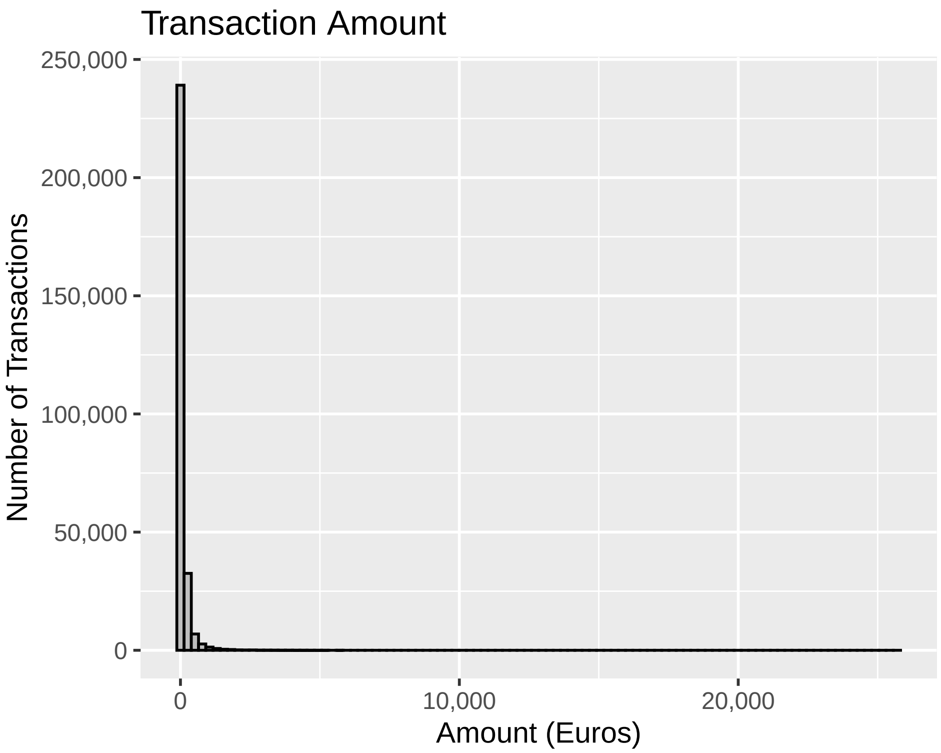


It would be interesting to look at whether fraud happens more often at different times of the day. We can plot transactions by hour of the day, divided into fraudulent or not fraudulent:

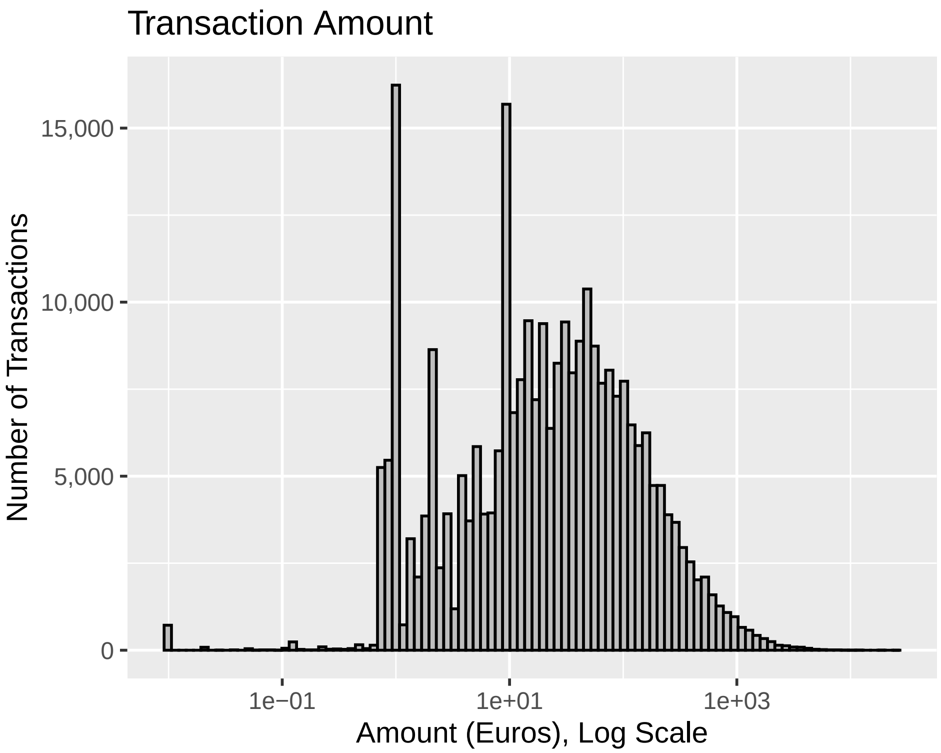


There seems to be more fraudulent transactions relative to valid ones in the early hours of the morning.

Now, let's look at the distribution of our last feature, *Amount*:



The skewness in the distribution is affecting our plot. Let's make the x axis logarithmic to get a better view of the distribution:



We can make a mental note to transform this variable with a power transformation in a later step. Additionally, when we ran the code we were warned about zeroes in the *Amount* column. Since we already know that the minimum value of the column is zero, let's see how many rows have a transaction amount of zero:

```
sum(df$Amount == 0)
```

```
[1] 1825
```

What should we do about these zeroes?

It would be interesting to talk to the creators of the dataset and find out how does it happen that the transaction value is 0? Was it a very small value that got rounded down to 0? Was it a missing value that was filled with a 0?

There is a small number of them, and without more information I will keep them in. We just need to make a mental note that by keeping the zeros can not use a Box-Cox transformation on this feature. We will need to use the Yeo-Johnson power transformation.

