**School of Computing and Engineering**

**Project Proposal Report**

**CP6CS46E**

Student Name: *Nikolay Nikolaev Ninov*

Student ID: *21361642*

Course: *Computer Science*

Project Title: *Fraud Detection using Machine Learning and Deep Learning for banking*

Supervisor Name: *Professor Massoud Zolgharni*

# Table of Contents

# Introduction

This project is related to my current job at Mitsubishi UFJ Financial Group (MUFG). MUFG is Japan's largest bank and one of the largest banks in the world. The bank focuses on building long term relationships with customers, which requires trust and honesty. Fraud detection is one of the most challenging tasks and can make customers lose the bank's trust that has been built for years.

Fraud is a criminal deception that is committed by a person who acts in a false and dishonest way. According to UK Finance (2019), unauthorised financial fraud losses for payment cards, remote banking and cheques make £844.8 million in 2018, which has increased 16 per cent from 2017. Nowadays, online transactions are even more preferred due to COVID-19 and the introduction of lockdowns all over the globe.

Before machine learning fraud detection, companies would use a rule-based approach to detect fraud by looking at common and evident clues. Pure rule-based algorithms perform countless different fraud scenarios which are manually written by a person. Nowadays rule-based scenarios require too much adjusting in order to keep up with the latest security measures. In addition, rule-based systems are not efficient on real-time data streaming, which can be crucial to banks.

On the other hand, machine learning can be used for real-time data streaming, can find hidden patterns within the user's behaviour and calculate the chance of fraudulent behaviour. Machine learning also takes less time compared to rule-based algorithms.

## Aims and Objectives

The aim of this project is to develop precise machine learning and deep learning models that can prevent fraudulent transactions and compare their differences. In order to accomplish the latter mentioned, I will need to:

- Get a financial dataset
- Remove redundant data fields from the dataset
- Find a suitable metrics to determine if the model is performing good or bad.
- Find the most successful models out of all of the tested ones and further improve the weaker ones.

## Definitions

**ANN** - Artificial Neural Network - also known as a Neural Network.

**RNN** - Recurrent Neural Network - also known as a Vanilla RNN - a class of Neural Networks which takes series of input with no size limit and gives a series of output vectors. These Neural Networks have a hidden state that is updated based on the input and uses it to compute the next input.

**LSTM** - Long Short-Term Memory - is a RNN architecture that has the ability to forget of previously stored memory and gain new information through the use of an input gate, output gate and forget gate.

**KNN** - K-Nearest Neighbour - is a nonparametric algorithm used for classification and regression.

**Random Forests** - an ensemble method made from multiple decision trees for classification and regression. Every tree produces a prediction, and the final result is the forest is based on the majority vote of the trees.

**Unsupervised learning** - algorithms that look for undetected patterns in data sets that do not have labels. These algorithms do not require humans to supervise them.

**Supervised learning** - algorithms that map an input to an output. The algorithm learns from the data and result to generate an inferred function.

**Hyperparameter** - parameter that controls the learning process of the algorithm.

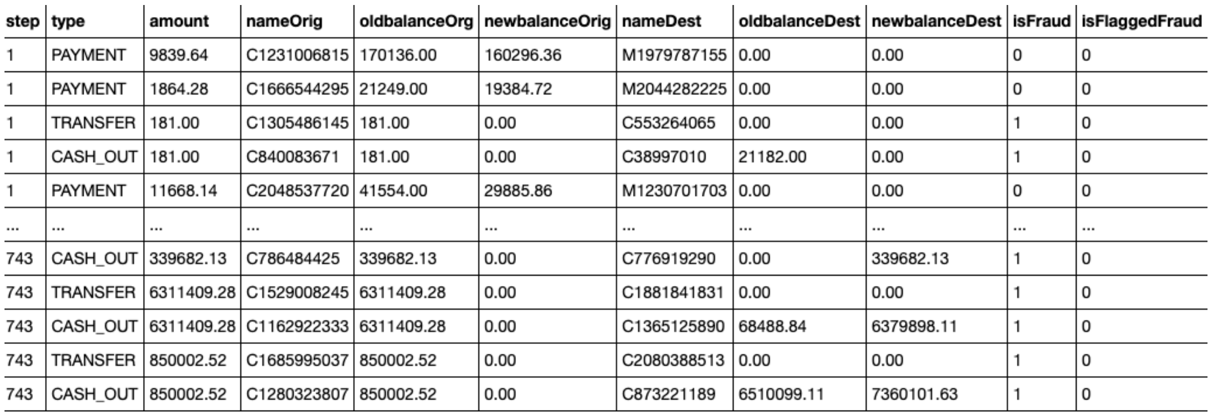
**Grid search** - is a method that tries every possible hyperparameter combination and returns the best one. This method takes a lot of time to process - based on how big the dataset is and how many hyperparameters there are.

**Random search** - is a method that tries random combinations of the hyperparameters to try and find the best ones. This method takes less time to process compared to grid search because it has a given a total amount of combinations that need to be tried.

**Accuracy paradox** - A paradox where the accuracy is not a good metric for classification models because it has a greater predictive power than models with higher accuracy.

## PaySim

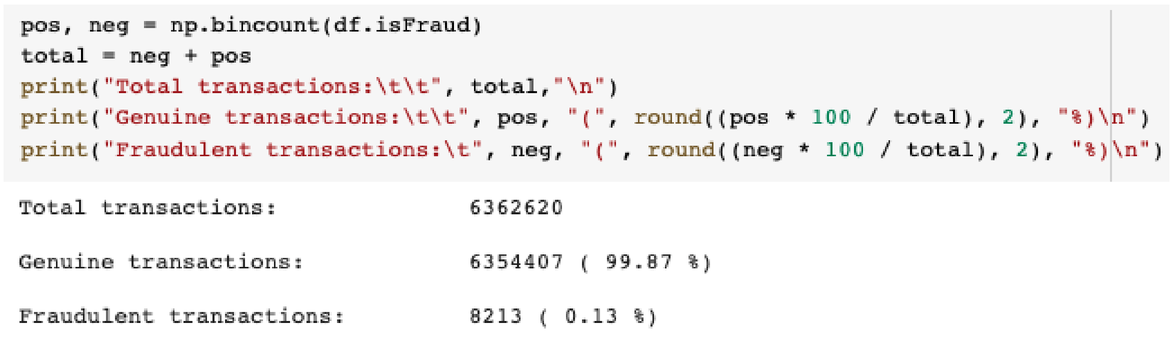
Unfortunately, there are not that many publicly available datasets for fraud detection due to General Data Protection Regulations and personal data. PaySim is a synthetic dataset that has been generated by the PaySim simulator. It uses a collection of private datasets to generate a dataset that is similar to a regular one. PaySim simulates mobile transactions that are sampled from real transactions. These transactions were extracted for one month from different financial logs from a mobile money service that is used in Africa. The logs were supplied by a multinational company which provides mobile financial services in more than 14 countries world-wide. The dataset was scaled down to 1/4th of the original dataset and has 11 columns and 6362620 rows.



| step | type | amount | nameOrig | oldbalanceOrg | newbalanceOrig | nameDest | oldbalanceDest | newbalanceDest | isFraud | isFlaggedFraud |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | PAYMENT | 9839.64 | C1231006815 | 170136.00 | 160296.36 | M1979787155 | 0.00 | 0.00 | 0 | 0 |
| 1 | PAYMENT | 1864.28 | C1666544295 | 21249.00 | 19384.72 | M2044282225 | 0.00 | 0.00 | 0 | 0 |
| 1 | TRANSFER | 181.00 | C1305486145 | 181.00 | 0.00 | C553264065 | 0.00 | 0.00 | 1 | 0 |
| 1 | CASH_OUT | 181.00 | C840083671 | 181.00 | 0.00 | C38997010 | 21182.00 | 0.00 | 1 | 0 |
| 1 | PAYMENT | 11668.14 | C2048537720 | 41554.00 | 29885.86 | M1230701703 | 0.00 | 0.00 | 0 | 0 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 743 | CASH_OUT | 339682.13 | C786484425 | 339682.13 | 0.00 | C776919290 | 0.00 | 339682.13 | 1 | 0 |
| 743 | TRANSFER | 6311409.28 | C1529008245 | 6311409.28 | 0.00 | C1881841831 | 0.00 | 0.00 | 1 | 0 |
| 743 | CASH_OUT | 6311409.28 | C1162922333 | 6311409.28 | 0.00 | C1365125890 | 68488.84 | 6379898.11 | 1 | 0 |
| 743 | TRANSFER | 850002.52 | C1685995037 | 850002.52 | 0.00 | C2080388513 | 0.00 | 0.00 | 1 | 0 |
| 743 | CASH_OUT | 850002.52 | C1280323807 | 850002.52 | 0.00 | C873221189 | 6510099.11 | 7360101.63 | 1 | 0 |

*Figure 1: Display the columns data from the dataset*

This dataset is highly unbalanced and will have to be modified in order for the models to make precise predictions. There are only 0.13% fraudulent transactions and 99.87% genuine transactions.



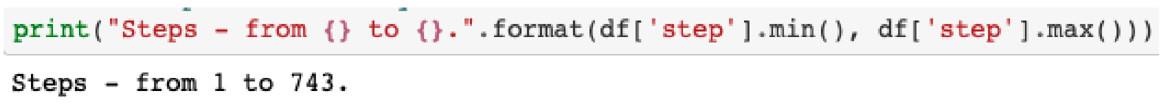
```
pos, neg = np.bincount(df.isFraud)
total = neg + pos
print("Total transactions:\t\t", total,"\n")
print("Genuine transactions:\t\t", pos, "(", round((pos * 100 / total), 2), "%)\n")
print("Fraudulent transactions:\t", neg, "(", round((neg * 100 / total), 2), "%)\n")
```

```
Total transactions:              6362620

Genuine transactions:            6354407 ( 99.87 %)

Fraudulent transactions:         8213 ( 0.13 %)
```

*Figure 2: Amount of genuine and fraudulent transactions*

## step

A step maps a unit of time to the real world. In this dataset 1 step is 1 hour of time. There are a total of 743 steps which is a 30-day simulation.

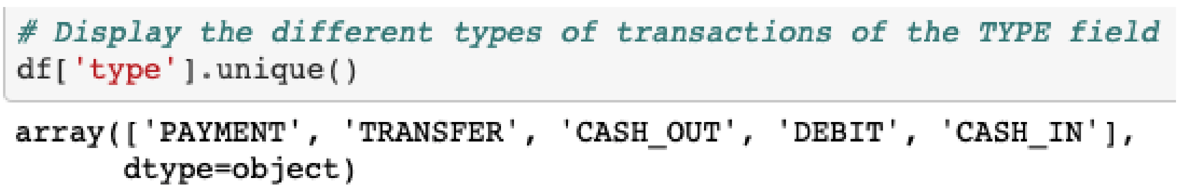


```
print("Steps - from {} to {}.".format(df['step'].min(), df['step'].max()))
```

```
Steps - from 1 to 743.
```

*Figure 3: Total number of steps*

## type



```
# Display the different types of transactions of the TYPE field
df['type'].unique()
```

```
array(['PAYMENT', 'TRANSFER', 'CASH_OUT', 'DEBIT', 'CASH_IN'],
      dtype=object)
```

*Figure 4: Different values in the 'type' column*

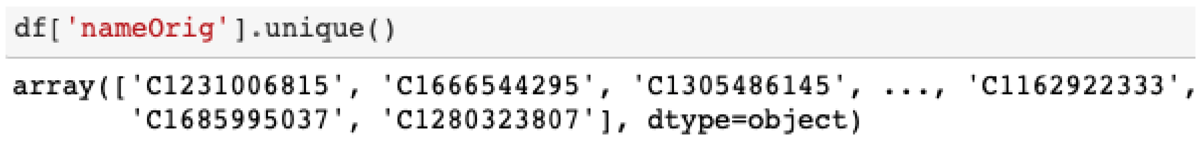
There are 5 different types of transactions:

- **PAYMENT** - When a customer pays to acquire goods or services from a merchant, the transaction is termed as Payment. It decreases the account balance of the sender while account balance of the receiver increases (i.e., amount got credited in his account).
- **TRANSFER** - When one user sends money to another user through mobile money service platform, the transaction is termed as Transfer.

  **CASH_OUT** - Merchant serves as an ATM to the customers and customers can decrease the balance of the account by withdrawing cash from the merchants.
- **CASH_IN** - Merchant serves as an ATM to the customers and customers can increase the balance of the account by paying in cash to the merchants.
- **DEBIT** - When customer send money from a mobile money service to a bank account, the transaction is termed as Debit. It decreases balance in the account just like CASH_OUT transaction.

## amount

The amount of the transaction in local currency.



```
df['nameOrig'].unique()

array(['C1231006815', 'C1666544295', 'C1305486145', ..., 'C1162922333',
       'C1685995037', 'C1280323807'], dtype=object)
```

*Figure 5: Different values in the 'amount' column*

## nameOrig

The name of the account who initiated the transaction.

## oldbalanceOrg

The initial balance of the sender.

## newbalanceOrig

The new balance of the sender.

## nameDest

The name of the account who received the transaction.

## oldbalanceDest

The initial balance of the receiver before the transaction.

## newbalanceDest

The new balance of the receiver after the transaction.

### isFraud

If the transaction is identified as fraudulent (1) or non-fraudulent (0).

### isFlaggedFraud

Flags suspicious transactions as fraud when a user illegally attempts to transfer more than 200.00 in a single transaction.

# Literature Survey

Both supervised and unsupervised learning methods are applied by banks to detect frauds in their data. Supervised learning is more common across different disciplines. Supervised learning requires a labelled dataset in order to make predictions. On the other hand, unsupervised learning detects anomalies and learns to detect patterns to determine if a transaction is fraud or not. Deep learning for fraud detection is associated with learning from customer's patterns and habits to determine if the made transaction is genuine or fraudulent.

SVM is a fast and dependable algorithm that works with a limited dataset. It is used to maximize the margin between different classes - in this case if a transaction is fraud or not. Random Forests are good for unbalanced fraud detection datasets, where there are almost no fraud examples.

The data can be represented in a sequence and LSTM models can be used to determine if a transaction is genuine or fraudulent based user's behaviour level instead of just looking at the transaction. This way no valuable information is lost. LSTM deals with the vanishing gradient problem that Vanilla RNNs face. The vanishing gradient problem occurs in the RNN weights in earlier layers. When layers use activation functions, the gradient of the loss function becomes closer to zero. This makes the network harder to train. However, the LSTM models are harder to implement and slower to train because they need to do additional computations to determine whether the data is kept or discarded. However, LSTMs are more accurate than traditional machine learning algorithms.

Awoyemi, Adetunmbi and Oluwadare (2017) compare the accuracy, sensitivity, specificity and precision for Naïve Bays, KNN and Logistic Regression. Out of the three algorithms, Logistic Regression had the weakest performance. However, paper focuses on accuracy, sensitivity, specificity and precision and does not use the PaySim dataset. It is a good starting point to determine which algorithms can be used as a starting point to detect fraud.

Bandyopadhyay and Dutta (2020) suggest the use of a Stacked-RNN with 12 layers to fully minimize fraud detection for the PaySim dataset. The RNN's accuracy is 99.87%, F1-score is 0.99 and the Mean Squared Error is 0.01. This paper is a good study for a RNN design, however Bandyopadhyay and Dutta use accuracy, F1-score and mean squared error to determine the model's performance. Accuracy in fraud detection is a poor metric due to the accuracy paradox that occurs in fraud detection. In addition, F1 score heavily relies on the True Positive values, which are always the greatest number because genuine transactions are always 99% more than fraudulent transactions - which are less than 1% in the dataset. In addition, mean squared error is mainly used in regression models rather than classifying. MSE does not penalize the model for misclassification as much as it has to.

Kaur (2019) proposes using Logistic Regression, Naïve Bayes, Random Forest, XGBoost algorithms to determine if a transaction is genuine or fraudulent. They use different data segmentation methods - test-train split and K-Folds and compares the differences. Kaur also compares the different algorithms by their accuracy from their research.  XGBoost has the highest accuracy of 99.95% (train-test split) and 96.46% (K-fold). Kaur also uses suggests using grid search and random search to determine which parameters are best for a specific algorithm. The research is a good starting point because it uses the PaySim dataset and has the best parameters for the suggested algorithms. However, it does not focus on the correct metrics such as False Negative Rate or the percentage of fraud that has been missed.

Baesens, Höppner and Verdonck (2021) compare different metrics based on the dataset itself and when the SMOTE technique is used on it to create synthetic fraudulent transactions. They also suggest a precise custom metrics to identify how well an algorithm is performed such as how much money a fraudulent transaction would cost (if fraud is committed), how much money would a genuine transaction that is thought fraudulent would cost and different weights for true negative, false negative, true positive and false positive values. This paper is a good starting point for considering the correct metrics when evaluating different models. However, they do not use the PaySim dataset and have more specific information such as admissions costs and real data.

Nordling (2020) suggests using decision trees, random forests and autoencoders to determine if a transaction is genuine or fraudulent. In order to overcome the imbalanced dataset, the Synthetic Minority Over-sampling Technique is proposed to create more transactions of the less dominant class until both classes have an equal amount of data points. However, the study uses a different dataset and uses Accuracy, Recall and AUROC as metrics.

Pambudi, Hidayah and Fauziati propose using SVM to determine if a transaction is fraudulent or genuine. They compare the performance of the different kernels with different gammas and C values by using Precision, Recall, F1-score and Area Under Precision-Recall Curve (AUPRC) as their metrics. However computing SVM is very long especially for large datasets with many columns. In addition the research does not go over how their data is cleaned and meaning that the model using the same parameters can be completely different.

Overall different studies propose to either use XGBoost, Random Forest, Recurrent Neural Network, SVM Logistic Regression and Naïve Bayes with metrics such as Accuracy, Recall and AUROC. Studies with custom datasets propose custom metrics that more precisely capture if a model is performing good or bad.

# Methodology

## Metrics

In order to discuss what metrics will be used, the confusion matrix must be explained:

- **True Positive (TP)** - How many genuine transactions were correctly predicted as genuine.
- **False Positive (FP)** - How many genuine transactions were predicted as fraud. This shows how many genuine transactions have been rejected, because the model thought that it was fraud.
- **False Negative (FN)** - This is the most important metric. It indicates how many fraud transactions were predicted as genuine. It shows that fraud has occurred and has not been detected.
- **True Negative (TN)** - How many fraud transactions were correctly predicted as fraud.

In traditional binary classification, one must minimize the loss function and maximize the metrics such as F1-score, accuracy, AUC, etc.

Accuracy is a metric that is used to predict how correct is our model. Unfortunately, accuracy will not work for this fraud detection dataset, because there are 99% genuine transactions and less than 1% fraudulent transactions. This implies that the accuracy will be 99% correct for any model. Hence the accuracy paradox occurs.

$$accuracy = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{TN} + \mathrm{FP} + \mathrm{FN}}$$

F1-score would not work as well because F1-score is a combination of precision and recall which emphasise on True Positive values (genuine transactions).

$$F1 = \frac{\mathrm{precision} * \mathrm{recall}}{\mathrm{precision} + \mathrm{recall}}$$

$$precision = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}$$

$$recall = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}$$

The AUC score would not work because it emphasises on the probability that a selected positive example will have a higher predicted probability of being positive than negative. In fraud detection the AUC score would tell us how more likely a transaction would be genuine than fraudulent. Usually, datasets are very unbalanced, and this will give every model at least 0.99 score or even 1 because fraudulent transactions are not enough.

