Mobile Money Fraud Detection

Vincent G. K.

20/06/2020

Contents

1 Overview

1.1 Problem statement

During the last decade, mobile money has become the go-to solution of choice to transfer money and make payments in many developing countries. In these countries, it has slowly filled-up the void left by scarce or difficult-to-access banking services. However, together with this welcomed development, new types of fraud have emerged, potentially impacting millions of daily transactions.

The purpose of this document is to design a machine learning algorithm to detect fraud on mobile money transactions. Several algorithms are developed and compared on a set of more than 6 million mobile money transactions. These transactions are generated by a simulator, PaySim[1]. This generator simulates mobile money transactions using real transactions provided by a large multi-national company operating in 14 countries all over the world. These synthetic transactions guarantees the users' anonymity while closely reproducing patterns seen in real data, allowing us to develop and refine machine learning algorithms which could then be used on real transactions.

The dataset used in this document is publicly available on Kaggle.[2]

1.2 Mobile money

While mobile money is not well known in western countries, it is widespread in developing countries where the banking system is not accessible to everyone. Mobile money works on a mobile telephone network. A mobile money account (MMA) is associated with a phone number and a SIM card, and every MMA holder can send money to any other MMA holder, for a fee, using their phone number as identifier. They can also pay school fees, electricity, taxes, and buy goods and services.

In order to put money on or withdraw money from her account, the user has to go to a mobile money agent. The mobile money agent is the interface between the electronic network and the world of notes and coins. Mobile money agents are widely available, with very high density of agents in urban areas.

The client can manage her MMA account using the provider's mobile application on her smartphone, or via a USSD menu using a feature phone. This simplicity and ease of access make mobile money accessible to everyone, including low income earners who usually do not have access to bank accounts. This State of the Industry Report on Mobile Money[3] from the GSM Association provides great insights on the current state and importance of the mobile money industry in the world.

1.3 Document structure

This document is organised as follows:

- Section **2. Workflow and methods** introduces the three models which are developed in this document and provides details about their algorithms, as well as the data workflow.
- Section **3. Preliminary analysis** describes the data preparation process and provides a detailed analysis of all the features in the dataset.
- Section **4. Modelling** develops and applies the models and their algorithms. It assesses the ability of the models to predict fraudulent transactions.

[1] https://www.researchgate.net/publication/313138956_PAYSIM_A_FINANCIAL_MOBILE_MONEY_SIMULATOR_FOR_FRAUD_DETECTION

[2] https://www.kaggle.com/ntnu-testimon/paysim1

[3] https://www.gsma.com/sotir/

- Section **5. Results and conclusion** presents the final results, when the best model is trained using a dataset as large as possible for the limited computing resources available for this task and applied to a dataset composed of unknown data.

As a general rule, most of the code required to get the results communicated in this document is made visible to the reader. The code for the main functions used in this work is shared in appendix to avoid interrupting the flow of the document with large chunks of code. Only code chuncks of secondary importance, such as cleaning code (`rm()`) and data saving code (`saveRDS()`, `loadRDS()`, etc.), are sometimes hidden. For the curious reader, they are still visible in the raw rmarkdown file or in the R script file associated with this report.

The rmarkdown file, R script and pdf file are available in the following github folder: https://github.com/vgkienzler/mob-money-fraud-detect

2 Workflow and methods

2.1 Workflow

The machine learning algorithms tested here are compared based on their ability to classify transactions in two different classes: fraudulent transaction (positive class) or genuine transaction (negative class). The classification is done based on the predictors available in the dataset. The workflow followed in this document, from data preparation to validating the final model, is the following:

1. **Data preparation.** This step includes downloading the dataset, cleaning and organising the data, e.g. editing or adding column names, changing the column types, splitting columns into several columns when necessary, dealing with missing values, etc. The original dataset is then split into three sets: a training set, a testing set and a validation set. The training and testing sets are used to develop and train the various models. The validation set is used only at the very end, once the best performing model has been identified. Using this validation set to make any sort of decision during the development of the algorithms would lead to overtraining: when applied to new, unknown data, the model would perform below expectations. For that reason, it will only be used to measure the performance of the model.

2. **Data exploration and visualisation.** This step involves exploring the various features in the dataset, such as their distribution, variation and correlation. It is important for the data scientist to become familiar with the dataset, identify which feature(s) can be discarded and which new feature(s) should be added, if any (feature engineering).

3. **Building and training the models.** This step involves actually developing the different algorithms and their variations for the different models, and comparing their performances. The best model is selected among all the different models and their variations. These models are developped on a dataset of limited size in order to not require long computing times.

4. **Validation of the best model.** Finally, the best model is trained on a larger dataset, which will take a significant amount of time, using the hyperparameters calculated in the previous section. This final model is then run to make predictions on the validation dataset and provide an estimate of the performance of the model on unknown data.

Steps 2 and 3 are intertwined to a certain extent. During step 3, it might be required to explore the data further, do additional visualisation and modify the format of the data to better fit the models under development.

2.2 Methods and tools

2.2.1 Algorithm selection

The problem at hand is a two-classes classification problem. To solve this problem, several algorithms could be used. These include: logistic regression, Naive Bayes, quadratic determinant analysis, linear determinant analysis, k-nearest neighbours, classification tree, and ensembles of trees such as random forests and gradient boosting machine (GBM), to mention just a few. For the current classification problem, the following machine learning algorithms have been selected:

1. Logistic regression
2. Random forests
3. Gradient boosting machine

Logistic regression will provide an interesting baseline, as it is a common classification algorithm, popular for its interpretability. Random forests and Gradient boosting machine are expected to provide better performance, but at the cost of interpretability and have been selected for their known high performance in classification problems such as this one.

For each of these algorithms different variations are tested.

First, these algorithms are tested on **different sets of features.** As the section on feature selection and engineering will show, new features are added to the initial dataset. However, adding new features increases computation time, and some models can see their performance decrease when non relevant features are added. The impact of the different features is assessed by running the different algorithms on three different sets of features. The exact composition of these sets is detailed later.

Secondly, various methods are used to **deal with data imbalance.** In this dataset, as the upcoming data exploration section will show, the number of positive observations (fraud) is much lower than the number of negative observations (genuine transactions). This means that the dataset is largely imbalanced. Because of this, additional technics such as class weights and re-sampling (SMOTE algorithm, which generates synthetic positive observations) are tested when an algorithm performs poorly, to see if its performance could be improved. The details of these methods are provided later.

Thirdly, **different values of the classification threshold** are tested. The aforementioned algorithms provide probability estimates for the two classes. Based on these estimates, and by default, the predicted class is the class with the highest probability, i.e. the class with a probability higher than 0.5. However, using this default threshold might not provide the best detection rate, especially when the dataset is strongly imbalanced. This is especially important for financial fraud, where false negatives are more damaging than false positives. For this reason, different classification thresholds are tested for the most promising models.

Finally, **different performance measures** are used for the selection of the best model. These are detailed in the next section.

2.2.2 Performance metric

Several metrics exist to measure the quality of a classification algorithm. When dealing with largely imbalanced data, the **area under the precision-recall curve** (PRAUC) is often cited as the best performance metric. PRAUC, by focusing on the positive class (minority class) is sensitive to the number of false positives (FP), false negatives (FN) and true positives (TP), which are not diluted by a high number of true negatives (TN).

$$Recall\ (True\ Positive\ Rate) = \frac{TP}{TP + FN}$$

$$Precision = \frac{TP}{TP + FP}$$

The precision-recall curve is calculated for various classification thresholds, and the area under this curve is estimated to get the PRAUC metric. Another performance measure, the **F1 score**, is also based on precision and recall, but does not take into account different classification threshold.

$$F1\ score = 2 * \frac{Precision * Recall}{Precision + Recall}$$

Alternative measures, such as **receiver operating characteristic area under the curve** (ROCAUC), which is based on the True Positive Rate (TPR) as a function of the True Negative Rate (TNR), and **accuracy** are usually considered to not be sensitive enough to the minority class. They might not be good performance metrics when dealing with imbalanced data. This can be explained by of the low occurrence of the positive class, which means that the number of FN are completely overshadowed by the number of TN. In addition, FP can be as numerous as TP (and therefore impact precision) but, because the negative class is several times more numerous than the positive class, have very little impact on accuracy and the TNR (see formulas below).

$$True\ Negative\ Rate = \frac{TN}{TN + FP}$$

$$Accuracy = \frac{TP + TN}{TN + FP + FP + FN}$$

In this document, **accuracy**, the **F1 score** and **PRAUC** are used to compare models. The best metric to achieve low FP and FN will be selected and applied to the final model.

2.2.3 Dataset sizes

The dataset available contains 6 million transactions. Training the models on 5 million transactions (to keep about 1 million aside to validate the model) would take a long time, depending on the computational complexity of the algorithm used. Some algorithms would possibly crash, such as RandomForest, which saturates between 500,000 and 1 million observations. Other algorithms, like `xgbTree` from the package `xgBoost` are more robust and can operate on much larger datasets.

In addition, some algorithms, like random forests, have a time complexity proportional to $n^2.p.n_{trees}$ whereas the complexity of logistic regression is only proportional to $p^2.n$.[4] This means that training a random forest algorithm on 1 million data points instead of 100,000 risk increasing execution time up to 100 folds. Using repeated k-fold cross-validation makes the situation worse: the computational time is multiplied by an additional $r.k$, where r is the number of repeats. If $k = 5$ and $r = 5$, the execution time is increased 25 times.

For these reasons, to keep computational time at a manageable level, a reduced sub-set of the initial dataset, containing 200,000 observations, is used to train and compare the models. A larger dataset is only used when the best model has been identified, to estimate the final performance.

Similarly, the validation dataset is kept to a reasonable size (10% of the total dataset) to limit computation time.

[4] https://www.thekerneltrip.com/machine/learning/computational-complexity-learning-algorithms/

2.2.4 Cross-validation

Cross-validation is used at two levels.

1. In order to minimise the variance of the models (the sensitivity of a model to the noise in a dataset), repeated k-fold cross-validation is used via the function `train()` of the `caret` package. The number of repeats and folds are discussed in the modelling section.

2. To ensure that the performance measure of the different models developed is not biased by overfitting, the original dataset is split between a training set, a test set and a validation set. The training set is used to fit the different models. The test set is used to measure the quality of the algorithms and compare them with one another. Finally, the validation test is left untouched until the end, to provide an unbiased measure of the best performing model.

2.2.5 Execution time

The performance of a model to predict the right class is an important aspect of the overall model performance, but the time it takes to train the algorithm and make the predictions can also be an important decision variable when selecting a machine learning model for a particular task.

In order to explore the computational time of the three types of algorithms in use, the time required to train these algorithms and run the predictions is measured using the `tictoc` package. To measure execution time two functions are required: `tic()` and `toc()`. `tic("label")` is introduced right before the code whose execution time is measured. `toc()` is placed after the code block, and triggers the stop of the timer. At the end, the times corresponding to all the labels (or their differences) can be displayed. As an additional note, the time it takes to train a single model is also accessible in the object returned by the `caret::train()` function, in the list labled `times`.

The execution time depends largely on the type of computer used to run the models, on the type of implementation of a particular algorithm and if parallel computing is used. For this analysis to make sense, it is therefore important to held all things other than the algorithm constant (size of the dataset, number of cores for parallel computing, etc.)

3 Preliminary analysis

3.1 Data preparation

Let's start with downloading and preparing the data. Running each of these blocks can be time consuming. For this reason some important datasets (`transactions.raw`, `train.set`, `test.set`, `validation` and a few others) are saved for future use in order to save time. In the rmarkdown document the `cache` function is also used to reduce the time required to generate this document between modifications.

Preliminary step - Loads the R packages needed, installs them if they are not yet installed, and initialises the theme used to display all the plots. Also, creates the `/data` folder in the working repository if it does not already exist. All the datasets and other data files (`.rds` or `.rdata`) will be stored in this `/data` folder.

```
# Required packages
if (!require(parallel)) install.packages('parallel')
library(parallel)
if (!require(doParallel)) install.packages('doParallel')
library(doParallel)
if (!require(class)) install.packages('class')
library(class)
```

```r
if (!require(tictoc)) install.packages('tictoc')
library(tictoc)
if (!require(knitr)) install.packages('knitr')
library(knitr)
if (!require(data.table)) install.packages('data.table')
library(data.table)
if (!require(stringr)) install.packages('stringr')
library(stringr)
if (!require(lubridate)) install.packages('lubridate')
library(lubridate)
if (!require(tidyverse)) install.packages('tidyverse')
library(tidyverse)
if (!require(ggridges)) install.packages('ggridges')
library(ggridges)
if (!require(ggthemes)) install.packages('ggthemes')
library(ggthemes)
if (!require(scales)) install.packages('scales')
library(scales)
if (!require(corrplot)) install.packages('corrplot')
library(corrplot)
if (!require(e1071)) install.packages('e1071')
library(e1071)
if (!require(randomForest)) install.packages('randomForest')
library(randomForest)
if (!require(xgboost)) install.packages('xgboost')
library(xgboost)
if (!require(PRROC)) install.packages('PRROC')
library(PRROC)
if (!require(caret)) install.packages('caret')
library(caret)

# Set the theme for all the graphs
theme_set(theme_fivethirtyeight(base_size = 10))
theme_update(axis.title = element_text())
theme_update(plot.title = element_text(margin=ggplot2::margin(0,0,15,0),
                                       hjust = 0.5))

# Creates "data" directory in the current directory if it doesn't exist
ifelse(!dir.exists(file.path("data")), dir.create(file.path("data")), FALSE)
```

Step 1 - Download the dataset. This can be done from the GitHub repository or from Kaggle, the original source of the data.

To download from the GitHub repository:

```r
# 1) Download the .rds file
url <- paste0("https://media.githubusercontent.com/media/vgkienzler/",
              "mob-money-fraud-detect/master/data/transactions-raw.rds")
download.file(url, destfile = "data/transactions-raw.rds")
# 2) Load the rds file
transactions.raw <- readRDS("data/transactions-raw.rds")
```

To download from Kaggle:

Log in to Kaggle and download the dataset at this address: https://www.kaggle.com/ntnu-testimon/paysim1. The data file "archive.zip" must be downloaded manually as it requires to log in using a personal account. Store the file in the `/data` folder.

```
# 1) Download and save the file from kaggle (archive.zip) in folder "data".
# 2) Unzip and save the csv file in the /data folder.
file.rename(unzip("data/archive.zip"),"data/transactions.csv")
# 3) Load the csv file as a data frame
transactions.raw <- read.csv(file="data/transactions.csv")
```

Step 2 - Clean and format the data properly. This includes making sure that column names are homogeneous and make sense, and splitting columns and column names if necessary.

```
# Display the current column names
colnames(transactions.raw)
```

```
##  [1] "step"           "type"           "amount"         "nameOrig"
##  [5] "oldbalanceOrg"  "newbalanceOrig" "nameDest"       "oldbalanceDest"
##  [9] "newbalanceDest" "isFraud"        "isFlaggedFraud"
```

Let's correct the spelling mistake in `oldbalanceOrg` and use a consistent naming scheme (camelCase) for all the column names:

```
# Correct spelling error and apply consistant naming scheme (camelCase) to
# all features:
transactions <- transactions.raw %>% rename(oldBalanceDest = oldbalanceDest,
                                            oldBalanceOrig = oldbalanceOrg,
                                            newBalanceOrig = newbalanceOrig,
                                            newBalanceDest = newbalanceDest)
```

Now check for NAs:

```
message("Is there any NA in the dataset? ", anyNA(transactions))
```

```
## Is there any NA in the dataset? FALSE
```

What is the dataset made of?

```
str(transactions, vec.len = 1)
```

```
## 'data.frame':    6362620 obs. of  11 variables:
##  $ step          : int  1 1 ...
##  $ type          : Factor w/ 5 levels "CASH_IN","CASH_OUT",..: 4 4 ...
##  $ amount        : num  9840 ...
##  $ nameOrig      : Factor w/ 6353307 levels "C1000000639",..: 757870 2188999 ...
##  $ oldBalanceOrig: num  170136 ...
##  $ newBalanceOrig: num  160296 ...
##  $ nameDest      : Factor w/ 2722362 levels "C1000004082",..: 1662095 1733925 ...
##  $ oldBalanceDest: num  0 0 ...
##  $ newBalanceDest: num  0 0 ...
##  $ isFraud       : int  0 0 ...
##  $ isFlaggedFraud: int  0 0 ...
```

As shown above, there are 11 predictors in the dataset:

- `step`: an integer. According to the dataset description:

  "Step - maps a unit of time in the real world. In this case 1 step is 1 hour of time. Total steps 744 (30 days simulation)."[5]

- `type`: a factor with 5 levels: `CASH-IN`, `CASH-OUT`, `DEBIT`, `PAYMENT`, `TRANSFER`. According to the dataset description, these types correspond to:

  "CASH-IN is the process of increasing the balance of account by paying in cash to a merchant.
  CASH-OUT is the opposite process of CASH-IN, it means to withdraw cash from a merchant which decreases the balance of the account.
  DEBIT is similar process than CASH-OUT and involves sending the money from the mobile money service to a bank account.
  PAYMENT is the process of paying for goods or services to merchants which decreases the balance of the account and increases the balance of the receiver.
  TRANSFER is the process of sending money to another user of the service through the mobile money platform."[6]

- `amount`: an integer, the amount of the transaction in local currency (undefined).

- `nameOrig`: factor, which contains information about the person sending money. According to the description, a sender who is a regular customer has a prefix "C", and a sender who is an agent or a merchant has a prefix "M".

- `oldBalanceOrig`: a numerical, which represents the amount of money on the account of the sender before the transaction.

- `newBalanceOrig`: a numerical, which represents the amount of money on the account of the sender after the transaction.

- `nameDest`: a factor, which contains information about the person receiving the money. According to the description, a recipient who is a regular customer has a prefix "C", and a recipient who is an agent or a merchant has a prefix "M".

- `oldBalanceDest`: a numerical, which represents the amount of money on the account of the recipient before the transaction. According to the description, if the recipient is a merchant (M), this information is "0" (no information).

- `newBalanceDest`: a numerical, which represents the amount of money on the account of the recipient after the transaction. According to the description, if the recipient is a merchant (M), this information is "0" (no information).

- `isFraud`: an integer, `1` represents a transaction made by a fraudulent actor. According to the description:

  "In this specific dataset the fraudulent behavior of the agents aims to profit by taking control of customers accounts and try to empty the funds by transferring to another account and then cashing out of the system."[7]

[5] https://www.kaggle.com/ntnu-testimon/paysim1
[6] https://www.kaggle.com/ntnu-testimon/paysim1
[7] https://www.kaggle.com/ntnu-testimon/paysim1

- `isFlaggedFraud`: an integer, flags a fraud noticed by the bank or mobile money operator. According to the official description, any single transaction above 200,000 in the local currency is automatically flagged as fraud. The next section verifies if this description actually matches the data.

  "The business model aims to control massive transfers from one account to another and flags illegal attempts. An illegal attempt in this dataset is an attempt to transfer more than 200.000 in a single transaction."[8]

Now let's update the type of the columns wherever it might be useful:

- `nameOrig` and `nameDest` are factors, but it is better to convert them to strings so that their type ("M" and "C") can be easily extracted. The idea behind this extraction is that the type of actor might provide important information to the classifier, but the rest of the factor might not be useful.
- `isFraud` and `isFlaggedFraud` are integers, but it is better to convert them to factors for machine learning. It is also recommended to put the positive class in the first position so that its factor index is "1", and "2" for the negative class, in second position.

```
# Update column type
# Update isFraud to factor, positive class = fraud (F1),
# negative class = genuine (G0)
# Using "0" and "1" as factors risks being confusing and triggers errors
# with some caret functions
transactions <- transactions %>%
  mutate(nameDest=as.character(nameDest),
         nameOrig=as.character(nameOrig)) %>%
  mutate(isFraud = factor(isFraud, levels = c(1, 0), labels = c("F1", "G0")),
         isFlaggedFraud = as.factor(isFlaggedFraud))
```

- Finally, let's separate the type of the originator and recipient ("C" for customer and "M" for merchant).

```
# Add type columns for nameOrig and nameDest and relocate them
transactions <- transactions %>%
  mutate(typeOrig = as.factor(str_sub(nameOrig,1,1)),
         typeDest = as.factor(str_sub(nameDest,1,1))) %>%
  relocate(typeOrig, .after=nameOrig) %>%
  relocate(typeDest, .after=nameDest)
```

Step 3 - Split the dataset between a validation set and a play set. Some additional changes to the dataset might be required, but they will be identified during data exploration, which comes next. In order to limit overtraining, it is better to separate the validation set from the dataset now, before doing any exploration or visualisation. By doing so, the information contained in the validation set is not used to inform any decision made during data exploration.

```
# Validation set is 10% of transactions dataset
# Set seed for reproducibility
set.seed(1)

in_validation <- createDataPartition(y = transactions$isFraud,
                                     times = 1, p = 0.1, list = FALSE)
```

[8] https://www.kaggle.com/ntnu-testimon/paysim1

```r
playset <- transactions[-in_validation,]
validation <- transactions[in_validation,]

# Save datasets for future use
saveRDS(playset, file = "data/playset.rds")
saveRDS(validation, file = "data/validation.rds")

# Do some cleaning
rm(in_validation)
```

```r
# Declare function "formated_nrow" to display number of rows using ","
formated_nrow <- function(x){
  x <- format(nrow(x), big.mark = ",")
  return(x)
}

# Display the number of observations in the different sets for control
message("'playset' contains ", formated_nrow(playset)," observations.")
```

```
## 'playset' contains 5,726,357 observations.
```

```r
message("'validation' contains ", formated_nrow(validation), " observations.")
```

```
## 'validation' contains 636,263 observations.
```

3.2 Data exploration

In this section, the following analyses are conducted:

1. Analysis of class prevalence for fraudulent transactions (F1) and genuine transactions (G0)
2. Distribution analysis of genuine transactions
3. Distribution analysis of fraudulent transactions
4. Balance analysis. This section focuses on analysing the balance and origin/destination for each transaction type (`PAYMENT`, `CASH_OUT`, `DEBIT`, `TRANSFER` and `CASH_IN`). This helps understanding the logic behind mobile money transactions, how they work, and how a fraud can occur and could be identified.
5. Looking for fraud patterns. Finally, this section looks specifically at patterns in fraudulent transactions. This helps identify possible patterns which are specific to fraudulent transactions, but were not obvious during the previous analysis which focused on each transaction types independently. This is motivated by the fact that several transactions are most likely required to complete a fraud (first a transfer and then a withdrawal).

3.2.1 Class prevalence

According to the data description, two features indicate a fraudulent transaction: `isFlaggedFraud` and `isFraud`. Let's look at the prevalence of each.

```r
# Look at the number of fraudulent/genuine transactions
freq_count <- data.frame(
  table(playset$isFraud,
        playset$isFlaggedFraud),
  prop.table(table(playset$isFraud,
                   playset$isFlaggedFraud)))[c(1,2,3,6)]

freq_count <- freq_count %>%
  mutate(Freq.1 = round(Freq.1, 4)) %>%
  rename(
    isFraud = Var1,
    isFlaggedFraud = Var2,
    Count = Freq,
    Prop. = Freq.1)

kable(freq_count, caption="Class prevalence",
      col.names=c("isFraud","isFlaggedFraud","Count", "Prop."))
```

Table 1: Class prevalence

isFraud	isFlaggedFraud	Count	Prop.
F1	0	7377	0.0013
G0	0	5718966	0.9987
F1	1	14	0.0000
G0	1	0	0.0000

As this table shows, the data is strongly imbalanced, with only 0.1% of all the observations corresponding to fraudulent transactions.