2.3 Data Cleaning and Feature Engineering

2.3.1 Data Cleaning

Let's start by looking for duplicates in our data:

```
sum(duplicated(df))
```

```
[1] 1081
```

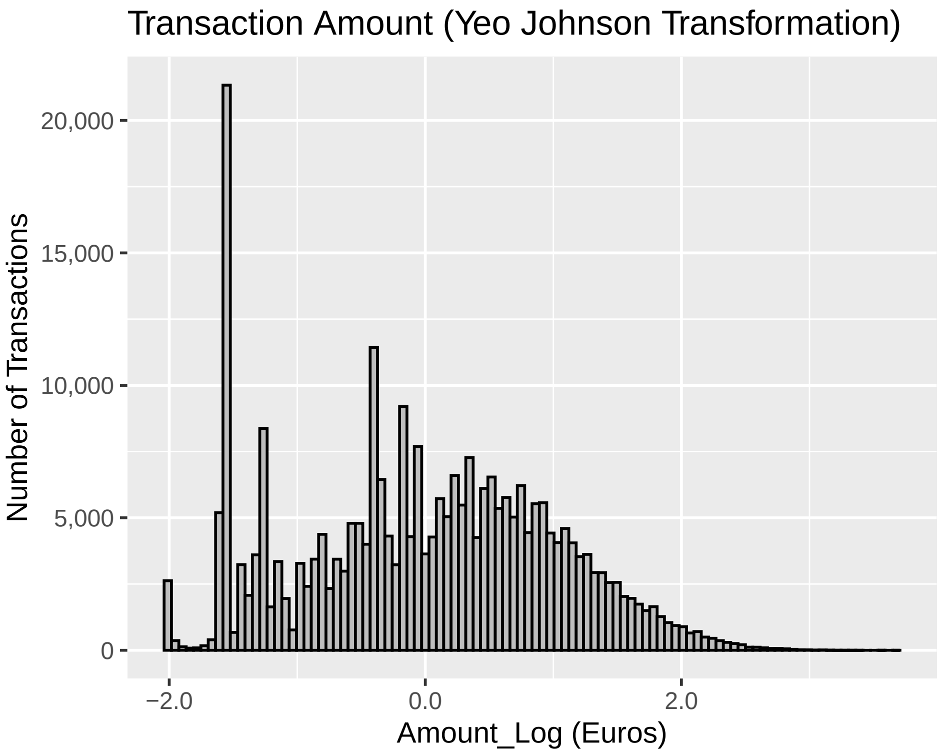
And we can get rid of those rows:

```
df <- df[!duplicated(df), ]
```

In the previous step, we saw that we can get a better sense of the distribution of *Amount* using log scale on the x axis. Let's explore using a power transformation on the data itself and create a new feature *Amount_Log*. We will use the yeojohnson function.

```
yj_obj <- yeojohnson(df$Amount, standardize = TRUE)
df$Amount_Log <- predict(yj_obj)
```

We can see that the resulting plot looks almost exactly like the one we made before:



2.3.2 Feature Engineering

The Time variable records the seconds elapsed for a given transaction from the first one in the dataset. It does not make sense to me that people making fraudulent transactions have knowledge of all the transactions taking place at that time. So any relationship would be just a coincidence.

In my opinion, leaving the feature as is would just be adding noise to our dataset. But I think it could make sense to add a feature that encodes whether a transaction happened during the night, during those two ~7 hour long periods of low transactions that we discovered in our visual exploration. This intuition seems to be supported by our fraud vs no fraud by hour of day plot.

We will create a new feature *Night*, that will have value of 1 if the transaction happened in either window of time or 0 otherwise.

```
Night <- (df$Time %in% 4000:27000) | (df$Time %in% 89000:112000)
df$Night <- as.integer(Night)
```

We can see that only about 8% (22,059 of 283,726) of transactions happened at "night", even though those two periods of time (a total of 12.8 hours) represent almost 27% of the time elapsed.

```
sum(df$Night)
```

```
[1] 22059
```

Before we start modeling, we can drop the features we will not need. We will drop the *Time* feature now that we have our *Night* variable.

To avoid data leakage, we should transform our *Amount* predictor after we split the dataset into train and test sets. So we will drop *Amount_Log* at this moment too.

```
df <- subset(df, select = -c(Time, Amount_Log))
```

Finally, verifiy everything looks right:

```
head(df)
```

```
# A tibble: 6 x 31
      V1      V2    V3     V4      V5      V6      V7      V8     V9     V10
   <dbl>   <dbl> <dbl>  <dbl>   <dbl>   <dbl>   <dbl>   <dbl>  <dbl>   <dbl>
1 -1.36  -0.0728 2.54   1.38  -0.338   0.462   0.240   0.0987  0.364  0.0908
2  1.19   0.266  0.166  0.448  0.0600 -0.0824 -0.0788  0.0851 -0.255 -0.167
3 -1.36  -1.34   1.77   0.380 -0.503   1.80    0.791   0.248  -1.51   0.208
4 -0.966 -0.185  1.79  -0.863 -0.0103  1.25    0.238   0.377  -1.39  -0.0550
5 -1.16   0.878  1.55   0.403 -0.407   0.0959  0.593  -0.271   0.818  0.753
6 -0.426  0.961  1.14  -0.168  0.421  -0.0297  0.476   0.260  -0.569 -0.371
# ... with 21 more variables: V11 <dbl>, V12 <dbl>, V13 <dbl>, V14 <dbl>,
#   V15 <dbl>, V16 <dbl>, V17 <dbl>, V18 <dbl>, V19 <dbl>, V20 <dbl>,
#   V21 <dbl>, V22 <dbl>, V23 <dbl>, V24 <dbl>, V25 <dbl>, V26 <dbl>,
#   V27 <dbl>, V28 <dbl>, Amount <dbl>, Class <fct>, Night <int>
```

2.4 Modeling Approach

This is a binary classification problem.

Algorithms to test

We will use the following algorithms and see which one performs best:

- Regularized Logistic Regression
- Support Vector Machines
- Random Forest Classifier
- Adaptive Boosting Classifier

Logistic Regression is a good choice in binary classification problems like this one.

Support Vector Machines can be very good sometimes at finding the rules that divide classes in classification problems. We will include it in our group of algorithms to test.

Random Forest and Adaptive Boosting Classifiers are normally good choices in classification problems. They can do a good job of modeling non-linear relationships in our features.