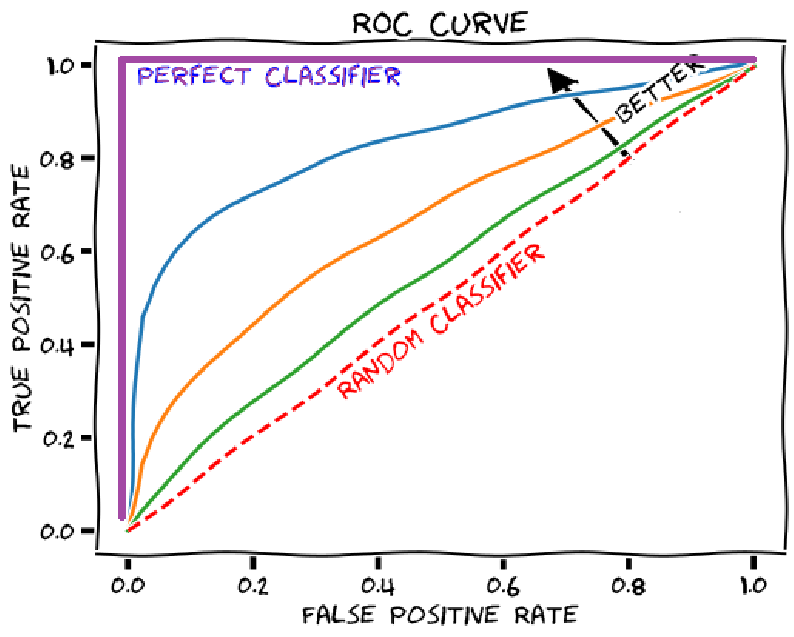


*Figure 6: Example of AUC curve*

Therefore, a higher AUC indicates superior classification performance. When dealing with highly imbalanced data as is the case with fraud detection, AUC (and ROC curves) may be too optimistic, and the Area under the Precision-Recall Curve (AUPRC) gives a more informative picture of a classifier's performance. AUPRC is a useful metric for imbalanced datasets where one class (especially the positive one) dominates the other (usually the negative one). It calculates the area under the PR curve, which shows the agreement between precision and recall. This curve starts from the top left corner (recall = 0 and precision = 1) and ends in the lower right corner (recall = 1 and precision = 0). The data between the start and end point are obtained by calculating the precision and recall of the different thresholds.

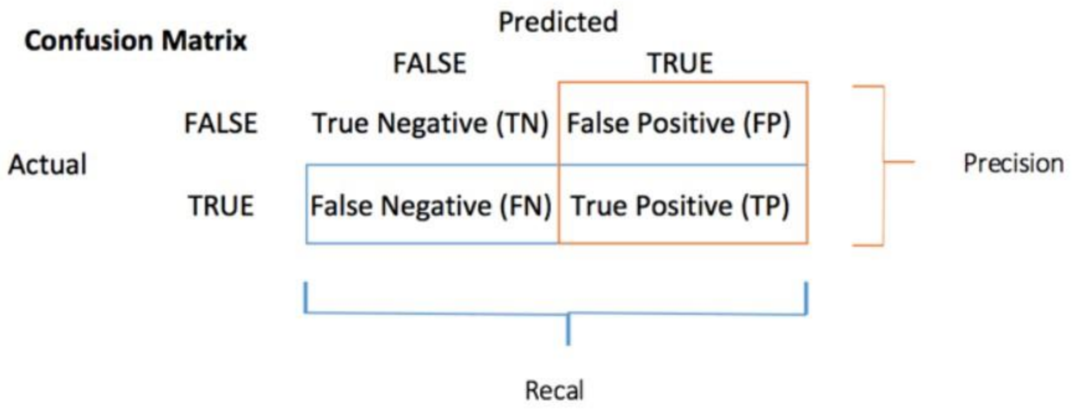


*Figure 7: Confusion Matrix example and show what precision and recall are created*

However, the most important metric is the ***% of fraud detected*** by the model. This metrics focuses on how the number of False Negative transactions. False Negatives are crucial because this indicates that fraudulent transactions are thought to be genuine. This leads to banks losing money. False Positives (genuine transactions that are thought to be fraudulent) are less crucial - the bank has to contact the customer to verify them or the other way around - customer calls the bank to verify that they are doing the "suspicious" transaction. The metric is calculated the following way:

$$Total\ fraud = \mathrm{FN} + \mathrm{TN}$$

$$\%\ Fraud = \frac{\mathrm{TN}}{\text{Total fraud}}$$

$$\%\ Missed = 1 - \%\ \mathrm{Fraud}$$

## Imbalanced data techniques

As previously mentioned, the PaySim dataset is highly unbalanced and will have to be modified in order for the models to make precise predictions. There are only 0.13% fraudulent transactions and 99.87% genuine transactions. Imbalanced data refers to a classification problem where the classes are not represented equally. With an imbalance the minority class can be completely ignored by the model and the results will be inaccurate. The data will have to be either oversampled, undersampled or the class weights have to be changed in order for the model to make accurate predictions.

### Random Undersampling

Random undersampling randomly selects and removes samples from the majority of the class. Samples are removed until the majority class has an equal amount of data as the minority class. In the PaySim dataset this means that majority of the genuine transactions will be discarded until there is an equal number of fraudulent and genuine transactions. This can be very problematic because data is lost, and the models will learn harder, and their performance will decrease. This means that 99% of the data will be discarded.

### Oversampling

#### *Random Oversampling*

One technique of oversampling is random oversampling. This technique consists of creating copies of the minority class so there is an equal amount of fraudulent and genuine transactions. However, this causes the problem of overfitting, meaning that the algorithm stops learning what are fraud and genuine transactions and starts memorising the dataset itself. Random oversampling will achieve perfect results because it has memories the data completely but when given new data it will have a very poor performance.

## *Synthetic Minority Oversampling Technique*

SMOTE is another approach to handle imbalanced datasets. It evens out the minority class but without copying already existing data. This technique synthesises new minority data points between existing minority instances.

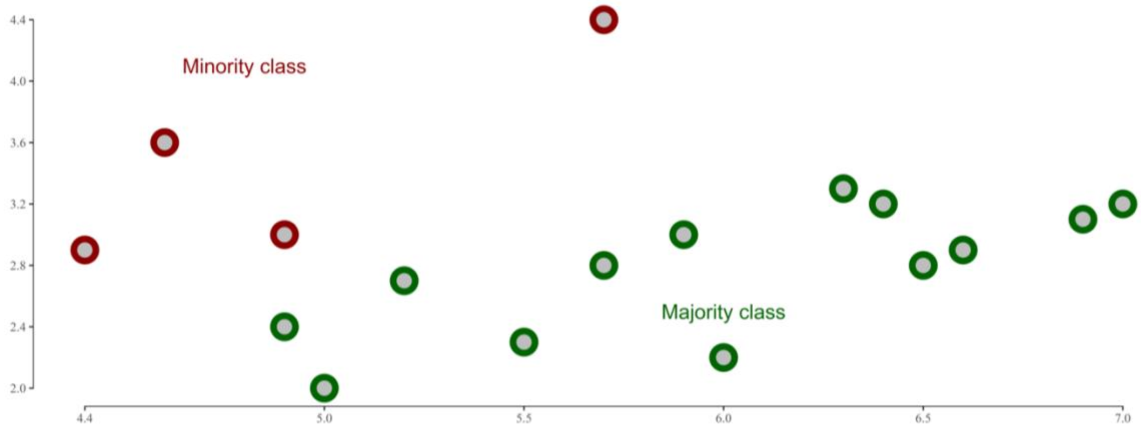


*Figure 8: Example of class imbalance*

Initially the algorithm plots down every data point. Then the feature vector and the data point's nearest neighbours are identified. The difference between the two vectors is calculated and their difference is multiplied by a random number between 0 and 1. Finally the new point is identified on the line segment by adding the random number to feature vector. For the PaySim dataset the latter steps are followed until the number of fraudulent transactions are equal to the number of genuine transactions.

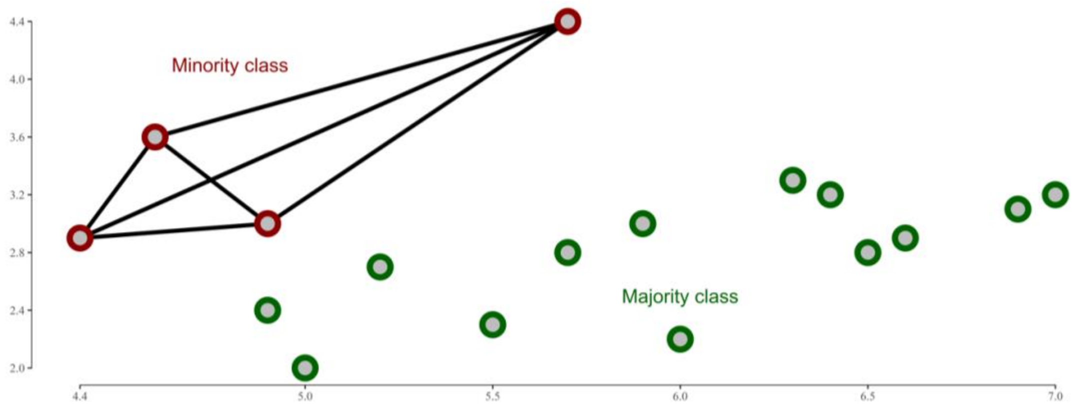


*Figure 9: Connecting minority class data points*

The SMOTE technique is used in this project so the dataset can be balanced. Random oversampling unfortunately will lead the model to overfitting, making it inaccurate on predicting actual data whereas random undersampling will remove a majority of the genuine transactions and the models will be undertrained.

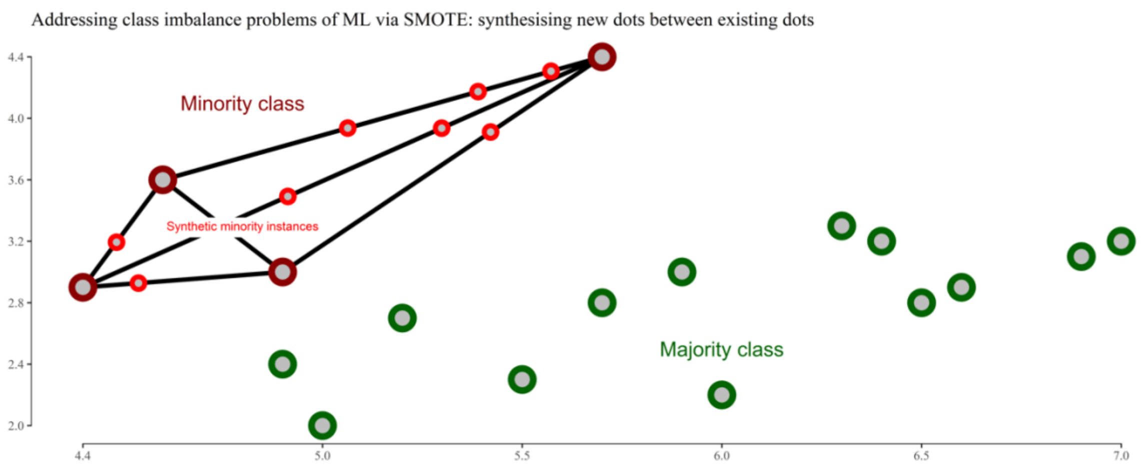


*Figure 10: Creating new synthetic data points of the minority class*

### Class weights

In general machine learning algorithms do not take into consideration if the dataset is imbalanced, meaning that there will be an equal penalization for misclassifying a correct and wrong example. However, in imbalanced datasets if a minority class is misclassified the penalization has to be higher than the penalization of misclassifying a correct example. When implementing the algorithm, the class weight is turned off by default, giving both classes equal weights. When the class weight is set to balanced, the weights are calculated the following way:

$$wj = \frac{\text{n\_samples}}{\text{n\_classes} * n\_samplesj}$$

- *wj* - weight for each class, j denotes the class
- *n_samples* - total number of samples in the dataset
- *n_classes* - total number of unique classes in the dataset
- *n_samplesj* - total number of classes of the selected class

## Data cleansing

In order to produce precises results, the dataset needs to be in an acceptable stage or improved. Removing noise, missing values and inconsistent fields will further improve the dataset. The dataset consists of 6362620 rows and 11 columns. In order to correctly clean the data, the following questions need to be answered:

### Which transactions are fraudulent?

The amount of fraud is transactions is 0.13%. Fraud only occurs in transactions that have a *TRANSACTION* type of either *TRANSFER* or *CASH_OUT*. From the second graph we can see that *TRANSFERs* happen 4 times less than *CASH_OUTs*. Hence *PAYMENT*, *DEBIT* and *CASH_IN* transaction types are redundant and are removed because they do not have any fraudulent behaviour.

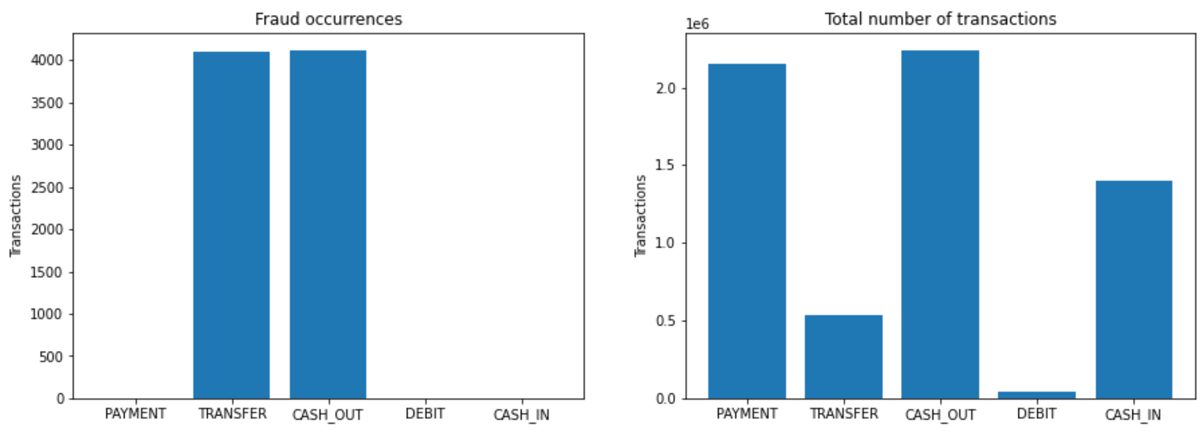
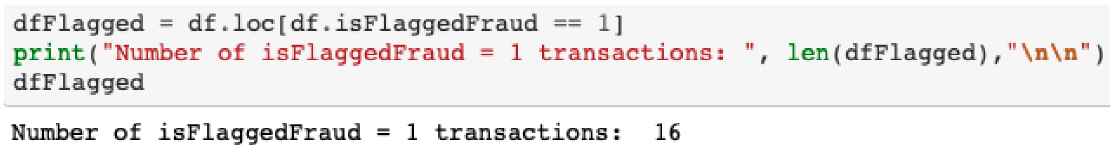


*Figure 11: Transactions that include fraud and display the total number of transactions*

## What determines whether the feature isFlaggedFraud gets set or not?

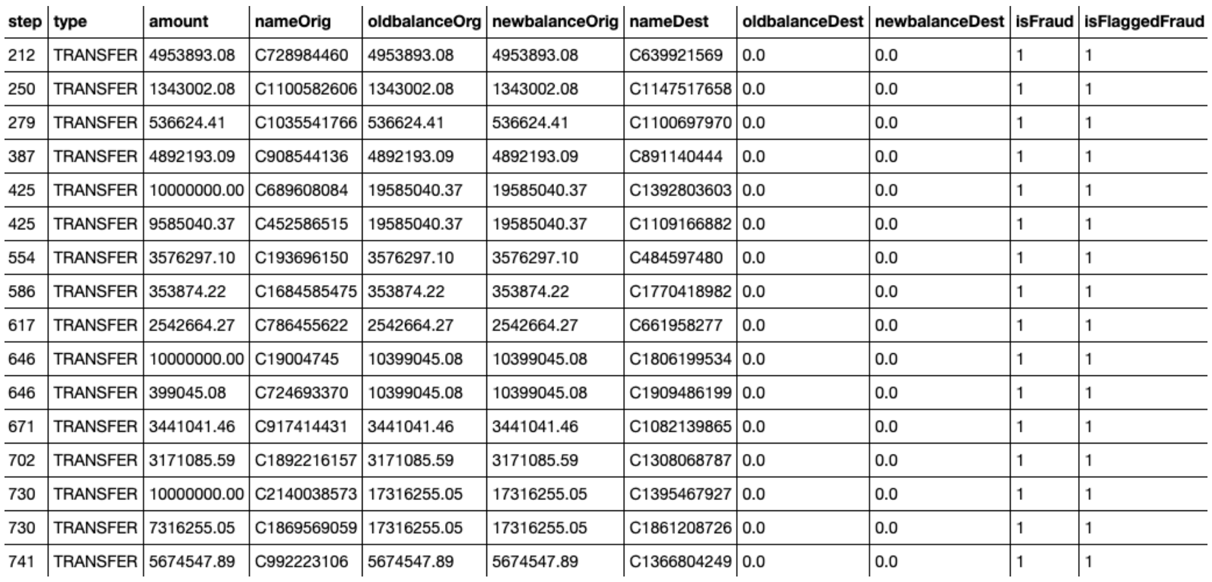
There are only 16 entries out of 6.3 million where *isFlaggedFraud* is set to 1. The *isFlaggedFraud* column is set only when a **TRANSFER** is done, and the *AMOUNT* is greater than 200,000 (local currency).



```
dfFlagged = df.loc[df.isFlaggedFraud == 1]
print("Number of isFlaggedFraud = 1 transactions: ", len(dfFlagged),"\n\n")
dfFlagged
```

```
Number of isFlaggedFraud = 1 transactions:  16
```

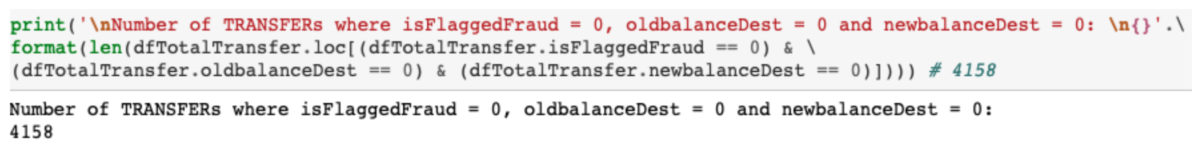
*Figure 12: Number of transactions which were considered to be fraudulent*



| step | type | amount | nameOrig | oldbalanceOrg | newbalanceOrig | nameDest | oldbalanceDest | newbalanceDest | isFraud | isFlaggedFraud |
|---|---|---|---|---|---|---|---|---|---|---|
| 212 | TRANSFER | 4953893.08 | C728984460 | 4953893.08 | 4953893.08 | C639921569 | 0.0 | 0.0 | 1 | 1 |
| 250 | TRANSFER | 1343002.08 | C1100582606 | 1343002.08 | 1343002.08 | C1147517658 | 0.0 | 0.0 | 1 | 1 |
| 279 | TRANSFER | 536624.41 | C1035541766 | 536624.41 | 536624.41 | C1100697970 | 0.0 | 0.0 | 1 | 1 |
| 387 | TRANSFER | 4892193.09 | C908544136 | 4892193.09 | 4892193.09 | C891140444 | 0.0 | 0.0 | 1 | 1 |
| 425 | TRANSFER | 10000000.00 | C689608084 | 19585040.37 | 19585040.37 | C1392803603 | 0.0 | 0.0 | 1 | 1 |
| 425 | TRANSFER | 9585040.37 | C452586515 | 19585040.37 | 19585040.37 | C1109166882 | 0.0 | 0.0 | 1 | 1 |
| 554 | TRANSFER | 3576297.10 | C193696150 | 3576297.10 | 3576297.10 | C484597480 | 0.0 | 0.0 | 1 | 1 |
| 586 | TRANSFER | 353874.22 | C1684585475 | 353874.22 | 353874.22 | C1770418982 | 0.0 | 0.0 | 1 | 1 |
| 617 | TRANSFER | 2542664.27 | C786455622 | 2542664.27 | 2542664.27 | C661958277 | 0.0 | 0.0 | 1 | 1 |
| 646 | TRANSFER | 10000000.00 | C19004745 | 10399045.08 | 10399045.08 | C1806199534 | 0.0 | 0.0 | 1 | 1 |
| 646 | TRANSFER | 399045.08 | C724693370 | 10399045.08 | 10399045.08 | C1909486199 | 0.0 | 0.0 | 1 | 1 |
| 671 | TRANSFER | 3441041.46 | C917414431 | 3441041.46 | 3441041.46 | C1082139865 | 0.0 | 0.0 | 1 | 1 |
| 702 | TRANSFER | 3171085.59 | C1892216157 | 3171085.59 | 3171085.59 | C1308068787 | 0.0 | 0.0 | 1 | 1 |
| 730 | TRANSFER | 10000000.00 | C2140038573 | 17316255.05 | 17316255.05 | C1395467927 | 0.0 | 0.0 | 1 | 1 |
| 730 | TRANSFER | 7316255.05 | C1869569059 | 17316255.05 | 17316255.05 | C1861208726 | 0.0 | 0.0 | 1 | 1 |
| 741 | TRANSFER | 5674547.89 | C992223106 | 5674547.89 | 5674547.89 | C1366804249 | 0.0 | 0.0 | 1 | 1 |

*Figure 13: All transactions that were considered fraudulent*

Even if the *isFlaggedFraud* condition is met, in some cases its value remains 0. In the table, the *oldbalanceDest* and *newbalanceDest* are identical (0.00). This is because the transaction is halted when the threshold is reached, however *isFlaggedFraud* can remain 0 in *TRANSFERs* where *oldbalanceDest* and *newbalanceDest* can both be 0. Hence these conditions do not determine the state of *isFlaggedFraud*.