When comparing `isFraud` to `isFlaggedFraud`, it does not seem that `isFlaggedFraud` brings any new information. There are only 16 transactions in this class, and all of them are included in `isFraud`. In addition, the data description indicates that any transaction above 200,000 in the local currency gets included in `isFlaggedFraud`. This does not seem to actually be the case.

```r
# Count and print the number of transactions above 200,000
trans_above <- nrow(transactions %>% filter(amount > 200000))
message("Number of transactions above 200,000: "
  , format(trans_above, big.mark = ","))
```

```
## Number of transactions above 200,000: 1,673,570
```

```r
# Count and print the number of transactions above 200,000
# and with isFlaggedFraud == 1
trans_both <- nrow(transactions %>%
                     filter(amount > 200000) %>%
                     filter(isFlaggedFraud == 1))
message(
  "Number of transactions above 200,000 and with isFlaggedFraud == 1: "
  , trans_both
  )
```

```
## Number of transactions above 200,000 and with isFlaggedFraud == 1: 16
```

```
# Do some cleaning
rm(freq_count, trans_above, trans_both)
```

In conclusion:

1. All the transactions in `isFlaggedFraud` are in `isFraud`, and there are only 16 transactions with `isFlaggedFraud == 1`.
2. The official description does not seem accurate: a significant number of transactions above 200,000 are not marked with `isFlaggedFraud == 1`.
3. All the transactions in `isFlaggedFraud` have an amount greater than 200,000.

Based on this analysis, it is not clear what additional information the predictor `isFlaggedFraud` brings. This predictor can be dropped.

3.2.2 Genuine transactions

Let's look first at the number of genuine (non-fraudulent) transactions for each type of transactions. As the table below shows, the most frequent types are `CASH-OUT` and `PAYMENT`, and the least frequent is `DEBIT` (this can be explained by the very low number of clients who also have a bank account and therefore can use `DEBIT`).

```
# Table of number of transactions per type
kable(sort(
  table(playset$type), decreasing = TRUE),
  caption="Number of transactions per type",
  col.names=c("Type","Count"))
```

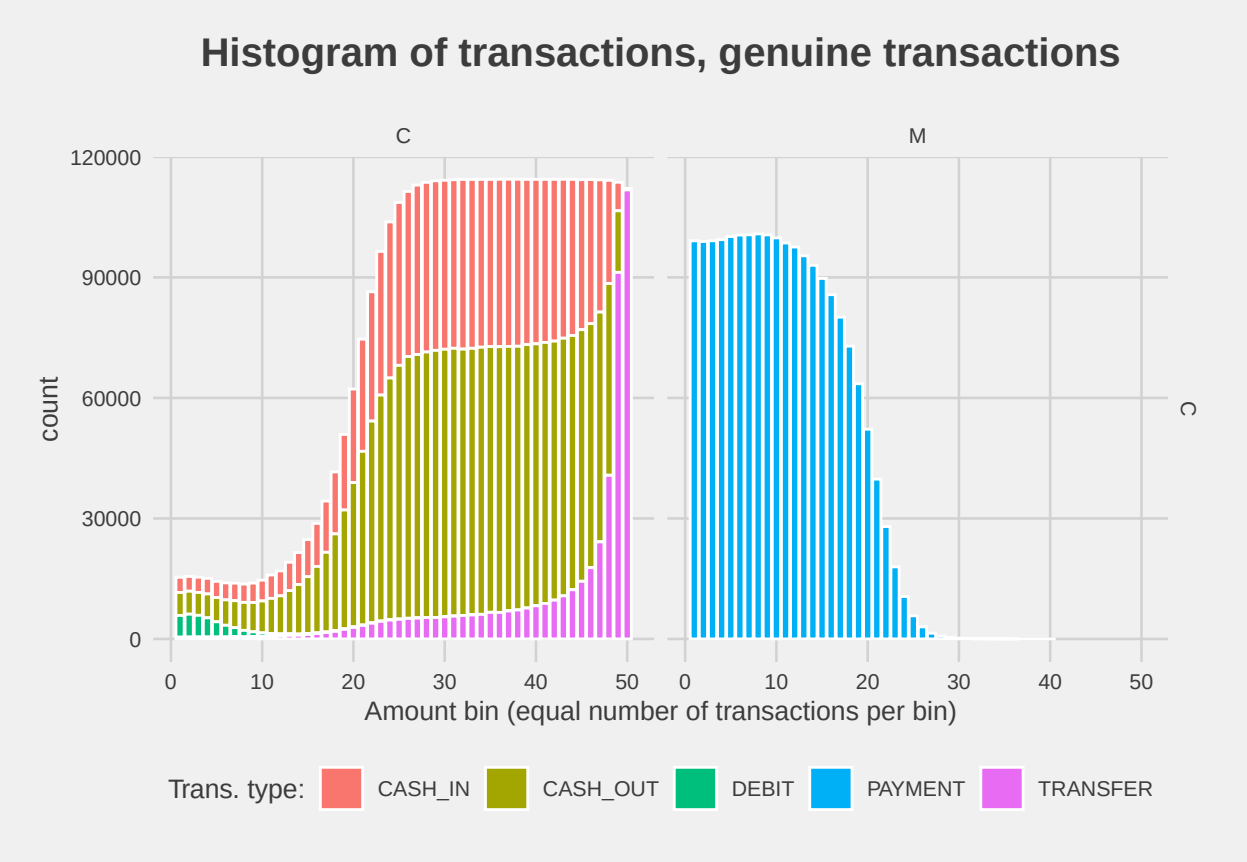
Table 2: Number of transactions per type

Type	Count
CASH_OUT	2013765
PAYMENT	1936094
CASH_IN	1259640
TRANSFER	479542
DEBIT	37316

3.2.2.1 Histograms of transactions for the different types Now let's look at the distribution of transactions between clients (C) and merchant (M) for `typeDest` and `typeOrig`. The facet-grid graph below draws an histogram of transactions, plotting one graph for each combination of (M) and (C). There are three possible combinations: C/C, M/C and C/M. In the histogram *all the bins contain the same number of transactions* ($N/50$ where N is the total number of transactions in `playset`, about 5.72 million), ordered by increasing amount. This means that the width of a bin does not reflect the spread of the values of the transactions included in that bin.

```
playset %>%
  mutate(bin = cut_number(amount, 50, labels=FALSE)) %>%
  filter(isFraud == "GO") %>%
  ggplot(aes(bin, fill = type)) +
```

```
geom_histogram(stat = "count", color="White") +
facet_grid(typeOrig ~ typeDest) +
labs(
  fill = "Trans. type:"
  , x = "Amount bin (equal number of transactions per bin)"
  , title = "Histogram of transactions, genuine transactions")
```



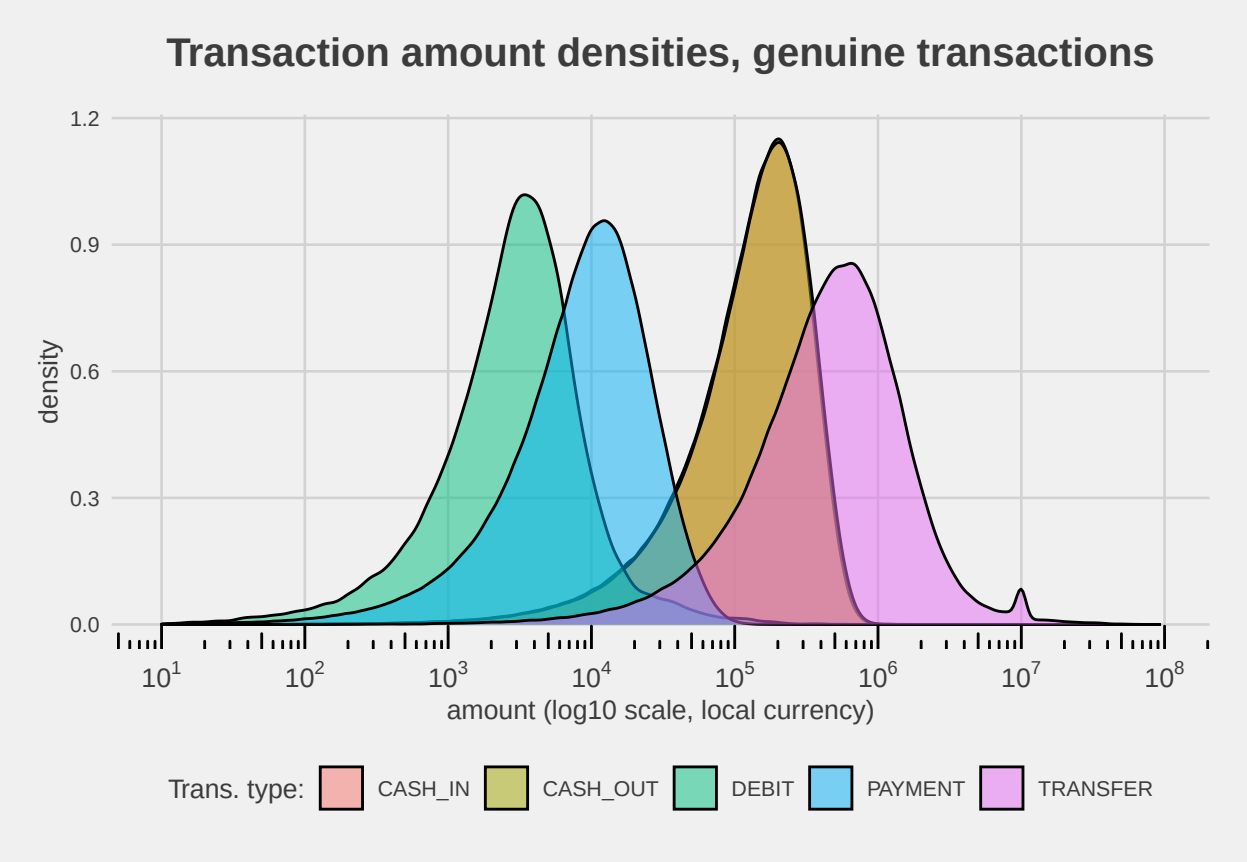
From this graph the following can be inferred:

1. `PAYMENT` is the only type of transactions which involve a merchant (factor `M`), as destination of a transaction. All other transactions types are between individuals (C).
2. `PAYMENT` is not used for transactions between individuals (no `C` to `C` transactions), confirming what is indicated in the official data description.
3. The most numerous transactions are `PAYMENT`, `CASH-OUT` and `CASH-IN`, confirming the table above.
4. Almost all of the highest amount transactions are of type `TRANSFER`.
5. Almost all of the lowest amount transactions are `PAYMENT` from an individual to a merchant.
6. `DEBIT` represents a very small number of transactions and are low amount transactions.

3.2.2.2 Amount distribution

Let's look into the distribution of the transaction amounts.

```r
# Plot the density vs. transaction amount using a log10 scale
# for non fraudulent transactions
playset %>%
  filter(isFraud == "G0") %>%
  ggplot(aes(amount, fill = type)) +
  geom_density(alpha = 0.5) +
  scale_x_log10(
    limits = c(10, NA)
    , n.breaks = 8
    , labels = trans_format("log10", math_format(10^.x))) +
  labs(
    fill = "Trans. type:"
    , x = "amount (log10 scale, local currency)"
    , title = "Transaction amount densities, genuine transactions") +
  annotation_logticks(sides = "b") +
  theme(axis.text.x= element_text(size=10))
```



As the graph above shows, the range of amounts is wide, from several hundreds to tens of millions in local currency, with significant differences between the different types of transaction. Please note that `CASH-IN` and `CASH-OUT` are overlapping and have almost identical distributions.

3.2.3 Fraudulent transactions

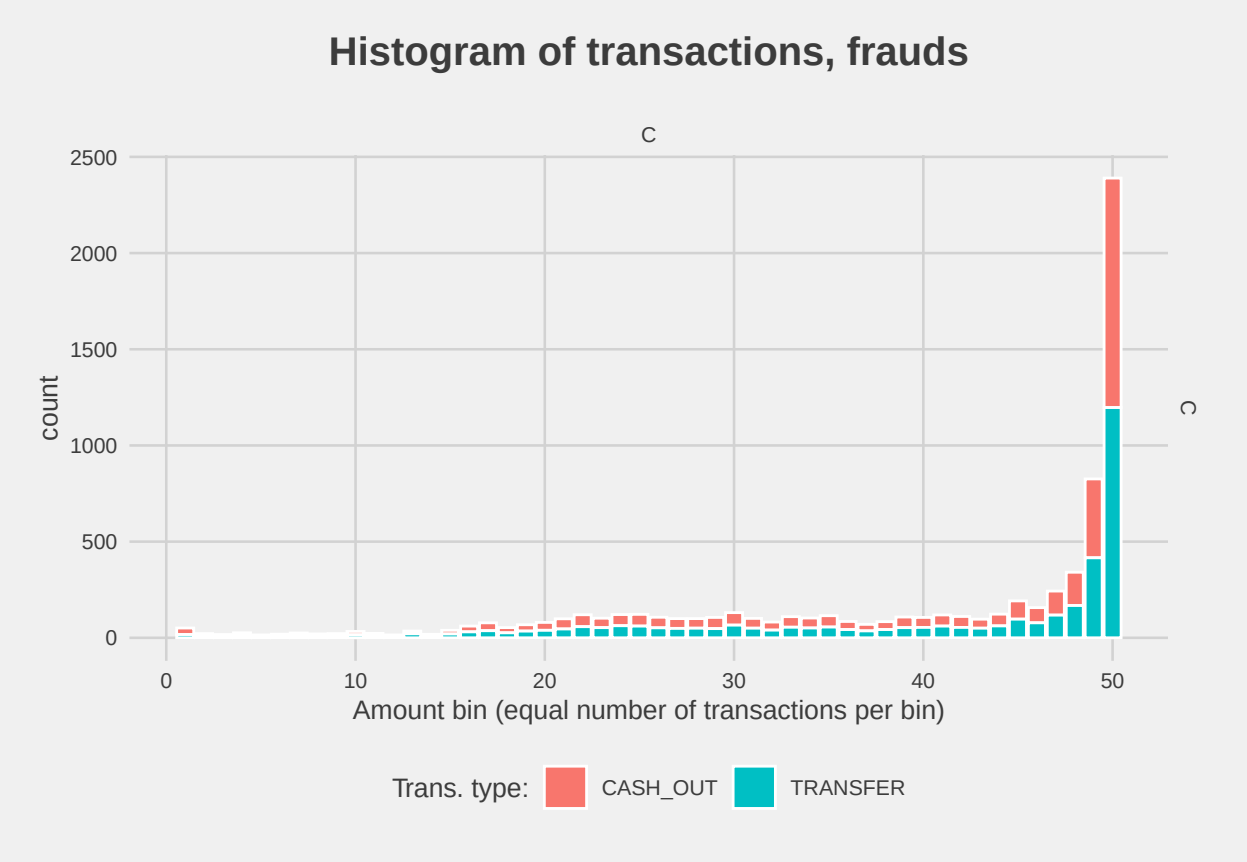
Let's now look specifically at fraudulent transactions to see if any specific pattern can be identified.

3.2.3.1 Histograms of transaction for the different types According to the official description, fraud occurs when money is transferred from a personal account (`typeOrig` C) to a merchant account (`typeDest` M) before the money is withdrawn:

> "In this specific dataset the fraudulent behavior of the agents aims to profit by taking control of customers accounts and try to empty the funds by transferring to another account and then cashing out of the system."[9]

Let's look first at the histogram of transactions `amount`. In the histogram below, `amount` has been split into 100 bins with an equal number of observations, as previously (each bin contains $N/100$ observations, where N is the total number of transactions in `playset`). Only fraudulent transactions are counted.

```
# Histogram of fraudulent transactions per amount and type,
# typeDest and typeOrig categories.
# Amount is split between 50 bins with the same number of transactions.
playset %>%
  mutate(bin = cut_number(amount, 50, labels=FALSE)) %>%
  filter(isFraud == "F1") %>%
  ggplot(aes(bin, fill = type)) +
  geom_histogram(stat = "count", color = "white") +
  facet_grid(typeDest ~ typeOrig) +
  labs(
    fill = "Trans. type:"
    , x = "Amount bin (equal number of transactions per bin)"
    , title = "Histogram of transactions, frauds")
```



[9] https://www.kaggle.com/ntnu-testimon/paysim1

This histogram shows that:

1. Fraudulent transactions are of type `CASH_OUT` and `TRANSFER` only.
2. The proportion of fraudulent transactions is higher for higher amounts (higher counts of fraudulent transactions in higher bins).
3. There is no fraudulent transactions with `typeDest` "M", or `facet_grid()` would have drawn an additional histogram. This observation is confirmed by the distribution table below: all 7,391 fraudulent transactions are of `typeOrig` and `typeDest` "C".

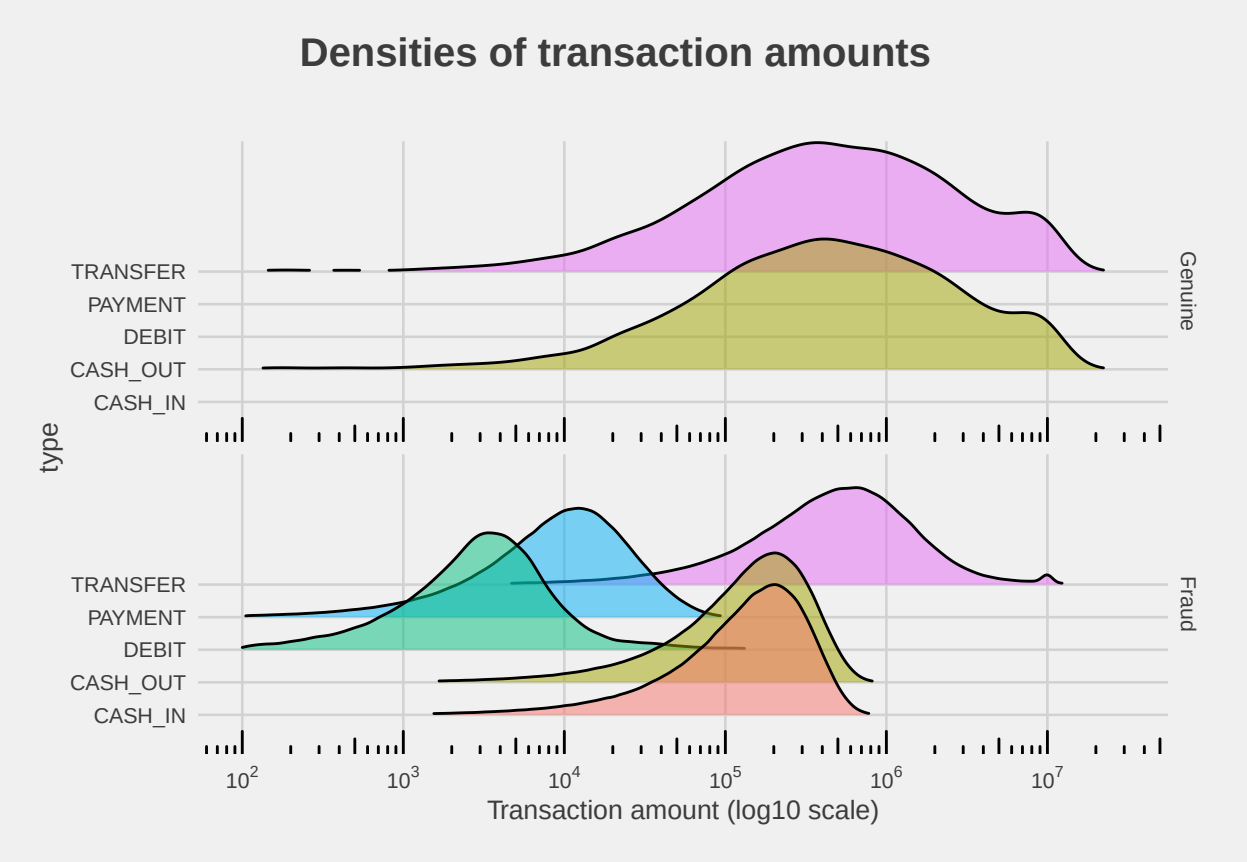
```
# Distribution table of fraudulent and genuine transactions
# between two perdictors: typeOrig, typeDest
kable(playset %>%
  group_by(isFraud, typeOrig, typeDest) %>%
  summarise(Count = n()))
```

isFraud	typeOrig	typeDest	Count
F1	C	C	7391
G0	C	C	3782872
G0	C	M	1936094

3.2.3.2 Amount distribution Let's look into the distribution of the transaction amounts.

```
# Plot the density vs. transaction amount using a log10 scale,
# for fraudulent and non-fraudulent transactions

# Define the labeller to display "Genuine" and "Fraud"
is_fraud <- list("0" = "Genuine", "1" = "Fraud")

my_labeller <- function(variable,value){
  return(is_fraud[value])
}

playset %>%
  ggplot(aes(x = amount, y = type, height = stat(density),
             fill = type)) +
  geom_density_ridges(alpha = 0.5,
                      stat = "density",
                      show.legend = FALSE,
                      scale = 4,
                      draw_baseline = FALSE,
                      rel_min_height = 0.01) +
  scale_x_log10(
    limits = c(10^2, 3*10^7),
    breaks = trans_breaks("log10", function(x) 10^x),
    labels = trans_format("log10", math_format(10^.x))
    ) +
  scale_y_discrete(expand = expand_scale(mult = c(0.3, 1))) +
  annotation_logticks(sides = "b") +
  xlab("Transaction amount (log10 scale)") +
  ggtitle("Densities of transaction amounts") +
```

```
theme(plot.title = element_text(margin=ggplot2::margin(0,0,25,0),
                                hjust = 0.3)) +
facet_grid(isFraud ~., labeller = my_labeller)
```



As the graph above shows, the range of amounts is narrower for fraudulent transactions than for genuine transactions: it varies from tens of thousands to tens of millions. The highest densities of genuine transactions depends on the type of transaction and varies largely. It is the highest for transfers, around several hundreds of thousands. In comparison, fraudulent transactions have the highest density around several hundreds of thousands both for `TRANSFER` and `CASH-OUT`.

3.2.4 Balance analysis

The previous sections focused on analysing the distributions of the transactions based on their amount, type and origin/destination. In this section the other aspects of the transactions are analysed (balance and patterns).

3.2.4.1 Cash-in transactions The previous analysis has shown that cash-in transactions are never flagged as fraud in the dataset and that these transactions never involve merchants (M).