Split the dataset

We can now split our data into train and test data. We will keep 20% for testing and we will train our models using the train split.

We have a good a mount of observations, we could use only 10% for validation but due to the huge class imbalance, I want to have more fraudulent observations in our validation set.

I will not use the test data for algorithm selection. I will only use it to measure the performance of the final model. Model selection will be done using the results from cross-validation.

```r
set.seed(775, sample.kind="Rounding")
test_index <- createDataPartition(y = df$Class,
                                  times = 1, p = 0.20,
                                  list = FALSE)
cc_train <- df[-test_index,]
cc_test <- df[test_index,]
```

We need to make sure that the number of fraudulent transactions is similar in both sets:

```r
sum(cc_train$Class == "Yes") / length(cc_train$Class)
```

```
[1] 0.001665345
```

```r
sum(cc_test$Class == "Yes") / length(cc_test$Class)
```

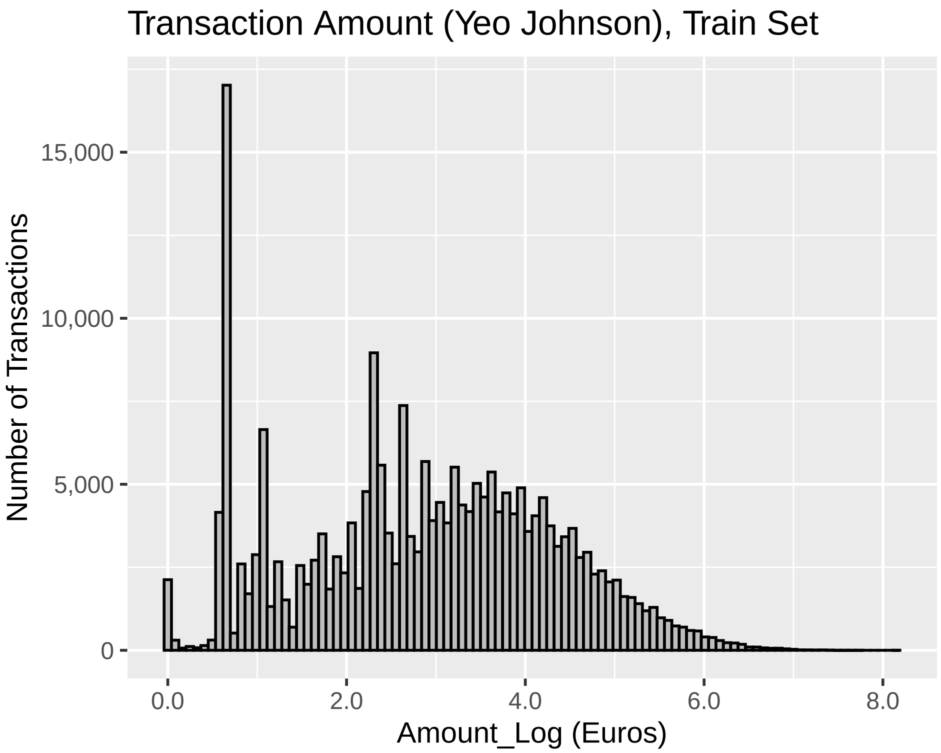
```
[1] 0.001674127
```

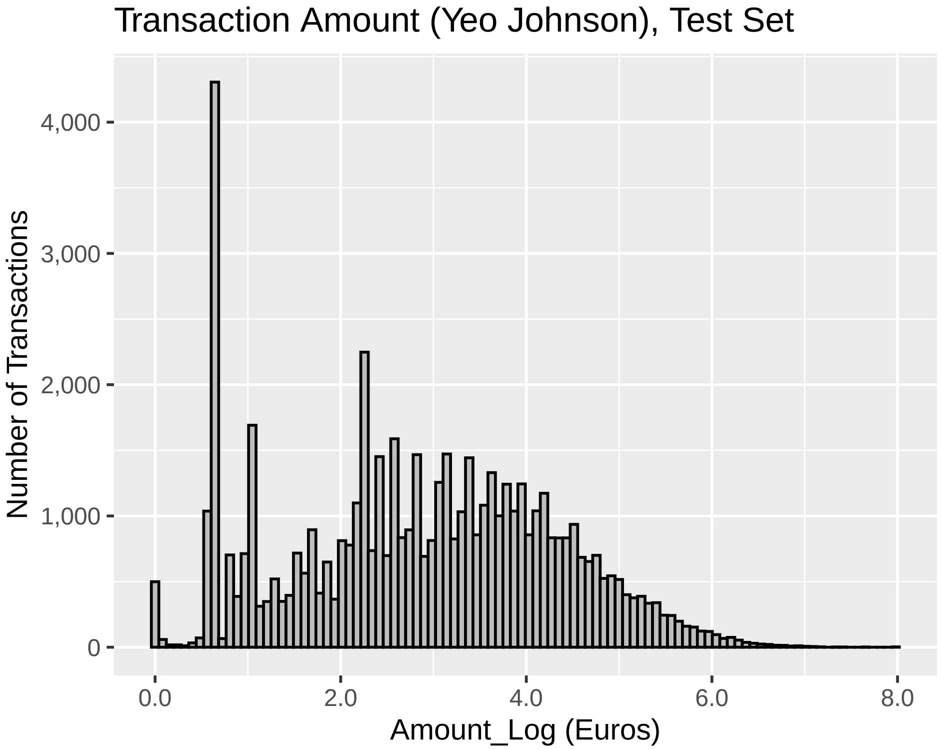
Pre-process

I will transform the *Amount* feature on both the train and test sets. First we fit the Yeo Johnson model in the train data and then we use that fit to transform both the train and test sets. We do not standardize the values at this point because we will do centering and scaling at the modeling stage.

```r
yj_obj <- yeojohnson(cc_train$Amount, standardize = FALSE)

cc_train$Amount <- predict(yj_obj)
cc_test$Amount <- predict(yj_obj, newdata = cc_test$Amount)
```

As a sanity check, I will plot the transformed data:





Setup

To better compare our models, I will use a shared trainControl object. This way we can use the exact same folds for cross-validation on all four algorithms. This is a large dataset and training takes several hours. We will use 5 folds instead of 10 to keep computation simpler.