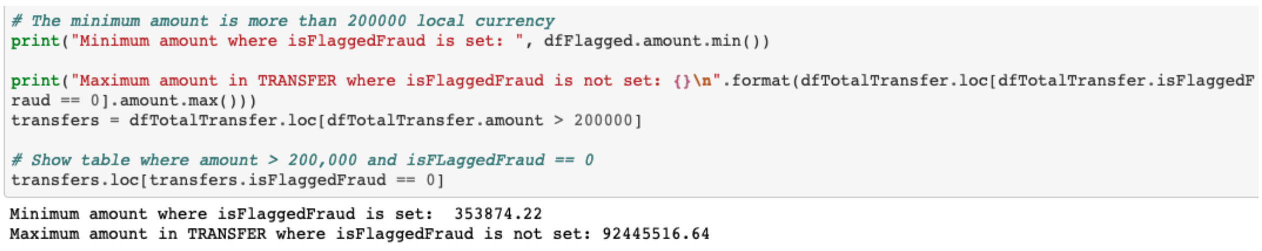


```
print('\nNumber of TRANSFERs where isFlaggedFraud = 0, oldbalanceDest = 0 and newbalanceDest = 0: \n{}'.\
format(len(dfTotalTransfer.loc[(dfTotalTransfer.isFlaggedFraud == 0) & \
(dfTotalTransfer.oldbalanceDest == 0) & (dfTotalTransfer.newbalanceDest == 0)]))) # 4158
```

```
Number of TRANSFERs where isFlaggedFraud = 0, oldbalanceDest = 0 and newbalanceDest = 0:
4158
```

*Figure 14: Show the total of transactions that were considered to be fraudulent and the previous and new balance is 0*

Even if the *isFlaggedFraud* condition is met, in some cases the flag value remains 0.

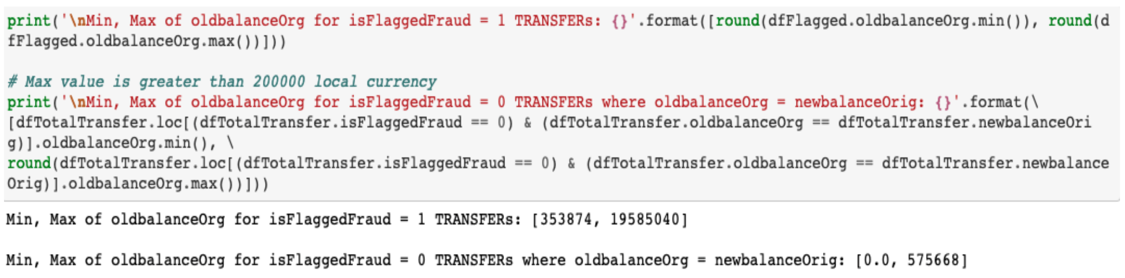


```
# The minimum amount is more than 200000 local currency
print("Minimum amount where isFlaggedFraud is set: ", dfFlagged.amount.min())

print("Maximum amount in TRANSFER where isFlaggedFraud is not set: {}\n".format(dfTotalTransfer.loc[dfTotalTransfer.isFlaggedF
raud == 0].amount.max()))
transfers = dfTotalTransfer.loc[dfTotalTransfer.amount > 200000]

# Show table where amount > 200,000 and isFLaggedFraud == 0
transfers.loc[transfers.isFlaggedFraud == 0]
```

```
Minimum amount where isFlaggedFraud is set:  353874.22
Maximum amount in TRANSFER where isFlaggedFraud is not set: 92445516.64
```

*Figure 15: Further investigation whether isFlaggedFraud is a useful column in the dataset*

There are 409094 rows where the amount is greater than 200000 and *isFlaggedFraud* is 0. Then *isFlaggedFraud* cannot be a threshold on *oldbalanceOrg* because values overlap. The maximum value of *oldbalanceOrg* is more than 200,000 when *isFlaggedFraud* is 0.

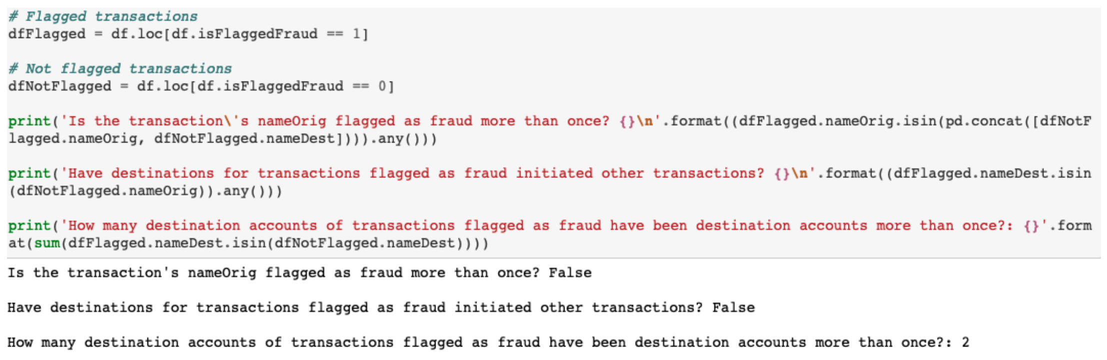


```
print('\nMin, Max of oldbalanceOrg for isFlaggedFraud = 1 TRANSFERs: {}'.format([round(dfFlagged.oldbalanceOrg.min()), round(d
fFlagged.oldbalanceOrg.max())]))

# Max value is greater than 200000 local currency
print('\nMin, Max of oldbalanceOrg for isFlaggedFraud = 0 TRANSFERs where oldbalanceOrg = newbalanceOrig: {}'.format(\
[dfTotalTransfer.loc[(dfTotalTransfer.isFlaggedFraud == 0) & (dfTotalTransfer.oldbalanceOrg == dfTotalTransfer.newbalanceOri
g)].oldbalanceOrg.min(), \
round(dfTotalTransfer.loc[(dfTotalTransfer.isFlaggedFraud == 0) & (dfTotalTransfer.oldbalanceOrg == dfTotalTransfer.newbalance
Orig)].oldbalanceOrg.max())]))
```

```
Min, Max of oldbalanceOrg for isFlaggedFraud = 1 TRANSFERs: [353874, 19585040]

Min, Max of oldbalanceOrg for isFlaggedFraud = 0 TRANSFERs where oldbalanceOrg = newbalanceOrig: [0.0, 575668]
```

*Figure 16: Get the minimum and maximum amount of where fraud is flagged and not flagged*

Another question to answer is if *isFlaggedFraud* is set based on seeing a customer doing a transaction more than once. Duplicate customers do not exist when *isFlaggedFraud* is equal to 0. However, duplicates exist when *isFlaggedFraud* is equal to 1.

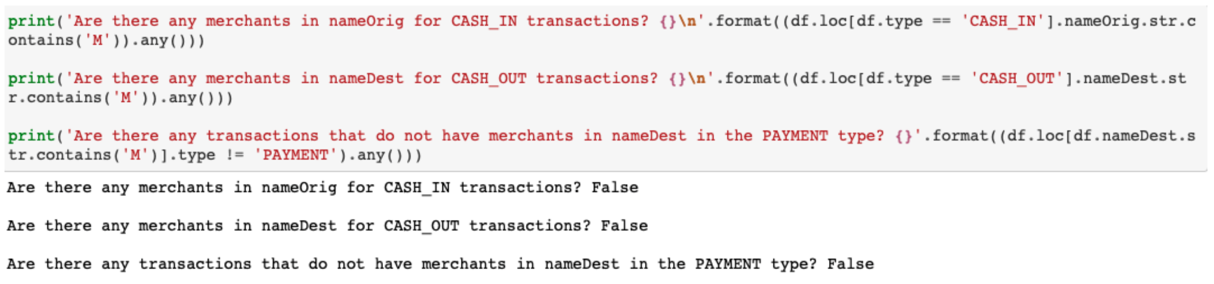

```
# Flagged transactions
dfFlagged = df.loc[df.isFlaggedFraud == 1]

# Not flagged transactions
dfNotFlagged = df.loc[df.isFlaggedFraud == 0]

print('Is the transaction\'s nameOrig flagged as fraud more than once? {}\n'.format((dfFlagged.nameOrig.isin(pd.concat([dfNotF
lagged.nameOrig, dfNotFlagged.nameDest]))).any()))

print('Have destinations for transactions flagged as fraud initiated other transactions? {}\n'.format((dfFlagged.nameDest.isin
(dfNotFlagged.nameOrig)).any()))

print('How many destination accounts of transactions flagged as fraud have been destination accounts more than once?: {}'.form
at(sum(dfFlagged.nameDest.isin(dfNotFlagged.nameDest))))
```

```
Is the transaction's nameOrig flagged as fraud more than once? False

Have destinations for transactions flagged as fraud initiated other transactions? False

How many destination accounts of transactions flagged as fraud have been destination accounts more than once?: 2
```

*Figure 17: Check if duplicate customers exist based on the isFlaggedFraud status*

## Are expected merchant accounts accordingly labelled?

Merchants ('*M*') are not involved in *CASH_IN* (paid by the merchant) transactions to customers ('*C*'). There are also no merchants among destination accounts for *CASH_OUT* transactions (paying a merchant). However, merchants exist for all *PAYMENTs* transactions in *nameDest*.

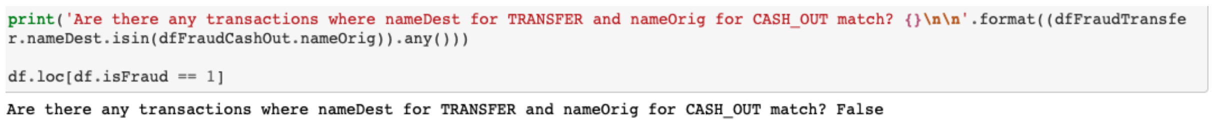

```
print('Are there any merchants in nameOrig for CASH_IN transactions? {}\n'.format((df.loc[df.type == 'CASH_IN'].nameOrig.str.c
ontains('M')).any()))

print('Are there any merchants in nameDest for CASH_OUT transactions? {}\n'.format((df.loc[df.type == 'CASH_OUT'].nameDest.st
r.contains('M')).any()))

print('Are there any transactions that do not have merchants in nameDest in the PAYMENT type? {}'.format((df.loc[df.nameDest.s
tr.contains('M')].type != 'PAYMENT').any()))
```

```
Are there any merchants in nameOrig for CASH_IN transactions? False

Are there any merchants in nameDest for CASH_OUT transactions? False

Are there any transactions that do not have merchants in nameDest in the PAYMENT type? False
```

*Figure 18: Check if merchants are involved in different types of transactions*

## Are there account labels common to fraudulent transactions?

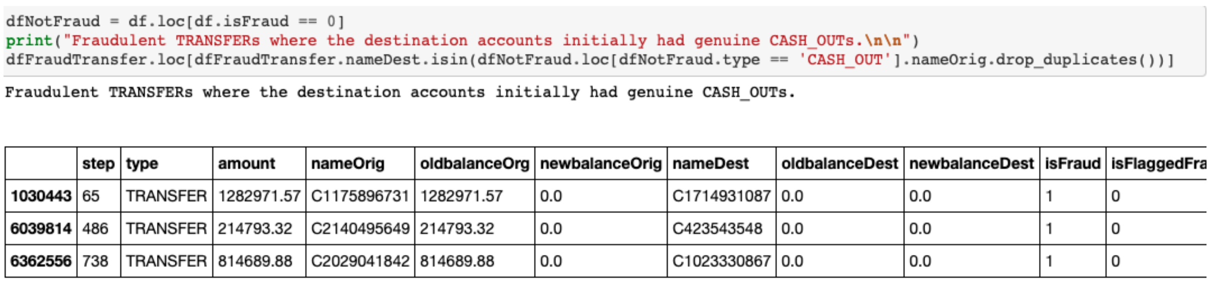
From the data, fraud involves initially *TRANSFERing* money to an account and then withdrawing (*CASH_OUT*) it, hence the *nameDest* for *TRANSFER* and *nameOrig* for *CASH_OUT* should match. However, this is not the case.


```
print('Are there any transactions where nameDest for TRANSFER and nameOrig for CASH_OUT match? {}\n\n'.format((dfFraudTransfe
r.nameDest.isin(dfFraudCashOut.nameOrig)).any()))

df.loc[df.isFraud == 1]
```

```
Are there any transactions where nameDest for TRANSFER and nameOrig for CASH_OUT match? False
```

*Figure 19: Check for transactions where the destination for Transfer transactions matches for CASH OUT*

There are 3 accounts where there is a fraudulent *TRANSFER* where the original *CASHOUT* is not detected and labelled as genuine. 2 out of 3 accounts initially made a genuine *CASH_OUT* and then receive a fraudulent *TRANSFER*. Fraudulent transactions cannot be indicated by the *nameOrig* and *nameDest* features.



```
dfNotFraud = df.loc[df.isFraud == 0]
print("Fraudulent TRANSFERs where the destination accounts initially had genuine CASH_OUTs.\n\n")
dfFraudTransfer.loc[dfFraudTransfer.nameDest.isin(dfNotFraud.loc[dfNotFraud.type == 'CASH_OUT'].nameOrig.drop_duplicates())]
```

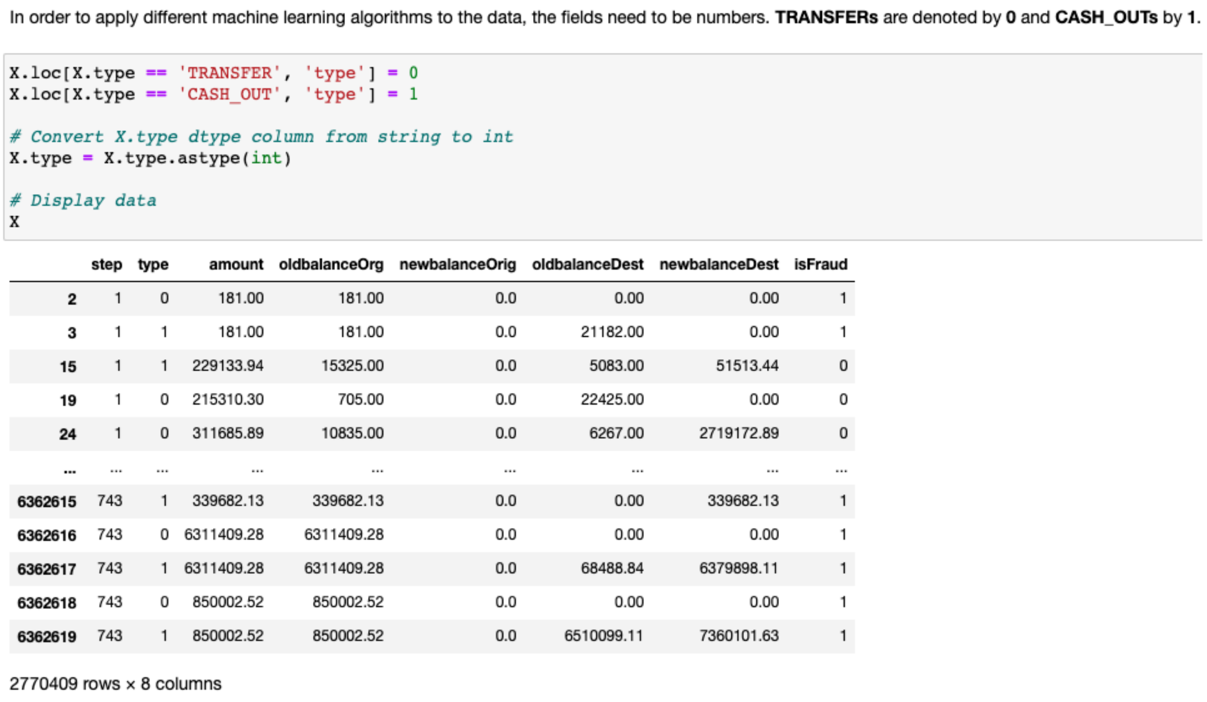
```
Fraudulent TRANSFERs where the destination accounts initially had genuine CASH_OUTs.
```

| | step | type | amount | nameOrig | oldbalanceOrg | newbalanceOrig | nameDest | oldbalanceDest | newbalanceDest | isFraud | isFlaggedFra |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **1030443** | 65 | TRANSFER | 1282971.57 | C1175896731 | 1282971.57 | 0.0 | C1714931087 | 0.0 | 0.0 | 1 | 0 |
| **6039814** | 486 | TRANSFER | 214793.32 | C2140495649 | 214793.32 | 0.0 | C423543548 | 0.0 | 0.0 | 1 | 0 |
| **6362556** | 738 | TRANSFER | 814689.88 | C2029041842 | 814689.88 | 0.0 | C1023330867 | 0.0 | 0.0 | 1 | 0 |

*Figure 20: Get fraudulent transactions that have a genuine CASH OUT status*

## Data Encoding

In order to apply different machine learning algorithms to the data, the fields need to be numbers. *TRANSFERs* are denoted by 0 and *CASH_OUTs* by 1.



In order to apply different machine learning algorithms to the data, the fields need to be numbers. **TRANSFERs** are denoted by **0** and **CASH_OUTs** by **1**.

```
X.loc[X.type == 'TRANSFER', 'type'] = 0
X.loc[X.type == 'CASH_OUT', 'type'] = 1

# Convert X.type dtype column from string to int
X.type = X.type.astype(int)

# Display data
X
```

| | step | type | amount | oldbalanceOrg | newbalanceOrig | oldbalanceDest | newbalanceDest | isFraud |
|---|---|---|---|---|---|---|---|---|
| **2** | 1 | 0 | 181.00 | 181.00 | 0.0 | 0.00 | 0.00 | 1 |
| **3** | 1 | 1 | 181.00 | 181.00 | 0.0 | 21182.00 | 0.00 | 1 |
| **15** | 1 | 1 | 229133.94 | 15325.00 | 0.0 | 5083.00 | 51513.44 | 0 |
| **19** | 1 | 0 | 215310.30 | 705.00 | 0.0 | 22425.00 | 0.00 | 0 |
| **24** | 1 | 0 | 311685.89 | 10835.00 | 0.0 | 6267.00 | 2719172.89 | 0 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... |
| **6362615** | 743 | 1 | 339682.13 | 339682.13 | 0.0 | 0.00 | 339682.13 | 1 |
| **6362616** | 743 | 0 | 6311409.28 | 6311409.28 | 0.0 | 0.00 | 0.00 | 1 |
| **6362617** | 743 | 1 | 6311409.28 | 6311409.28 | 0.0 | 68488.84 | 6379898.11 | 1 |
| **6362618** | 743 | 0 | 850002.52 | 850002.52 | 0.0 | 0.00 | 0.00 | 1 |
| **6362619** | 743 | 1 | 850002.52 | 850002.52 | 0.0 | 6510099.11 | 7360101.63 | 1 |

2770409 rows × 8 columns

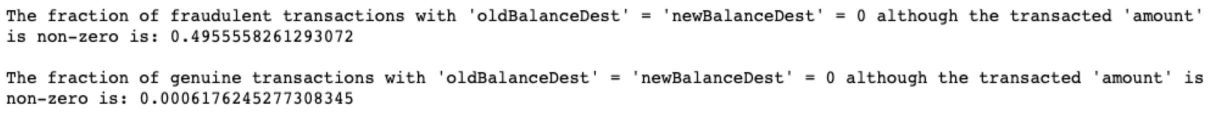
*Figure 21: Encode TRANSFER and CASH_OUT types and show the new data*

## Data cleaning

As mentioned before fraud only occurs in *TRANSFERs* and *CASH_OUTs* meaning that *PAYMENT*, *DEBIT* and *CASH_IN* fields will be removed from the data. From the latter questions we find out that the fields *nameOrig*, *nameDest* and *isFlaggedFraud* are irrelevant and will be removed as well.