As the result of the query below shows, `CASH_IN` transactions contains both sender and recipient information (`nameOrig` and `nameDest`). This raises a question as depositing money on an account only involves the client, unless the recipient of the transaction (the merchant who credit the client's account) is included somehow. However, neither the name of the recipient or originator starts with "M" so this does not seem to be the explanation. Maybe the person who credited the client's account is not officially registered as a merchant but informally acts as such?

```r
# Display 10 cash-in transactions
head(
  playset %>%
    select(-step, -isFlaggedFraud, -isFraud,
           -typeDest, -typeOrig) %>%
    filter(type == "CASH_IN")
  , 10)
```

```
##       type    amount    nameOrig oldBalanceOrig newBalanceOrig    nameDest
## 1  CASH_IN 143236.26 C1862994526            0.0       143236.3 C1688019098
## 2  CASH_IN  35902.49  C839771540       371688.2       407590.7 C2001112025
## 3  CASH_IN 232953.64 C1037163664       407590.7       640544.3   C33524623
## 4  CASH_IN 193492.68 C1200546947       706457.2       899949.9 C1531333864
## 5  CASH_IN  60836.64  C443713699       899949.9       960786.6  C488044861
## 6  CASH_IN  62325.15  C695530017       960786.6      1023111.7  C564160838
## 7  CASH_IN 349640.51 C1493042329      1023111.7      1372752.2  C909295153
## 8  CASH_IN 135324.19 C1751403001      1372752.2      1508076.4  C453211571
## 9  CASH_IN 380015.14 C1717433286      1508076.4      1888091.6  C401424608
## 10 CASH_IN 418688.27 C1756819670      1888091.6      2306779.8  C401424608
##    oldBalanceDest newBalanceDest
## 1        608932.2       97263.78
## 2         49003.3           0.00
## 3       1172672.3     1517262.16
## 4       1247284.1       55974.56
## 5        143436.0        5203.54
## 6       1880271.7     1254956.07
## 7        360950.6     5602234.95
## 8       1356747.9     3461666.05
## 9       1336470.2     1178808.14
## 10       956455.0     1178808.14
```

Looking beyond these 10 transactions confirms this analysis. This is done by looking at the number of transactions which have a balance error on the sender or recipient account (because of rounding-up, an error means any difference higher than 0.1). As the result below shows, there are only a few errors on the origin account (`errBalanceOrig`). However, it appears that there are many more balance errors on the destination account (`errBalanceDest`).

```r
# Filter cash-in transactions
playset_CASH_IN <- playset %>% filter(type == "CASH_IN")

# Display total number of cash-in transactions
count_cash_in <- formated_nrow(playset_CASH_IN)
message("Total number of cash-in transactions in the dataset: "
  , count_cash_in)
```

```
## Total number of cash-in transactions in the dataset: 1,259,640
```

```r
# Display number of transactions with a balance error on the origin account
# that is greater than 0.1 to omit rounding errors.
orig_error <- formated_nrow(
  playset_CASH_IN %>%
    mutate(errBalanceOrig = oldBalanceOrig + amount - newBalanceOrig) %>%
```

```
    filter(abs(errBalanceOrig) > 0.1)
  )

message(
  "Nb. of cash-in transactions with an error in the origin balance: "
  , orig_error
  )
```

```
## Nb. of cash-in transactions with an error in the origin balance: 31
```

```
# Display number of transactions with a balance error on the destination account
dest_error <- formated_nrow(
  playset_CASH_IN %>%
    mutate(errBalanceDest = oldBalanceDest - amount - newBalanceDest) %>%
    filter(abs(errBalanceDest) > 0.1)
  )

message(
  "Nb. of cash-in transactions with an error in the destination balance: "
  , dest_error
  )
```

```
## Nb. of cash-in transactions with an error in the destination balance: 330,591
```

Let's look at errors on the destination account. As the result of the query below shows, the error amount can be significant. No easy or obvious explanation for these errors can be identified.

```
# Display the 10 largest errors on the destination account
kable(head(
  playset_CASH_IN %>%
  mutate(errBalanceDest = - newBalanceDest - amount + oldBalanceDest) %>%
  filter(abs(errBalanceDest) > 0.1) %>%
  select(-step, -type, -nameDest, -nameOrig, -typeOrig, -typeDest,
         -isFraud, -isFlaggedFraud) %>%
  arrange(desc(abs(errBalanceDest))), 10))
```

amount	oldBalanceOrig	newBalanceOrig	oldBalanceDest	newBalanceDest	errBalanceDest
21791.57	828107.4	849899.02	113554059.0	167882762	-54350495
91666.99	204.0	91870.99	41405174.7	76111985	-34798477
113311.50	2784447.8	2897759.32	251855.0	34720903	-34582360
361850.31	5399110.5	5760960.84	609857.3	34641271	-34393264
1371.97	227.0	1598.97	11369.0	30302606	-30292609
27337.82	5629503.3	5656841.11	147552.2	29448089	-29327874
29827.58	8040820.9	8070648.50	158709.0	29408777	-29279896
465693.50	24560.0	490253.50	18981.0	28591377	-29038089
9660.61	93111.0	102771.61	31033204.4	59919426	-28895883
328978.72	8303390.6	8632369.34	459949.4	28783585	-28652615

Finally, let's look at the error on the origin account. From the result of the query below, it seems that errors occur when the balance is not updated, i.e. when the amount of the transaction is not added to the

account. In other words, `oldBalanceOrig == newBalanceOrig`. This is confirmed by the last query of the code chunk below.

```
# Filter only transactions with a balance error on the origin account
origin_error_df <- playset_CASH_IN %>%
       select(-step, -type, -oldBalanceDest, -newBalanceDest, -nameDest,
              -isFlaggedFraud, -isFraud) %>%
       mutate(errBalanceOrig = newBalanceOrig - amount - oldBalanceOrig) %>%
       filter(abs(errBalanceOrig) > 0.1)

# Display transactions displaying a balance error on the origin account
kable(head(origin_error_df, 10))
```

amount	nameOrig	typeOrig	oldBalanceOrig	newBalanceOrig	typeDest	errBalanceOrig
110226.34	C1475192960	C	1601450.6	1601450.6	C	-110226.34
770537.37	C2015999862	C	8499043.1	8499043.1	C	-770537.37
61505.91	C1790387225	C	4910403.4	4910403.4	C	-61505.91
404167.60	C984412970	C	4910403.4	4910403.4	C	-404167.60
42012.45	C1137667747	C	350501.2	350501.2	C	-42012.45
187256.62	C1106468732	C	521018.6	521018.6	C	-187256.62
208664.25	C1216611732	C	4800759.2	4800759.2	C	-208664.25
301755.98	C1544031987	C	6509377.5	6509377.5	C	-301755.98
305762.25	C690051320	C	1957713.7	1957713.7	C	-305762.25
275712.54	C1378612814	C	6269058.3	6269058.3	C	-275712.54

```
# Check if all the errors are due to amount not being substracted
# from oldBalanceOrig
amount_not_reflected <- formated_nrow(
  origin_error_df %>% filter(oldBalanceOrig == newBalanceOrig)
  )

message(
  "Nb. of cash-in transactions displaying an error on the origin balance: "
  , orig_error
  )
```

```
## Nb. of cash-in transactions displaying an error on the origin balance: 31
```

```
# Display the number of transactions with balance error
message("Nb. of these transactions where 'newBalanceOrig' is not updated: "
  , amount_not_reflected)
```

```
## Nb. of these transactions where 'newBalanceOrig' is not updated: 31
```

```
# Do some cleaning
rm(dest_error, orig_error, origin_error_df, count_cash_in,
   playset_CASH_IN, amount_not_reflected)
```

As a conclusion, this analysis shows that transactions of type `CASH-IN` do not involve merchants but only agents of type `C`. A few transactions also contains balance errors on the origin account, and these errors seem to come from the fact that `newBalanceOrig` is not updated after the transaction. A lot more transaction contains errors on the destination account, errors which can be rather large, and cannot be easily explained.

3.2.4.2 Payment transactions Let's conduct a similar analysis for payment transactions. From the previous analysis we know that the origin of all these transactions are clients (`typeDest == C`) and that their destination are always merchants (`typeDest == M`). As the query results below show, the old and new balance of the destination account are null, which is in line with the data description (all merchant accounts are null).

```
# Display 10 payment transactions
kable(head(playset %>%
       select(-step, -isFlaggedFraud, -isFraud, -nameOrig, -nameDest,
              -typeDest, -typeOrig) %>%
       filter(type == "PAYMENT"), 10))
```

type	amount	oldBalanceOrig	newBalanceOrig	oldBalanceDest	newBalanceDest
PAYMENT	9839.64	170136.0	160296.36	0	0
PAYMENT	1864.28	21249.0	19384.72	0	0
PAYMENT	11668.14	41554.0	29885.86	0	0
PAYMENT	7817.71	53860.0	46042.29	0	0
PAYMENT	7861.64	176087.2	168225.59	0	0
PAYMENT	3099.97	20771.0	17671.03	0	0
PAYMENT	2560.74	5070.0	2509.26	0	0
PAYMENT	11633.76	10127.0	0.00	0	0
PAYMENT	4098.78	503264.0	499165.22	0	0
PAYMENT	1563.82	450.0	0.00	0	0

```
# Filter payment transactions
playset_PAYMENT <- playset %>% filter (type == "PAYMENT")

# Count how many payment transactions have oldBalanceDest > 0
number_oldBalanceDest <- nrow(playset_PAYMENT %>% filter(oldBalanceDest > 0))
message("Nb. of payment transactions where oldBalanceDest > 0: "
  , number_oldBalanceDest)
```

```
## Nb. of payment transactions where oldBalanceDest > 0: 0
```

```
# Count how many payment transactions have newBalanceDest > 0
number_newBalanceDest <- nrow(playset_PAYMENT %>% filter(newBalanceDest > 0))
message("Nb. of payment transactions where oldBalanceDest > 0: "
  , number_newBalanceDest)
```

```
## Nb. of payment transactions where oldBalanceDest > 0: 0
```

Now let's look at the balance error on the origin account. As the query below shows, almost half the transactions display such an error.

```
# Display total number of payment transactions
count_payment <- formated_nrow(playset_PAYMENT)
message("Total number of payment transactions in the dataset: "
  , count_payment)
```

```
## Total number of payment transactions in the dataset: 1,936,094
```

```r
# Caculate the number of transactions with
# balance error on the origin account
orig_error <- formated_nrow(
  playset_PAYMENT %>%
    mutate(errBalanceOrig = oldBalanceOrig - amount - newBalanceOrig) %>%
    filter(abs(errBalanceOrig) > 0.1)
  )

# Display the number of such transactions
message(
  "Nb. of payment transactions displaying an error on the origin balance: "
  , orig_error
  )
```

```
## Nb. of payment transactions displaying an error on the origin balance: 990,609
```

Let's look at the cause of these errors. As the results of the queries below show, it seems that errors occur when a payment is made from an account which does not have enough money for the transaction to clear, so the new balance is zero. This is confirmed by the last query. According to this query, the number of such payments is equal to the number of payments which display origin errors.

```r
# Select what to display
payment_errors_df <- playset_PAYMENT %>%
  select(- step, -isFraud, -isFlaggedFraud, -nameDest, -typeDest,
         -oldBalanceDest, -newBalanceDest) %>%
  mutate(errBalanceOrig = oldBalanceOrig - amount - newBalanceOrig) %>%
  filter(abs(errBalanceOrig) > 0.1)

# Display payment transactions with errors
head(payment_errors_df, 5)
```

```
##      type   amount    nameOrig typeOrig oldBalanceOrig newBalanceOrig
## 1 PAYMENT 11633.76 C1716932897        C          10127              0
## 2 PAYMENT  1563.82  C761750706        C            450              0
## 3 PAYMENT  6061.13 C1043358826        C            443              0
## 4 PAYMENT  9920.52  C764826684        C              0              0
## 5 PAYMENT  3448.92 C2103763750        C              0              0
##   errBalanceOrig
## 1       -1506.76
## 2       -1113.82
## 3       -5618.13
## 4       -9920.52
## 5       -3448.92
```

```r
not_enough <- formated_nrow(
  payment_errors_df %>% filter(amount > oldBalanceOrig)
  )

message("Nb. of payment transactions where 'amount' > initial balance: "
  , not_enough)
```

```
## Nb. of payment transactions where 'amount' > initial balance: 990,609
```

```r
# Do some cleaning
rm(playset_PAYMENT, orig_error, count_payment, number_oldBalanceDest,
   number_newBalanceDest, payment_errors_df, not_enough)
```

As a conclusion, it appears that transactions of type `PAYMENT` mostly contain pieces of information which are in line with the official data description: account balances of merchant account are 0, and all the destination agents are of type 'merchant'. A few transactions contains errors on the origin account, and these errors seem to come from the fact that `amount` is greater than `oldBalanceOrig`, and therefore `newBalanceOrig` is equal to 0. It is not clear though how the difference is paid for, as the transaction does not seem to be cancelled.

3.2.4.3 Debit transactions

Debit transactions are similar to cash-out transactions except that the money is wired to a bank account. As the previous analysis has shown, no debit transaction is associated with fraud and all these transactions are from an individual (C) to another individual (c). No merchant (M) is involved, which contradict the official description of the dataset.

As the results of the queries below show, there are significant balance errors on both the origin and destination accounts.

```r
# Filter debit transactions
playset_DEBIT <- playset %>% filter(type == "DEBIT")

# Display 10 DEBIT transactions
head(
  playset_DEBIT %>% select(-step, -isFraud, -isFlaggedFraud)
  , 10)
```

```
##     type  amount    nameOrig typeOrig oldBalanceOrig newBalanceOrig    nameDest
## 1  DEBIT 5337.77  C712410124        C          41720       36382.23  C195600860
## 2  DEBIT 9644.94 C1900366749        C           4465           0.00  C997608398
## 3  DEBIT 9302.79 C1566511282        C          11299        1996.21 C1973538135
## 4  DEBIT 1065.41 C1959239586        C           1817         751.59  C515132998
## 5  DEBIT 5758.59 C1466917878        C          32604       26845.41 C1297685781
## 6  DEBIT 5529.13  C867288517        C           8547        3017.87  C242131142
## 7  DEBIT 4510.22  C280615803        C          10256        5745.78 C1254526270
## 8  DEBIT 8727.74  C166694583        C         882770      874042.26 C1129670968
## 9  DEBIT 4874.49  C811207775        C            153           0.00 C1971489295
## 10 DEBIT 5149.66 C1955990522        C           4782           0.00 C1330106945
##    typeDest oldBalanceDest newBalanceDest
## 1         C          41898       40348.79
## 2         C          10845      157982.12
## 3         C          29832       16896.70
## 4         C          10330           0.00
## 5         C         209699       16997.22
## 6         C          10206           0.00
## 7         C          10697           0.00
## 8         C          12636           0.00
## 9         C         253104           0.00
## 10        C          52752       24044.18
```

```r
# Display total number of debit transactions
count_debit <- formated_nrow(playset_DEBIT)
```

```r
message("Total number of debit transactions in the dataset: "
  , count_debit)
```

```
## Total number of debit transactions in the dataset: 37,316
```

```r
# Add errBalanceOrig and errBalanceDest to playset_DEBIT
playset_DEBIT <- playset_DEBIT %>%
  mutate(errBalanceOrig = oldBalanceOrig - amount - newBalanceOrig) %>%
  mutate(errBalanceDest = oldBalanceDest + amount - newBalanceDest)

# Display the number of transactions showing a balance error
# on the origin account
orig_error <- formated_nrow(
  playset_DEBIT %>% filter(abs(errBalanceOrig) > 0.1)
  )
message(
  "Nb. of debit transactions with an error on the origin balance: ",
  orig_error
  )
```

```
## Nb. of debit transactions with an error on the origin balance: 10,635
```

```r
# Display the number of transactions showing a balance error
# on the destination account
dest_error <- formated_nrow(
  playset_DEBIT %>% filter(abs(errBalanceDest) > 0.1)
  )
message(
  "Nb. of debit transactions with an error on the destination balance: ",
  dest_error
  )
```

```
## Nb. of debit transactions with an error on the destination balance: 3,112
```

Let's have a look at the errors on the origin account. As the results of the query below shows, it seems that the errors are due to an initial account balance too low for the entire transaction to take place. Consequently, the final account balance is zero. This is confirmed by the last query of the code block. This explanation account for all the errors on the origin account.

```r
# Display the 10 largest error on the origin account
orig_errors_df <- playset_DEBIT %>%
       filter(abs(errBalanceOrig) > 0.1) %>%
       arrange(desc(abs(errBalanceOrig)))

kable(head(
  orig_errors_df %>%
    select(-type, -step, -isFraud, -isFlaggedFraud, -nameOrig,
           -nameDest, -typeDest, -typeOrig, - errBalanceDest)
  , 10))
```

amount	oldBalanceOrig	newBalanceOrig	oldBalanceDest	newBalanceDest	errBalanceOrig
569077.5	0	0	1215873.2	1784950.7	-569077.5
547139.8	0	0	4059422.5	4606562.3	-547139.8
417825.1	0	0	542309.2	960134.2	-417825.1
402836.2	0	0	576521.4	979357.7	-402836.2
389762.7	118	0	1890046.4	2279809.1	-389644.7
349853.3	0	0	3566031.6	3915884.8	-349853.3
365375.8	20316	0	499503.3	864879.0	-345059.8
346384.6	12858	0	4633316.1	4979700.7	-333526.6
328523.4	0	0	481960.7	810484.0	-328523.4
278659.1	2281	0	6618783.9	6897443.0	-276378.1

```
not_enough <- formated_nrow(
  orig_errors_df %>% filter(amount > oldBalanceOrig)
  )

message(
  "Nb. of debit transactions where 'amount' > account initial balance: ",
  not_enough
  )
```

```
## Nb. of debit transactions where 'amount' > account initial balance: 10,635
```

As far as errors on the destination account are concerned, there is no easy or straightforward explanation, and these errors can be large.

```
# Display the 10 largest error on the destination account
dest_errors_df <- playset_DEBIT %>%
       mutate(errBalanceDest = newBalanceDest - amount - oldBalanceDest) %>%
       filter(abs(errBalanceDest) > 0.1)

# Display the 10 largest error on the destination account
kable(head(
  dest_errors_df %>%
    select(-type, -step, -isFraud, -isFlaggedFraud, -typeDest, -typeOrig,
           -nameOrig, -nameDest, -oldBalanceOrig, -newBalanceOrig) %>%
    arrange(desc(abs(errBalanceDest))), 10))
```

amount	oldBalanceDest	newBalanceDest	errBalanceOrig	errBalanceDest
3108.91	479935.31	18169202	-3108.91	17686158
5252.43	45862.00	16793942	0.00	16742828
17122.27	145036.00	16420813	0.00	16258655
677.17	643543.84	11530852	0.00	10886631
466.76	161270.71	10408505	-364.76	10246767
3083.13	5471.89	10178554	0.00	10169999
891.93	55432.00	8813777	0.00	8757453
146.51	221.00	6882449	0.00	6882081
3035.96	329055.00	7074755	-2145.96	6742664
1739.48	332090.96	7074755	0.00	6740924

```r
# Do some cleaning
rm(dest_error, orig_error, count_debit, playset_DEBIT,
   dest_errors_df, orig_errors_df)
```

As a conclusion, transactions of type `DEBIT` only involve agents of type 'client' (C) and none of type 'merchant' (M) which contradicts the official description of the dataset.

It appears that a large proportion of these transactions contains balance errors. Errors on the destination account cannot be easily explained. All errors on the origin account occurs when `amount` is greater than `oldBalanceOrig`, and therefore `newBalanceOrig` is equal to `0`. This explanation covers all the error cases. It can be assumed that the difference has been paid in cash by the client.

3.2.4.4 Cash-out transactions

The previous analysis has shown that cash-out transactions can be flagged as fraud, and that cash-out transactions never involve merchants. As the queries below show, `CASH_OUT` transactions contains errors on both the origin and destination accounts.

```r
# Filter cash-out transactions
playset_CASH_OUT <- playset %>% filter(type == "CASH_OUT")

# Display total number of cash-out transactions
count_cash_out <- formated_nrow(playset_CASH_OUT)
message("Total number of cash-out transactions in the dataset: "
  , count_cash_out)
```

```
## Total number of cash-out transactions in the dataset: 2,013,765
```

```r
# Add errBalanceOrig and errBalanceDest
playset_CASH_OUT <- playset_CASH_OUT %>%
  mutate(errBalanceOrig = oldBalanceOrig - amount - newBalanceOrig) %>%
  mutate(errBalanceDest = oldBalanceDest + amount - newBalanceDest)

# Number of transactions displaying a balance error on the origin account
orig_error <- formated_nrow(
  playset_CASH_OUT %>%filter(abs(errBalanceOrig) > 0.1)
  )

message(
  "Nb. of cash-out transactions with an error on the origin balance: "
  , orig_error
  )
```

```
## Nb. of cash-out transactions with an error on the origin balance: 1,783,044
```

```r
# Number of transactions displaying a balance error on the destination account
dest_error <- formated_nrow(
  playset_CASH_OUT %>% filter(abs(errBalanceDest) > 0.1)
  )

message(
  "Nb. of cash-out transactions with an error on the dest. balance: ",
  dest_error
  )
```

```
## Nb. of cash-out transactions with an error on the dest. balance: 190,461
```

Looking at the transactions with errors on the origin account, it seems that they are due to transactions with `amount` higher than `oldBalanceOrig`. This explanation account for all but 2 errors. It is not clear how such a transaction can occur, unless the client is given only the amount of cash left on her account. For the two remaining errors, it seems that `newBalanceOrig` has not been updated after the transaction. Maybe in this cases the transaction was cancelled for some reason.

```
# Select the transactions with errors on the origin account
orig_errors_df <- playset_CASH_OUT %>%
       filter(abs(errBalanceOrig) > 0.1)

# Display the largest error on the origin account
kable(head(
  orig_errors_df %>%
    select(-type, -step, -isFraud, -isFlaggedFraud, -nameOrig,
    -nameDest, -typeDest, -typeOrig, - errBalanceDest)
  , 10))
```

amount	oldBalanceOrig	newBalanceOrig	oldBalanceDest	newBalanceDest	errBalanceOrig
229133.94	15325.00	0	5083	51513.44	-213808.94
110414.71	26845.41	0	288800	2415.16	-83569.30
56953.90	1942.02	0	70253	64106.18	-55011.88
5346.89	0.00	0	652637	6453430.91	-5346.89
23261.30	20411.53	0	25742	0.00	-2849.77
82940.31	3017.87	0	132372	49864.36	-79922.44
94253.33	25203.05	0	99773	965870.05	-69050.28
28404.60	0.00	0	51744	0.00	-28404.60
75405.10	0.00	0	104209	46462.23	-75405.10
50101.88	0.00	0	67684	9940339.29	-50101.88

```
# Look at how many errors are explained by a transaction amount greater than
# the initial account balance
not_enough <- formated_nrow(
  orig_errors_df %>% filter(amount > oldBalanceOrig)
  )

message(
  "Nb. of cash-out transactions where 'amount' > initial balance: "
  , not_enough
  )
```

```
## Nb. of cash-out transactions where 'amount' > initial balance: 1,783,042
```

```
# Look at the errors which are not explained by the previous explanation
orig_errors_df %>%
  filter(amount < oldBalanceOrig) %>%
  select(-step, -type, -nameOrig, -nameDest, -typeOrig, -typeDest,
         -isFlaggedFraud, -errBalanceDest)
```

```
##      amount oldBalanceOrig newBalanceOrig oldBalanceDest newBalanceDest isFraud
```

```
## 1  30308.45        88926.2        88926.2        88926.2       122750.5      G0
## 2 144659.77       149736.0       149736.0       149736.0            0.0      G0
##   errBalanceOrig
## 1      -30308.45
## 2     -144659.77
```

As far as errors on the destination account are concerned, there is no easy nor straightforward explanation, and these errors can be large: about a third of them have an error larger than twice the amount of the transaction in absolute value.

```
# Display the 10 largest error on the destination account
dest_errors_df <- playset_CASH_OUT  %>%
       filter(abs(errBalanceDest) > 0.1)

# Display the 10 largest error on the destination account
kable(head(
  dest_errors_df %>%
    select(-type, -step, -isFraud, -isFlaggedFraud,
           -typeDest, -typeOrig, -nameOrig, -nameDest,
           -oldBalanceOrig, -newBalanceOrig, -errBalanceOrig) %>%
    arrange(desc(abs(errBalanceDest)))
  , 10))
```

amount	oldBalanceDest	newBalanceDest	errBalanceDest
115714.47	136757866.4	194661382	-57787801
336898.24	68944281.0	120004461	-50723282
147473.45	91204806.3	141218514	-49866235
296975.90	0.0	42661092	-42364117
272808.72	58363434.8	95787350	-37151106
648453.90	47862612.2	85466784	-36955718
170259.60	46688798.7	81496526	-34637468
149879.00	46670604.1	80280526	-33460043
99776.21	115310.0	31839617	-31624531
145505.69	215086.2	31839617	-31479025

```
count_twice_amount <- formated_nrow(
  playset_CASH_OUT %>% filter(abs(errBalanceDest) > 2*amount)
  )

message("Nb. of transactions where abs(errBalanceDest) > 2 * amount: "
  , count_twice_amount)
```

```
## Nb. of transactions where abs(errBalanceDest) > 2 * amount: 63,782
```

Finally, let's look more into details at the fraudulent cash-out transactions. It turns out that all but one fraudulent cash-out transactions have `newBalanceOrig == 0`. However, not all cash-out transactions with `newBalanceOrig == 0` are fraudulent.

```
# Display total number of fraudulent cash-out transactions
count_cash_out_fraud <- formated_nrow(
  playset_CASH_OUT %>% filter(isFraud == "F1")
```

```
  )

message(
  "Total number of fraudulent cash-out transactions in the dataset: "
  , count_cash_out_fraud
  )
```

```
## Total number of fraudulent cash-out transactions in the dataset: 3,710
```

```
# Visualise a few fraudulent cash-out transactions
head(
  playset_CASH_OUT %>%
    filter(isFraud == "F1") %>%
    select(-step, -type, -typeOrig, -typeDest, -nameOrig,
           -nameDest, -isFlaggedFraud)
  , 10)
```

```
##        amount oldBalanceOrig newBalanceOrig oldBalanceDest newBalanceDest
## 1      181.00         181.00              0       21182.00           0.00
## 2     2806.00        2806.00              0       26202.00           0.00
## 3    20128.00       20128.00              0        6268.00       12145.85
## 4   416001.33           0.00              0         102.00     9291619.62
## 5  1277212.77     1277212.77              0           0.00     2444985.19
## 6    35063.63       35063.63              0       31140.00        7550.03
## 7    25071.46       25071.46              0        9083.76       34155.22
## 8   132842.64        4499.08              0           0.00      132842.64
## 9   235238.66      235238.66              0           0.00      235238.66
## 10 1096187.24     1096187.24              0           0.00     1096187.24
##    isFraud errBalanceOrig errBalanceDest
## 1       F1            0.0       21363.00
## 2       F1            0.0       29008.00
## 3       F1            0.0       14250.15
## 4       F1      -416001.3    -8875516.29
## 5       F1            0.0    -1167772.42
## 6       F1            0.0       58653.60
## 7       F1            0.0           0.00
## 8       F1      -128343.6           0.00
## 9       F1            0.0           0.00
## 10      F1            0.0           0.00
```

```
# Confirm how many cash-out transactions have newBalanceOrig == 0
fraud_newBal <- formated_nrow(
  playset_CASH_OUT %>%
    filter(isFraud == "F1") %>%
    filter(newBalanceOrig == 0)
  )

message(
  "Total number of fraudulent cash-out transactions with 'newBalanceOrig' = 0: "
  , fraud_newBal
  )
```

```
## Total number of fraudulent cash-out transactions with 'newBalanceOrig' = 0: 3,709
```

```r
# Compare this to the total number of genuine cash-out transactions with
# newBalanceOrig == 0
genuine_newBal <- formated_nrow(
  playset_CASH_OUT %>%
    filter(isFraud == "G0") %>%
    filter(newBalanceOrig == 0)
  )

message(
  "Total number of genuine cash-out transactions with 'newBalanceOrig' = 0: "
  , genuine_newBal
  )
```

```
## Total number of genuine cash-out transactions with 'newBalanceOrig' = 0: 1,783,019
```

```r
# Do some cleaning
rm(dest_error, orig_error, count_cash_out, not_enough, txt,
   orig_errors_df, playset_CASH_OUT, count_cash_out_fraud, genuine_newBal,
   fraud_newBal, dest_errors_df, count_twice_amount)
```

```
## Warning in rm(dest_error, orig_error, count_cash_out, not_enough, txt,
## orig_errors_df, : object 'txt' not found
```

As a conclusion, transactions of type `CASH-OUT` only involve agents of type 'client' (C) and none of type 'merchant' (M) which contradicts the official description of the dataset.