I am using prSummary to use Area Under the Precision Recall Curve (AUPRC) as the performance metric for cross-validation.

```r
set.seed(111, sample.kind="Rounding")
myFolds <- createFolds(cc_train$Class, k = 5)

myControl <- trainControl(
  summaryFunction = prSummary,
  classProbs = TRUE,
  verboseIter = FALSE,
  savePredictions = TRUE,
  index = myFolds
)
```

2.4.1 Regularized Logistic Regression with glmnet

This algorithm has two tuning parameters: alpha and lambda.

When alpha is equal to zero, the model uses Ridge regularization. When alpha is equal to 1, it uses Lasso. A value between 0 and 1 combines the two. Lambda is the regularization strength.

First, we create a tuning grid:

```r
myGrid_glmnet <- expand.grid(
  alpha = c(0, 0.5, 1),
  lambda = seq(0.0001, 0.1, length = 10)
)
```

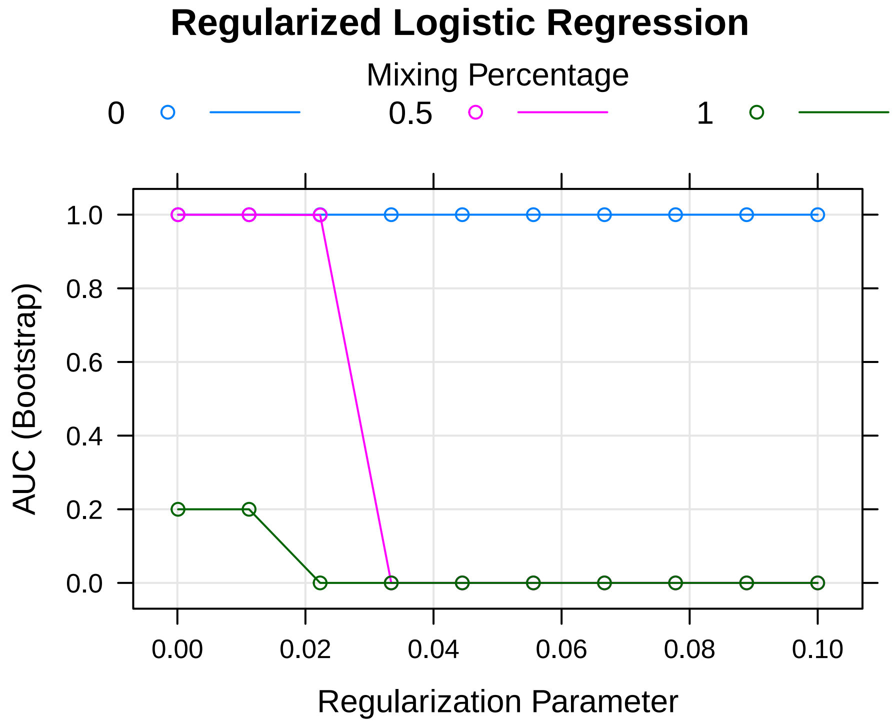
An interesting fact about glmnet is that it trains all values of lambda at the same time. This feature makes this algorithm faster than others. Our next step it to train the model using this tuning grid and the shared train control object.

I am also using a pre-processing pipeline with three steps:

- Eliminate features with no variance
- Center
- Scale

```r
model_glmnet <- train(
  Class ~ .,
  cc_train,
  method = "glmnet",
  metric = "AUC",
  preProcess = c("zv", "center", "scale"),
  tuneGrid = myGrid_glmnet,
  trControl = myControl
)
```

We can plot the main hyper parameters. We can see that Ridge works better in this model:



We can see the tuning parameters of the best model:

```
  alpha lambda
2     0 0.0112
```

And the area under the precision-recall curve (AUC), and the Precision, Recall and F1 scores from the training data:

AUPRC:

```
 [1] 0.9999121 0.9999216 0.9999208 0.9999195 0.9999185 0.9999181 0.9999176
 [8] 0.9999161 0.9999132 0.9999109 0.9998948 0.9998028 0.9991676 0.0000000
[15] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.1999841
[22] 0.1999603 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
[29] 0.0000000 0.0000000
```

Precision:

```
 [1] 0.9993142 0.9990245 0.9988560 0.9987680 0.9987031 0.9986679 0.9986525
 [8] 0.9986239 0.9986052 0.9985986 0.9993825 0.9985667 0.9983347 0.9983347
[15] 0.9983347 0.9983347 0.9983347 0.9983347 0.9983347 0.9983347 0.9985834
[22] 0.9983347 0.9983347 0.9983347 0.9983347 0.9983347 0.9983347 0.9983347
[29] 0.9983347 0.9983347
```

Recall:

```
 [1] 0.9998808 0.9998886 0.9999029 0.9999261 0.9999305 0.9999360 0.9999404
 [8] 0.9999481 0.9999713 0.9999967 0.9998312 0.9999570 1.0000000 1.0000000
```

```
[15] 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000 0.9999371
[22] 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000
[29] 1.0000000 1.0000000
```

F1:

```
 [1] 0.9995974 0.9994563 0.9993792 0.9993467 0.9993164 0.9993015 0.9992960
 [8] 0.9992856 0.9992878 0.9992972 0.9996068 0.9992613 0.9991666 0.9991666
[15] 0.9991666 0.9991666 0.9991666 0.9991666 0.9991666 0.9991666 0.9992597
[22] 0.9991666 0.9991666 0.9991666 0.9991666 0.9991666 0.9991666 0.9991666
[29] 0.9991666 0.9991666
```