In order to apply different machine learning algorithms to the data, the fields need to be numbers. The only column that does not have numbers as input is type. We denote *TRANSFERs* as 0s and *CASH_OUTs* as 1.

The data has transactions with 0.00 in *newbalanceDest* before and after a non-zero amount is transacted, meaning that the *newbalanceDest* field is possibly related to fraud. We replace the value of 0 with -1 which will be more useful for the different machine learning algorithms.



```
The fraction of fraudulent transactions with 'oldBalanceDest' = 'newBalanceDest' = 0 although the transacted 'amount'
is non-zero is: 0.4955558261293072

The fraction of genuine transactions with 'oldBalanceDest' = 'newBalanceDest' = 0 although the transacted 'amount' is
non-zero is: 0.0006176245277308345
```

*Figure 22: Get the fraction of fraudulent transactions where the old and new balance destinations match*

There are also several transactions where the balance is 0 in *oldbalanceOrg* and *newbalanceOrig*.

## Feature Engineering

Feature engineering is transforming the data into features which describe the data's structure. 2 new features have been created - *errorBalanceOrig* and *errorBalanceDest*. Two new columns are created which record errors in the originating (*errorBalanceOrig*) and destination (*errorBalanceDest*) accounts for each transaction. This helps distinguish zero-balanced fraudulent and genuine transactions. These features are important to get the best performance from Machine Learning algorithms.

$$errorBalanceOrig = newBalanceOrig + amount - oldBalanceOrig$$

$$errorBalanceDest = newBalanceDest + amount - oldBalanceDest$$

Two additional features have been created - *DayOfWeek* and *HourOfDay*.

$$HourOfDay = \frac{\text{step}}{24}$$

$$DayOfWeek = \frac{\text{step}}{7}$$

The following histograms display how fraud is spread over the course of a month.

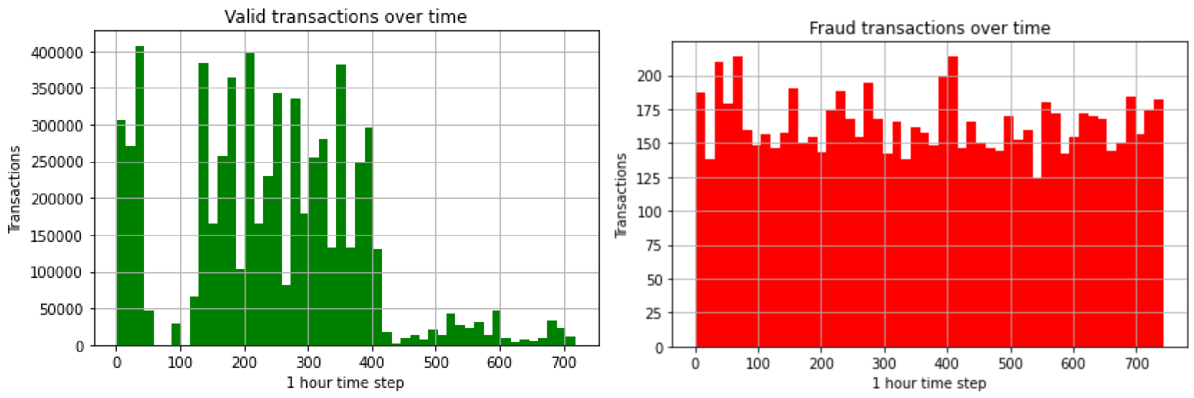


*Figure 23: Get the genuine and fraudulent transactions rate based on the current time*

Most valid transactions occur around from 0 to 60 time step and from 110 to 410. The visualizations represent the number of transactions for every time step of the month. Frequency of occurrence of fraudulent transactions does not change a lot over time.

However, day 0 in the histograms does not mean that Monday or Sunday. From the latter histograms there is not enough evidence that fraudulent transactions occur at a particular day of the week. Hence showing what day of the week is unimportant.

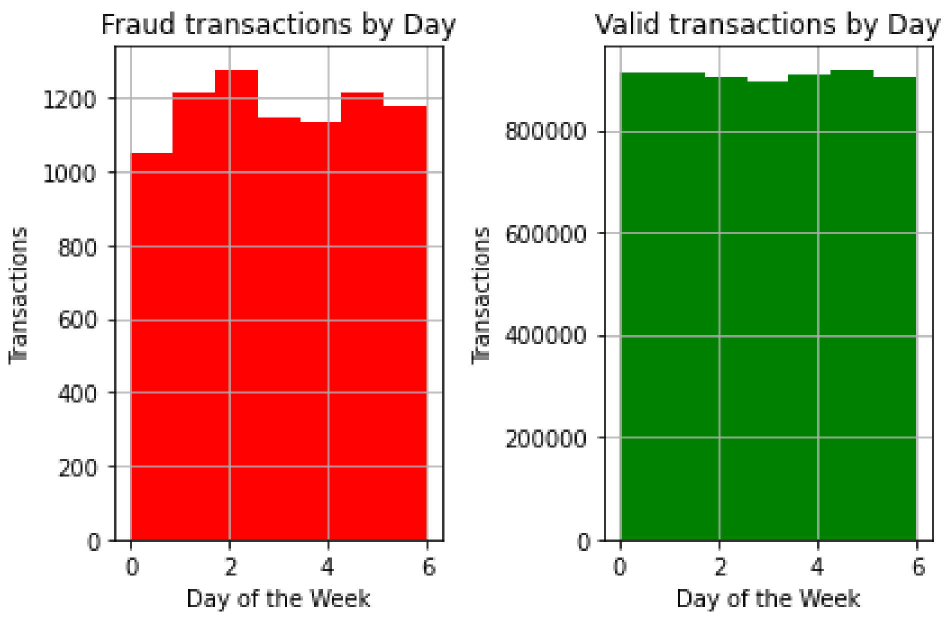


*Figure 24: Get the number of transactions on a weekly basis*

However, when transactions are looked based on the time of the day fraud can happen during any time of a day, whereas genuine transactions mostly occur between hour 10 and 20.

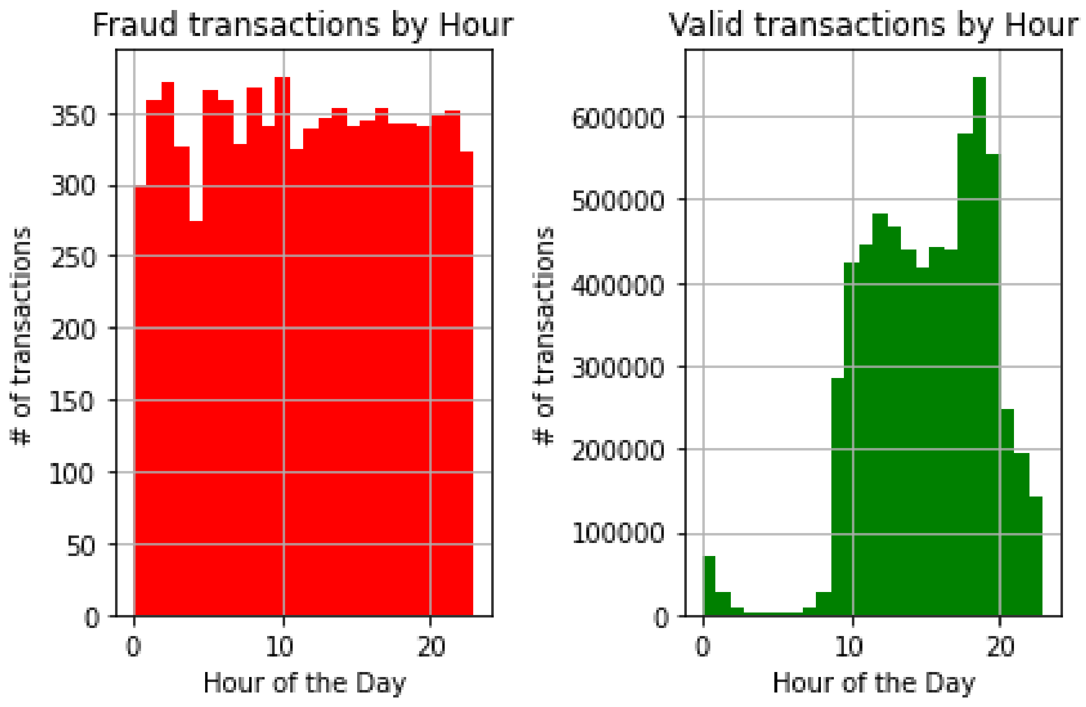


*Figure 25: Get the number of transactions based on hours*

## Data visualisation

In order to see if the Machine Learning algorithms will make strong predictions, the fraudulent and genuine transactions need to be visualized. The scatterplot displays the distribution of between the different transaction types.

Genuine *CASH_OUT* transactions exceed genuine *TRANSFERs*. It tries to make separation between transactions that occur at the same time with different abscissae (X axis). Fraudulent transactions are more homogeneously distributed compared to genuine ones.

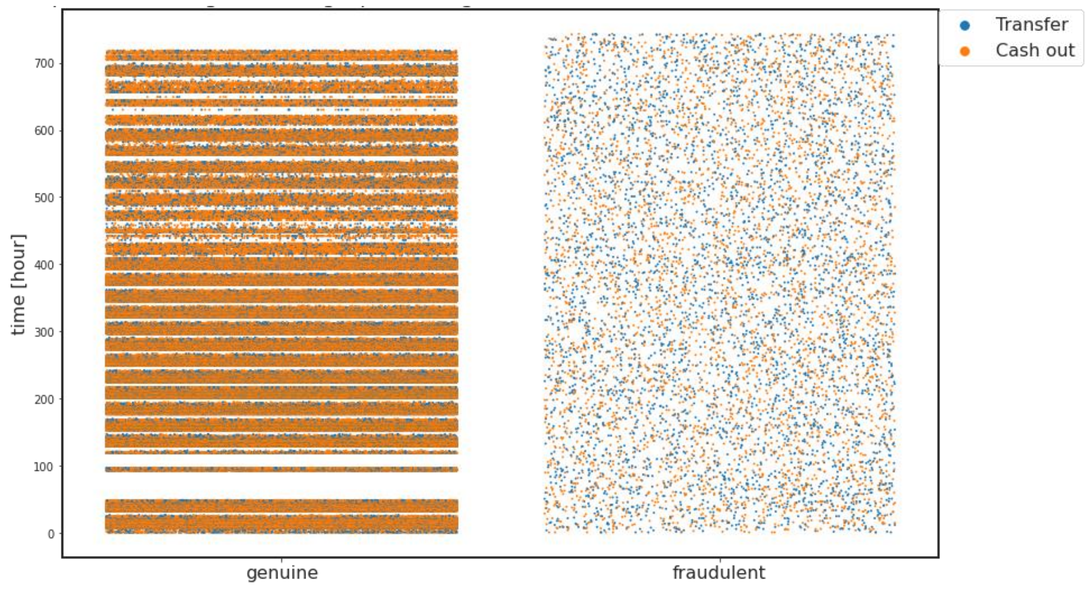


*Figure 26: Display the striped and homogenous fingerprints of genuine and fraudulent transactions over time*

The following scatterplot shows that the *errorBalanceDest* is more effective than *amount* for identifying fraud when a comparison is made between *amount* over genuine transactions and *amount* over fraud transactions.

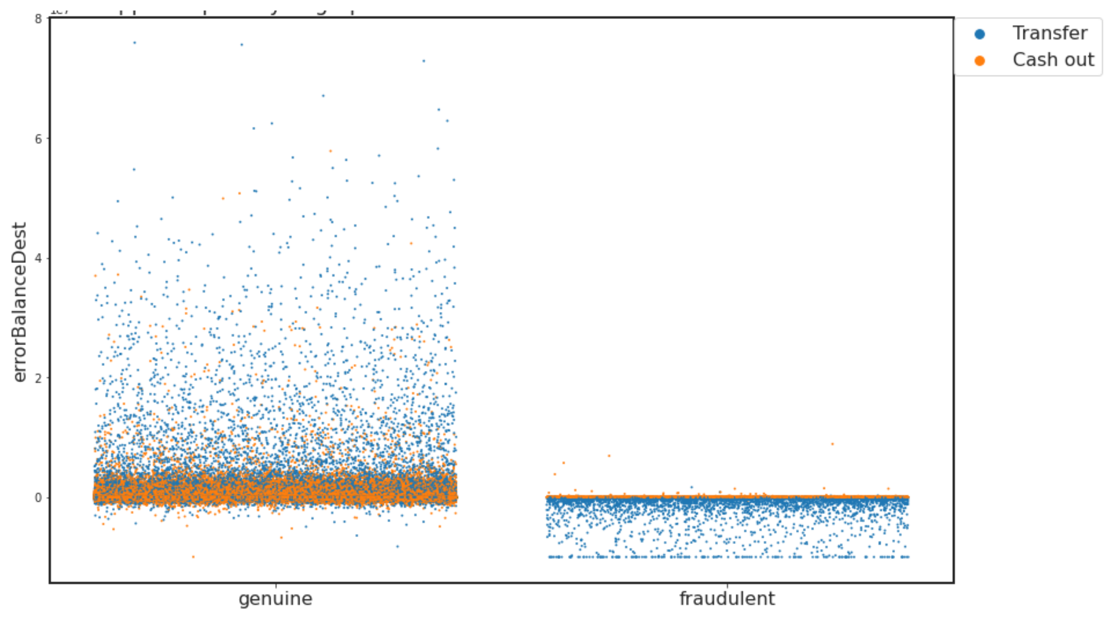


*Figure 27: Get the fingerprints over the error in destination account balances*

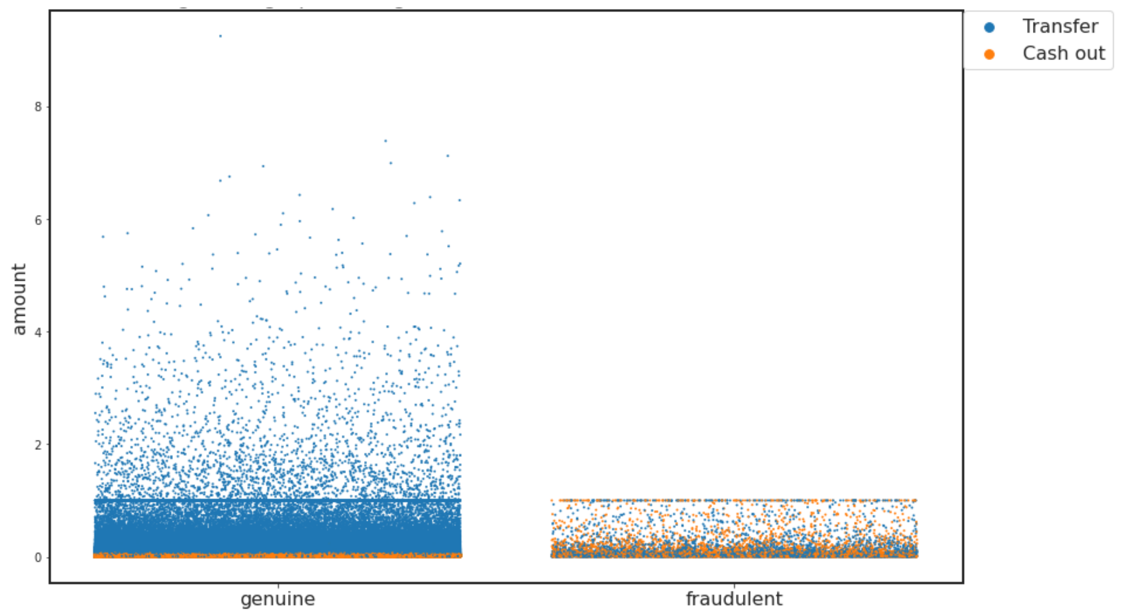


*Figure 28: Fingerprint transactions over amount*

When the fraud and genuine transactions are put on a 3D plot by using *errorbalanceDest* and *errorBalanceOrig*, the initial *step* column is inefficient in separating out fraud.

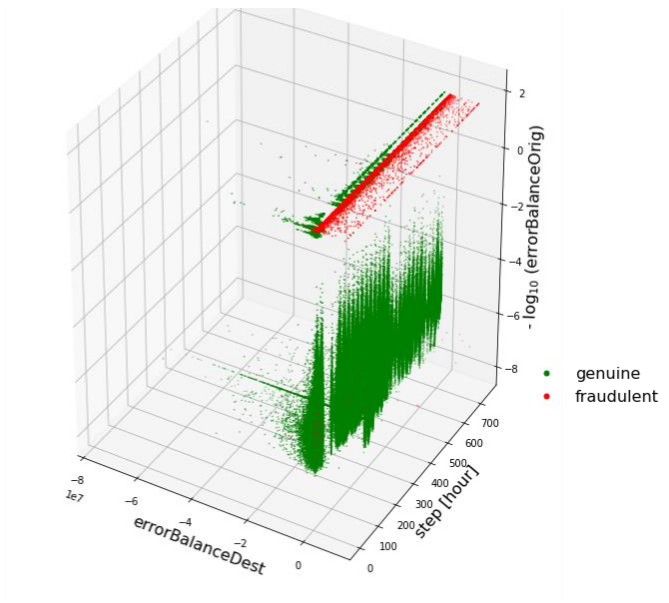


*Figure 29: Error-based feature separating genuine and fraudulent transactions*

The correlation heatmaps below represent the difference between fraudulent and genuine transactions. If a transaction is genuine then it has high correlation with *errorBalanceOrig*. Fraudulent transactions, have a high correlation with *oldBalanceOrig*. The heatmap also highlights the important features. *DayOfWeek* is not as important as it was initially thought to be.