These transactions contain a large number of balance errors on the destination account which cannot be easily explained. All errors on the origin account occurs when `amount` is greater than `oldBalanceOrig`, and therefore `newBalanceOrig` is equal to 0. It is not clear how such transactions can occur, unless the client is given only the amount of cash left on her account. This explanation covers all the error cases but 1. In the transaction with the remaining errors, it seems that the new account balance on the origin account is not updated.

As far as fraudulent cash-out transactions are concerned, all but one in the dataset have a new balance of the origin account equal to 0. However, a lot of *genuine* cash-out transactions also have a new balance of the origin account equal to 0, so this aspect on its own is not enough to identify fraudulent cash-out transactions.

3.2.4.5 Transfer transactions

The previous analysis has shown that transfer transactions can be flagged as fraud, and that transfer transactions never involve merchants. As the query below shows, `TRANSFER` transactions contains errors on both the origin and destination accounts.

```r
# Filter transfer transactions
playset_TRANSFER <- playset %>% filter(type == "TRANSFER")

# Display total number of transfer transactions
count_transfer <- formated_nrow(playset_TRANSFER)
message("Total number of transfer transactions in the dataset: "
  , count_transfer)
```

```
## Total number of transfer transactions in the dataset: 479,542
```

```r
# Add errBalanceOrig and errBalanceDest
playset_TRANSFER <- playset_TRANSFER %>%
  mutate(errBalanceOrig = oldBalanceOrig - amount - newBalanceOrig) %>%
  mutate(errBalanceDest = oldBalanceDest + amount - newBalanceDest)

# Number of transactions displaying a balance error on the origin account
orig_error <- formated_nrow(
  playset_TRANSFER %>% filter(abs(errBalanceOrig) > 0.1)
  )

message(
  "Nb. of transfer transactions displaying an error on the origin balance: "
  , orig_error
  )
```

```
## Nb. of transfer transactions displaying an error on the origin balance: 456,730
```

```r
# Display number of transactions displaying a balance error on the destination account
dest_error <- formated_nrow(
  playset_TRANSFER %>% filter(abs(errBalanceDest) > 0.1)
  )

message(
  "Nb. of transfer transactions displaying an error on the destination balance: "
  , dest_error
  )
```

```
## Nb. of transfer transactions displaying an error on the destination balance: 52,200
```

Looking at the transactions with errors on the origin account, it seems that they are due to the transaction amount being higher than `oldBalanceOrig`. This explanation accounts for all but 6 errors. It is not clear how such a transaction can occur in practice, unless the client gives the complement in cash.
For the 6 remaining errors, it seems that `newBalanceOrig` has not been updated after the transactions took place. Interestingly, 5 out of 6 transactions are frauds, and for all these transactions `oldBalanceDest` and `newBalanceDest == 0`.

```r
# Select the transactions with errors on the origin account
orig_errors_df <- playset_TRANSFER %>%
       filter(abs(errBalanceOrig) > 0.1)

# Display the largest error on the origin account
kable(head(
  orig_errors_df %>%
    select(-type, -step, -isFraud, -isFlaggedFraud, -nameOrig,
    -nameDest, -typeDest, -typeOrig, - errBalanceDest)
  , 20))
```

amount	oldBalanceOrig	newBalanceOrig	oldBalanceDest	newBalanceDest	errBalanceOrig
311685.89	10835.00	0	6267.00	2719172.89	-300850.89
42712.39	10363.39	0	57901.66	24044.18	-32349.00
77957.68	0.00	0	94900.00	22233.65	-77957.68

amount	oldBalanceOrig	newBalanceOrig	oldBalanceDest	newBalanceDest	errBalanceOrig
17231.46	0.00	0	24672.00	0.00	-17231.46
78766.03	0.00	0	103772.00	277515.05	-78766.03
224606.64	0.00	0	354678.92	0.00	-224606.64
125872.53	0.00	0	348512.00	3420103.09	-125872.53
379856.23	0.00	0	900180.00	19169204.93	-379856.23
1505626.01	0.00	0	29031.00	5515763.34	-1505626.01
554026.99	0.00	0	579285.56	0.00	-554026.99
147543.10	0.00	0	223220.00	16518.36	-147543.10
761507.39	0.00	0	1280036.23	19169204.93	-761507.39
1429051.47	0.00	0	2041543.62	19169204.93	-1429051.47
358831.92	0.00	0	474384.53	3420103.09	-358831.92
367768.40	0.00	0	370763.10	16518.36	-367768.40
209711.11	0.00	0	399214.71	2415.16	-209711.11
583848.46	0.00	0	667778.00	2107778.11	-583848.46
1724887.05	0.00	0	3470595.10	19169204.93	-1724887.05
710544.77	0.00	0	738531.50	16518.36	-710544.77
581294.26	0.00	0	5195482.15	19169204.93	-581294.26

```
# Look at how many errors are explained by a transaction amount greater than
# the initial account balance
not_enough <- formated_nrow(
  orig_errors_df %>% filter(amount > oldBalanceOrig)
  )

message("Nb. of cash-out transactions where 'amount' > initial balance: "
  , not_enough)
```

```
## Nb. of cash-out transactions where 'amount' > initial balance: 456,715
```

```
# Look at the errors which are not explained by the previous explanation
orig_errors_df %>%
  filter(amount < oldBalanceOrig) %>%
  select(-step, -type, -nameOrig, -nameDest, -typeOrig, -typeDest,
         -isFlaggedFraud, -errBalanceDest)
```

```
##       amount oldBalanceOrig newBalanceOrig oldBalanceDest newBalanceDest
## 1   169107.0       575667.5       575667.5       575667.5       22190.99
## 2 10000000.0     19585040.4     19585040.4            0.0           0.00
## 3  9585040.4     19585040.4     19585040.4            0.0           0.00
## 4 10000000.0     10399045.1     10399045.1            0.0           0.00
## 5   399045.1     10399045.1     10399045.1            0.0           0.00
## 6  7316255.0     17316255.1     17316255.1            0.0           0.00
##   isFraud errBalanceOrig
## 1      G0      -169107.0
## 2      F1    -10000000.0
## 3      F1     -9585040.4
## 4      F1    -10000000.0
## 5      F1      -399045.1
## 6      F1     -7316255.1
```

As far as errors on the destination account are concerned, there is no easy nor straightforward explanation, and a significant number of these errors are larger than twice the amount of the transaction in absolute value.

```
# Display the 10 largest error on the destination account
dest_errors_df <- playset_TRANSFER  %>%
       filter(abs(errBalanceDest) > 0.1)

# Display the 10 largest error on the destination account
kable(head(
  dest_errors_df %>%
    select(-type, -step, -isFraud, -isFlaggedFraud,
           -typeDest, -typeOrig, -nameOrig, -nameDest,
           -oldBalanceOrig, -newBalanceOrig, -errBalanceOrig) %>%
    arrange(desc(abs(errBalanceDest)))
  , 10))
```

amount	oldBalanceDest	newBalanceDest	errBalanceDest
23496309	212359450.98	311404901	-75549141
10000000	186123.55	83016435	-72830312
10000000	716484.46	77736657	-67020172
10000000	15110.76	74722956	-64707845
10000000	10186123.55	83016435	-62830312
10000000	87396.39	72491028	-62403632
10000000	4988154.50	76556199	-61568045
21058477	33755978.87	113007240	-58192785
10000000	10716484.46	77736657	-57020172
10000000	124465.46	66834206	-56709741

```
count_twice_amount <- formated_nrow(
  playset_TRANSFER %>%
  filter(abs(errBalanceDest) > 2 * amount)
  )

message("Nb. of transactions where abs(errBalanceDest) > 2*amount: "
  , count_twice_amount)
```

```
## Nb. of transactions where abs(errBalanceDest) > 2*amount: 7,074
```

Finally, let's look more into details at the fraudulent transfer transactions. It turns out that a large proportion of fraudulent transfer transactions have `newBalanceOrig == 0` and `errBalanceDest == amount`. However, not all transfer transactions with these characteristics are fraudulent.

```
# Display total number of fraudulent transfer transactions
count_transfer <- formated_nrow(
  playset_TRANSFER %>% filter(isFraud == "F1")
  )
message("Total number of fraudulent transfer transactions in the dataset: "
  , count_transfer)
```

```
## Total number of fraudulent transfer transactions in the dataset: 3,681
```

```r
# Visualise a few fraudulent transfer transactions
head(
  playset_TRANSFER %>%
    filter(isFraud == "F1") %>%
    select(-step, -type, -typeOrig, -typeDest, -nameOrig,
           -nameDest, -isFlaggedFraud)
  , 10)
```

```
##        amount oldBalanceOrig newBalanceOrig oldBalanceDest newBalanceDest
## 1      181.00         181.00              0              0             0
## 2     2806.00        2806.00              0              0             0
## 3    20128.00       20128.00              0              0             0
## 4  1277212.77     1277212.77              0              0             0
## 5    35063.63       35063.63              0              0             0
## 6    25071.46       25071.46              0              0             0
## 7   963532.14      963532.14              0              0             0
## 8    14949.84       14949.84              0              0             0
## 9    18627.02       18627.02              0              0             0
## 10   10539.37       10539.37              0              0             0
##    isFraud errBalanceOrig errBalanceDest
## 1       F1              0         181.00
## 2       F1              0        2806.00
## 3       F1              0       20128.00
## 4       F1              0     1277212.77
## 5       F1              0       35063.63
## 6       F1              0       25071.46
## 7       F1              0      963532.14
## 8       F1              0       14949.84
## 9       F1              0       18627.02
## 10      F1              0       10539.37
```

```r
# Confirm how many transfer transactions have newBalanceOrig == 0
# and errBalanceDest == amount
fraud_selection <- formated_nrow(
  playset_TRANSFER %>% filter(isFraud == "F1" &
                              newBalanceOrig == 0 &
                              errBalanceDest == amount)
  )
message(
  "Fraudulent transfers with 'newBalanceOrig' and 'errBalanceDest' = 'amount': "
  , fraud_selection
  )
```

```
## Fraudulent transfers with 'newBalanceOrig' and 'errBalanceDest' = 'amount': 3,511
```

```r
# Compare this to the total number of genuine transfer transactions with
# newBalanceOrig == 0
genuine_selection <- formated_nrow(
  playset_TRANSFER %>% filter(isFraud == "G0" &
                              newBalanceOrig == 0 &
                              errBalanceDest == amount)
  )
```

```r
message(
  "Genuine transfer transactions with these characteristics: "
  , genuine_selection
  )
```

```
## Genuine transfer transactions with these characteristics: 86
```

```r
# Do some cleaning
rm(dest_error, orig_error, count_transfer, not_enough, orig_errors_df,
   playset_TRANSFER, dest_errors_df, count_twice_amount, fraud_selection,
   genuine_selection, txt)
```

```
## Warning in rm(dest_error, orig_error, count_transfer, not_enough,
## orig_errors_df, : object 'txt' not found
```

As a conclusion, transactions of type `TRANSFER` only involve agents of type 'client' (C) and none of type 'merchant' (M,) which is in line with the official description of the dataset.

These transactions contain a large number of balance errors on the destination account which cannot be easily explained. All errors on the origin account occurs when `amount` is greater than `oldBalanceOrig`, and therefore `newBalanceOrig` is equal to 0. It is not clear how such transactions can occur in practice, unless the client gives in cash what is missing. This explanation covers all the error cases but 6. In the transactions with the 6 remaining errors, it seems that the new account balance on the origin account is not updated.

As far as fraudulent transfer transactions are concerned, a large proportion has a new balance of the origin account equal to 0, as well as a balance error of the destination account equal to the amount of the transaction. However, some genuine transfer transactions also have these characteristics.

3.2.5 Other fraud patterns

Finally let's verify the last piece of information from the official dataset description. It indicates that fraudsters would start by transferring the fraud amount to a merchant account before cashing-out the money. As the queries below show, there is indeed a pattern of equal amounts for 1 transaction of type `TRANSFER` and 1 transaction of type `CASH-OUT`. However, this is not always the case. In addition, no merchant is ever involved in any of the frauds.

```r
# Filter all fraud transactions
playset_FRAUD <- playset %>% filter(isFraud == "F1")

# Display the number of fraudulent transactions
fraud_count <- formated_nrow(playset_FRAUD)
message("Total number of fraudulent transactions: ", fraud_count)
```

```
## Total number of fraudulent transactions: 7,391
```

```r
# Display a few of them
kable(head(
  playset_FRAUD %>%
    select(-step, -typeOrig, -typeDest, -isFraud, -isFlaggedFraud,
           - nameDest, -oldBalanceDest, -newBalanceDest)
  , 15), caption = "Analysis of fraudulent transactions - Origin side")
```

Table 13: Analysis of fraudulent transactions - Origin side

type	amount	nameOrig	oldBalanceOrig	newBalanceOrig
TRANSFER	181.00	C1305486145	181.00	0
CASH_OUT	181.00	C840083671	181.00	0
TRANSFER	2806.00	C1420196421	2806.00	0
CASH_OUT	2806.00	C2101527076	2806.00	0
TRANSFER	20128.00	C137533655	20128.00	0
CASH_OUT	20128.00	C1118430673	20128.00	0
CASH_OUT	416001.33	C749981943	0.00	0
TRANSFER	1277212.77	C1334405552	1277212.77	0
CASH_OUT	1277212.77	C467632528	1277212.77	0
TRANSFER	35063.63	C1364127192	35063.63	0
CASH_OUT	35063.63	C1635772897	35063.63	0
TRANSFER	25071.46	C669700766	25071.46	0
CASH_OUT	25071.46	C1275464847	25071.46	0
CASH_OUT	132842.64	C13692003	4499.08	0
CASH_OUT	235238.66	C1499825229	235238.66	0

```
kable(head(
  playset_FRAUD %>%
    select(-step, -typeOrig, -typeDest, -isFraud, -isFlaggedFraud,
           - nameOrig, -oldBalanceOrig, -newBalanceOrig)
  , 15), caption = "Analysis of fraudulent transactions - Destination side")
```

Table 14: Analysis of fraudulent transactions - Destination side

type	amount	nameDest	oldBalanceDest	newBalanceDest
TRANSFER	181.00	C553264065	0.00	0.00
CASH_OUT	181.00	C38997010	21182.00	0.00
TRANSFER	2806.00	C972765878	0.00	0.00
CASH_OUT	2806.00	C1007251739	26202.00	0.00
TRANSFER	20128.00	C1848415041	0.00	0.00
CASH_OUT	20128.00	C339924917	6268.00	12145.85
CASH_OUT	416001.33	C667346055	102.00	9291619.62
TRANSFER	1277212.77	C431687661	0.00	0.00
CASH_OUT	1277212.77	C716083600	0.00	2444985.19
TRANSFER	35063.63	C1136419747	0.00	0.00
CASH_OUT	35063.63	C1983025922	31140.00	7550.03
TRANSFER	25071.46	C1384210339	0.00	0.00
CASH_OUT	25071.46	C1364913072	9083.76	34155.22
CASH_OUT	132842.64	C297927961	0.00	132842.64
CASH_OUT	235238.66	C2100440237	0.00	235238.66

```
# Repetition of fraud amount
same_amount_summary <- playset_FRAUD %>%
  group_by(amount) %>%
  summarise(count = n()) %>%
  arrange(desc(count))
```

```r
# Display the most frequent amounts and their respective counts
kable(head(same_amount_summary, 10))
```

amount	count
10000000.00	253
0.00	14
1165187.89	4
429257.45	3
119.00	2
119.65	2
164.00	2
170.00	2
174.92	2
181.00	2

```r
# Display how many times these counts are repeated
kable(table(
  same_amount_summary %>% pull(count)
  ), caption = "Number of times a given count is repeated"
  , col.names = c("count","frequency"))
```

Table 16: Number of times a given count is repeated

count	frequency
1	763
2	3177
3	1
4	1
14	1
253	1

```r
# Do some cleaning
rm(fraud_count, playset_FRAUD, same_amount_summary, fraud_count)
```

A closer look at the amounts of fraudulent transactions reveals that a couple amounts are repeated a lot: 0 and $10,000,000$. For the rest, a lot of these amounts are repeated exactly twice. For amounts which are repeated twice, most of the time, there is a transfer and a cash-out transaction. This confirms the official dataset description. However, a look through all these double transactions shows that almost no destination account matches the origin account in these pairs of transfer/cash-out transactions of same amount (this can be verified on all the fraudulent transactions using additional code, but it is not shown here). This contradicts the official description.

3.3 Feature engineering

3.3.1 Transactions per agent

The `nameDest` and `nameOrig` features can be used to count the number of time a particular agent appear in a transaction. The intuition here is that fraudsters might create a new account just for a fraud and then

close it. Similarly, maybe accounts involved in multiple transactions are not likely to be involved in fraud, as fraudsters might avoid using several time the same account to avoid being identified. The corresponding new features are created and added to the dataset.

```
# Create two new features: countOrig and countDest
playset <- playset %>% group_by(nameOrig) %>% mutate(countOrig = n())
playset <- playset %>% group_by(nameDest) %>% mutate(countDest = n())

validation <- validation %>% group_by(nameOrig) %>% mutate(countOrig = n())
validation <- validation %>% group_by(nameDest) %>% mutate(countDest = n())

# Move the features to a more relevant location in the dataframe
playset <- data.frame(playset %>%
  relocate(countOrig, .after=nameOrig) %>%
  relocate(countDest, .after=nameDest) %>%
  mutate(countOrig = as.integer(countOrig), countDest = as.integer(countDest)))
validation <- data.frame(validation %>%
  relocate(countOrig, .after=nameOrig) %>%
  relocate(countDest, .after=nameDest) %>%
  mutate(countOrig = as.integer(countOrig), countDest = as.integer(countDest)))
```

3.3.2 Account balance error

As the previous analysis has shown, the balance error both on the origin account and destination account turned out to be important. These two features can be added to the dataset as they could be useful to train the ML algorithms.

Because the amount of a transaction is always positive, the way the error is calculated should depend on whether money is moving out (cash-out, debit) or whether money is moving in (cash-in). For this reason different cases need to be considered.

```
# Cash-out, Debit, transfer, payment transactions:
# money moving out of origin account
# Cash-in: money moving in origin account, out of destination account
# use 'ifelse' clause to distinguish the two cases
playset <- playset %>%
  mutate(errBalanceOrig = ifelse(type == "CASH_IN",
                                 oldBalanceOrig + amount - newBalanceOrig,
                                 oldBalanceOrig - amount - newBalanceOrig)) %>%
  mutate(errBalanceDest = ifelse(type == "CASH_IN",
                                 oldBalanceDest - amount - newBalanceDest,
                                 oldBalanceDest + amount - newBalanceDest))

validation <- validation %>%
  mutate(errBalanceOrig = ifelse(type == "CASH_IN",
                                 oldBalanceOrig + amount - newBalanceOrig,
                                 oldBalanceOrig - amount - newBalanceOrig)) %>%
  mutate(errBalanceDest = ifelse(type == "CASH_IN",
                                 oldBalanceDest - amount - newBalanceDest,
                                 oldBalanceDest + amount - newBalanceDest))
```

3.3.3 Feature selection

As the previous analysis has shown, `typeDest` and `typeOrig` do not carry any relevant information, they can be dropped.

Similarly, now that the number of times a given agent appear in transactions is saved in the features `countDest` and `countOrig`, the two features `nameOrig` and `nameDest` do not carry any relevant information. The analysis of fraudulent transactions has shown that there is no obvious link between the accounts of transfer and cash-out fraudulent transactions of same amount. They can be dropped as well.