2.4.2 Support Vector Machines with svmLinear

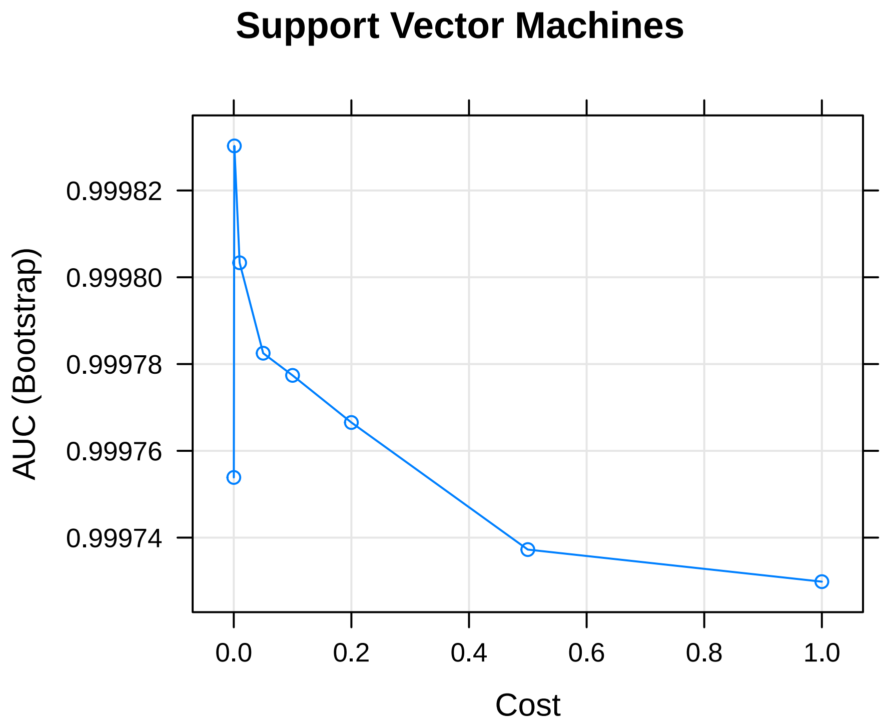
Now it's time for Support Vector Machines. This algorithm has only one tuning parameter: C or cost. We first create the tuning grid:

```
myGrid_svm <- data.frame(C = c(0.0001, 0.001, 0.01, 0.05, 0.1, 0.2, 0.5, 1))
```

Then we train the model using this tuning grid and the shared train control object. I am also using the same pre-processing pipeline as the previous algorithm:

```
model_svm <- train(
  Class ~ .,
  cc_train,
  method = "svmLinear",
  metric = "AUC",
  preProcess = c("zv", "center", "scale"),
  tuneGrid = myGrid_svm,
  trControl = myControl
)
```

We can see how the AUC varies with different values of C:



We can also see the tuning parameters of the best model:

```
      C
2 0.001
```

Here are the AUPRC, the Precision, Recall and F1 scores from the training data:

AUPRC:

```
[1] 0.9997539 0.9998303 0.9998033 0.9997825 0.9997774 0.9997665 0.9997372
[8] 0.9997299
```

Precision:

```
[1] 0.9993638 0.9993142 0.9993946 0.9993539 0.9992889 0.9993131 0.9992547
[8] 0.9992999
```

Recall:

```
[1] 0.9998444 0.9998599 0.9998742 0.9998786 0.9998831 0.9998820 0.9998842
[8] 0.9998786
```

F1:

```
[1] 0.9996040 0.9995870 0.9996344 0.9996162 0.9995859 0.9995974 0.9995693
[8] 0.9995892
```

2.4.3 Random Forest with ranger

I am using the **ranger** package for our random forest model. ranger has 3 tunable parameters:

- mtry: the number of randomly selected predictors to use for each tree
- splitrule: is the method used to choose splits in the decision trees
- min.node.size: used to stop splitting nodes based on the number of observations in a node

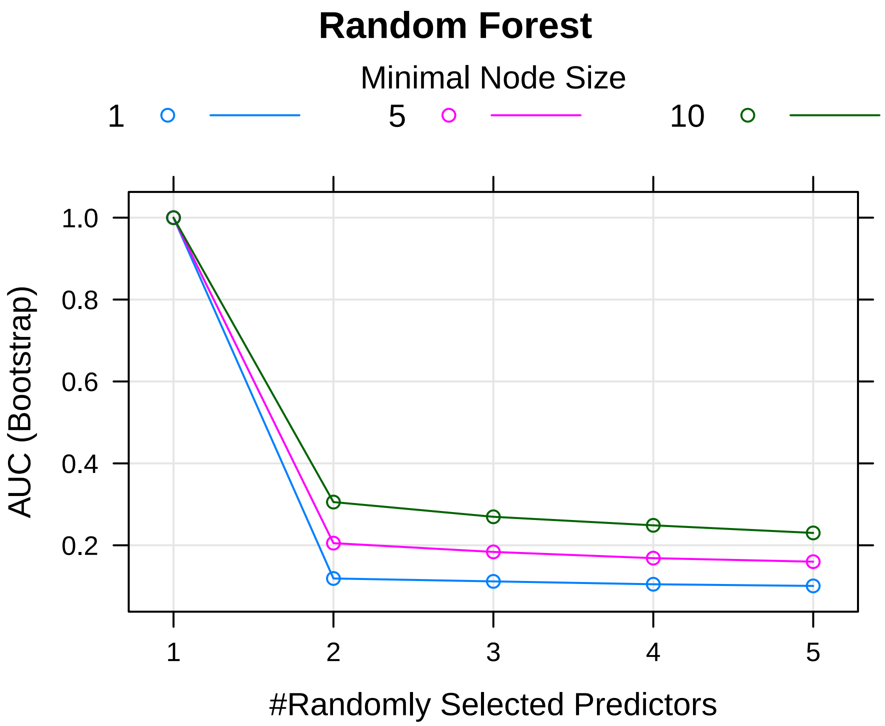
The first step is to create our tuning grid. In my tests the performance decreased as *mtry* increased beyond 5, so I am using the range 1-5. I will use *extratrees* as the *splitrule* because that was the rule that worked best. I have also decided to use minimal node sizes of 1, 5, and 10.

```
myGrid_rf <- expand.grid(
  mtry = c(1, 2, 3, 4, 5),
  splitrule = "extratrees",
  min.node.size = c(1, 5, 10)
)
```