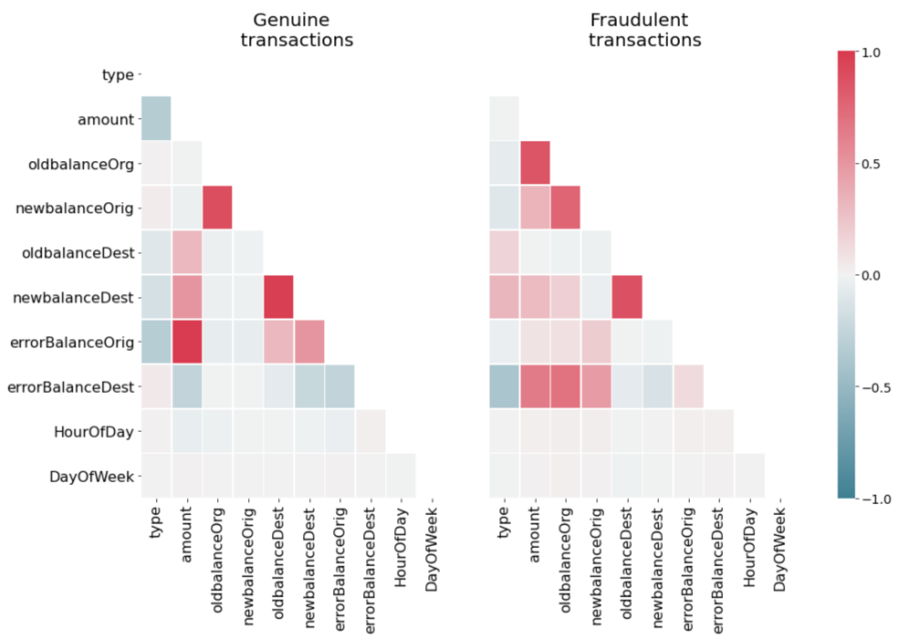


*Figure 30: Heatmap based on column correlation*

## Results

The same hyper parameters are used in the traditional machine learning algorithms in two different ways - class weights and SMOTE. Class weights use the standard PaySim dataset but there is a higher penalization if the algorithm misclassifies a fraudulent transaction. The other way of determining if the algorithm is precise or not is by using SMOTE and creating synthetic fraudulent transactions. In order a model to be considered precise it has to produce good metrics (% of fraud detected, % of fraud missed and average precision-recall score).

In order to split the data into training and testing, the train test split method is used to split the data into two. The training set is 70% of the dataset and 30% is used for testing the algorithm's performance. This prevents the dataset from overfitting and underfitting.

## Random Forests

Random Forests is a supervised machine learning algorithm that is based on ensemble learning. This algorithm combines multiple decision trees, which forms a forest. Initially every tree has the same weight towards the end result. The final result is processed by checking which answer has highest votes. Random Forests can be used for both regression and classification tasks.

Random Forests have a time complexity of *O(t*u*n*log(n))* where:

- **t** - number of trees
- **u** - number of features considered for splitting

The following hyper parameters have been used:

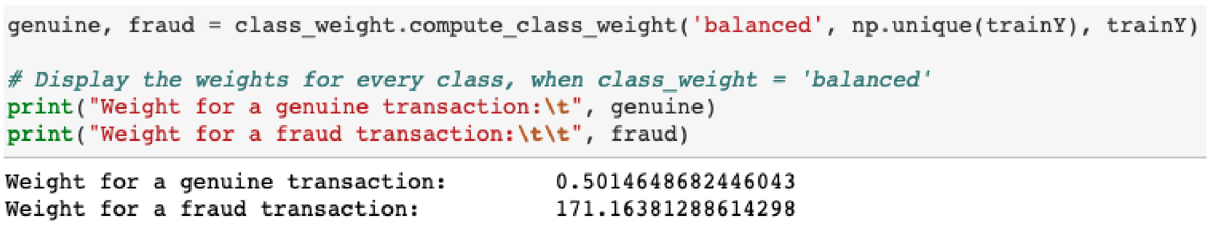
| | |
|---|---|
| **bootstrap** | True |
| **max_depth** | 100 |
| **max_features** | 2 |
| **min_samples_leaf** | 3 |
| **min_samples_split** | 10 |
| **n_estimators** | 200 |
| **random_state** | 10 |
| **verbose** | 1 |

*Table 1: Random Forest parameters*

In order to make the Random forest algorithm take into consideration that the dataset is unbalanced, the *class_weight* hyper parameter needs to be set. Fraud transactions have a higher weight than genuine transactions.

| | |
|---|---|
| **class_weight** | balanced |

*Table 2: Random Forest class weight parameter*



```
genuine, fraud = class_weight.compute_class_weight('balanced', np.unique(trainY), trainY)

# Display the weights for every class, when class_weight = 'balanced'
print("Weight for a genuine transaction:\t", genuine)
print("Weight for a fraud transaction:\t\t", fraud)
```

```
Weight for a genuine transaction:        0.5014648682446043
Weight for a fraud transaction:          171.16381288614298
```

*Figure 31: Weights for genuine and fraudulent transactions (Random Forest)*

The random forest with class weights matrix table barely has any fraudulent transactions that are thought to be genuine. Out of 2548 fraudulent transactions, the algorithm has only misclassified 6 examples. In addition, the second classifier- Average Precision-Recall score also produces nearly perfect score - 0.9976524310789946 out of 1.

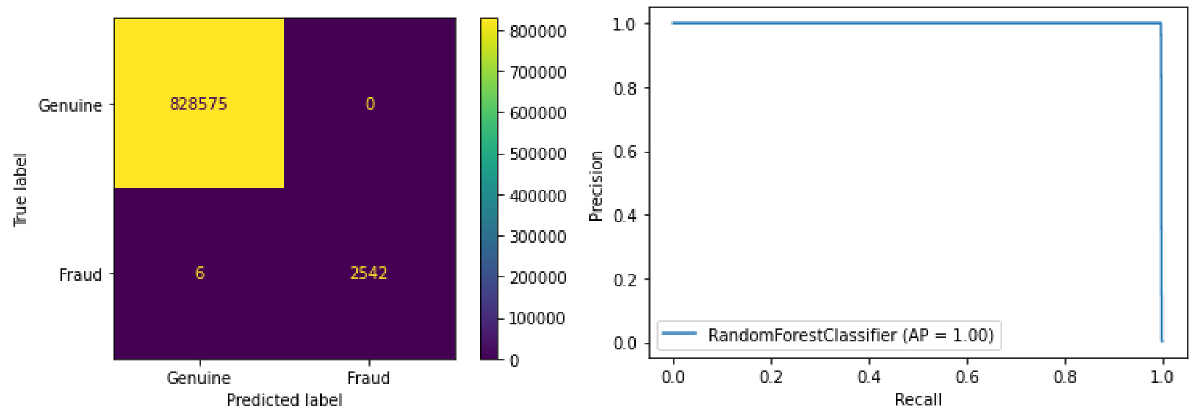
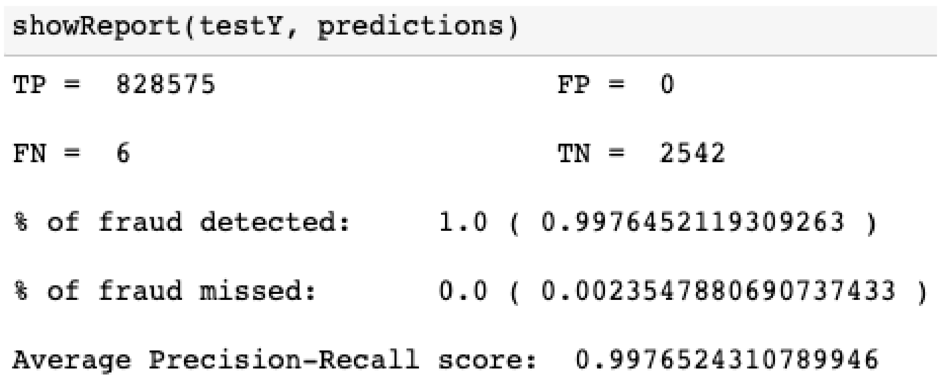


*Figure 32: Confusion matrix and Average Precision-Recall score for Random Forest (class weights)*

Overall class weights capture at least 99.5% of all the fraudulent transactions.



```
showReport(testY, predictions)

TP =  828575                        FP =  0

FN =  6                             TN =  2542

% of fraud detected:     1.0 ( 0.9976452119309263 )

% of fraud missed:       0.0 ( 0.0023547880690737433 )

Average Precision-Recall score:  0.9976524310789946
```

*Figure 33: Get the fraud detected and missed for Random Forest (class weights)*

When SMOTE is used to even out the number of fraudulent transactions to the genuine ones, the results are the same as the *class_weight* ones. There are slightly more false negative transactions and only 3 false positives, but the dataset has rapidly grown in size.

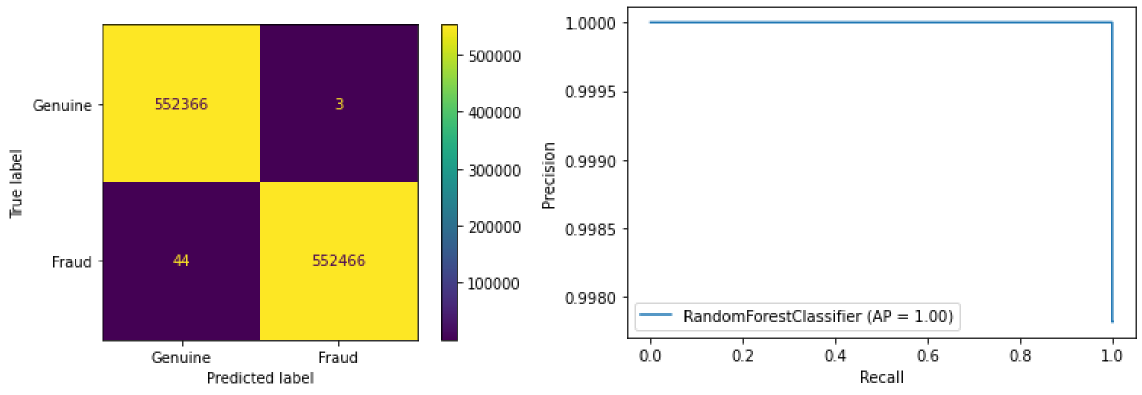
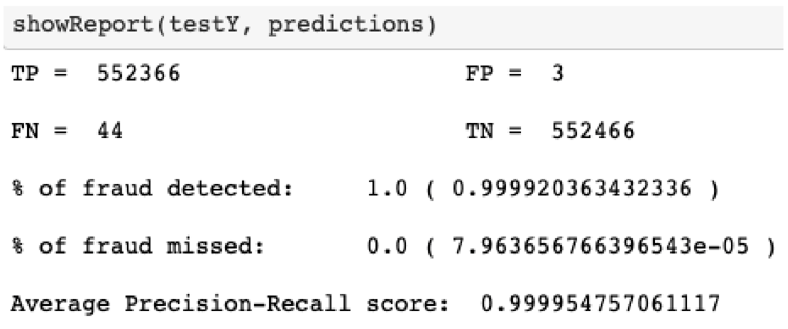


*Figure 34: Confusion matrix and Average Precision-Recall score for Random Forest (SMOTE)*



```
showReport(testY, predictions)

TP =  552366                        FP =  3

FN =  44                            TN =  552466

% of fraud detected:     1.0 ( 0.999920363432336 )

% of fraud missed:       0.0 ( 7.963656766396543e-05 )

Average Precision-Recall score:  0.999954757061117
```

*Figure 35: Get the fraud detected and missed for Random Forest (SMOTE)*

Overall Random forest is a very precise algorithm which gives similar results for both class weights and the SMOTE technique. It

## Extreme Gradient Boosting

Boosting is used to in supervised learning. It is an ensemble algorithm that combines weak learners to form a strong learner and increase the accuracy of the model. Rules help identify a specific class. If a class is identified on an individual learner, then the prediction is flawed. Multiple rules are used before determining the class of the final output.

In Gradient Boosting, the base learners are generated sequentially that the current base learner is more effective from its predecessor. The loss function is optimized of the previous learner. Extreme Gradient Boosting has high speed and performance. It is used for regression and classification problems. XGBoost is used as a starting point to make accurate predictions for other models. This algorithm supports:

- Parallelization - the algorithm uses maximum available computational power
- Cache optimization - keeps its calculations in the cache and fetches it fast to perform calculations
- Out of memory computation - XGBoost can work with data that is more than the available space.
- Regularization - prevents the model of overfitting
- Missing values - takes care of missing values in a dataset

XGBoost has a time complexity of *O(t*d*x*log(n))* where:

- **t** - number of trees
- **d** - height of trees
- **x** - is the number of non-missing entries in the training data

A new sample takes *O(t*d)* time. The following hyper parameters have been used:

| | |
|---|---|
| **n_estimators** | 100 |
| **learning_rate** | 0.01 |
| **subsample** | 0.3 |
| **max_depth** | 5 |
| **colsample_bytree** | 0.5 |
| **min_child_weight** | 3 |

*Table 3: XGBoost parameters*

In order to make the Random forest algorithm take into consideration that the dataset is unbalanced, the *scale_pos_weight* hyper parameter needs to be set, giving fraud transactions bigger weight than genuine transactions.

| | |
|---|---|
| **scale_pos_weight** | (Y == 0).sum() / (1.0 * (Y == 1).sum()) |

*Table 4: XGBoost class weight parameter*

The algorithm produces almost the same amount of False Negative transactions such as Random Forests, however XGBoost returns more false positive examples compared to Random Forests. This is less crucial because genuine transactions are considered to be fraudulent, meaning that the customer would have to contact the bank to confirm that the transaction is legitimate.

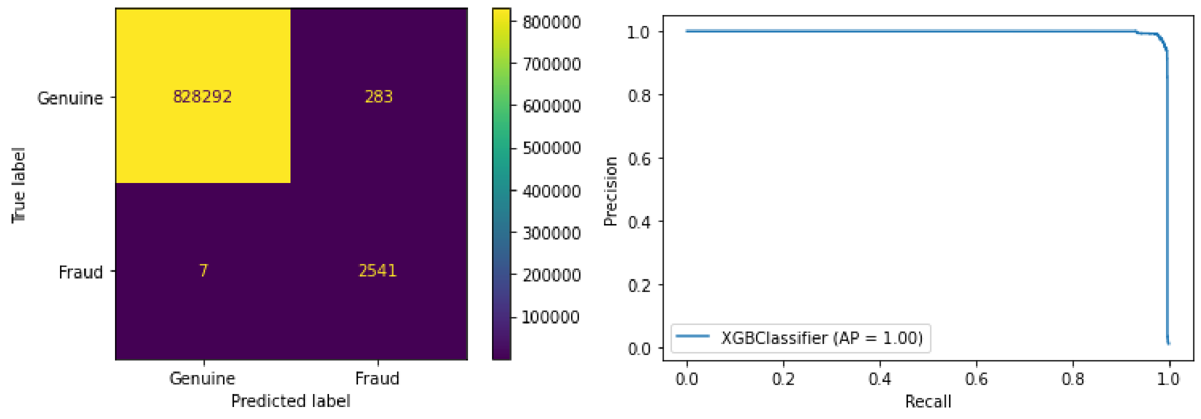
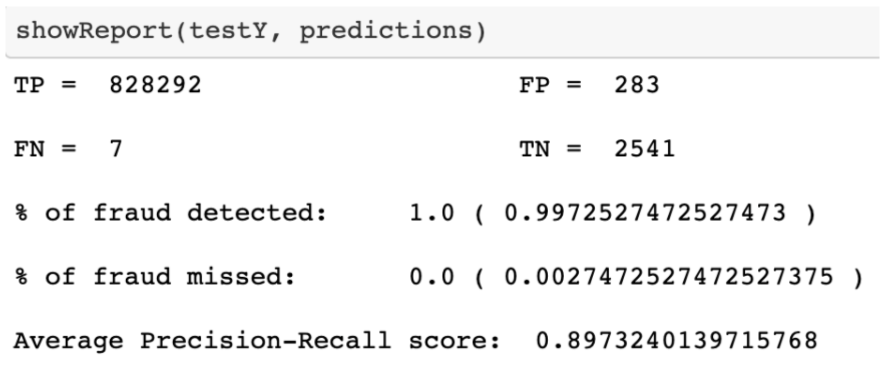


*Figure 36: Confusion matrix and Average Precision-Recall score for XGBoost (class weights)*

Although the average precision score is lower than the Random forest one, XGBoost with class weights still manages to correctly to identify fraudulent transactions correctly and captures 99.7% of the fraudulent transactions.



```
showReport(testY, predictions)

TP =  828292                         FP =  283

FN =  7                              TN =  2541

% of fraud detected:     1.0 ( 0.9972527472527473 )

% of fraud missed:       0.0 ( 0.0027472527472527375 )

Average Precision-Recall score:  0.8973240139715768
```

*Figure 37: Get the fraud detected and missed for XGBoost (class weights)*

Even when using the SMOTE technique XGBoost manages to capture at least 99.6% of the fraud. Although there are more False Negative transactions, the algorithm has classified majority of the fraudulent and genuine transactions correctly. It has misclassified less false positive transactions which is less crucial.

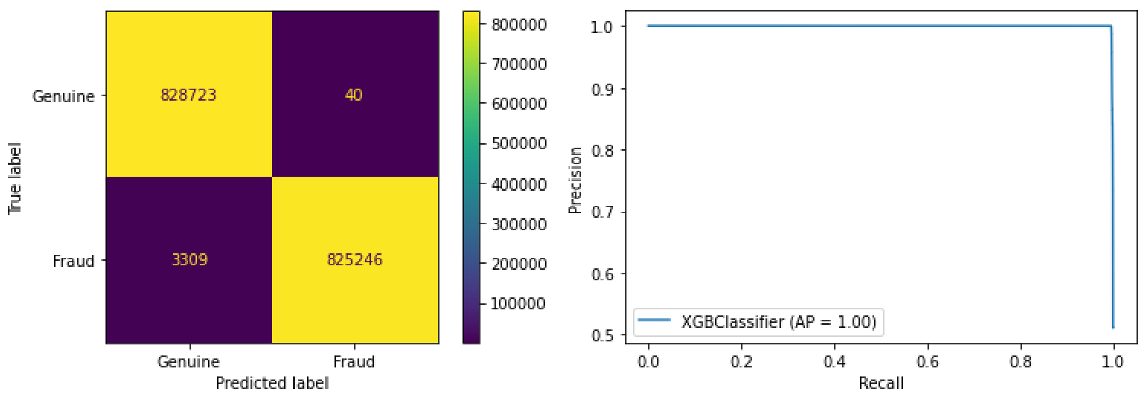
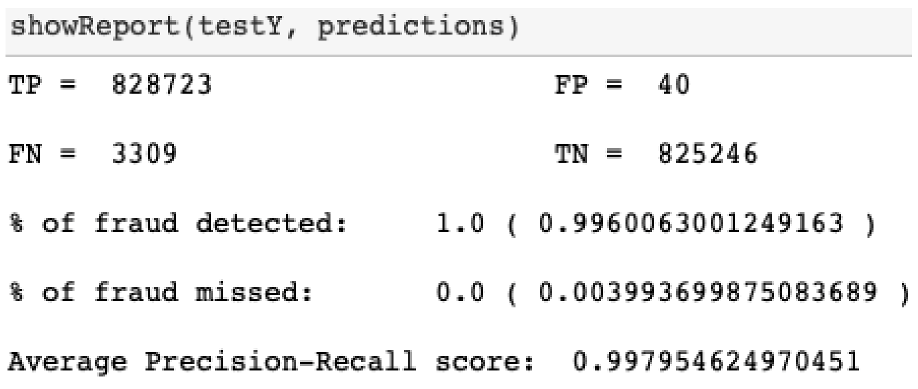


*Figure 38: Confusion matrix and Average Precision-Recall score for XGBoost (SMOTE)*



```
showReport(testY, predictions)

TP =   828723                       FP =   40

FN =   3309                         TN =   825246

% of fraud detected:     1.0 ( 0.9960063001249163 )

% of fraud missed:       0.0 ( 0.003993699875083689 )

Average Precision-Recall score:  0.997954624970451
```

*Figure 39: Get the fraud detected and missed for XGBoost (SMOTE)*

Overall XGBoost is a reliable algorithm that performs well both with class weights and SMOTE, but the class weights example gives a slightly better performance.

## Logistic Regression

This algorithm is similar to Linear Regression, but Logistic Regression classifies binary dependent variables, rather than predicting continuous data. Instead of fitting a line to the data, Logistic Regression fits a S shaped logistic function. It predicts if something is true or false instead of predicting continuous data like linear regression. The curve tells the probability of the transaction to be fraudulent or genuine by using the maximum likelihood.