Finally, `isFlaggedFraud` turned out to not being useful either, it can be dropped.

```
# Dropping uninformative features
playset <- playset %>%
  select(-typeOrig, -typeDest, -nameOrig, -nameDest, -isFlaggedFraud)
validation <- validation %>%
  select(-typeOrig, -typeDest, -nameOrig, -nameDest, -isFlaggedFraud)

# Save the clean playset and validation dataset in directory "data"
# Creates "data" directory in the current working directory if it doesn't exist
ifelse(!dir.exists(file.path("data")), dir.create(file.path("data")), FALSE)
saveRDS(validation, file = "data/validation-clean.rds")
saveRDS(playset, file = "data/play-clean.rds")
```

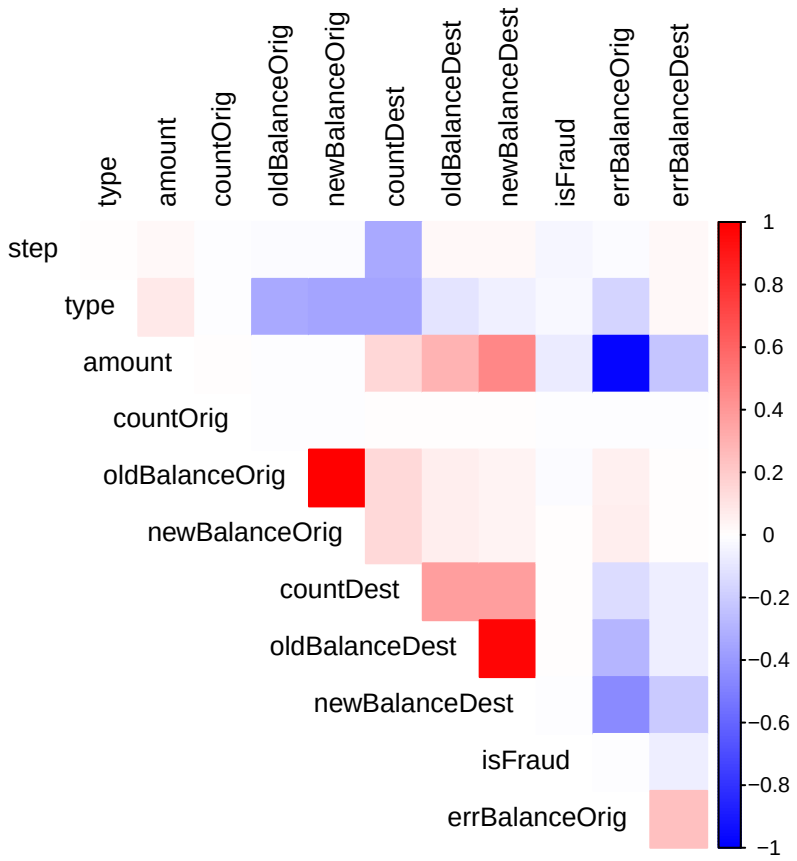
3.4 Feature correlation

In order to finish exploring the dataset, let's look at the correlation matrix between the various features.

```
# Define function to run correlation and plot correlation heatmap
# Takes as argument the dataset as dataframe and the vector of features
# to include in the heatmap (vector of integer)
corr_heat <- function(data_frame, features_vect, add_to = FALSE){
  # Keep from data_frame the features of interest (features_vect)
  # and convert them to integer so cor() can run.

  df_correl <- sapply(
    as.data.frame(data_frame[, features_vect]),
    as.integer)

  # Calculate correlation matrix
  correl_mat <- cor(df_correl)

  # visualize correlation matrix
  corrplot(correl_mat, method = "color", add = add_to,
           col = colorRampPalette(c("blue","white","red"))(200),
           type = "upper", diag = F,
           tl.cex=0.80, tl.col = "black",
           tl.offset = 0.6, cl.cex = 0.7)
}
```

```
# Define what predictors to include
# 1: step, 2: type, 3: amount,
# 4: countOrig, 5: oldBalanceOrig, 6: newBalanceOrig,
# 7: count Dest, 8: oldBalanceDest, 9: newBalanceDest,
```

```
# 10: isFraud, 11: errBalanceOrig, 12: errBalanceDest
#
features_select <- c(1:12)

# Run function corr_heat()
corr_heat(playset, features_select)
```

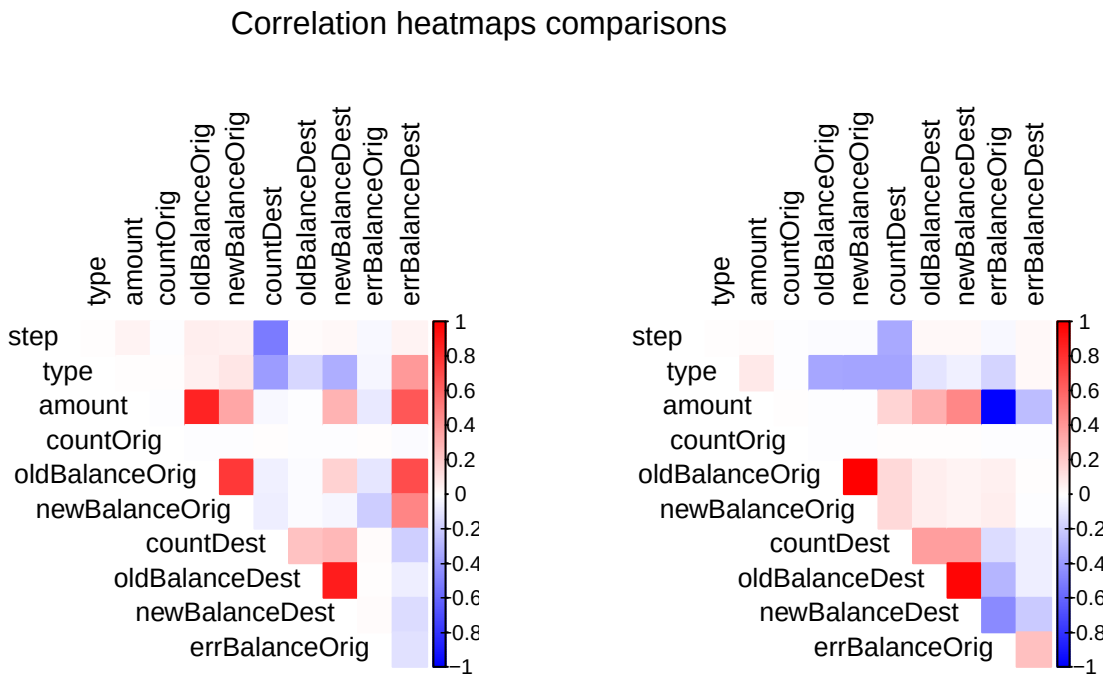


The correlation pattern is interesting: error balance on the origin account is highly correlated with old and new balance of the destination account. It is also strongly correlated with the amount of the transaction. Balance error on the destination accounts is slightly correlated to the transaction amount, to the balance error on the origin accounts and to the new balance of destination accounts. Also, `countOrig` does not appear to be correlated to any feature, but `countDest` is correlated to `step`, `type`, `oldBalanceDest` and `newBalanceDest`. Finally, `isFraud`is only very slightly correlated with `errBalanceDest`, `amount` and `step`, but the amount is so low that it probably does not deserve to be mentioned.

Let's keep these correlations in mind when it is time to train the ML algorithms, as some algorithms do not work well with highly correlated features. Now let's look at the correlation matrix for fraudulent transactions and genuine transactions separately.

```
# Keep only fraudulent transactions
playset_fraud <- playset %>% filter(isFraud == "F1")

# Keep only genuine transactions
playset_genuine <- playset %>% filter(isFraud == "G0")

# Select predictors of interest
```

```r
features_select <- c(1:9, 11, 12)

# Splitting the graph are in two to allow account-by-account comparison
par(mfrow=c(1,2))

# Call corr_heat() on fraudulent transactions
corr_heat(playset_fraud, features_select)

# Call corr_heat() on regular transactions
corr_heat(playset_genuine, features_select)

mtext("Correlation heatmaps comparisons",
      account = 3, line = -2, outer = TRUE, cex = 1)
```



The correlation pattern is different between the two maps, especially as far as the `errBalanceOrig`, `errBalanceDest`, `type` and `amount` are concerned.

In the case of fraudulent transactions (left map), `oldBalanceOrig` is strongly correlated with `amount` which confirms what the previous analysis has shown (for fraudulent transactions `oldBalanceOrig` is often equal to `amount`). Also, `errBalanceDest` is strongly correlated with amount and `oldBalanceOrig` and significantly correlated with `type` and `newBalanceOrig`. This is not the case for non-fraudulent transactions.

4 Modelling

In this section several models are trained and compared. These models are the following:

1. Logistic,
2. Random Forest,
3. Gradient Boosting Machine.

For each of these models, 3 versions are calculated, based on a different set of features:

- The first version, `base`, includes only the non-engineered features: `step`, `type`, `amount`, and all the old and new balances.
- Version 2, `err`, includes all non engineered features plus the balance error `errBalanceOrig` and `errBalanceDest`.
- Version 3, `all`, includes all the features: non engineered features plus balance error features and count features (`countOrig`, `countDest`).

Comparing these three versions makes it possible to assess the contribution of the engineered features to the performance of the different models.

Some additional improvements are also tested:

1. **Re-classification**: As the data analysis section has shown, no fraudulent transaction is of type debit, cash-in or payment. Therefore, any prediction of such a transaction as fraud is likely wrong. Re-classification consists in correcting this mistake, by making sure that no debit, cash-in or payment transaction is classified as fraud.
2. **Classification threshold**: The models used here all return class probabilities. By default, the classifier uses the default threshold of 0.5: the predicted class is the class the probability of which is higher than 0.5. However, with datasets containing highly imbalanced data, changing the threshold to make the classifier more sensitive to the rare class could improve performance. The performance of the different models is tested for various values of the threshold.
3. **Re-sampling and class weights**: In order to further deal with imbalance, weighting classes and re-sampling (SMOTE) is also used in an additional attempt to deal with the strong class imbalance of the dataset.

4.1 Learning set-up

4.1.1 Data sets

Now that all the relevant features have been created or selected, the dataset is final. `playset` can be split between the train and test sets, and these sets are saved for future use. To control that everything is as expected, the number of observations in the new sets is checked.

```
# Set seed for reproducibility
set.seed(1)

# The test set will be 20% of 'playset'
in_test <- createDataPartition(y = playset$isFraud,
                               times = 1, p = 0.2, list = FALSE)
train.set <- playset[-in_test,]
test.set <- playset[in_test,]

# Do some cleaning
rm(in_test, transactions, transactions.raw)
```

- Check the lengths of the different sets

```
# Display the number of observations in all sets for control
message("'train.set' contains ", formated_nrow(train.set), " observations.")
```

```
## 'train.set' contains 4,581,084 observations.
```

```
message("'test.set' contains ", formated_nrow(test.set), " observations.")
```

```
## 'test.set' contains 1,145,273 observations.
```

```
message("'validation' contains ", formated_nrow(validation), " observations.")
```

```
## 'validation' contains 636,263 observations.
```

- Save the clean `train.set` and `test.set`:

```
# Save the clean transactions dataset in directory "data"
# Creates "data" directory in the current directory if it doesn't exist
ifelse(!dir.exists(file.path("data")), dir.create(file.path("data")), FALSE)
saveRDS(train.set, file = "data/train.rds")
saveRDS(test.set, file = "data/test.rds")
```

4.1.2 Cross-validation

Repeated cross-validation is used to make sure that the models are not too sensitive to noise in the data. This is especially important for the random forest algorithms which can easily be overtrained. The number of repeats and folds is set for all models. Since the training set is large (4 million + observations) there is not really a lack of data, but training the models on the entire dataset would take too long. For this reason, the different models are trained on a smaller dataset. Because the smaller dataset is only 200,000 observations large, the models could have been trained on completely different sets of data extracted from the original training set. However, this would increase programming complexity whereas using re-sampling via the cross-validation options of the caret package would simplify programming.

Based on trial and errors, a train set of 200k observations turns out to be a good compromise between quality of model fits and computing time. Because the availability of observations is not the limiting resources (but time is) it is better to optimise the use of time by using a larger dataset, an intermediate number of folds (5) and a small number of repeats (3).

On a 200k observations dataset, 5 folds means that each iteration of model training is based on 160k observations, which should contain on average 160 fraudulent transactions. The `caret::train()` function stratifies the folds[10] to ensure that they all contain a similar amount of minority class observations, so no model is trained on a set containing only a handful fraudulent transactions.

```
# Create smaller sub-set for training to lower execution time
# Set seed for repeatability and comparability
seed_general <- 123
set.seed(seed_general)
train.small <- sample_n(train.set, small_sample_size)

message("'train.small' contains ", formated_nrow(train.small), " observations.")
```

[10] see caret manual, https://topepo.github.io/caret/data-splitting.html

```
## 'train.small' contains 200,000 observations.
```

```
# Set-up k-fold and nb of repeats for repeated cross-validation
k_fold <- 5
n_repeat <- 3
```

4.1.3 Formulas

As mentioned previously, three versions of the different models are tested to assess the contribution of the various features. These three versions are declared here.

```
# Create list of forumla
# Create the 3 different formulas for the 3 versions
form.base <- isFraud ~ .-countOrig -countDest -errBalanceDest -errBalanceOrig
form.err <- isFraud ~ .-countOrig -countDest
form.all <- isFraud ~ .

# Create list with all the formulas
form.list <- c("base" = form.base, "err" = form.err, "all" = form.all)

# Do some cleaning
rm(form.base, form.err, form.all)
```

4.1.4 Parallel computing

In order to reduce execution time and make the most of parallel computing, the package `doParallel` is used. More information about how to use it with caret is available here.[11]

```
# Start parallel computing
# Detect max number of cores and leave 1 out so
# the computer can still work on other things
nb_cores <- detectCores()-2
# Create cluster
parallel_clusters <- makePSOCKcluster(nb_cores)
# Start parallel processing
registerDoParallel(parallel_clusters)
```

4.2 Functions and classes

4.2.1 Classes

In order to structure and clarify the training environment a set of classes is created. These classes are the following:

- **Model** is a class that contains all the details of a particular type of model, including model fits, predictions, performance, and all the parameters which have been used to train the model.
- **LabEnv** is a class that contains all the parameters common to all the models as well as a selection (to save memory space) of the relevant information contained in the Model object which have been created.

[11] https://topepo.github.io/caret/parallel-processing.html

- **LabParam** is a list of parameters required to create a `LabEnv` object. It is created using a function to ensure that no parameters is forgotten. These include seed, training set and testing set common to all models.
- **ModParam** is a list of parameters required to create a `Model` object, generated via a function to ensure that no parameters is forgotten. These include model name, type of prediction for the `caret::predict()` function (`raw`or `prob`), the list of formulas for the different versions of the model, and the list of training arguments to be passed to the `caret::train()` function.

The code for these classes is given below.

```
# Create S3 class "LabParam"
# Just a way to make sure the list structure is as required
LabParam <- function(seed, train_set, test_set){
  # Create LabParam list/object with the proper structure
  object <- list(
    "Seed" = seed,
    "TrainSet" = train_set,
    "TestSet" = test_set)

  # Set class name
  class(object) <- append(class(object), "LabParam")
  return(object)
}

# Create S3 class "ModelParam"
# Just a way to make sure the list structure is as required
ModelParam <- function(model_name, raw_prob, form_list, args_list){
  # Create ModelParam list/object with the proper structure
  object <- list("ModName" = model_name,
                 "RawProb" = raw_prob,
                 "FormulaList" = form_list,
                 "TrainArgs" = args_list)

  # Set class name
  class(object) <- append(class(object), "ModelParam")
  return(object)
}

# Create S3 class "LabEnv"
LabEnv <- function(lab_param){
  # Will contain all the model objects and common parameters
  object <- list()

  object["Params"] <- list(lab_param)

  # Set class name
  class(object) <- append(class(object), "LabEnv")
  return(object)
}

# Create S3 class "Model"
Model <- function(param_list){
  # Function to create S3 class with name "ModelObject"
```

```
  object <- list()

  object["Params"] <- list(param_list)
  object["ModFits"] <- list("Model details not updated yet.")
  object["Preds"] <- list("Predictions not updated yet.")
  object["ExecTimes"] <- list("Execution times not updated yet.")
  object["Perfs"] <- list("Performances not updated yet.")

  # Set the name for the class
  class(object) <- append(class(object), "Model")
  return(object)
}
```

Now that the classes have been introduced a `LabEnv` object is initiated.

```
# Create list of lab parameters using class LabParam.
# Seed, train_set, test_set will be used with all models
# trained in this lab environment.
lab_parameters <- LabParam(seed = seed_general,
                           train_set = train.small,
                           test_set = test.set)

# Create LabEnv object
lab_env <- LabEnv(lab_param = lab_parameters)

# Do some cleaning
rm(lab_parameters, train.small, test.set)
```

4.2.2 Functions

The following functions are used in the next sections. The code for these functions is provided in appendix:

1. **run_model()**, which wraps the `caret::train()` function and returns a Model object including model fit, predictions and execution time information. This is a method for class Model. Takes as argument an empty `Model` object and a `LabEnv` object.
2. **measure_perf()**, which measures the performance of a model. Takes as argument a `Model` object, a `LabEnv` object and a classification threshold.
3. **threshold_effect()**, which calculates the number of false positives and negatives for various values of the classification threshold. This is only possible for models trained with `savePrediction = TRUE` in `caret::train()`.
4. **plot_th_effect()**, which plots the number of false positives and negatives for various values of the classification threshold, using as argument the result of `threshold_effect()`.
5. **get_performance()**, which extracts from the list of results the classification performance of the model passed in argument. Note that execution time and the confusion matrix are not included in this function.
6. **get_exec_time()**, which extracts from the list of results the execution time of the model passed in argument.
7. **get_conf_matrix()**, which extracts the confusion matrix for the model passed in argument form the list of results.

8. **display_performance()**, which uses all the previous functions to extract performance information (including execution time and the confusion matrix) and display it.
9. **preds_to_cm()**, which calculates the confusion matrix from predictions and reference values.
10. **plot_var_imp()**, which draw a bar plot of the variable importance of a model.
11. **get_reclass_FP()**, to calculate the number of false positives which could be reclassified (more details in the next section).
12. **display_best_perf()**, to display the performance of the best version of a model and return the best version.
13. **shrink_model()**, to delete some information from a `Model` object before it is added to the `LabEnv` object in order to save space.
14. **add_mod_to_lab()**, to append a `Model` object to a `LabEnv` object.

4.3 Logistic classifier

Three versions of a logistic regression algorithm are tested.

1. **Logistic regression**: The first model is a standard logistic regression model.
2. **Logistic with reclassification**: This model reclassifies any fraud prediction of transactions of type cash-in, debit or payment. The next two models test the impact on performance of common techniques used to deal with class imbalance: weighting and re-sampling.
3. **Weighted logistic**: In the second version, the model gives more weight to observations of the minority class by imposing a heavier cost on errors made on the minority class. This method is often used when dealing with strongly imbalanced datasets.
4. **Logistic SMOTE**: The third version uses re-sampling to attempt to deal with data imbalance. Several types of re-sampling are available: up-sampling, which consist in randomly replicating observations of the minority class; down sampling, which consists in reducing the number of observations of the majority class; and synthetic minority sampling using the SMOTE method. Only "smote" is detailed here. The other options have also been tested but they do not perform better.

4.3.1 Logistic regression

The first model, which serves as a baseline, is based on logistic regression. As the performance metrics below shows, this classifier misclassifies a lot of fraudulent transactions as genuine (false negatives).

As far as features are concerned, it seems that only error balances improve the performance of the classification. When the counts features are added, the performance decreases. The counts features seem to confuse the algorithm.

```
# Set the general trainControl parameters for the train() function
trc <- trainControl("repeatedcv",
                    number = k_fold,
                    repeats = n_repeat,
                    allowParallel = TRUE,
                    classProbs = TRUE)

# Set model name for data retrieval and storage
model_name <- "Logistic"
```

```r
raw_prob <- "prob"

# Set the arguments list for the train() function.
args_list <- list(
      method = "glm",
      # data = train.small,
      family = "binomial",
      trControl = trc)

# Aggregate all these parameters into a "ModelParam" object
parameters <- ModelParam(model_name = model_name,
                  raw_prob = raw_prob,
                  form_list = form.list,
                  args_list = args_list)

# Initiate a new Model object
model_object <- Model(parameters)

# Update Model object with execution time,
# predictions and model fit
model_object <- run_model(model_object, lab_env)

# Update Model object with performances
model_object <- measure_perf(model_object, lab_env)
```

```r
# Display performance of the different versions
best_version <- display_best_perf(model_object = model_object,
                                  lab_env = lab_env,
                                  comparison_metric = "Accuracy")
```

```
##   Features   PRAUC  Recall Precision Accuracy      F1
## 1     base 0.34031 0.56457   0.59857  0.99895 0.58107
## 2      err 0.28037 0.35091   0.79116  0.99904 0.48618
## 3      all 0.32604 0.39892   0.81044  0.99910 0.53466
## Execution time: 0h 5m 25.1s.
## Best version: 'all' when using 'Accuracy' as comparison metric.
## Confusion matrix of best version:
##           Reference
## Prediction      F1      G0
##         F1     590     138
##         G0     889 1143656
```

```r
# Shrink model to save memory space and add to lab_env for easy tracking
lab_env <- add_mod_to_lab(shrink_model(model_object = model_object,
                                       version_keep = best_version),
                          lab_object = lab_env)
```

4.3.2 Re-classification

The previous results show that a large part of misclassifications are false positives. However, data analysis has shown that in the entire `playset` dataset of 5 million+ transactions, no transaction of type `PAYMENT`, `DEBIT` or `CASH-IN` is fraudulent. From there it can be inferred that any transaction of any one of these types

classified as fraud is *necessarily* a false positive. This model could therefore be improved by forcing any such positive, if there is any, to class G0 (non-fraudulent). But first, how many of the positives are of type `PAYMENT`, `DEBIT` or `CASH-IN`? The function `get_reclass_FP()` in appendix calculates the number of positive susceptible of re-classification.

```
nfp <- get_reclass_FP(model_object = model_object,
                      model_version = best_version,
                      lab_env = lab_env)

message("Number of positives of type 'CASH_IN', 'DEBIT' or 'PAYMENT': ", nfp)
```

```
## Number of positives of type 'CASH_IN', 'DEBIT' or 'PAYMENT': 0
```

```
# Do some cleaning
# saveRDS(model_object, "data/log-mod.rds")
rm(trc, model_name, model_object, args_list,
   raw_prob, parameters, nfp, best_version)
```

As the previous results show, re-classification does not improve the performance of the model.

4.3.3 Weighted logistic

Let's start with weighted classes. The weights are calculated based on the frequency of each class. The positive class is 1000 times less numerous than the negative class, so a weight of 1000 is applied to the positive class.

```
# Set model name for data retrieval and storage
model_name <- "Logistic_weighted"
raw_prob <- "prob"

# Set the general trainControl parameters for the train() function
trc <- trainControl("repeatedcv",
                    number = k_fold,
                    repeats = n_repeat,
                    classProbs = TRUE)

# Define model weights vector in line with the proportion of each class
model_weights <- ifelse(lab_env$Params$TrainSet$isFraud == "F1", 1000, 1)

# Define the arguments list for train()
args_list <- list(
      method = "glm",
      # data = train.small,
      weights = model_weights,
      family = "binomial",
      trControl = trc)

# Aggregate all these parameters into a "ModelParam" object
parameters <- ModelParam(model_name = model_name,
                   raw_prob = raw_prob,
                   form_list = form.list,
                   args_list = args_list)
```

```r
# Initiate a new Model object
model_object <- Model(parameters)

# Update Model object with execution time,
# predictions and model fit
model_object <- run_model(model_object, lab_env)

# Update Model object with performances
model_object <- measure_perf(model_object, lab_env)
```

```r
model_name <- "Logistic_weighted"
# Display performance of the models
best_version <- display_best_perf(model_object = model_object,
                                  lab_env = lab_env,
                                  comparison_metric = "Accuracy")
```

```
##   Features   PRAUC  Recall Precision Accuracy      F1
## 1     base 0.00356 0.99730   0.00357  0.64016 0.00711
## 2      err 0.03499 0.70588   0.04789  0.98150 0.08969
## 3      all 0.00373 0.99256   0.00374  0.65848 0.00745
## Execution time: 0h 4m 40.47s.
## Best version: 'err' when using 'Accuracy' as comparison metric.
## Confusion matrix of best version:
##           Reference
## Prediction      F1      G0
##         F1    1044   20756
##         G0     435 1123038
```

```r
# Shrink model to save memory space and add to lab_env for easy tracking
lab_env <- add_mod_to_lab(shrink_model(model_object = model_object,
                                       version_keep = best_version),
                          lab_object = lab_env)

# Identify positives candidate for re-classification
message("Number of positives of type 'CASH_IN', 'DEBIT' or 'PAYMENT': ",
        get_reclass_FP(model_object = model_object,
                       model_version = best_version,
                       lab_env = lab_env,
                       threshold = 0.5))
```

```
## Number of positives of type 'CASH_IN', 'DEBIT' or 'PAYMENT': 52
```

There is a noticeable improvement in terms of true positives and false negatives. However, this comes at the cost of a significant increase in false positives, and the PRAUC is much lower. The increase in false positives is so large that re-classification would not be sufficient to make this model worth considering.