The next step is to train our model. For decision trees we do not need to center and scale our data so we will omit those steps from the pre-process pipeline.

```
model_rf <- train(
  Class ~ .,
  cc_train,
  method = "ranger",
  metric = "AUC",
  preProcess = c("zv"),
  tuneGrid = myGrid_rf,
  trControl = myControl
)
```

Here we can see how higher values of *mtry* lower our performance:



These are the parameters for our best model:

```
  mtry  splitrule min.node.size
2    1 extratrees             5
```

Finally, we have the AUPRC, the Precision, Recall and F1 scores from the training data:

AUPRC:

```
 [1] 0.9999070 0.9999174 0.9999108 0.1189191 0.2051878 0.3054568 0.1118320
 [8] 0.1837564 0.2695275 0.1047702 0.1685576 0.2487215 0.1007325 0.1599649
[15] 0.2301532
```

Precision:

```
 [1] 0.9991423 0.9990818 0.9990542 0.9993539 0.9993241 0.9992932 0.9994266
 [8] 0.9994134 0.9993671 0.9994707 0.9994585 0.9994189 0.9994993 0.9994883
[15] 0.9994574
```

Recall:

```
 [1] 0.9999250 0.9999139 0.9999106 0.9999029 0.9998996 0.9998908 0.9998930
 [8] 0.9998952 0.9998831 0.9998908 0.9998797 0.9998764 0.9998842 0.9998786
[15] 0.9998676
```

F1:

```
 [1] 0.9995335 0.9994977 0.9994822 0.9996283 0.9996118 0.9995919 0.9996597
 [8] 0.9996542 0.9996250 0.9996807 0.9996691 0.9996476 0.9996917 0.9996834
[15] 0.9996625
```

2.4.4 Adaptive Boosting with adaboost

For the final algorithm, I chose adaboost. adaboost has 3 tunable parameters:

- nIter: is the number of trees in the ensemble
- method: is the technique used by the algorithm

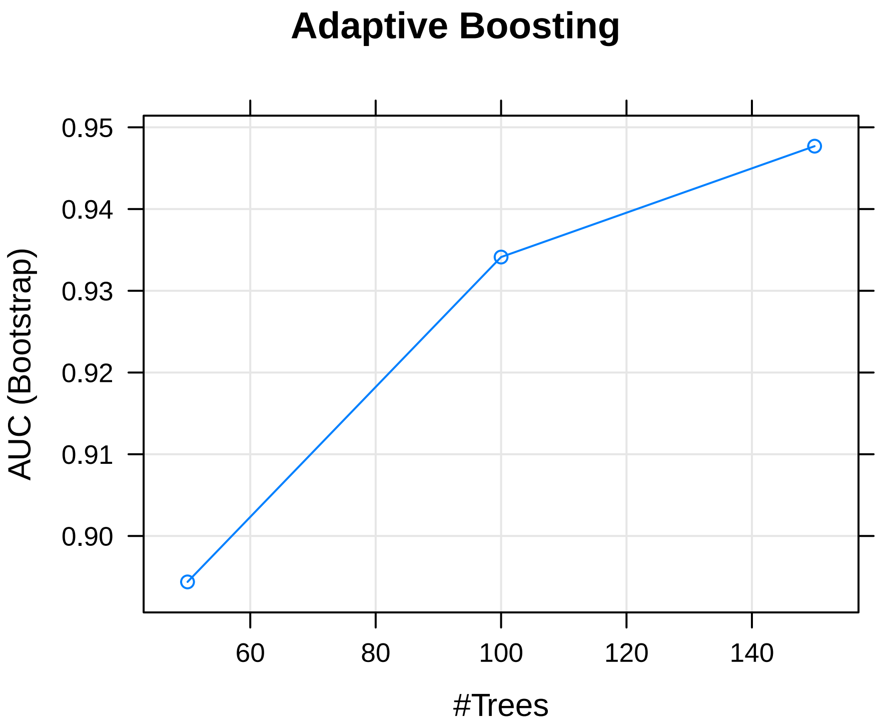
Again, the first step is to create our tuning grid. I will use the default 50, 100, and 150 values for *nIter*. In my tests *Real Adaboost* was the method that worked best.

```
myGrid_ada <- expand.grid(
  nIter = c(50, 100, 150),
  method = "Real adaboost"
)
```

The next step is to train the model. Like we did for random forest, we will only perform zero variance feature removal as pre-processing:

```
model_ada <- train(
  Class ~ .,
  cc_train,
  method = "adaboost",
  metric = "AUC",
  preProcess = c("zv"),
  tuneGrid = myGrid_ada,
  trControl = myControl
)
```

We can see the interaction of the tuning parameters here:



Here are the parameters of our best model:

```
  nIter        method
3   150 Real adaboost
```

And our results for adaboost:

AUPRC:

```
[1] 0.8943820 0.9341253 0.9476931
```

Precision:

```
[1] 0.9993858 0.9994112 0.9994178
```

Recall:

```
[1] 0.9999250 0.9999360 0.9999349
```

F1:

```
[1] 0.9996553 0.9996735 0.9996763
```

2.4.5 Algorithm Selection

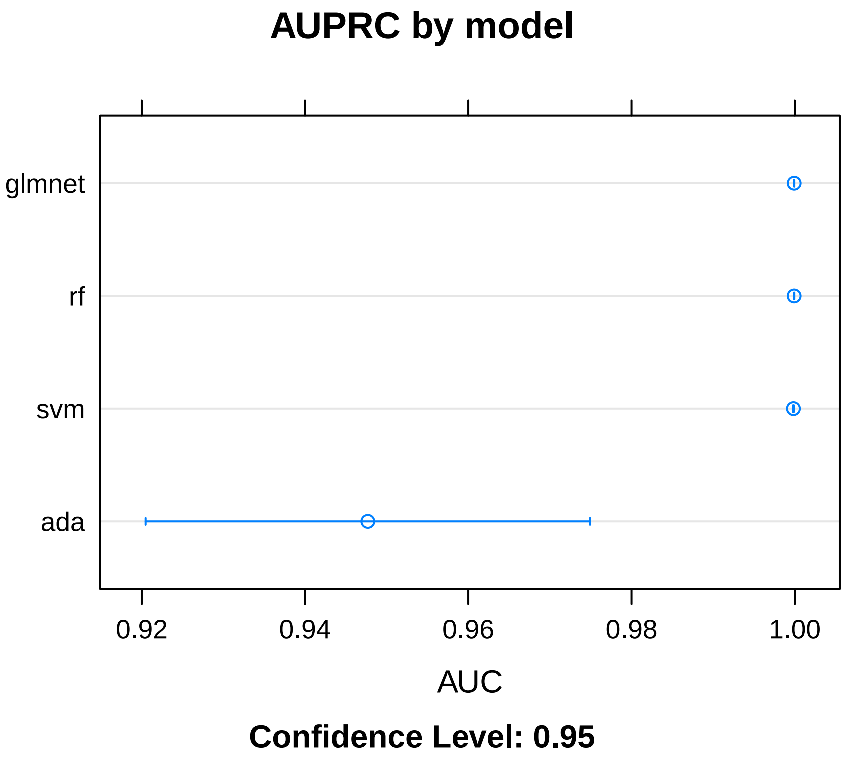
The final step is to select the algorithm that performed better. I will use the resamples function in caret to compare the performance of the 4 algorithms. Here is a complete comparison:

```
model_list <- list(
  glmnet = model_glmnet,
  svm = model_svm,
  rf = model_rf,
  ada = model_ada
)

resamps <- resamples(model_list)

summary(resamps)
```

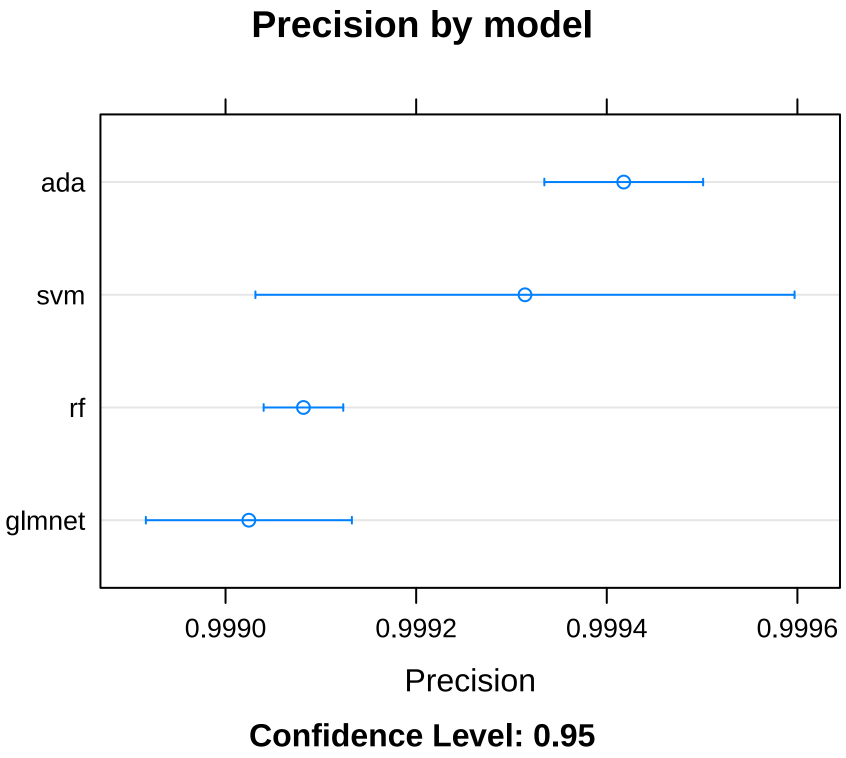
```

Call:
summary.resamples(object = resamps)

Models: glmnet, svm, rf, ada
Number of resamples: 5

AUC
            Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
glmnet 0.9999030 0.9999207 0.9999226 0.9999216 0.9999269 0.9999351    0
svm    0.9997969 0.9998063 0.9998068 0.9998303 0.9998479 0.9998935    0
```

```
rf     0.9998956 0.9998986 0.9999202 0.9999174 0.9999341 0.9999385    0
ada    0.9308246 0.9310205 0.9407053 0.9476931 0.9523086 0.9836064    0

F
            Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
glmnet 0.9994101 0.9994321 0.9994651 0.9994563 0.9994707 0.9995037    0
svm    0.9994652 0.9995395 0.9995809 0.9995870 0.9996388 0.9997104    0
rf     0.9994679 0.9994899 0.9995038 0.9994977 0.9995065 0.9995203    0
ada    0.9996333 0.9996608 0.9996774 0.9996763 0.9996884 0.9997215    0

Precision
            Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
glmnet 0.9988979 0.9989859 0.9990520 0.9990245 0.9990575 0.9991291    0
svm    0.9990135 0.9992172 0.9992888 0.9993142 0.9994321 0.9996194    0
rf     0.9990465 0.9990466 0.9990905 0.9990818 0.9991071 0.9991181    0
ada    0.9993274 0.9993660 0.9994486 0.9994178 0.9994707 0.9994762    0

Recall
            Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
glmnet 0.9998731 0.9998786 0.9998786 0.9998886 0.9998897 0.9999228    0
svm    0.9998014 0.9998455 0.9998621 0.9998599 0.9998731 0.9999173    0
rf     0.9998897 0.9998897 0.9999062 0.9999139 0.9999228 0.9999614    0
ada    0.9999007 0.9999062 0.9999393 0.9999349 0.9999559 0.9999724    0
```