The time complexity is *O(n*d)* where:

- n - number of training examples
- d - number of dimensions of the data

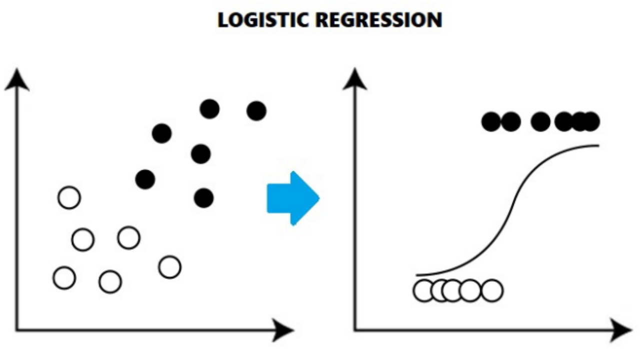


*Figure 40: Transforming data into a Logistic Regression model*

The following hyper parameters have been used:

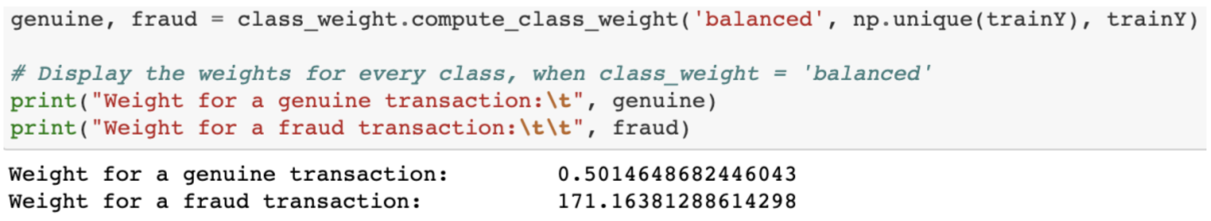
| | |
|---|---|
| **C** | 3 |
| **Penalty** | 11 |
| **solver** | liblinear |

*Table 5: Logistic Regression parameters*

When using class weights fraudulent transactions have a higher priority than genuine transactions.

|  |  |
|---|---|
| **class_weight** | balanced |

*Table 6: Logistic Regression class weight parameter*



```
genuine, fraud = class_weight.compute_class_weight('balanced', np.unique(trainY), trainY)

# Display the weights for every class, when class_weight = 'balanced'
print("Weight for a genuine transaction:\t", genuine)
print("Weight for a fraud transaction:\t\t", fraud)
```

```
Weight for a genuine transaction:        0.5014648682446043
Weight for a fraud transaction:          171.16381288614298
```

*Figure 41: Weights for genuine and fraudulent transactions (Logistic Regression)*

Logistic Regression with weights identified 90% of the fraudulent transactions but the Average Precision-Recall score is very low - 0.042. This algorithm has the highest amount of False Negative transactions compared to Random Forest and XGBoost.

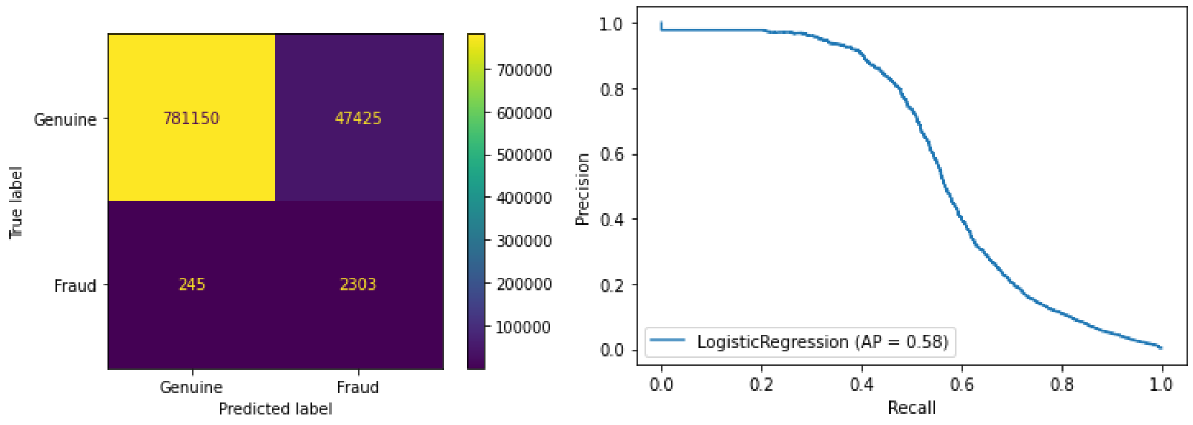
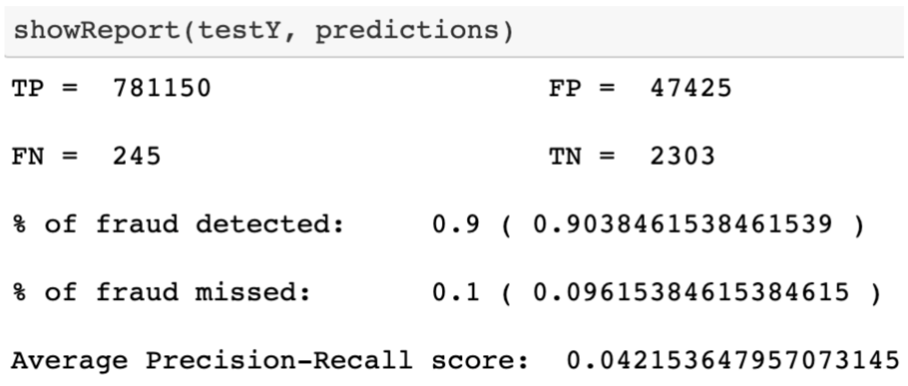


*Figure 40: Confusion matrix and Average Precision-Recall score for Logistic Regression (class weights)*



```
showReport(testY, predictions)

TP =   781150                        FP =   47425

FN =   245                           TN =   2303

% of fraud detected:      0.9 ( 0.9038461538461539 )

% of fraud missed:        0.1 ( 0.09615384615384615 )

Average Precision-Recall score:   0.042153647957073145
```

*Figure 41: Get the fraud detected and missed for Logistic Regression (class weights)*

However, when the SMOTE technique is used, Logistic Regression has a slightly higher fraud detection percentage but has misclassified more fraudulent transactions as genuine than genuine transaction identified as fraudulent.

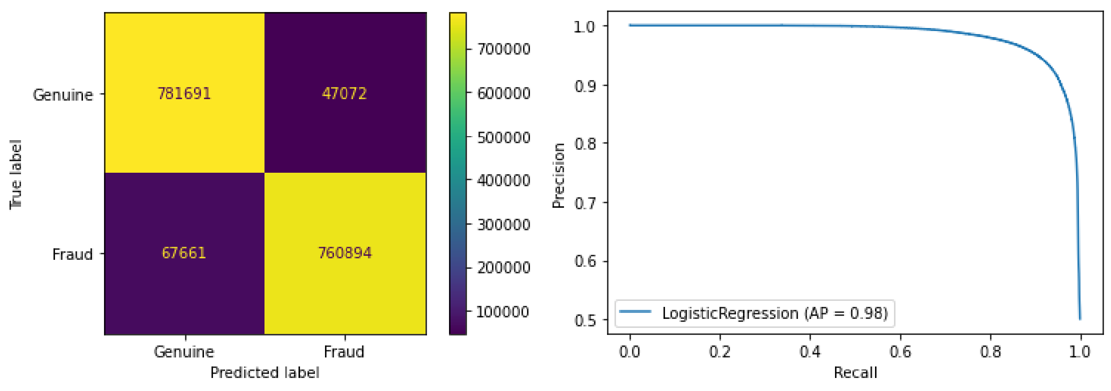
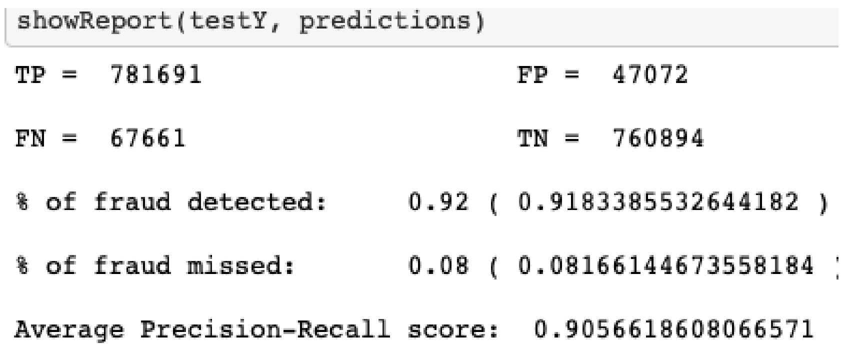


*Figure 42: Confusion matrix and Average Precision-Recall score for Logistic Regression (SMOTE)*



```
showReport(testY, predictions)

TP =   781691                        FP =   47072

FN =   67661                         TN =   760894

% of fraud detected:      0.92 ( 0.9183385532644182 )

% of fraud missed:        0.08 ( 0.08166144673558184

Average Precision-Recall score:   0.9056618608066571
```

*Figure 43: Get the fraud detected and missed for Logistic Regression (SMOTE)*

Logistic Regression with class weights and the SMOTE technique overall performs poorer compared to the XGBoost and Random Forest algorithms.

## Naïve Bayes

Naive Bayes works on the principles of conditional probability as given by Bayes' Theorem. It calculates the conditional probability of the occurrence of an event based on prior knowledge of conditions that might be related to the event.

$$P(A|B) = \frac{P(A \bigcap B)}{P(B)} = \frac{P(A) \cdot P(B|A)}{P(B)}$$

**where:**

$P(A) =$ The probability of A occurring

$P(B) =$ The probability of B occurring

$P(A|B) =$ The probability of A given B

$P(B|A) =$ The probability of B given A

$P\left(A \bigcap B\right)) =$ The probability of both A and B occurring

*Figure 44: Naïve Bayes equation*

The Gaussian Naive Bayes is a variant of Naive Bayes that follows the Gaussian normal distribution and supports continuous data. Out of the different Naive Bayes types, Gaussian Naive Bayes is used for Fraud Detection.

$$P(x_i \mid y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right)$$

*Figure 45: Gaussian Naïve Bayes equation*

The time complexity is *O(n*d)* where:

- **n** - number of training examples
- **d** - number of dimensions of the data

Gaussian Naïve Bayes does not have class weights and calculates if a transaction is fraudulent or genuine by using the latter equation. The algorithm has misclassified more fraudulent transactions as genuine than correctly classified fraudulent transactions. It has detected 40% of the fraudulent transactions and has a very poor Average Precision-Recall score (0.04). Even with SMOTE Gaussian Naïve Bayes has misclassified more False Negative transactions than True Negative ones.

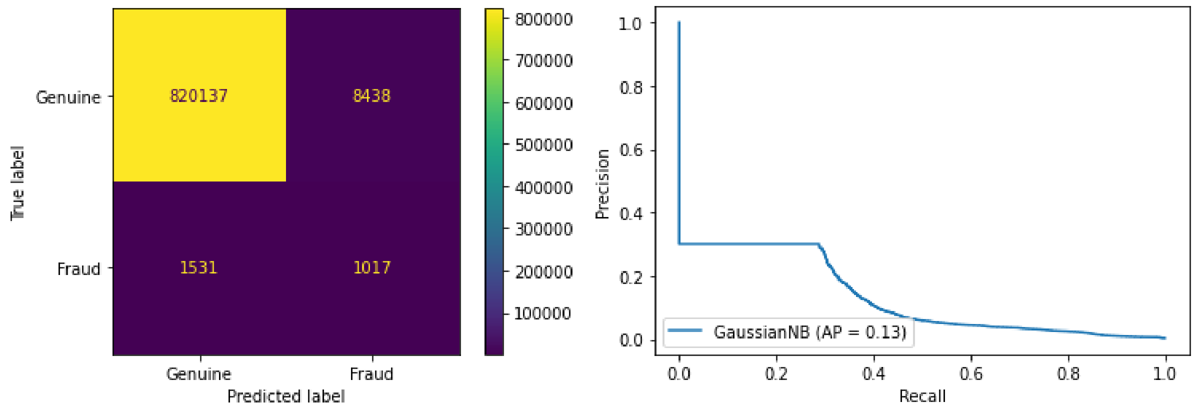
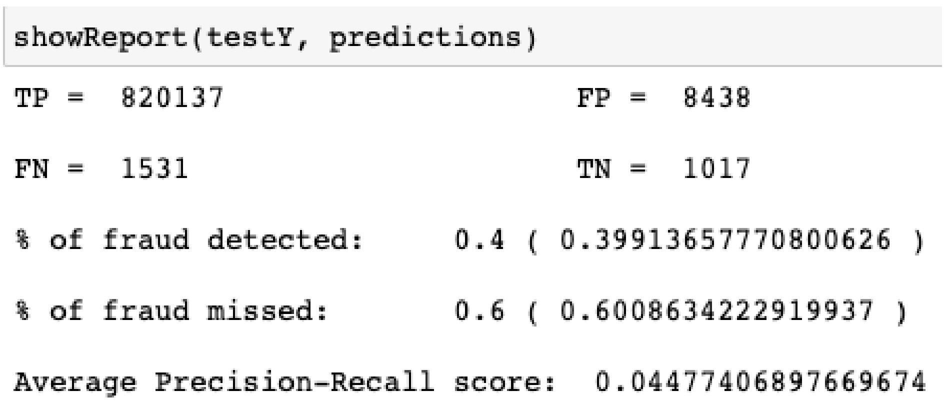


*Figure 46: Confusion matrix and Average Precision-Recall score for Gaussian Naïve Bayes*



```
showReport(testY, predictions)

TP =  820137                        FP =  8438

FN =  1531                          TN =  1017

% of fraud detected:      0.4 ( 0.39913657770800626 )

% of fraud missed:        0.6 ( 0.6008634222919937 )

Average Precision-Recall score:  0.04477406897669674
```

*Figure 47: Get the fraud detected and missed for Gaussian Naïve Bayes*

When using SMOTE to create more fraudulent transactions the algorithm does not performs slightly better but not enough to be used. It has correctly classified 46% of the fraud and the Average Precision-Recall score is higher (0.71).

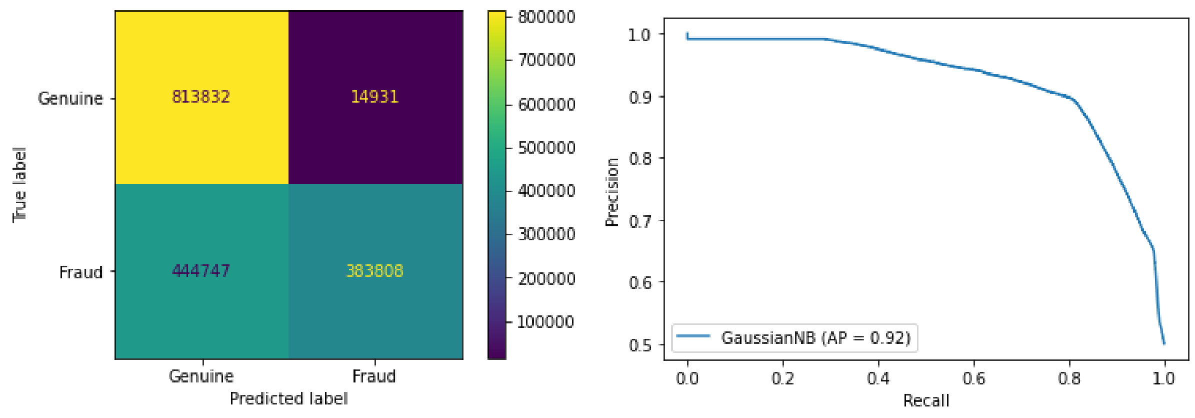
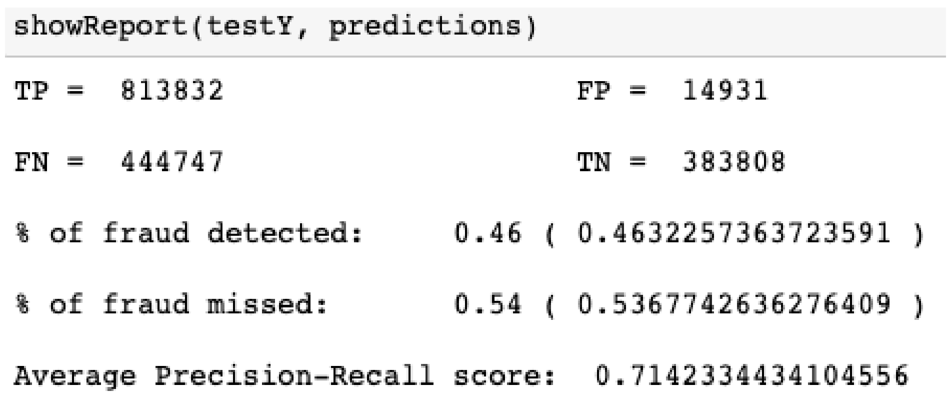


*Figure 48: Confusion matrix and Average Precision-Recall score for Gaussian Naïve Bayes (SMOTE)*



```
showReport(testY, predictions)

TP =   813832                    FP =   14931

FN =   444747                    TN =   383808

% of fraud detected:      0.46 ( 0.4632257363723591 )

% of fraud missed:        0.54 ( 0.5367742636276409 )

Average Precision-Recall score:   0.7142334434104556
```

*Figure 49: Get the fraud detected and missed for Gaussian Naïve Bayes (SMOTE)*

In general Gaussian Naïve Bayes performs worse than Logistic Regression and does not give reliable results.

## K-Nearest Neighbour

K-Nearest Neighbours is a simple supervised machine learning algorithm that is used to solve classification and regression problems. This algorithm identifies the k nearest neighbours of new unlabelled data. K is the number of nearest neighbours that will be considered when deciding the data's class. A good practice is to choose an odd K number so when deciding the class of the unlabelled data, one class has more votes than the other by one. This algorithm is very easy to use and does not require a lot of maths. However, the cons are that it is memory intensive, new predictions take longer time and there is no real pre-processing. The bigger the K parameter is, the more time it will take to compute the label.

The time complexity is *O(k*n*d)* where:

- k - number of neighbours
- n - number of training examples
- d - number of dimensions of the data

Like Gaussian Naïve Bayes, K-Nearest Neighbour does not have class weights. The following hyper parameters have been used:

| | |
|---|---|
| **n_neighbors** | 5 |
| **weights** | distance |
| **p** | 2 |

*Table 7: Logistic Regression parameters*

The algorithm on the standard dataset performs significantly better than Logistic Regression and Gaussian Naïve Bayes but it still has a low fraud detected percentage (69%) compared to XGBoost and Random Forest. By itself this algorithm is not sufficient enough to correctly identify fraud.