As the query above shows, some positives could be forced to class "G0", however these do not make up a significant part of all false positives, so this option is not pushed further with this model.

```r
# Back-up Model object for later use
# saveRDS(model_object, "data/log_weighted_mod.rds")
```

```
# Do some cleaning - remove unused object and run garbage collection
rm(trc, model_name, model_object, args_list, raw_prob,
   parameters, model_weights, best_version)
```

As mentioned before, re-sampling and weighting methods are sometimes useful to improve classification when there is a strong imbalance of classes. Could these methods help with our dataset?

4.3.4 SMOTE re-sampling

```
model_name <- "Logistic_SMOTE"
raw_prob <- "prob"

# Set the general trainControl parameters for the train() function
trc <- trainControl("repeatedcv",
                    number = k_fold,
                    repeats = n_repeat,
                    sampling = "smote",
                    classProbs = TRUE)

# Define the arguments list for train()
args_list <- list(
      method = "glm",
      # data = train.small,
      family = "binomial",
      trControl = trc)

# Aggregate all these parameters into a "ModelParam" object
parameters <- ModelParam(model_name = model_name,
                  raw_prob = raw_prob,
                  form_list = form.list,
                  args_list = args_list)

# Initiate a new Model object
model_object <- Model(parameters)

# Update Model object with execution time,
# predictions and model fit
model_object <- run_model(model_object, lab_env)
```

```
## Loading required package: grid

## Registered S3 method overwritten by 'quantmod':
##   method            from
##   as.zoo.data.frame zoo
```

```
# Update Model object with performances
model_object <- measure_perf(model_object, lab_env)
```

As the performance measure and confusion matrix below show, using synthetic sampling (SMOTE) does decrease significantly the number of false negatives, but this comes at the cost of a significant increase of false

positives. The PRAUC is lower than for the initial logistic model, but higher than for the weighted logistic model. This model performs better than weighted logistic, but it is still not good enough. It is interesting to note that the best version of this model is the "base" version, without any additional engineered feature.

It is worth noting that when PRAUC is used to compare models, "base" has the best performance. However, if F1 is used as performance measure, then "all" is the best model. However, in both cases the performance is poor.

```
model_name <- "Logistic_weighted"
# Display performance of the models
best_version <- display_best_perf(model_object,
                                  lab_env,
                                  comparison_metric = "Accuracy")
```

```
##   Features   PRAUC  Recall Precision Accuracy      F1
## 1     base 0.44068 0.93577   0.04331  0.97322 0.08278
## 2      err 0.02573 0.88776   0.02848  0.96075 0.05520
## 3      all 0.03360 0.69844   0.04639  0.98107 0.08699
## Execution time: 0h 2m 50.99s.
## Best version: 'all' when using 'Accuracy' as comparison metric.
## Confusion matrix of best version:
##           Reference
## Prediction      F1      G0
##         F1    1033   21237
##         G0     446 1122557
```

```
# Shrink saved model to save memory space
lab_env <- add_mod_to_lab(shrink_model(model_object = model_object,
                                       version_keep = best_version),
                          lab_object = lab_env)
# Identify an positive candidate for reclassification
message("Number of positives of type 'CASH_IN', 'DEBIT' or 'PAYMENT': ",
        get_reclass_FP(model_object = model_object,
                       model_version = best_version,
                       lab_env = lab_env,
                       threshold = 0.5))
```

```
## Number of positives of type 'CASH_IN', 'DEBIT' or 'PAYMENT': 3
```

In this case again some false positives could be reclassified, but the number is low compared to the total number of false negatives so this would not make a significant difference.

```
# Back-up Model object for later use
# saveRDS(model_object, "data/log_smote_mod.rds")
# Do some cleaning
rm(trc, model_name, model_object, args_list, raw_prob,
   parameters, best_version)
```

4.3.5 Conclusion

In this section four models based on logistic regression have been tested, using accuracy as performance metric:

- Logistic regression,
- Weighted logistic,
- Logistic with SMOTE re-sampling.

For each of these models, re-classification was also tested. None of these models performed well. There is a significant improvement in terms of false negatives when using SMOTE re-sampling, but the high number of false positives make these models not suitable for our needs. These results are not encouraging, so no additional variation of logistic regression is considered (alternative performance metric, other re-sampling option or modifying the classification threshold).

4.4 Random Forests

By relying on an ensemble of trees trained on a randomly selected sub-set of features and observations, random forests should provide better results than the algorithms tested so far.

The `caret::train()` function offers the possibility of choosing the metric to select the best fit during model training. In this section two metrics are tested:

1. The default metric (accuracy)
2. The area under the Precision-Recall curve (PRAUC).

Re-classification is also tested.

4.4.1 Accuracy metric

```
# Set model name
model_name <- "Random_Forest_ACC"
raw_prob <- "prob"

# Set the general trainControl parameters for the train() function
trc <- trainControl("repeatedcv",
                    number = k_fold,
                    repeats = n_repeat,
                    classProbs = TRUE,
                    savePredictions = TRUE,
                    allowParallel = TRUE,
                    verboseIter = TRUE)

tune_grid <- expand.grid(mtry = c(2, 4, 7))

# Define the arguments list for train()
args_list <- list(
      method = "rf",
      tuneGrid = tune_grid,
      # The models have run with up to 500 trees
      # 150 is largely enough
      ntree = 150,
      trControl = trc)
```

```r
# Aggregate all these parameters into a "ModelParam" object
parameters <- ModelParam(model_name = model_name,
                  raw_prob = raw_prob,
                  form_list = form.list,
                  args_list = args_list)

# Initiate a new Model object
model_object <- Model(parameters)

# Update Model object with execution time,
# predictions and model fit
model_object <- run_model(model_object, lab_env)

# Update Model object with performances
model_object <- measure_perf(model_object, lab_env)
```

```r
# Display performance of the models
model_name <- "Random_Forest_ACC"
# Display performance of the models
best_version <- display_best_perf(model_object,
                                  lab_env,
                                  comparison_metric = "Accuracy")
```

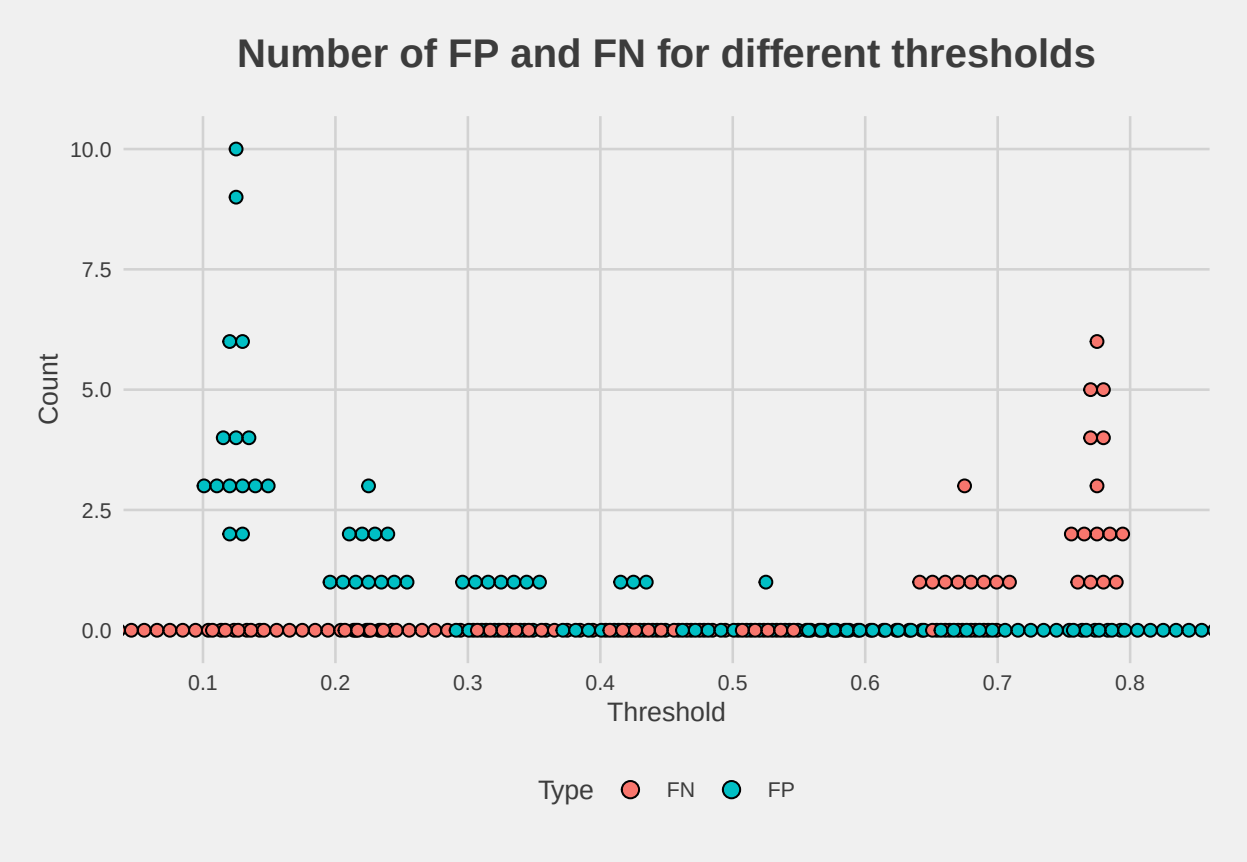
```
##   Features   PRAUC  Recall Precision Accuracy      F1
## 1     base 0.86507 0.74983   0.96017  0.99964 0.84207
## 2      err 0.99597 0.99594   0.99729  0.99999 0.99662
## 3      all 0.99597 0.99594   0.99932  0.99999 0.99763
## Execution time: 0h 57m 24.14s.
## Best version: 'all' when using 'Accuracy' as comparison metric.
## Confusion matrix of best version:
##           Reference
## Prediction      F1      G0
##         F1    1473       1
##         G0       6 1143793
```

```r
# Shrink saved model to save memory space
lab_env <- add_mod_to_lab(shrink_model(model_object = model_object,
                                       version_keep = best_version),
                          lab_object = lab_env)
# Identify an positive candidate for reclassification
message("Number of positives of type 'CASH_IN', 'DEBIT' or 'PAYMENT': ",
        get_reclass_FP(model_object = model_object,
                       model_version = best_version,
                       lab_env = lab_env,
                       threshold = 0.5))
```

```
## Number of positives of type 'CASH_IN', 'DEBIT' or 'PAYMENT': 0
```

```r
# Calcualte the number of FP and FN for various value of the classification
# threhold
th_effect_df <- threshold_effect(model_object = model_object,
                                 lab_env = lab_env,
```

```
                                    model_version = best_version)
# Plot the graph of FN and FP vs. threshold value
plot_th_effect(th_effect_df)
```



This model performs much better than the previous ones. The version including all the features performs the best, when models are compared using accuracy as performance metric. It is interesting to note that the five digits displayed are not enough to distinguish the best model if accuracy is used (value of 0.99999 for both the `err` and `all` versions). The same remark can be made about the PRAUC metric (identical value of 0.99597). However, F1 score is more sensitive to the difference of performance.

The number of positives which could be re-classified is null.

Looking at the dot plot of FP and FN for the various values of the threshold is interesting. This graph is plotted using the results of the 15 models run by the `caret::train()` function (15 models from 5 folds and 3 repeats). In this graph, the FN are plotted on the left side of the threshold value, in red, and the FP are plotted on the right side of the threshold value, in blue.

A threshold of 0.6 would give no FP and no FN on the various folds and repeats of the training set. Let's see if this is also valid when applied to the test set. The query below retrieves the predictions made on the test set, for the best version of the model, in order to calculate the new confusion matrix using the threshold passed as argument.

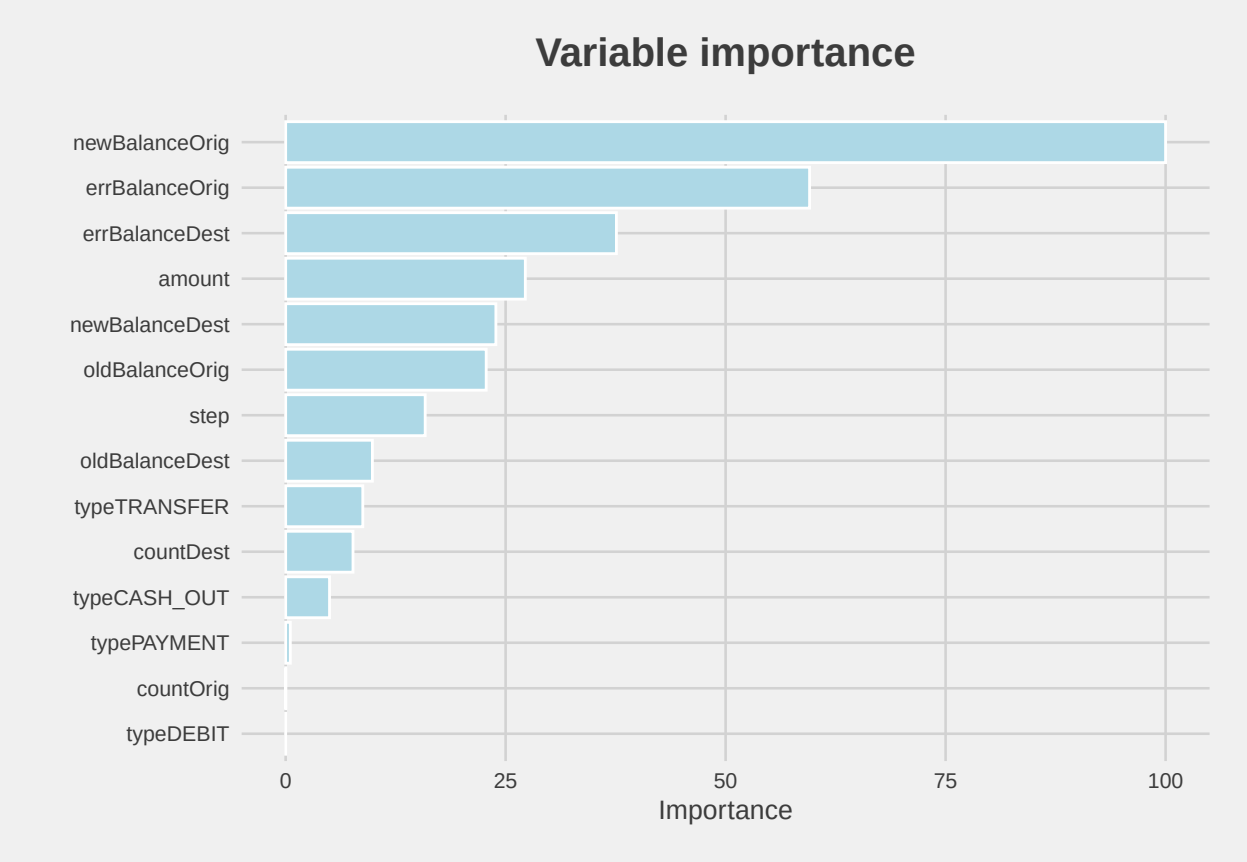
```
preds_to_cm(model_object = model_object,
            model_version = best_version,
            lab_env = lab_env,
            threshold = 0.6)
```

```
##             Reference
## Predictions      F1      G0
##          F1    1469       0
##          G0      10 1143794
```

As the results above show, using a different threshold does not improve the results when applied to the test set. In fact it makes these results worst. The improvement observed is therefore likely due to the variance of the model. For this reason the default threshold of 0.5 is kept.

Random Forests are difficult to interpret, but it is possible to check the relative importance of each feature in the forest (which features have the most predictive power). The graph below shows that the three most important features are `newBalanceOrig` and the two calculated error features, `errBalanceOrig` and `errBalanceDest`. The features `type` (payment, cash-in and debit) and `countOrig` have (almost) no importance.

```
# Use function plot_var_imp() (see code in annex) to draw
# graph of variable importance.
plot_var_imp(model_object = model_object,
             model_version = best_version,
             lab_env = lab_env)
```




```
# Back-up Model object for later use
saveRDS(model_object, "data/rf_acc_mod.rds")

# Do some cleaning
```

```
rm(trc, model_name, model_object, args_list, raw_prob,
   parameters, tune_grid, th_object_df, best_version)
```

The `train()` function has selected the best tuning parameters based on the accuracy performance metric, by default. However, the accuracy metric might not be the best metric to compare models. In the next section the PRAUC metric is used as performance metric to tune the models.

4.4.2 PRAUC metric

As the results below show, using PRAUC as performance metric has an impact on performance, but not for the best. The number of false positives is null. However, the number of false negatives has jumped, which is not the right behaviour for a fraud detection model. Also, it is worth noting that if F1 is used as a performance metric to select which of the three versions is best, then `all` would be selected. It has less false negatives, and still no false positives, which is better for a fraud detection model. The fact that PRAUC gives a higher performance can be explained by the way this metric is estimated, for various values of the classification threshold (PRAUC is a curve, not a point), and not for a single value of the classification threshold, like is the case for F1.

```
# Set model name
model_name <- "Random_Forest_PR"
raw_prob <- "prob"

# Set the general trainControl parameters for the train() function
# This time we specify "prSummary" as summaryFunction to make it
# possible to tune parameters on PRAUC
trc <- trainControl("repeatedcv",
                    number = k_fold,
                    repeats = n_repeat,
                    classProbs = TRUE,
                    savePredictions = TRUE,
                    allowParallel = TRUE,
                    verboseIter = TRUE,
                    summaryFunction = prSummary)

tune_grid <- expand.grid(mtry = c(2, 4, 7))

# Define the arguments list for train()
# This time metric is "AUC" instead of default "Accuracy"
args_list <- list(
      method = "rf",
      tuneGrid = tune_grid,
      ntree = 150,
      metric = "AUC",
      # data = train.small,
      trControl = trc)

# Aggregate all these parameters into a "ModelParam" object
parameters <- ModelParam(model_name = model_name,
                   raw_prob = raw_prob,
                   form_list = form.list,
                   args_list = args_list)
```

```r
# Initiate a new Model object
model_object <- Model(parameters)

# Update Model object with execution time,
# predictions and model fit
model_object <- run_model(model_object, lab_env)

# Update Model object with performances
model_object <- measure_perf(model_object, lab_env)
```

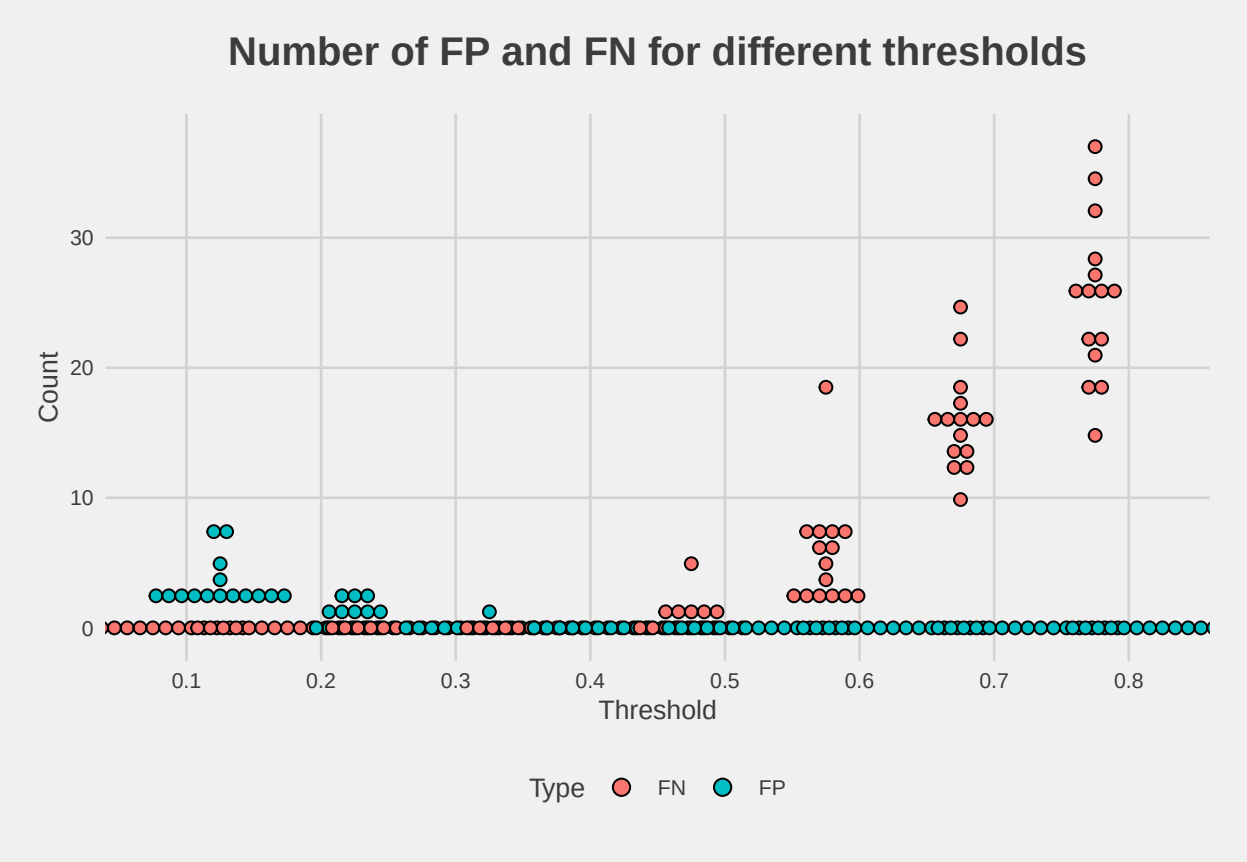
```r
# Display performance of the models
model_name <- "Random_Forest_PR"
# Display performance of the models
best_version <- display_best_perf(model_object,
                                  lab_env,
                                  comparison_metric = "PRAUC")
```

```
##   Features   PRAUC  Recall Precision Accuracy      F1
## 1     base 0.83120 0.61866   0.99782  0.99951 0.76377
## 2      err 0.99597 0.99256   1.00000  0.99999 0.99627
## 3      all 0.99596 0.99256   1.00000  0.99999 0.99627
## Execution time: 1h 7m 1.45s.
## Best version: 'err' when using 'PRAUC' as comparison metric.
## Confusion matrix of best version:
##           Reference
## Prediction      F1      G0
##         F1    1468       0
##         G0      11 1143794
```

```r
# Shrink saved model to save memory space
lab_env <- add_mod_to_lab(shrink_model(model_object = model_object,
                                       version_keep = best_version),
                          lab_object = lab_env)
# Identify an positive candidate for reclassification
message("Number of positives of type 'CASH_IN', 'DEBIT' or 'PAYMENT': ",
        get_reclass_FP(model_object = model_object,
                       model_version = best_version,
                       lab_env = lab_env,
                       threshold = 0.5))
```

```
## Number of positives of type 'CASH_IN', 'DEBIT' or 'PAYMENT': 0
```

```r
# Calculate the number of FP and FN for various value of the classification
# threhold
th_effect_df <- threshold_effect(model_object = model_object,
                                 lab_env = lab_env,
                                 model_version = best_version)
# Plot the graph of FN and FP vs. threshold value
plot_th_effect(th_effect_df)
```



Could the performance be improved using re-classification or by changing the classification threshold? The performance can be improved by modifying the classification threshold to 0.4. Using re-classification does not improve the performance.

As things stand, using PRAUC as performance metric to tune the model decreases the performance of the overall model - especially if one also takes into consideration the possibility of using re-classification - and makes it less suitable for fraud detection (more false negatives). Changing the threshold to 0.4 improves slightly the results on `test.set` but not much.

```
preds_to_cm(model_object = model_object,
            model_version = best_version,
            lab_env = lab_env,
            threshold = 0.4)
```

```
##            Reference
## Predictions      F1      G0
##          F1    1470       0
##          G0       9 1143794
```

```
# Back-up Model object for later use
saveRDS(model_object, "data/rf_pr_mod.rds")

# Do some cleaning
rm(trc, model_name, model_object, args_list, raw_prob,
   parameters, tune_grid, th_object_df, best_version)
```

4.4.3 Conclusion

Random Forest based models perform better than models based on logistic regression. PRAUC is regularly higher than 0.995. Various versions have been tested:

- With accuracy as performance metric to tune the model,
- With PRAUC as performance metric to tune the model,
- With reclassification to avoid predicting a fraudulent transaction when the transaction is of type cash-in, debit or payment,
- Changing the classification threshold to see if it impacts performance.