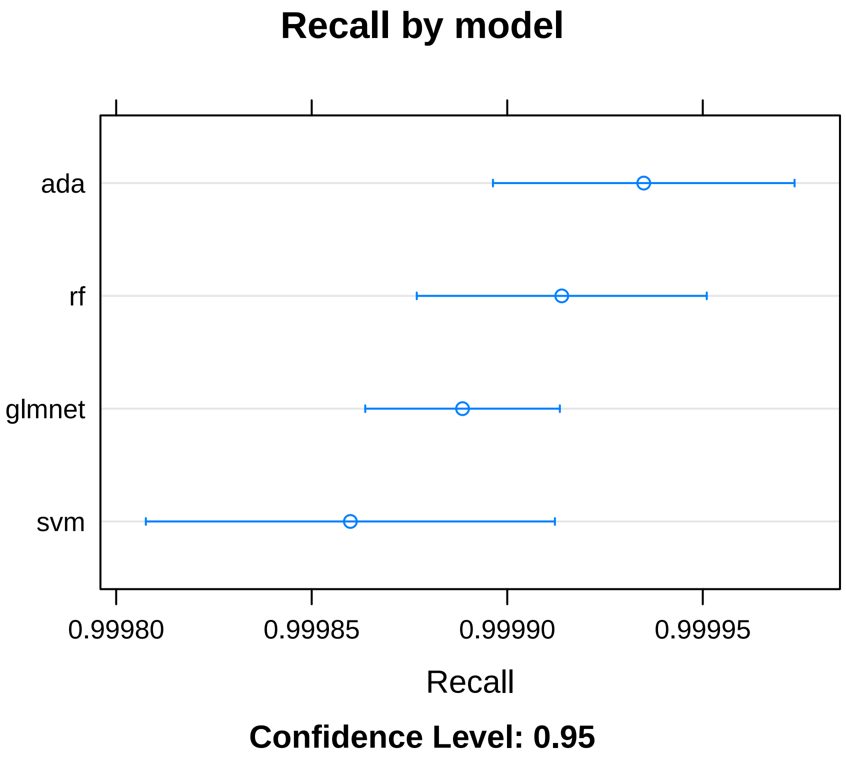
To better see which algorithm performed best we can plot the results. This is the plot for the AUPRC:



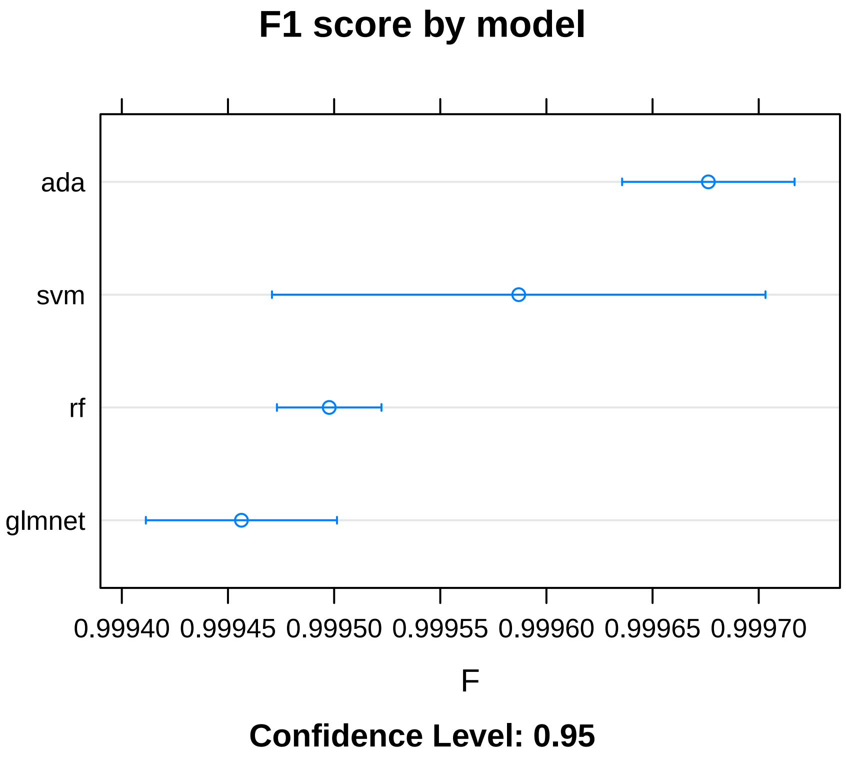
These are the Precision scores of the different models:



And also the Recall scores:



Finally, we can see the F1 score for each algorithm:



Based of the F1 results, we will select the adaboost algorithm for our final model.

3 Results and Final Model

With our algorithm selected, we can finally use our validation set to test the performance of the model in unseen data. The first step is to generate the predictions and then we build a confusion matrix:

```
final_preds <- predict(model_ada, cc_test)

final_cm <- confusionMatrix(final_preds,
                            as.factor(cc_test$Class),
                            positive = "Yes")
```

Here we have our final scores for Kappa, Precision, Recall and F1.

```
    Kappa
0.8503472
```

```
Precision
0.9367089
```

```
   Recall
0.7789474
```

```
       F1
0.8505747
```

4 Conclusion

Adaptive Boosting performed better than random forest, regularized logistic regression, and support vector machines.

It was really interesting to see how accuracy is not a good metric when the classes are so unbalanced. A model that predicts 0 (not fraud) for every transaction would get an accuracy of 99.82% on this dataset. Once we look at Kappa and F1, the scores are lower and we get a better sense for the relative performance of the algorithms.

The best model had an F1 score of **0.8506** on the validation set using an Adaptive Boosting algorithm. F1 is the harmonic mean of Precision and Recall.

One could say that Recall/Sensitivity is the metric we need to focus on since we want to minimize fraudulent transactions. But false positives (normal transactions flagged as fraud) are also important because they interfere with a client's business, and it could also lead to lost revenue for the Bank if the client uses an alternative method of payment. That is why I focused on the F1 metric to find a balance between Precision and Recall.

There are more than 280,000 transactions in the dataset. Once you run the cross-validation search over 5 folds, you end up running about 80 versions of the model for ranger, and 30 for adaboost. It took several hours to train and tune all 4 algorithms.

There are many ways this model could be improved:

- Investigate the possibility of using even more data, so we can train our model with even more fraudulent transactions.
- With the original features, we could look into better feature engineering to try to improve the performance.
- Try other algorithms or even create an ensemble of different models to see if we can get better results.
- With a more powerful computer or cluster, run more hyper parameter tuning.