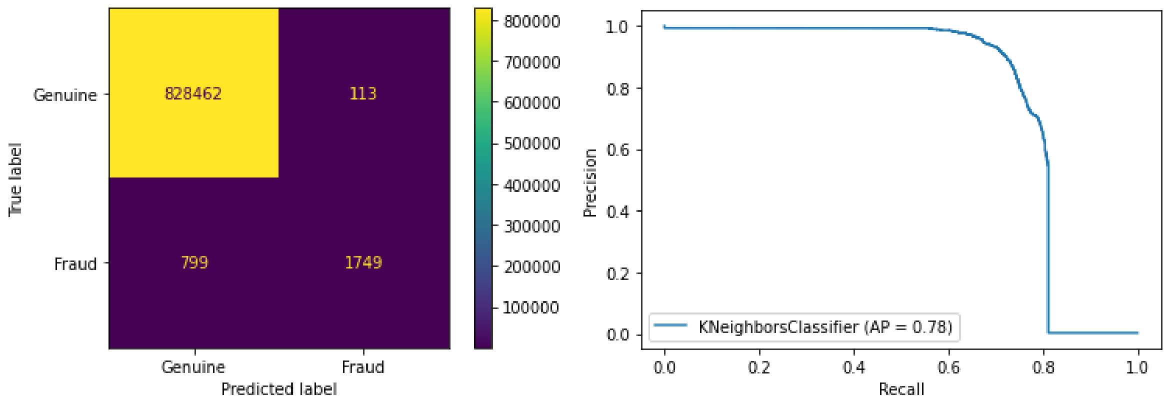
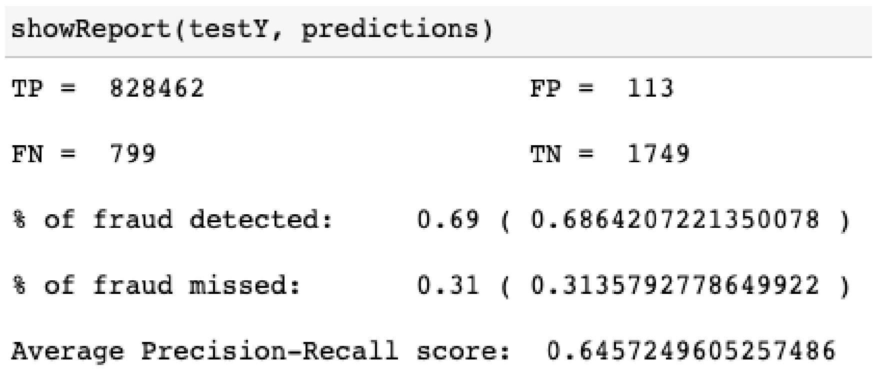


*Figure 50: Confusion matrix and Average Precision-Recall score for KNN*



```
showReport(testY, predictions)
TP =  828462                        FP =  113

FN =  799                           TN =  1749

% of fraud detected:      0.69 ( 0.6864207221350078 )

% of fraud missed:        0.31 ( 0.3135792778649922 )

Average Precision-Recall score:  0.6457249605257486
```

*Figure 51: Get the fraud detected and missed for KNN*

However, when the SMOTE technique is used the results change rapidly. There are lesser False Negative classifications and majority of the transactions have been classified correctly. The algorithm has managed to detect 99% of the fraud and the Average Precision-Recall score is higher than the result without SMOTE.

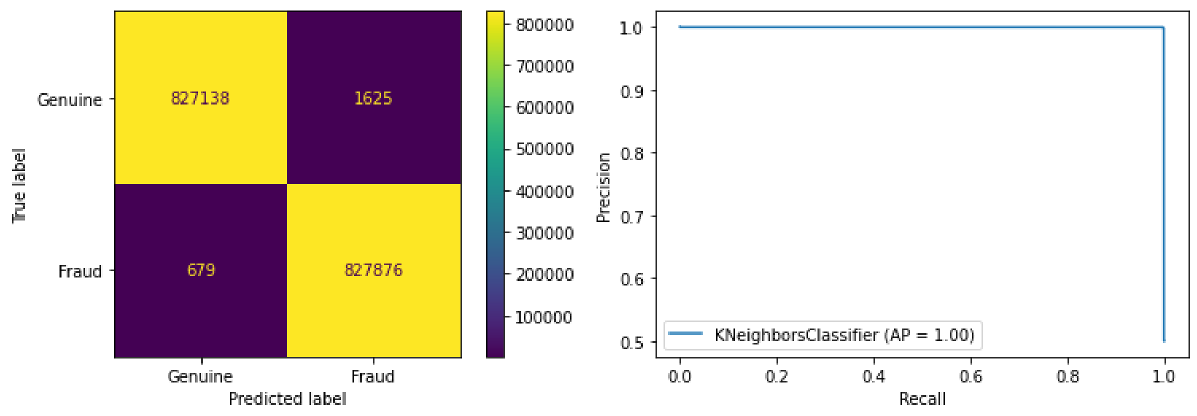
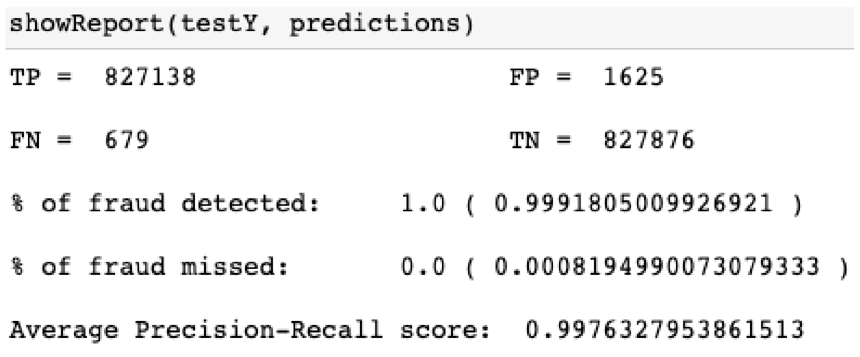


*Figure 52: Confusion matrix and Average Precision-Recall score for KNN (SMOTE)*



```
showReport(testY, predictions)
TP =  827138                          FP =  1625

FN =  679                             TN =  827876

% of fraud detected:     1.0 ( 0.9991805009926921 )

% of fraud missed:       0.0 ( 0.0008194990073079333 )

Average Precision-Recall score:  0.9976327953861513
```

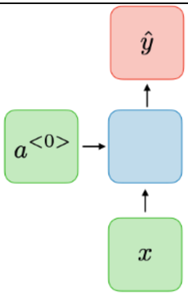
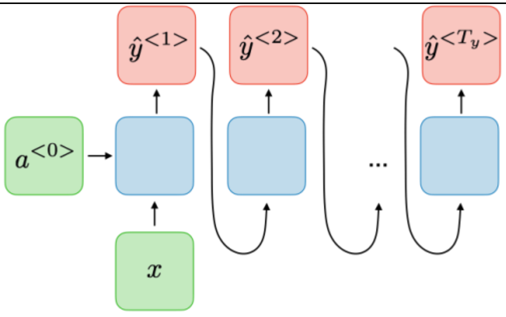
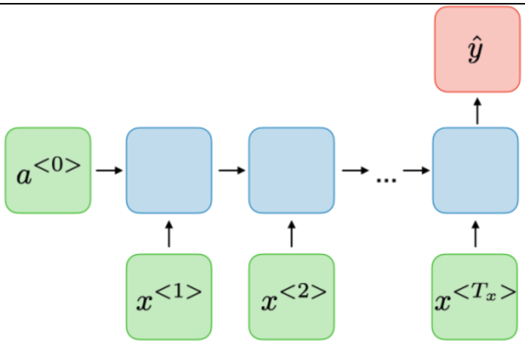
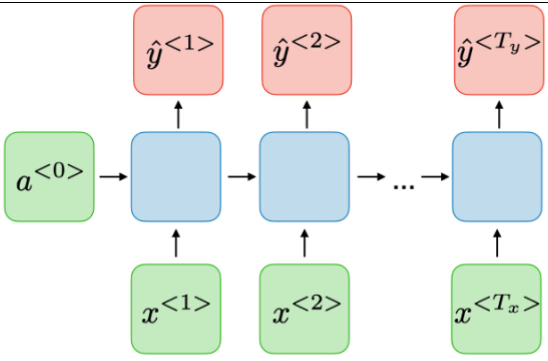
*Figure 52: Get the fraud detected and missed for KNN (SMOTE)*

When using KNN with SMOTE the algorithm can correctly classify what a fraudulent and genuine transaction is, however, the computation is very slow, compared to the previously mentioned algorithms.

## Recurrent Neural Networks

Neural Networks are the most promising fraud detection solution because it can work in real time and does not take a long time to process newly incoming transactions. However Neural Networks take the longest to train but give the most precise results. For fraud detection Neural Networks by itself will not be enough - there is too much information to be processed and a single pass through the neural network will not be enough. On the other hand, Recurrent Neural Networks are Neural Networks that have a feedback loop where data can be fed back as input at some point before it is fed forward again for further processing and final output. This feature is useful because more data can be taken into consideration such as how much time the user spent making the transaction and looking at different customer patterns to determine if a transaction is fraudulent or genuine.

There are 4 different types of Recurrent Neural Networks:

| **RNN Type** | Illustration | **Example** |
|---|---|---|
| One to one<br><br>$T_x = T_y = 1$ |  | **Traditional neural network**<br><br>There is one input and one output |
| One to many<br><br>$T_x = 1, T_y > 1$ |  $\hat{y}^{<1>}$<br>$\hat{y}^{<2>}$<br>$\hat{y}^{<T_y>}$<br>$a^{<0>}$<br>...<br>$x$ | **Music generation**<br><br>There is one input and multiple outputs |
| Many to one<br><br>$T_x > 1, T_y = 1$ |  $\hat{y}$<br>$a^{<0>}$<br>...<br>$x^{<1>}$<br>$x^{<2>}$<br>$x^{<T_x>}$ | **Sentiment classification**<br><br>There is more than one inputs and one input |
| Many to many<br><br>$T_x = T_y$ |  $\hat{y}^{<1>}$<br>$\hat{y}^{<2>}$<br>$\hat{y}^{<T_y>}$<br>$a^{<0>}$<br>...<br>$x^{<1>}$<br>$x^{<2>}$<br>$x^{<T_x>}$ | **Name entity**<br><br>There is an equal number of inputs and outputs |

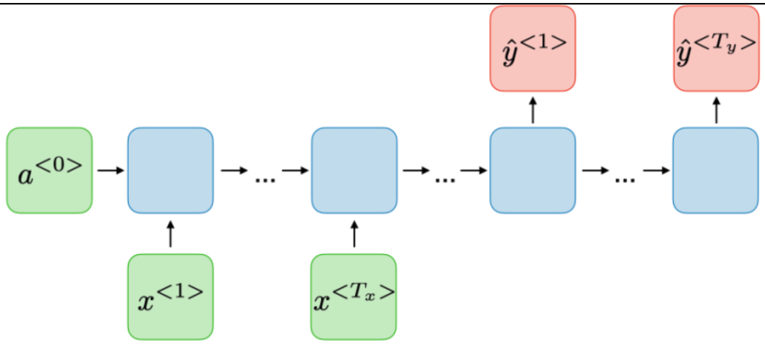
| | | |
|---|---|---|
| Many to many<br><br>$T_x \neq T_y$ |  $\hat{y}^{<1>}$<br>$\hat{y}^{<T_y>}$<br>$a^{<0>}$<br>$x^{<1>}$<br>$x^{<T_x>}$ | **Machine translation**<br><br>There is a different number of inputs and outputs |

*Table 8: Different types of Recurrent Neural Networks*

For fraud detection the Many to one type will be used because there are multiple inputs and a single output - either a true or false value to determine if the transaction is fraudulent or not.

In order to get the correct weights, a precise loss function is required to calculate the loss at every point in time. Mean Squared Error is a good starting point but not the best loss function because it equally penalizes the model for misclassifying a label. However Binary Cross Entropy penalizes the model a lot if it is incorrect. If there is a mistake the value of the gradient will increase rapidly.

To get more accurate and faster results the data needs to be scaled down. There are two different ways of scaling data - by using standardization or by using normalization. Normalization scales the data's features between 0 and 1 and does not follow the Gaussian distribution whereas standardization scales the features between -1 and 1 and follows the Gaussian distribution.

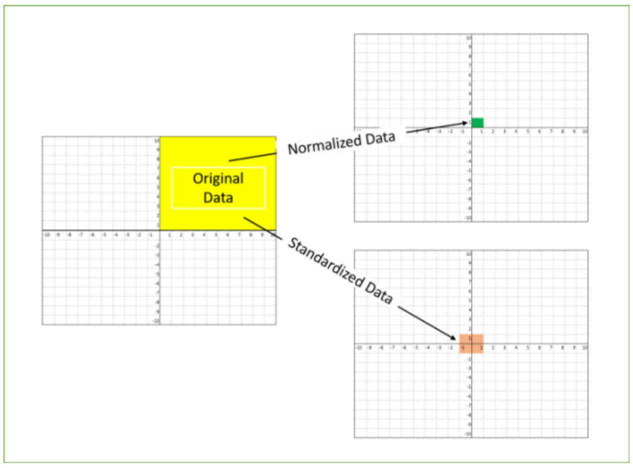


*Figure 53: Show the difference between Normalized and Standardized data*

Scaling down the data helps when calculating the gradient descent. When the data is scaled down, the model can minimize the gradient faster and smoother rather than when the data is not scaled down. The gradient descent does backpropagation so the weights of the networks can be changed for every epoch.

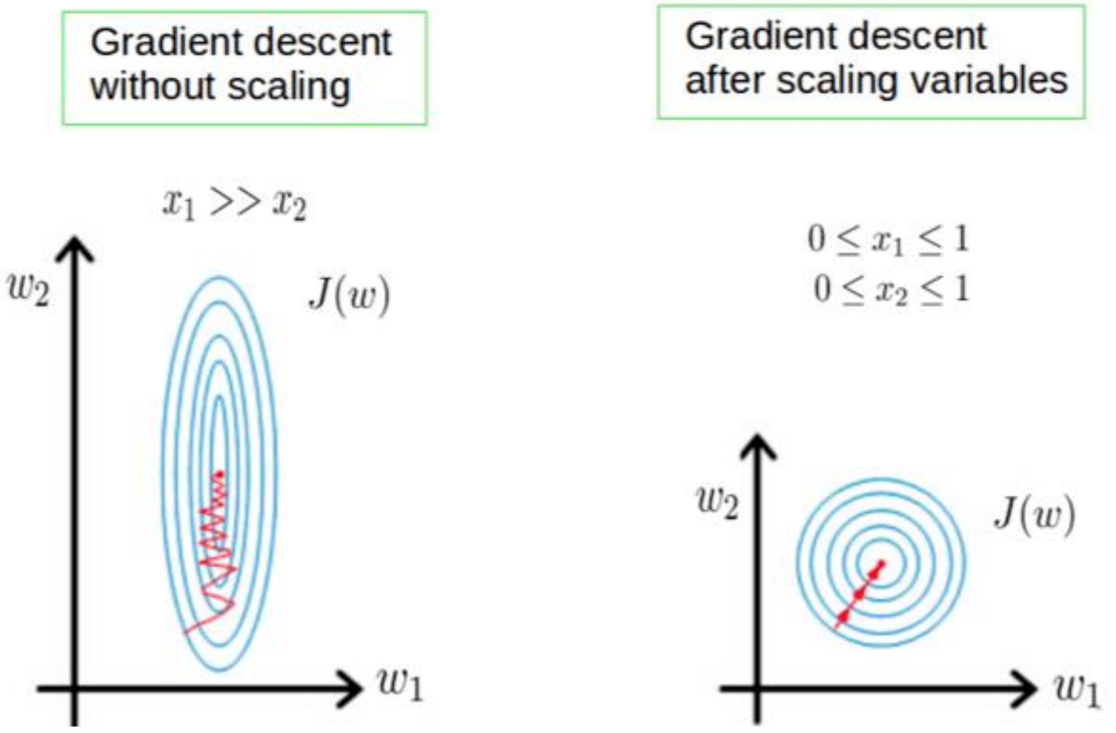


*Figure 54: Show the difference in Gradient descent between with and without scaling*

Without an activation function at the end of the neural network, a neural network would just be a linear regression model. Activation functions converts the output to a non-linear one and can do more complicated tasks. The three commonly used activation functions are Sigmoid, Tanh and RELU.

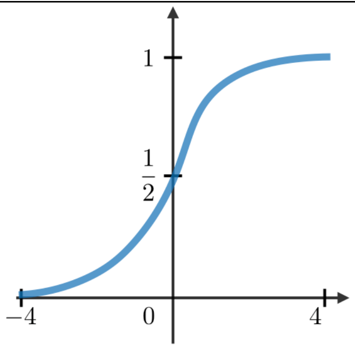
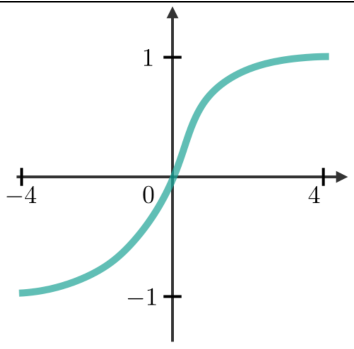
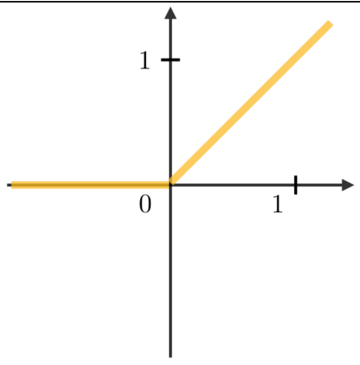
| **Sigmoid** | **Tanh** | **RELU** |
|---|---|---|
| $g(z) = \frac{1}{1+ e^{-z}}$ | $g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$ | $g(z) = \max(0, z)$ |
|  |  |  |
| Produces an output between 0 and 1 | Produces an output between -1 and 1 | Produces an output of 0 if the input is negative or the value of z |

*Table 9: Common activation functions*

Sigmoid works best for classification problems and will be used in the following models to squish the network's calculations.

Neural Networks are beneficial in fraud detection because they can catch a fraudulent transaction the fastest however, they require the most amount of data, modifications and training time in order to produce accurate results.

Another hyperparameter that has to be taken into consideration is the batch size. It is responsible for controlling the accuracy of estimating the gradient's error when the network is training. The smaller the batch size, the more network tweaks are made. A small batch size can skip the minimum gradient descent value whereas a larger batch could possibly never find.

Finally, are important because it indicates the number of iterations that the model goes over the dataset. If there are too many epochs the model can memorise the data and perform poorly when new data is fed to the model whereas not enough epochs could lead to not fully understanding the data and its patterns.

Dropout is used in the following models, so the data does not overfit, meaning that instead of understanding the data set, the algorithm will memorise it, leading to perfect training results and very poor testing ones. During training random output layers will be completely ignored and the output will float through the other ones. This way the model will not rely on specific paths to feed the next layer.

Finally, some models are using Long Short-Term Memory (LSTM) architecture instead of the main Recurrent Neural Network one. This helps the model to selectively control the flow of information. Different gates either add or remove the data to the next output layer.

All of the following models have been given only the SMOTE variant of the dataset.

## Model 1

This model is based on the Bandyopadhyay and Dutta (2020) article that suggest a 12-layer Recurrent Neural network. However, they use MSE, Accuracy and F1-Score to measure the model's performance. In this project I will be using Binary Cross Entropy, % of fraud detected and Average Precision-Recall score. The initial proposed model is:

| Layer | Output Shape | Parameters | Activation Function |
|---|---|---|---|
| Simple RNN | (None, 10, 128) | 16640 | Sigmoid |
| Dropout | 0.2 | 0 | None |
| Simple RNN | (None, 10, 64) | 12352 | Sigmoid |
| Dropout | 0.2 | 0 | None |
| Simple RNN | (None, 10, 32) | 3104 | Sigmoid |
| Dropout | 0.2 | 0 | None |
| Simple RNN | (None, 10, 16) | 784 | Tanh |
| Dropout | 0.2 | 0 | None |
| Dense | (None, 10, 8) | 136 | None |
| Dense | (None, 10, 4) | 36 | None |
| Dense | (None, 10, 2) | 10 | None |
| Dense | (None, 10, 1) | 3 | Sigmoid |

*Table 10: Layers of model 1*

This model has completed 2 epochs with a batch size of 64. A higher batch size rapidly can reduce the training time of the model and a small batch size can increase it.

After 2 iterations over the dataset, the model's loss falls just below 0.40 but it poorly performs to correctly identify transactions. Considering the dataset is 5,524,392 transactions (with SMOTE) - an equal number of fraudulent and genuine transactions, it has managed to identify half of the fraudulent transactions as fraudulent and misclassified the rest as genuine ones. The loss has barely fallen to 40%.