Random forests do not seem to be impacted by useless features. `countOrig`, `type.CASH_IN`, `type.DEBIT` and `countDest` do not contribute to the performance of the model. Both the `err` and `all` versions provide identical results.

Re-classification does not seem to work for the dataset that have been used here.

Using PRAUC instead of accuracy modifies the performance profile, by decreasing the number of false positives and increasing the number of false negatives. This does not make the model suitable for fraud detection, where false negatives should be minimised.

Overall, the best performing model is the random forest model, `all` version tuned using accuracy as performance metric, with `mtry = 4` and with a classification threshold of 0.5.

4.5 Gradient Boosting Machine

The last algorithm trained and tested is Gradient Boosting Machine (GBM). This algorithm, like Random Forest, uses an ensemble of trees to make predictions. However, instead of making prediction using each individual tree separately, in parallel, GBM works iteratively. The first tree is fitted to the training data, the second builds on the predictions of the first model and focuses on where it performed poorly. This process of boosting is repeated a number of times. Let's see if boosting would help get better classification performance.

4.5.1 Accuracy metric

This algorithm, too, works well. As the results below show, it performs very well in terms of performance measures (PRAUC and F1). Looking at the confusion matrix for the best model, there are a few false negatives and positives.

Execution time, however, is much longer than for the equivalent random forest model Random_Forest_ACC. This is likely due to the larger number of hyperparameters to optimize.

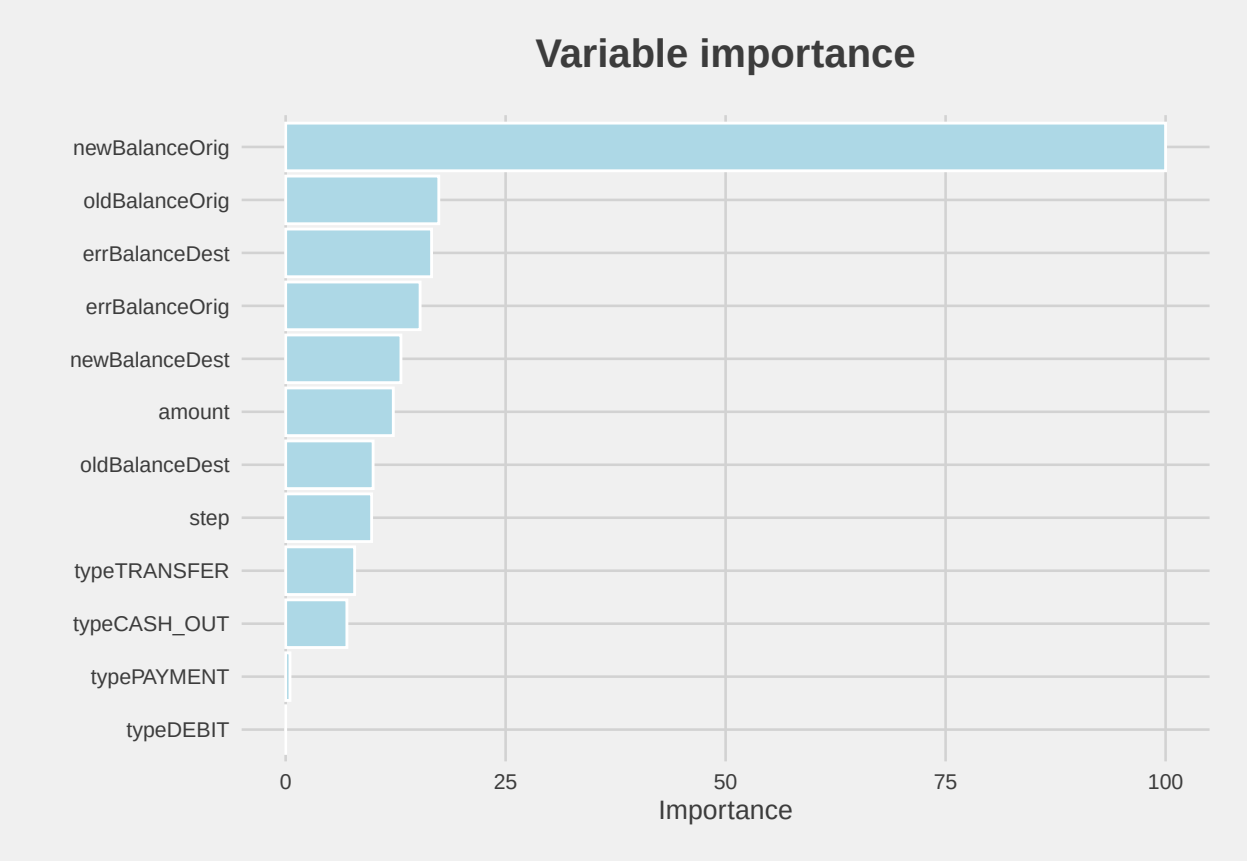
```
# Set model name
model_name <- "GBM_ACC"
raw_prob <- "prob"

# Set the general trainControl parameters for the train() function
# This time we specify "prSummary" as summaryFunction to make it
# possible to tune parameters on PRAUC
trc <- trainControl("repeatedcv",
                    number = k_fold,
                    repeats = n_repeat,
```

```
                    verboseIter = TRUE,
                    savePredictions = TRUE,
                    allowParallel = TRUE,
                    classProbs = TRUE)

# Grid space for the search for best hyperparameters
tune_grid <- expand.grid(nrounds = c(100, 150, 180),
                         max_depth = c(10, 15, 20),
                         colsample_bytree = seq(0.5, 0.9, length.out = 5),
                         eta = 0.1,
                         gamma=0,
                         min_child_weight = 1,
                         subsample = 1
                        )

# Define the arguments list for train()
args_list <- list(
      method = "xgbTree",
      trControl = trc,
      tuneGrid = tune_grid)

# Aggregate all these parameters into a "ModelParam" object
parameters <- ModelParam(model_name = model_name,
                  raw_prob = raw_prob,
                  form_list = form.list,
                  args_list = args_list)

# Initiate a new Model object
model_object <- Model(parameters)

# Update Model object with execution time,
# predictions and model fit
model_object <- run_model(model_object, lab_env)

# Update Model object with performances
model_object <- measure_perf(model_object, lab_env)
```

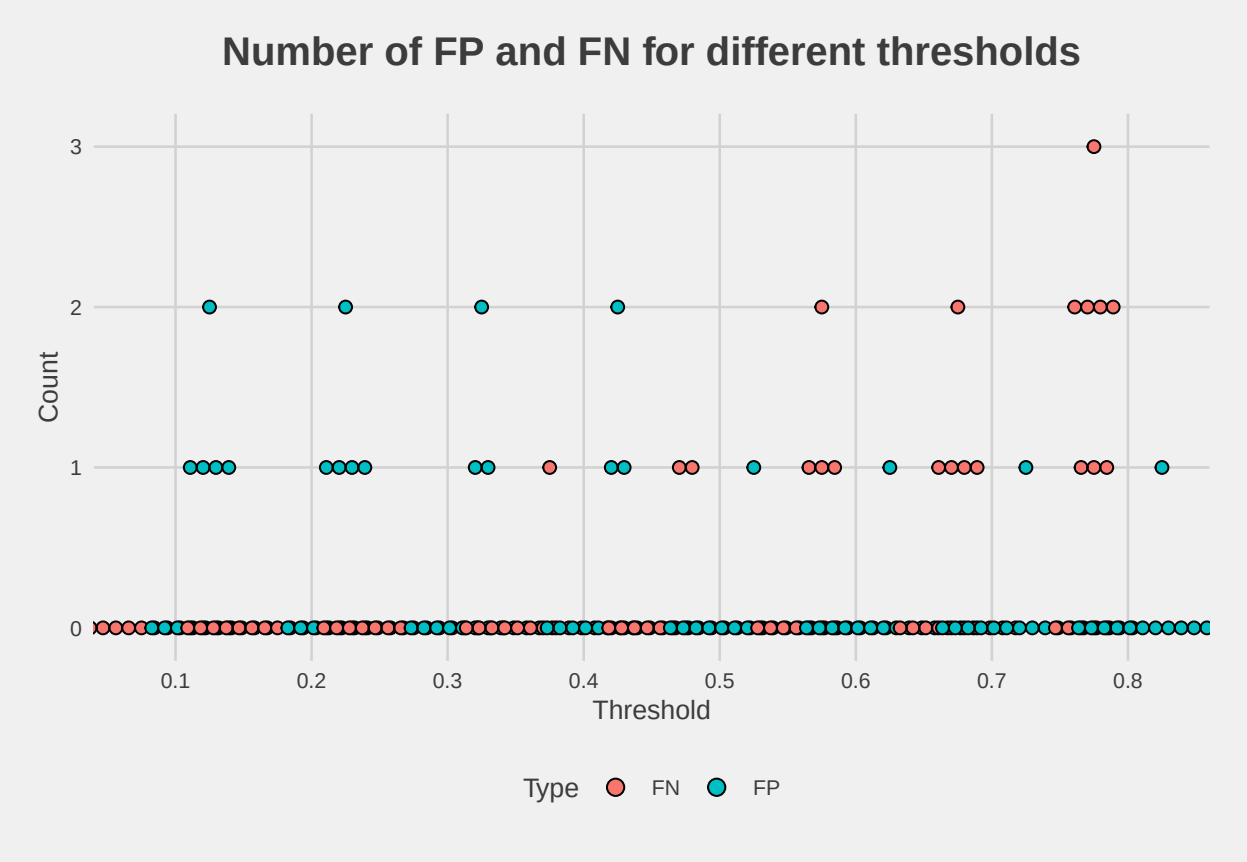
```
# Display performance of the models
best_version <- display_best_perf(model_object,
                                  lab_env,
                                  comparison_metric = "Accuracy")
```

```
##   Features   PRAUC  Recall Precision Accuracy      F1
## 1     base 0.90694 0.75997   0.98510  0.99968 0.85802
## 2      err 0.99598 0.99594   1.00000  0.99999 0.99797
## 3      all 0.99596 0.99594   0.99864  0.99999 0.99729
## Execution time: 1h 47m 33.09s.
## Best version: 'err' when using 'Accuracy' as comparison metric.
## Confusion matrix of best version:
##           Reference
## Prediction      F1      G0
##         F1    1473       0
##         G0       6 1143794
```

```
# Look at variable importance
plot_var_imp(model_object = model_object,
             model_version = best_version,
             lab_env = lab_env)
```




```
# Shrink saved model to save memory space
lab_env <- add_mod_to_lab(shrink_model(model_object = model_object,
                                       version_keep = best_version),
                          lab_object = lab_env)
# Identify an positive candidate for reclassification
message("Number of positives of type 'CASH_IN', 'DEBIT' or 'PAYMENT': ",
        get_reclass_FP(model_object = model_object,
                       model_version = best_version,
                       lab_env = lab_env,
                       threshold = 0.5))

# Calcualte the number of FP and FN for various value of the classification
# threhold
th_effect_df <- threshold_effect(model_object = model_object,
                                 lab_env = lab_env,
                                 model_version = best_version)
# Plot the graph of FN and FP vs. threshold value
plot_th_effect(th_effect_df)
```



As far as variable importance is concerned, `newBalanceOrig` is the most important and `countDest` and `countOrig` are never used. All other features are moderately important.

The model's performance is stable with the threshold, and there is no gain from reclassification. However, the number of FP and FN is low which means that the need for reclassification is low anyway.

The execution time to train the 3 models and run the predictions is about 1.5 times longer than for RF.

Given the importance of detecting all cases of fraud (no FN), the best threshold seems to be 0.3, which would result in 3 FP and 0 FN. However, the gains observed on the training sets do not translate to `test.set`: 2 false positives appear.

```
preds_to_cm(model_object = model_object,
            model_version = best_version,
            lab_env, 0.3)
```

```
##            Reference
## Predictions      F1      G0
##          F1    1473       2
##          G0       6 1143792
```

4.5.2 PRAUC metric

Would xgBoost perform better if the PRAUC metric is used to select the best tuning parameters? As the results below shows, the PRAUC is slightly improved when this metric is used to fit the model. However, the confusion matrix shows that this comes at a cost of more false negatives. It is worth noting that the F1 performance value decreases slightly, reflecting the increase of false negatives.

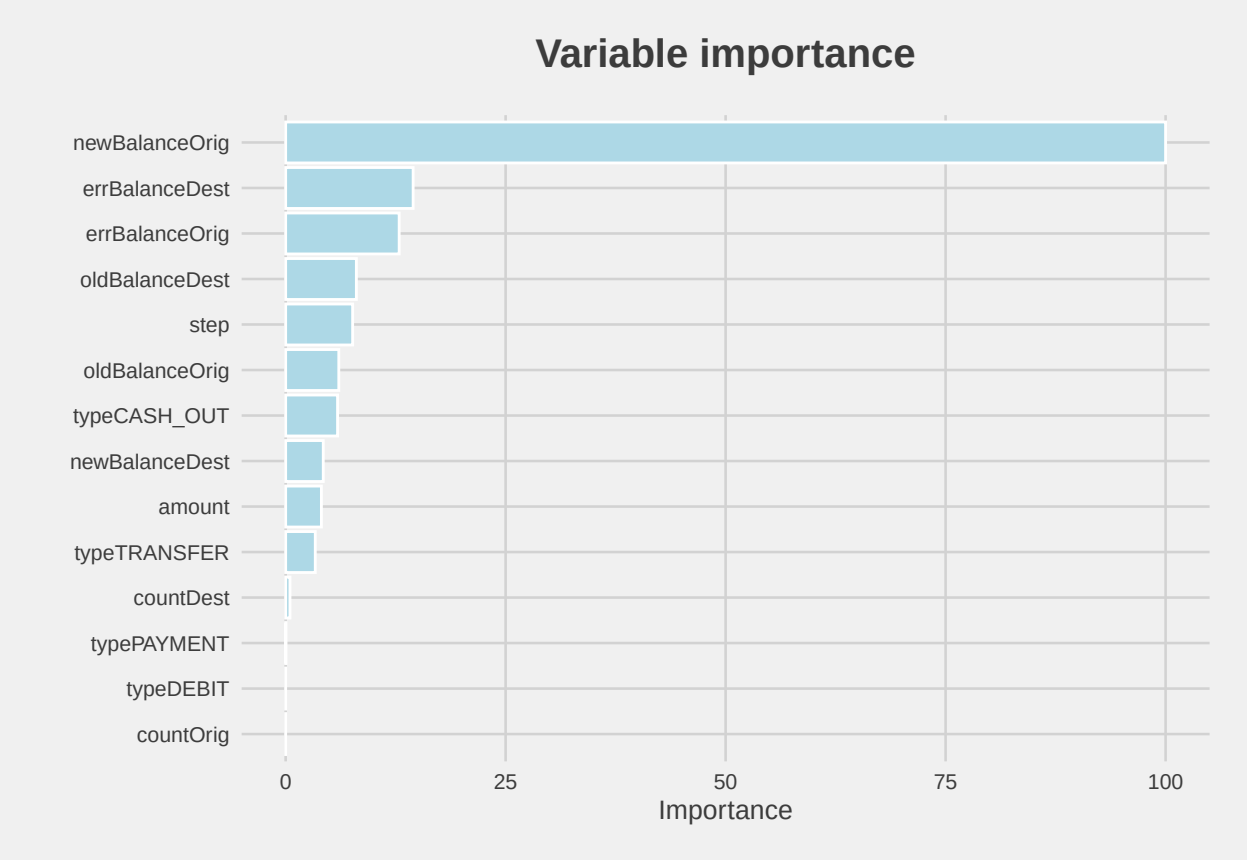
```r
# Set model name
model_name <- "GBM_PR"
raw_prob <- "prob"

# Set the general trainControl parameters for the train() function
# This time we specify "prSummary" as summaryFunction to make it
# possible to tune parameters on PRAUC
trc <- trainControl("repeatedcv",
                    number = k_fold,
                    repeats = n_repeat,
                    savePredictions = TRUE,
                    allowParallel = TRUE,
                    classProbs = TRUE,
                    verboseIter = TRUE,
                    summaryFunction = prSummary)

# Grid space for the search for best hyperparameters
tune_grid <- expand.grid(nrounds = c(100, 150, 180),
                         max_depth = c(10, 15, 20),
                         colsample_bytree = seq(0.5, 0.9, length.out = 5),
                         eta = 0.1,
                         gamma=0,
                         min_child_weight = 1,
                         subsample = 1
                        )

# Define the arguments list for train()
# This time metric is "AUC" instead of default "Accuracy"
args_list <- list(
      method = "xgbTree",
      metric = "AUC",
      trControl = trc,
      tuneGrid = tune_grid)

# Aggregate all these parameters into a "ModelParam" object
parameters <- ModelParam(model_name = model_name,
                    raw_prob = raw_prob,
                    form_list = form.list,
                    args_list = args_list)

# Initiate a new Model object
model_object <- Model(parameters)

# Update Model object with execution time,
# predictions and model fit
model_object <- run_model(model_object, lab_env)

# Update Model object with performances
model_object <- measure_perf(model_object, lab_env)
```

```r
# Display performance of the models
best_version <- display_best_perf(model_object,
                                  lab_env,
```

```
                                    comparison_metric = "PRAUC")
```

```
##   Features   PRAUC  Recall Precision Accuracy      F1
## 1     base 0.91031 0.77011   0.97684  0.99968 0.86125
## 2      err 0.99597 0.99594   0.99864  0.99999 0.99729
## 3      all 0.99597 0.99594   0.99864  0.99999 0.99729
## Execution time: 2h 4m 13.06s.
## Best version: 'all' when using 'PRAUC' as comparison metric.
## Confusion matrix of best version:
##           Reference
## Prediction      F1      G0
##         F1    1473       2
##         G0       6 1143792
```

```
# Look at variable importance
plot_var_imp(model_object = model_object,
             model_version = best_version,
             lab_env = lab_env)
```

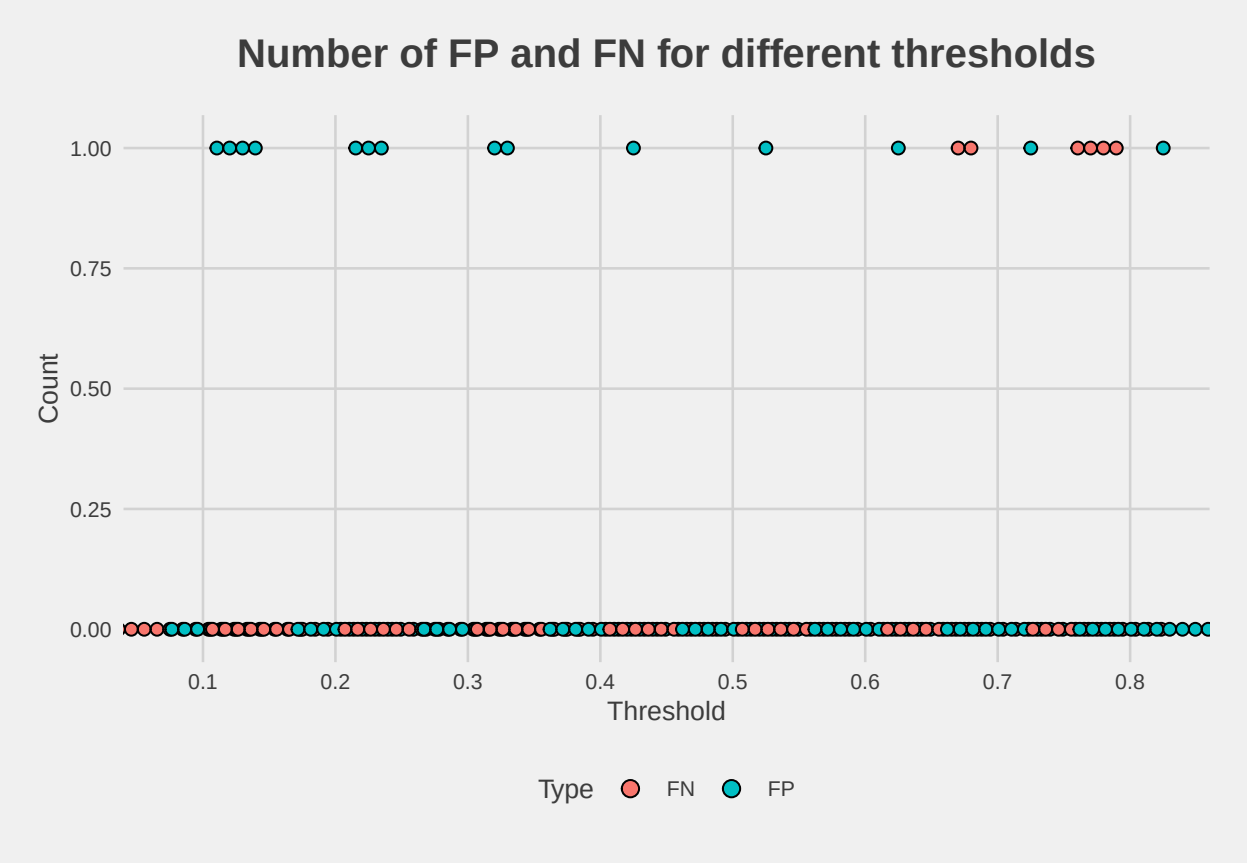



```
# Shrink saved model to save memory space
lab_env <- add_mod_to_lab(shrink_model(model_object = model_object,
                                       version_keep = best_version),
                          lab_object = lab_env)
# Identify an positive candidate for reclassification
```

```
message("Number of positives of type 'CASH_IN', 'DEBIT' or 'PAYMENT': ",
        get_reclass_FP(model_object = model_object,
                       model_version = best_version,
                       lab_env = lab_env,
                       threshold = 0.5))
```

```
## Number of positives of type 'CASH_IN', 'DEBIT' or 'PAYMENT': 2
```

```
# Calcualte the number of FP and FN for various value of the classification
# threhold
th_effect_df <- threshold_effect(model_object = model_object,
                                 lab_env = lab_env,
                                 model_version = best_version)
# Plot the graph of FN and FP vs. threshold value
plot_th_effect(th_effect_df)
```



It is interesting to notice that the confusion matrix on the test set shows 2 false positives for GBM_PR, but none with GBM_ACC. However, the plot of the numbers of FN and FP as a function of the threshold seems to show opposite results: there are less FN and FP with GBM_PR than with GBM_ACC. Based on this plot, the best threshold seems to be between 0.4 and 0.5. Let's see if the gains apply to `test.set`. It does not seem to be the case, as the confusion matrix remains the same. All this indicates that these observations are due to model variance.

```
preds_to_cm(model_object = model_object,
            lab_env = lab_env,
            model_version = best_version,
            threshold = 0.4)
```

```
##            Reference
## Predictions      F1      G0
##          F1    1473       2
##          G0       6 1143792
```

```
#{r, warning=FALSE, cache=FALSE, eval=models_rerun}
# Back-up Model object for later use
saveRDS(model_object, "data/gbm_pr_mod.rds")

# Do some cleaning
rm(trc, model_name, model_object, args_list, raw_prob,
   parameters, tune_grid, th_object_df, best_version)
```

4.5.3 Conclusion

The performance achieved with GBM is similar to the performance achieved with random forests. PRAUC is regularly higher than 0.995. However, computational time is significantly larger (between 1.5 to 2 times longer). Various versions have been tested:

- With accuracy as performance metric to tune the model,
- With PRAUC as performance metric to tune the model,
- Adding reclassification to avoid predicting a fraudulent transaction when the transaction is of type cash-in, debit or payment,
- Changing the classification threshold to see if it impacts performance.

There seem to be room for a slight improvement by adjusting the classification threshold to further improve performance, but this improvement doesn't transfer to the test set. This is likely due to the fact that the difference is too small, and probably due to the variance of the model.

GBM does not seem to be impacted by useless features. `countOrig`, `type.CASH_IN`, `type.DEBIT` and `type.PAYMENT` do not contribute to the performance of the model.