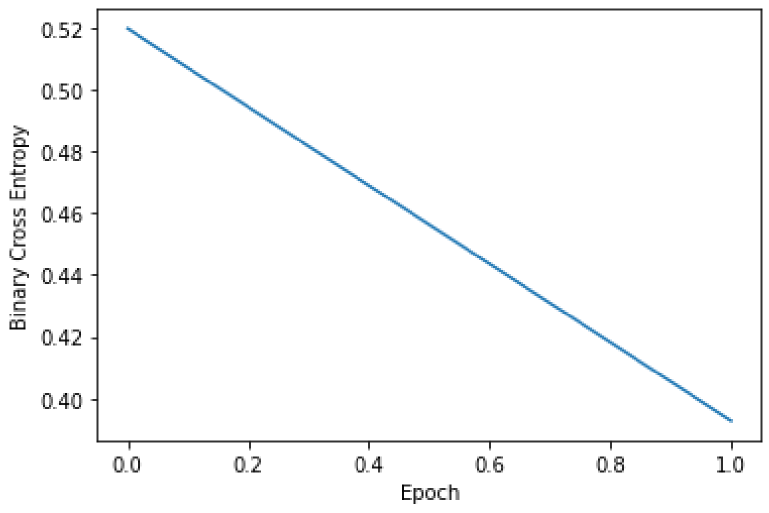
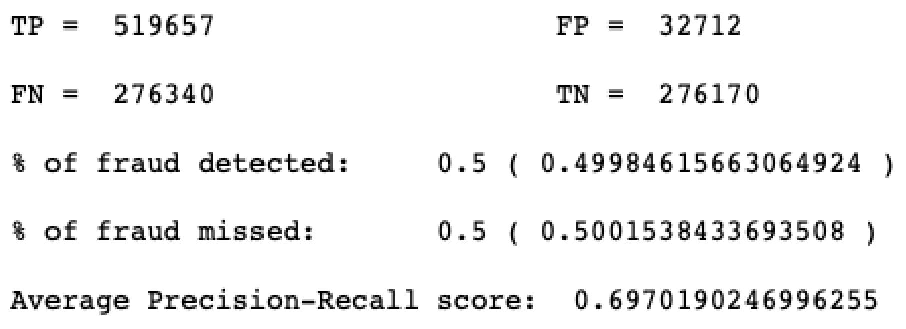


*Figure 55: Loss over time for model 1*



```
TP =  519657                    FP =  32712

FN =  276340                    TN =  276170

% of fraud detected:     0.5 ( 0.49984615663064924 )

% of fraud missed:       0.5 ( 0.5001538433693508 )

Average Precision-Recall score:  0.6970190246996255
```

*Figure 56: Get the fraud detected and missed for RNN model 1 (SMOTE)*

### *Model 2*

Model 1 is a good starting point, but the model could possibly forget important information due to too much data. This model uses LSTMs to tackle the latter mentioned problem. Now data that contributes to the final decision will be taken into consideration when the data is passed from one layer to the other.

| **Layer** | **Output Shape** | **Parameters** | **Activation Function** |
|---|---|---|---|
| LSTM | (None, 10, 128) | 16640 | Sigmoid |
| Dropout | 0.2 | 0 | None |
| LSTM | (None, 10, 64) | 12352 | Sigmoid |
| Dropout | 0.2 | 0 | None |
| LSTM | (None, 10, 32) | 3104 | Sigmoid |
| Dropout | 0.2 | 0 | None |
| LSTM | (None, 10, 16) | 784 | Tanh |
| Dropout | 0.2 | 0 | None |
| Dense | (None, 10, 8) | 136 | None |
| Dense | (None, 10, 4) | 36 | None |
| Dense | (None, 10, 2) | 10 | None |
| Dense | (None, 10, 1) | 3 | Sigmoid |

*Table 11: Layers of model 2*

As the previous model, this one has completed 2 epochs using a batch of 64 samples. However, model 2 took longer to train due to the LSTM layers, which do additional computations.

The model has a lower loss of 34% but it still performs slightly worser than the previous model. Model 2 has misclassified more than half of the fraudulent transactions are genuine, although the average precision-recall score is the same - 0.69. Even with LSTMs the model still performs the same.

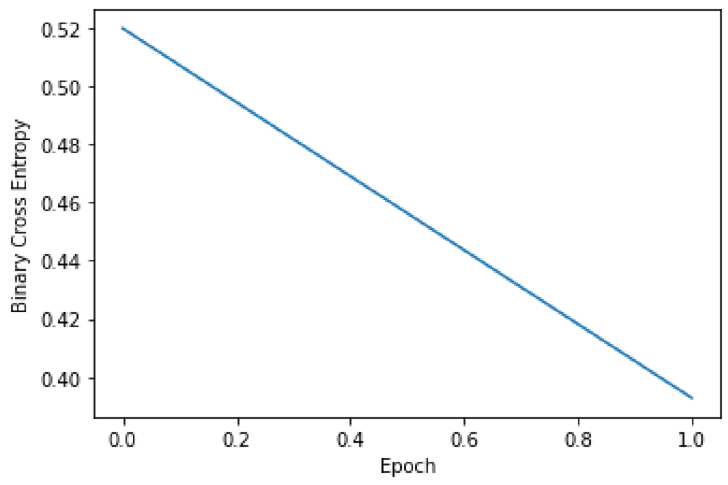
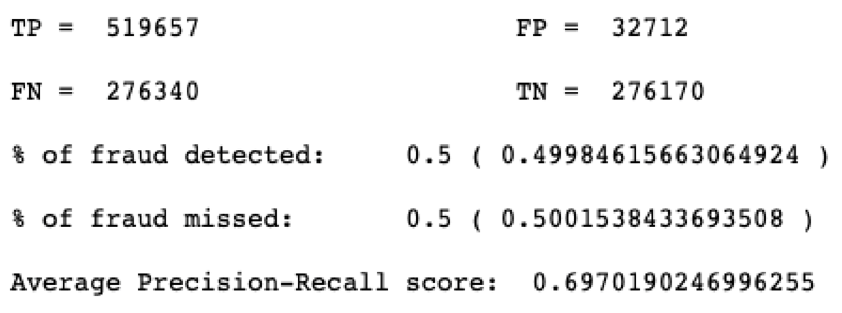


*Figure 57: Loss over time for model 2*



```
TP =  519657                    FP =  32712

FN =  276340                    TN =  276170

% of fraud detected:     0.5 ( 0.49984615663064924 )

% of fraud missed:       0.5 ( 0.5001538433693508 )

Average Precision-Recall score:  0.6970190246996255
```

*Figure 58: Get the fraud detected and missed for RNN model 2 (SMOTE)*

### *Model 3*

Models 1 and 2 have only went through 2 epochs and the models may still not be to correctly identify a fraudulent and a genuine transaction. This model is the same as model 1 but it has gone through 20 epochs.

| Layer | Output Shape | Parameters | Activation Function |
|---|---|---|---|
| Simple RNN | (None, 10, 128) | 16640 | Sigmoid |
| Dropout | 0.2 | 0 | None |
| Simple RNN | (None, 10, 64) | 12352 | Sigmoid |
| Dropout | 0.2 | 0 | None |
| Simple RNN | (None, 10, 32) | 3104 | Sigmoid |
| Dropout | 0.2 | 0 | None |
| Simple RNN | (None, 10, 16) | 784 | Tanh |
| Dropout | 0.2 | 0 | None |
| Dense | (None, 10, 8) | 136 | None |
| Dense | (None, 10, 4) | 36 | None |
| Dense | (None, 10, 2) | 10 | None |
| Dense | (None, 10, 1) | 3 | Sigmoid |

*Table 12: Layers of model 3*

However, even more epochs have no impact on the final result. Although the loss drops under 15% the end result is the same as the latter models.

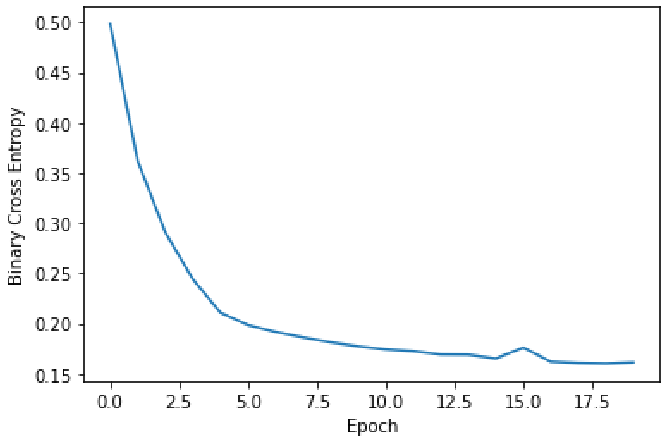
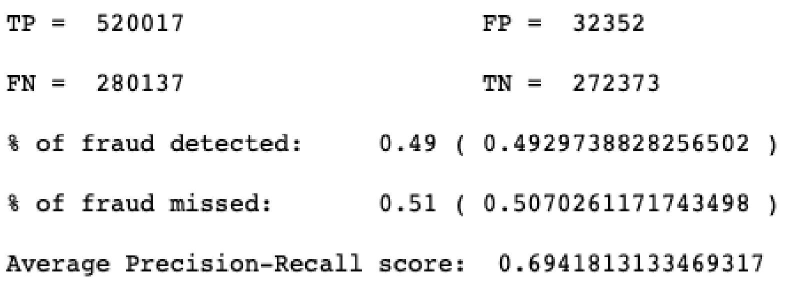


*Figure 59: Loss over time for model 3*



```
TP =  520017                    FP =  32352

FN =  280137                    TN =  272373

% of fraud detected:     0.49 ( 0.4929738828256502 )

% of fraud missed:       0.51 ( 0.5070261171743498 )

Average Precision-Recall score:  0.6941813133469317
```

*Figure 60: Get the fraud detected and missed for RNN model 3 (SMOTE)*

### Model 4

Model 4 uses the same RNN architecture as model 2 but it goes through 10 epochs.

| Layer | Output Shape | Parameters | Activation Function |
|---|---|---|---|
| LSTM | (None, 10, 128) | 16640 | Sigmoid |
| Dropout | 0.2 | 0 | None |
| LSTM | (None, 10, 64) | 12352 | Sigmoid |
| Dropout | 0.2 | 0 | None |
| LSTM | (None, 10, 32) | 3104 | Sigmoid |

| | | | |
|---|---|---|---|
| Dropout | 0.2 | 0 | None |
| LSTM | (None, 10, 16) | 784 | Tanh |
| Dropout | 0.2 | 0 | None |
| Dense | (None, 10, 8) | 136 | None |
| Dense | (None, 10, 4) | 36 | None |
| Dense | (None, 10, 2) | 10 | None |
| Dense | (None, 10, 1) | 3 | Sigmoid |

*Table 13: Layers of model 4*

However even with more epochs, the LSTM model was still unable to correctly identify fraudulent transactions even though the loss is below 20%.

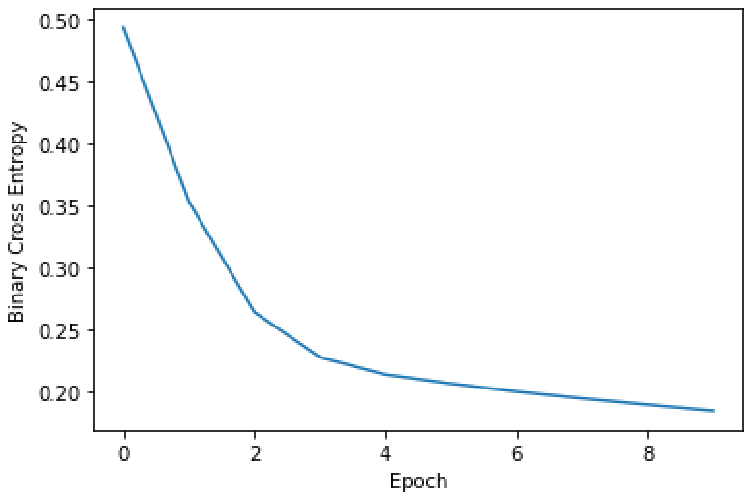
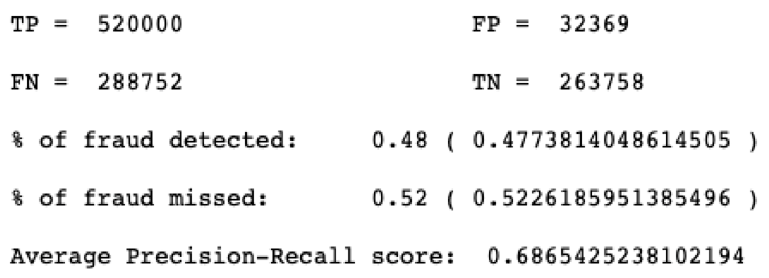


*Figure 61: Loss over time for model 4*



```
TP =  520000                      FP =  32369

FN =  288752                      TN =  263758

% of fraud detected:      0.48 ( 0.4773814048614505 )

% of fraud missed:        0.52 ( 0.5226185951385496 )

Average Precision-Recall score:   0.6865425238102194
```

*Figure 61: Get the fraud detected and missed for RNN model 4 (SMOTE)*

# Conclusion and Future Work

Fraud detection is crucial for banks in order to prevent money laundering from customers. During the pandemic online transactions have become even more common amongst us and there is even a higher risk of fraud or stealing a customer's information. This project aims to compare and develop precise machine learning and deep learning models that can be used to detect fraud by using the PaySim dataset. In addition, different metrics are discussed to determine whether an algorithm is performing good or bad.

Overall ensemble methods such as Random Forest, XGBoost and K-Nearest Neighbours perform better than the other algorithms that take a single choice (Gaussian Naïve Bayes, Logistic Regression and Recurrent Neural Networks). Although RNNs are the fastest in detecting real time fraud, they require more testing in order to develop a precise model that can outperform the other algorithms. Class weights and SMOTE show how the algorithms interact with the data and how they differ. The biggest difference between the two methods can be seen in K Nearest Neighbours.

Additional work can be done by creating a custom loss function for Traditional Machine Learning and Deep Learning that calculates the amount of money which is lost based on False Negative transactions when a model misclassifies a transaction. This can be a challenge because PaySim is a mobile money simulator and real-life administrative costs, and it does not give any additional information regarding the customer. Administrative costs, such as utilities, insurance, salaries, office space, electricity, etc. play a crucial part in calculating the total loss of a bank when a transaction is fraudulent. If a transaction is misclassified by the algorithm the total loss is equal to the amount of money lost and the administrative costs. On the other hand, true negative results only cost administrative expenses. Secondly the RNN models can be tested with different batch sizes and more epochs to see if the model would improve or not. Smaller batch sizes or more epochs would take longer to compute.

Another task is to investigate and improve the currently proposed Deep Learning models. Training takes a very long time, and more experiments need to be ran with different epochs, batch sizes and create a new RNN model. In addition, RNN might not be the most suitable model for this dataset because RNN learns sequential data. The RNN can be successfully used on data that also contains the user’s behaviour which can also lead to catching a fraudulent transaction before it has even happened, just by looking at data such as how long it takes a customer to go from one page to another and even how much time they spend filling up card information or staying on the website.

Finally, the K-Nearest Neighbours algorithm can be tried with more than 5 neighbours. The more neighbours the algorithm takes into consideration, the slower the algorithm is.

# References

Afonja, T. (2017) *Accuracy Paradox*

Available at: https://towardsdatascience.com/accuracy-paradox-897a69e2dd9b (Accessed: 3 February 2021)

Altexsoft (no date) *Fraud Detection: How Machine Learning Systems Help Reveal Scams in Fintech, Healthcare, and eCommerce.*

Available at: https://www.altexsoft.com/whitepapers/fraud-detection-how-machine-learning-systems-help-reveal-scams-in-fintech-healthcare-and-ecommerce/ (Accessed: 10 November 2020)

Amidi, A., Amidi, S., (No date) *Recurrent Neural Network cheatsheet*

Available at: https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks (Accessed: 5 January 2021)

Analytics Vidhya (2020) *How to Improve Class Imbalance using Class Weights in Machine Learning*

Available at: https://www.analyticsvidhya.com/blog/2020/10/improve-class-imbalance-class-weights/ (Accessed: 14 February 2021)

Awoyemi, J. O., Adetunmbi, A. O. and Oluwadare, S. A. (2017) ‘Credit card fraud detection using machine learning techniques: A comparative analysis’, *International Conference on Computing Networking and Informatics (ICCNI).* Lagos, 2017, pp. 1-9, doi: 10.1109/ICCNI.2017.8123782.

Baesens, B., Höppner, S., Verdonck, T. (2021) '*data engineering for fraud detection*', Decision Support Systems pp. 1-12, doi: 10.1016/j.dss.2021.113492

Bronshtein, A. (2017) *Train/Test Split and Cross Validation in Python*

Available at: https://towardsdatascience.com/train-test-split-and-cross-validation-in-python-80b61beca4b6 (Accessed: 24 December 2020)

Brownlee, J. (2018) *Difference Between a Batch and an Epoch in a Neural Network*

Available at: https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/ (Accessed: 15 December 2020)

Brownlee, J. (2021) *Random Oversampling and Undersampling for Imbalanced Classification*

Available at: https://machinelearningmastery.com/random-oversampling-and-undersampling-for-imbalanced-classification/ (Accessed: 28 February 2021)

Brownlee, J. (2020) *SMOTE for Imbalanced Classification with Python*

Available at: https://machinelearningmastery.com/smote-oversampling-for-imbalanced-classification/ (Accessed: 1 January 2021)

Draelos, R. (2019) *Measuring Performance: AUC (AUROC)*

Available at: https://glassboxmedicine.com/2019/02/23/measuring-performance-auc-auroc/ (Accessed: 15 November 2020)

GeeksforGeeks (2020) *Activation functions in Neural Networks*

Available at: https://www.geeksforgeeks.org/activation-functions-neural-networks/ (Accessed: 10 December 2020)

Huoh, Y. (2017) *Adjustments, error types, and aggressiveness in fraud modelling*

Available at: https://fin.plaid.com/articles/model-tradeoffs-and-error-types/ (Accessed: 24 December 2020)

Wang, Y. (2019) *Rethinking the Right Metrics for Fraud Detection*

Available at: https://medium.datadriveninvestor.com/rethinking-the-right-metrics-for-fraud-detection-4edfb629c423 (Accessed: 16 November 2020)

Pambudi, B., Hidayah, I., Fauziati, S. (2019) *Improving Money Laundering Detection Using Optimized Support Vectror Machine*. Universitas Gadjah Mada

Available at: https://www.researchgate.net/publication/339979898_Improving_Money_Laundering_Detection_Using_Optimized_Support_Vector_Machine (Accessed: 15th March 2021)

Pykes, K. (2020) *Oversampling and Undersampling*
Available at: https://towardsdatascience.com/oversampling-and-undersampling-5e2bbaf56dcf (Accessed: 13 December 2020)

Rich Data (2017) *SMOTE explained for noobs - Synthetic Minority Over-samplingTEchnique line by line*

Available at: https://rikunert.com/SMOTE_explained (Accessed: 2 January 2021)

Jain, A. (2016) *Complete Guide to Parameter Tuning in XGBoost with codes in Python*

Available at: https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/ (Accessed: 2 January 2021)

Kaggle (no date) *Synthetic Financial Dataset For Fraud Detection*

Available at: https://www.kaggle.com/ntnu-testimon/paysim1 (Accessed 15 November 2020)

Medium (2019) *An introduction to Grid Search*

Available at: https://medium.com/datadriveninvestor/an-introduction-to-grid-search-ff57adcc0998 (Accessed: 18 March 2021)

Nordling, C. (2020) *Anomaly Detection in Credit Card Transactions using Autoencoders.* KTH Royal Institute of Technology

Available at: https://www.diva-portal.org/smash/get/diva2:1466914/FULLTEXT01.pdf

(Accessed: 15 April 2021)

Olah, C. (2015) *Understanding LSTM Networks*

Available at: https://colah.github.io/posts/2015-08-Understanding-LSTMs/ (Accessed: 1 January 2021)

Ray, S. (2015) *Quick Introduction to Boosting Algorithms in Machine Learning*

Available at: https://www.analyticsvidhya.com/blog/2015/11/quick-introduction-boosting-algorithms-machine-learning/ (Accessed: 31 October 2020)

Shen, K. (2018) *Effects of batch size on training dynamics*

Available at: https://medium.com/mini-distill/effect-of-batch-size-on-training-dynamics-21c14f7a716e (Accessed: 20 December 2020)