The best GBM model is GBM_ACC (i.e. using the default accuracy metric) with a threshold of 0.5, `err` version, and the following tuning parameters:`max_depth = 20`, `nrounds = 20`, `colsample_bytree = 0.8`. It performs slightly better than the best random forest model.

Finally, the longer computational time can be largely explained by the high number of hyperparameters which were tested, compared to random forests. When GBM is run with only 1 set of hyperparameters, computational time is largely reduced.

```
# Stop parallel computing
stopCluster(parallel_clusters)
```

5 Results and conclusion

5.1 Summary

5.1.1 Models and algorithms

In the previous section, three different algorithms have been trained and tested. For some of these algorithms, different options were tested.

1. **Logistic classification**:

- Standard logistic regression,
- Logistic regression with class weights,
- Logistic regression with SMOTE re-sampling,
- Logistic regression with re-classification.

2. **Random Forest**:

- Using the accuracy metric,
- Using the PRAUC metric,
- For each of these models re-classification and threshold modification has been tested.

3. **Gradient Boosting Machine**:

- Using the accuracy metric,
- Using the PRAUC metric,
- For each of these models re-classification and threshold modification has been tested.

For each of these algorithms, 3 different sets of features were used:

1. `base`: the base set of features (`step`, `type`, `amount` and balance features),
2. `err`: the base set of features and the `errBalanceOrig` and `errBalanceDest` features which contains the balance error amounts on the origin and destination accounts respectively,
3. `all`: all the previous features (base and `errBalanceOrig` and `errBalanceDest`) plus the `countDest` and `countOrig` features which counts how many time a user or agent appears as originator or destination of a transaction, respectively.

5.1.2 Results

Class weights and **SMOTE re-sampling** have been used in an attempt to compensate for the strong class imbalance of the dataset. However, it did not improve enough the performance of the logistic regression models to make them useful. They were also applied to the other models but did not show any positive impact so they were left out of this document.

Re-classification was designed to leverage the initial data analysis, which shown that fraudulent transactions can only be cash-out or transfer transactions. It consists in identifying all the predictions of fraud for observations which are payments, cash-in or debits and re-classify these predictions as genuine. This works well for the logistic model but it is not enough to make it reach the level of performance of random forests and GBM based models. It also works to some extent to reduce the number of false positives of the other random forests and GBM models, but the impact is small and inconsistent.

The random forests and GBM models were run using two different performance metrics: accuracy and PRAUC. PRAUC was expected to perform better as is it usually preferred for imbalanced data. However, this has not been observed, and the models trained using the accuracy metric seem to perform slightly better.

Finally, changing the **threshold** seems to slightly impact the performance, but it is not clear if this change is due to anything other than model variance. For this reason it is left unchanged, at the default value of 0.5.

In most cases, the best predictions were obtained with the `err` and `all` sets of features. In only two cases, when using SMOTE re-sampling and class weights on the logistic classification was the classification performance better with the `base` set of features.

5.1.3 Best models

All in all, the **best models** are:

- **Random forest with all the features**, using the **accuracy metric** and `mtry = 4`, `ntree = 150`. The accuracy achieved is 0.99999, PRAUC is 0.99597 and the F1-score is 0.99763.
- **Gradient Boosting Machine, `err` version**, using the **accuracy metric** and `max_depth = 20`, `nrounds = 20`, `colsample_bytree = 0.8`. The accuracy achieved is 0.99999, PRAUC is 0.99598 and F1-score is 0.99797.

Ideally, the models should be trained with the entire dataset available, in order not to lose any information. Because of computational limitations, the models were only trained on a smaller sub-set of 200,000 observations and repeated k-fold cross-validation (`k = 5`, `repeats = 3`).

Additional attempts were made on the training and test sets, with `savePredictions = FALSE`, `classProbs = FALSE`, to limit memory use, and with a limited number of folds and repeats, for the same reason. All the tuning parameters where set at the best values identified in the previous section. The training sets were made of 500,000, 1 million and 2 million observations.

With **Random Forest**, using a dataset larger than 500,000 observations is not possible because of memory limitations. With **Gradient Boosting Machine**, it is possible to use up to 2 million observations with repeated cross-validation. Also, once the hyperparameters are fixe, the `xgBoost` algorithm (GBM) runs much faster than the implementation of random forest used here (`randomForest` method from the `randomForest` package)d. In both cases there is no improvement of performance.

Since the GBM model performs slightly better and is faster, this model is selected as final model.

5.2 Validation

Table 17: Confusion matrix validation set

	F1	G0
F1	819	0
G0	3	635441

```
##        F1
## 0.9981718
```

The F1-score measured on the validation set (0.99817) is higher than the F1-score measured on the test set during the development of the model. This is a good sign, showing that there has been no over-training. The fraud detection algorithm works well, with few wrong classifications. On the validation set, there are no false positive and only 3 false negatives, out of 822 fraudulent transactions, or slightly **less than 0.4% of false negatives**.

5.3 Conclusion

Three different types of models have been tested to solve this fraud classification problem: logistic regression, random forests and gradient boosting machines. Logistic regression performed poorly, but random forests and GBM performed well. The performance of these two types of models are similar, with GBM being slightly better. The final performance achieved is 0.9981 (F1-score) on the validation set, with an observed false negatives rate of less than 0.4% and no false positives.

In fine only a small amount of the data available in the dataset has been used. Additional work could be done to use the dataset better. This could help to better measure the variance of the model as well as estimate if the performance could be improved further. Such work could consist of:

1. Re-running the same algorithms and code changing only the seed (`seed_general`) to get a better sense of how stable these results are when data is changed (model variance).
2. If GBM is confirmed as the best algorithm and if the hyperparameters are confirmed, then this set of hyperparameters could be applied using GBM and k-fold cross-validation (with k = 4 and repeats = 4 for example) on a set of 1 million observations, repeated T times (up to five times) to make sure that all the observations of the dataset are used. Then a final model can be built, as an ensemble of these T new models. The classification would be made based on majority voting. Alternatively, if the time required to run predictions using these T models is too long, then the model whose performance is closer to the mean of this population of T models can be selected as final model.
3. Another option is to use the entire dataset and divide it into blocks of 200,000, and the GBM algorithm could be run repeatedly on all the blocks, with the code used here. Then, again, the final model could either be an ensemble of models, or the model whose performance is closer to the mean of the population of models.
4. Finally, since training the models is much slowlier than calculating predictions, whenever possible the models could be tested on all the remaining observations (validation set excepted). This would offer a better sense of the performance of the models.

Appendix

Function definitions

```r
# Function pr_auc() to return the area under the precision-recall curve
pr_auc <- function(true_classes, pred_pc, score_type, include_curve = FALSE){
  # Depending on pred_type ("raw" or "prob"), pred_pc is either probabilities or classes
  # true_classes are the classes of the observations, "G0" or "F1"
  # scores.weights0 must be numerical values (1 for positive class, 0 for other)
  if(score_type == "prob"){
    # Here pred_pc are the probabilities for the positive class (classification scores)
    # given by the classifier
    PRAUC <- pr.curve(scores.class0 = pred_pc$F1
                      , weights.class0 = ifelse(true_classes == "F1", 1, 0)
                      , curve = include_curve)
  }
  if(score_type == "raw"){
    # Here pred_pc is the class prediction for all classes
    PRAUC <- pr.curve(scores.class0 = ifelse(pred_pc == "F1", 1, 0)
                      , weights.class0 = ifelse(true_classes == "F1", 1, 0)
                      , curve = include_curve)
  }
  return(PRAUC)
}


# Function to try each type of model on the formula list, based on caret::train()
# formula_list must be a list of formula with names to work properly
# pred_type can be "prob" or "raw"
run_model <- function(model_object, lab_object)
{
  # print("Calling the base run_model function")
  UseMethod("run_model", model_object)
}


run_model.default <- function(model_object, lab_object)
{
  print("Wrong object type.")
}


run_model.Model <- function(model_object, lab_object){
  # Initiate the three lists which will receive the fits,
  # predictions and performance
  fit.list <- list()
  pred.list <- list()
  time.list <- list()

  # Get parameters from model_object
  formula_list <- model_object$Params$FormulaList
  args_caret_train <- model_object$Params$TrainArgs
  pred_type <- model_object$Params$RawProb

  # Get general parameters from lab_object
  data_test <- lab_object$Params$TestSet
  data_train <- lab_object$Params$TrainSet
```

```
  set.seed(lab_object$Params$Seed)

  # Set the training dataset and fit the model
  # 1) Without engineered features
  # 2) With balance error features only
  # 3) With all features
  tic("Train 3 models")
  for (i in 1:length(formula_list)){
    # Update args_caret_train with formula and data to run model
    args_train <- c(list("form" = as.formula(formula_list[[i]]))
                    , "data" = list(data_train), args_caret_train)

    # In order to allow running caret::train() on different lists of arguments
    # the base::do.call() function is used.
    # Call train on list of arguments args_caret_train
    fit <- do.call(train, args_train)

    # Update list of fits with training results
    fit.list[names(formula_list)[i]] <- list(fit)
  }
  time_to_train <- toc()
  time.list["TrainTime"] <- list(
    c("msg" = list(time_to_train$msg)
      , "time.sec" = list(time_to_train$toc-time_to_train$tic))
    )

  # Run the 3 models on the test dataset to get predictions
  # Request probabilities with type = "prob" to make it possible to estimate PRAUC
  # Measure exectuation time and update time.list with elapsed time
  tic("Predict 3 models")
  for (i in 1:length(formula_list)){
    pred.list[names(formula_list)[i]] <- list(
      predict.train(fit.list[[i]], data_test, type = pred_type)
      )
  }
  time_to_predict <- toc()
  time.list["PredTime"] <- list(
    c("msg" = list(time_to_predict$msg)
      , "time.sec" = list(time_to_predict$toc-time_to_predict$tic))
    )

  # Update model_object with new attributes
  model_object["ModFits"] <- list(fit.list)
  model_object["Preds"] <- list(pred.list)
  model_object["ExecTimes"] <- list(time.list)

  # Return all information as a list
  return (model_object)
}


add_mod_to_lab <- function(model_object, lab_object){
  lab_object[[model_object$Params$ModName]] <- model_object
```

```r
  return(lab_object)
}


# Define function measure_perf() that returns PR-AUC and the confusion matrix
# of a given model a list of formulas. Performance is measured using the predictions
# calculated based on the test set. Takes into account a classification threshold when
# predictions are probabilities (as opposed to classes)
measure_perf <- function(model_object, lab_object, threshold = 0.5){

  # Get parameters from model_object
  model_name <- model_object$Params$ModName
  formula_list <- model_object$Params$FormulaList
  pred_type <- model_object$Params$RawProb

  # Get parameters from lab_object
  data_test <- lab_object$Params$TestSet

  # Initiate performance list
  perf.list <- list()

  # Measure performance and update the performance list for each of
  # the three model versions
  for (i in 1:length(formula_list)){
    # 1) Calculate PRAUC performance measure
    PRAUC <- pr_auc(true_classes = data_test$isFraud
                    , pred_pc = model_object$Preds[[i]]
                    , score_type = pred_type
                    , include_curve = TRUE)

    # 2) Depending on pred_type, measure performance using either
    # probabilities or class
    if(pred_type == "prob"){
      # Confusion Matrix is calculated from probabilities with the threshold
      # passed as argument
      cm <- confusionMatrix(
        factor(ifelse(model_object$Preds[[i]]$F1 > threshold
                      , "F1"
                      , "G0")
               , levels = c("F1", "G0")
               , labels = c("F1", "G0"))
        , data_test$isFraud
        , positive = "F1"
        )
      Threshold <- threshold
    } else if(pred_type == "raw"){
      # Confusion Matrix is calculated from predicted classes
      cm <- confusionMatrix(model_object$Preds[[i]]
                            , data_test$isFraud
                            , positive = "F1")
      Threshold <- "None (class)"
    } else {
      warning("Error: only 'raw' or 'prob' possible.")
```

```
    }
    # Update perf.list with performance for formula i
    perf.list[names(formula_list)[i]] <- list(
      c("PRAUC" = list(PRAUC)
        , "ConfusionMatrix" = list(cm)
        , "Threshold" = list(Threshold))
      )
  }
  # Update model_object with performance
  model_object["Perfs"] <- list(perf.list)

  # Return model_object updated with performance
  return (model_object)
}


# Function to display variable importance
# Returns a ggplot graph object
plot_var_imp <- function(model_object,
                         model_version,
                         lab_env){
  var_imp <- cbind(
    Variable = row.names(varImp(model_object$ModFits[[model_version]])[[1]])
    , varImp(model_object$ModFits[[model_version]])[[1]]
    ) %>%
    arrange(desc(Overall))

  row.names(var_imp) <- NULL

  # Set the ggplot object
  g <- var_imp %>%
    ggplot(aes(x = Overall, y = reorder(Variable, Overall))) +
    geom_bar(color = "white", fill="lightblue", stat = "identity") +
    xlab("Importance") +
    ylab("") +
    ggtitle("Variable importance")
  return(g)
}


# Return the performance of a model when different thresholds are used for
# classification. savePredictions=TRUE is required in trainControl() when
# training the models since this function re-uses the probabilities computed
# by train() to save time. These probabilities are saved under in
# "ModFits[[model_version]]$pred" in the Model object
threshold_effect <- function(model_object,
                             model_version,
                             lab_env,
                             th.lim = c(0.1, 0.9)){

  # Get variables from Model and LabEnv objects:
  model_method = model_object$Params$TrainArgs$method
  nb_repeats <- model_object$Params$TrainArgs$trControl$repeats
```

```r
nb_folds <- model_object$Params$TrainArgs$trControl$number

threshold <- seq(from = th.lim[1], to = th.lim[2], by = 0.1)
df <- data.frame("Threshold" = numeric()
                 , "Type" = factor(levels=c("FP", "FN"))
                 , "Count" = integer()
                 , "Rep_ID" = integer())

# Extract the probabilities for the model and version for all
# nb_fold and nb_repeats of the cross-validation process
# Select tunning parameters of bestTune

best_tune <- model_object$ModFits[[model_version]]$bestTune

if(model_method == "rf"){
  best_mtry <- best_tune$mtry
  preds_df <- model_object$ModFits[[model_version]]$pred %>%
    filter(mtry == best_mtry)
} else if(model_method == "xgbTree"){
  best_nrounds <- best_tune$nrounds
  best_maxdepth <- best_tune$max_depth
  best_eta <- best_tune$eta
  best_gamma <- best_tune$gamma
  best_colsample <- best_tune$colsample_bytree
  best_minchildweight <- best_tune$min_child_weight
  best_subsample <- best_tune$subsample

  preds_df <- model_object$ModFits[[model_version]]$pred %>%
    filter(nrounds == best_nrounds
           & max_depth == best_maxdepth
           & eta == best_eta
           & gamma == best_gamma
           & colsample_bytree == best_colsample
           & min_child_weight == best_minchildweight
           & subsample == best_subsample)
} else {
  warning("No acceptable model method found.")
}

# Create predictions based on probabilities and thresholds for all thresholds
for(th in threshold){
  rep_ID <- 0
  # Loop through each rep of nb_repeats
  for(rep in seq(1, nb_repeats, 1)){
    rep_ID <- rep_ID + 1
    # Loop through each fold of nb_folds
    for(fold in seq(1, nb_folds, 1)){
      name_resample <- paste("Fold", fold, ".Rep", rep, sep="")
      # Extract probabilities
      F1_prob <- (preds_df %>% filter(Resample == name_resample))$F1
      # Extract observations
      obs <- (preds_df %>% filter(Resample == name_resample))$obs
      # Make predictions based on threshold value and save as factor
```

```
        preds_th <- factor(ifelse(F1_prob > th, "F1", "GO")
                           , levels = c("F1", "GO")
                           , labels = c("F1", "GO"))
        # Binds observations and predictions and compute false positives
        # and false negatives
        # When using binding, factors are lost, but factor indexes remain
        bind <- data.frame(cbind(preds_th, obs))
        # Save false positives (fp)
        fp <- nrow(bind %>% filter(preds_th == 1 & obs == 2))
        df <- df %>% add_row("Threshold" = th
                             , "Type" = "FP"
                             , "Count" = fp
                             , "Rep_ID" = rep_ID)
        # save false negatives (fn)
        fn <- nrow(bind %>% filter(preds_th == 2 & obs == 1))
        df <- df %>% add_row("Threshold" = th
                             , "Type" = "FN"
                             , "Count" = fn
                             , "Rep_ID" = rep_ID)
      }
    }
  }
  return(df)
}


# Plot threshold effect
plot_th_effect <- function(dataframe, threshold.lim = c(0.1, 0.8)){

  df <- dataframe %>%
    filter((Threshold >= threshold.lim[1])
           & (Threshold <= threshold.lim[2])) %>%
    mutate(Threshold = factor(Threshold))

  max_count <- max(df$Count)

  df %>% ggplot(aes(x = Threshold, y = Count, fill = Type)) +
    geom_dotplot(binaxis='y'
                 , method = "histodot"
                 , stackdir='center'
                 , binwidth = max_count/30
                 , dotsize= 0.8
                 , position=position_dodge(1)
                 , alpha = 1) +
    xlab("Threshold") +
    ylab("Count") +
    ggtitle("Number of FP and FN for different thresholds") +
    theme(legend.position = "bottom")
}


# Return confusion matrix based on probablity observations and class of reference
# for the specified threshold
```

```r
preds_to_cm <- function(model_object,
                        model_version,
                        lab_env,
                        threshold){
  # Get the test set
  test_set <- lab_env$Params$TestSet

  # Get the predictions
  F1_prob <- model_object$Preds[[model_version]]$F1
  preds_th <- factor(ifelse(F1_prob > threshold, "F1", "GO")
                     , levels = c("F1", "GO")
                     , labels = c("F1", "GO"))

  obs <- test_set$isFraud

  # Using table calculates the confusion matrix
  df <- data.frame("Predictions" = preds_th
                   , "Reference" = obs
                   , stringsAsFactors = FALSE)
  return(table(df))
}


# Delete the parts of ModFits which are not needed for the version of the
# models that is not "version_keep". Helps manage memory space.
shrink_model <- function(model_object,
                         version_keep){
  model_fits <- model_object$ModFits
  for (i in 1:length(model_fits)){
    if (names(model_fits)[[i]] != version_keep){
      # Remove less important details
      model_object$ModFits <- "Removed to reduce size."
    }
  }
  return(model_object)
}


# Returns the number of reclassifiable false positives for a specific model and version
# based on a specific threshold
get_reclass_FP <- function(model_object,
                           model_version,
                           lab_env,
                           threshold = 0.5){
  # Get parameters from model and lab_env
  type <- model_object$Params$RawProb
  test_set <- lab_env$Params$TestSet

  pred <- model_object$Preds[[model_version]]
  new_df <- cbind(test_set, pred)
  # Let's see how many of the positives are of type payment, debit or cash-in
  if (type == "prob"){
    nfp <- nrow(new_df %>%
```

```
                  filter(F1 > 0.5) %>%
                  filter(type=="PAYMENT"|type=="CASH_IN" | type=="DEBIT"))
  } else if (type == "raw"){
    nfp <- nrow(new_df %>%
                  filter(pred == "F1") %>%
                  filter(type=="PAYMENT"|type=="CASH_IN" | type=="DEBIT"))
  } else {
    warning(
      "Type argument error in get_reclass_FP(): only 'raw' or 'prob' expected.\n"
      )
  }
  return(nfp)
}


# Function to extract and return performance based on the name of the model
get_performance <- function(model_object, lab_env){
  # Create data frame that will be returned
  return_df <- data.frame("Features" = character()
                          , "PRAUC" = numeric()
                          , "Recall" = numeric()
                          , "Precision" = numeric()
                          , "Accuracy" = numeric()
                          , "F1" = numeric()
                          , stringsAsFactors=FALSE)

  # Extract the relevant information from lab_env object
  for (i in 1:length(model_object$Perfs)){
    model_version <- names(model_object$Perfs)[i]
    prauc <- round(
      model_object$Perfs[[i]]$
        PRAUC$auc.integral, 5)
    recall <- round(
      model_object$Perfs[[i]]$
        ConfusionMatrix$byClass["Recall"], 5)
    precision <- round(
      model_object$Perfs[[i]]$
        ConfusionMatrix$byClass["Precision"], 5)
    accuracy <- round(
      model_object$Perfs[[i]]$
        ConfusionMatrix$overall["Accuracy"], 5)
    F1_perf <- round(
      model_object$Perfs[[i]]$
        ConfusionMatrix$byClass["F1"], 5)
    return_df <- return_df %>%
      add_row(Features = model_version
              , PRAUC = prauc
              , Recall = recall
              , Precision = precision
              , Accuracy = accuracy
              , F1 = F1_perf)
  }
  return(return_df)
```

```
}


# Function to extract and return execution time based on the name of the model
# Execution time expressed in hours, minutes, seconds
# Returns a named list
get_exec_time <- function(model_object, lab_env){
  exec_time <- round(model_object$ExecTime$TrainTime$time.sec +
                       model_object$ExecTime$PredTime$time.sec, 5)
  # Get numbers of hours by using %/% to get quotient
  exec_time_hour <- exec_time %/% 3600
  # Get numbers of minutes using %% (modulo)
  exec_time_min <- (exec_time - exec_time_hour*3600) %/% 60
  # Get remaining seconds
  exec_time_sec <- (exec_time - exec_time_hour*3600) %% 60

  exec_time <- list("hour" = exec_time_hour
                    , "min" = exec_time_min
                    , "sec" = exec_time_sec)
  return (exec_time)
}


# Function to extract the confusion matrix based on the name of the model
# and it's option ("base", "err", "all" or "best" - will select the best model)
# Metric for selection of best model can be specified, default value: PRAUC
get_conf_matrix <- function(model_object,
                            model_option,
                            lab_env,
                            comparison_metric="PRAUC"){
  best_perf <- 0

  # If "best" is passed as model_option this loops finds the best version
  # using comparison_metric for the comparison
  if(model_option == "best"){
    for (i in 1:length(model_object$Perfs)){
      model_version <- names(model_object$Perfs)[i]

      # Get performance measure
      if(comparison_metric == "PRAUC"){
        perf_measure <- model_object$Perfs[[i]]$
          PRAUC$auc.integral
      } else if(comparison_metric == "Accuracy"){
        perf_measure <- model_object$Perfs[[i]]$
          ConfusionMatrix$overall[["Accuracy"]]
      } else {
        perf_measure <- model_object$Perfs[[i]]$
          ConfusionMatrix$byClass[[comparison_metric]]
      }

      # If performance version i better than previous, update model_option
      model_option <- ifelse(perf_measure > best_perf
                             , model_version
```

```
                              , model_option)
      # If performance version i better than previous, update performance
      best_perf <- ifelse(perf_measure > best_perf
                          , perf_measure
                          , best_perf)
    }
  }

  return (list("Version" = model_option
               , "Metric" = comparison_metric
               , "Table" = model_object$Perfs[[model_option]]$ConfusionMatrix$table))
}


# Function to display the performance of a model
# "comparison_metric" required to compare the versions to get
# the best version
# Returns the best version of the model

display_best_perf <- function(model_object,
                              lab_env,
                              comparison_metric){
  # Retrieve performance data
  print(get_performance(model_object, lab_env))

  # Dispaly execution time list
  time_list <- get_exec_time(model_object)
  cat("Execution time: "
      , time_list[[1]], "h "
      , time_list[[2]], "m "
      , time_list[[3]], "s.\n"
      , sep = "")

  # Display confusion matrix for the best version of the
  # specified model
  conf_matrix <- get_conf_matrix(model_object = model_object
                                 , model_option = "best"
                                 , lab_env = lab_env
                                 , comparison_metric = comparison_metric)
  cat(
    "Best version: '"
    , conf_matrix[["Version"]]
    , "' when using '"
    , conf_matrix[["Metric"]]
    , "' as comparison metric.\n"
    , "Confusion matrix of best version: \n"
    , sep = ""
    )
  print(conf_matrix[["Table"]])
  return(conf_matrix[["Version"]])
}
```