Financial Fraud Detection using Machine Learning

Adelene Ng, Gareth Jonathan Halim

1 Introduction

With the advent of e-commerce and online purchases, both businesses and consumers alike are increasingly reliant on digital payments for their financial transactions. With the increasing use of online payments, there is a corresponding increase in financial fraud. The sheer volume of transactions together with increasingly sophisticated methods employed by cyber criminals, manual detection or even static rule-based fraud detection is no longer effective. In order to combat increasingly sophisticated methods adopted by cyber-criminals, we will use machine learning and artificial intelligence techniques and tools to fight the war against fraud.

2 Problem Framing

We want the machine learning model to be able to predict if a financial transaction is fraudulent or not. Our problem can be best framed as a binary classification problem which predicts if a transaction is genuine or not. The ideal outcome would be to correctly predict a transaction which is fraudulent as actually fraudulent.

Success of the model is measured by the reduction in fraud volume by up to 60% & keeping customers secure with real-time fraud detection.

Our machine learning model is deemed a failure if it fails to detect any fraudulent transactions.

The output from our machine learning model will return true if a given transaction is deemed fraudulent, false otherwise. Our machine learning model will take as input (e.g. transaction amount, transaction type) and the output will be a value of 1 if a transaction is fraudulent, 0 otherwise. The output will be used to decide if the transaction should be approved or denied.

If no machine learning is used, then we either have to resort to manual or rule-based detection, using heuristics such as sudden unusual transaction amounts, unexpected spikes in transactional activities, increased frequency of transactions that are potential red flags.

3 Datasets

Since financial datasets cannot be easily collected by individuals, we will use pre-prepared datasets. The lack of public datasets in the financial sector is due to the intrinsically private nature of financial transactions. We will use the dataset from Kaggle, “Synthetic Financial Datasets For Fraud Detection” [1], which has been generated by the PaySim mobile generator. It simulates mobile money transactions based on a sample of real transactions extracted from one month of financial logs from a mobile money service implemented in an African country. The original logs were provided by a multinational company, who is the provider of the mobile financial service which is currently running in more than 14 countries all around the world.

Table 1 shows the features and their descriptions in the dataset used for this project.

Features	Description
step	Maps a unit of time in the real world. In this case 1 step is 1 hour of time.
type	CASH-IN, CASH-OUT, DEBIT, PAYMENT and TRANSFER
amount	Amount of the transaction in local currency
nameOrig	Customer who started the transaction
oldbalanceOrg	Initial balance before the transaction
newbalanceOrig	New balance after the transaction
nameDest	Customer who is the recipient of the transaction
oldbalanceDest	Initial balance of recipient before the transaction
newbalanceDest	New balance of recipient after the transaction
isFraud	This is the transaction made by the fraudulent agents inside the simulation. In this specific dataset the fraudulent behaviour of the agents aims to profit by taking control or customers' accounts and try to empty the funds by transferring to another account and then cashing out of the system.
isFlaggedFraud	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.

Table 1 – Description of Dataset Features

4 Team members' contributions

Table 2 shows how we split the tasks amongst the team members.

Tasks	Person Responsible
Exploratory Data Analysis	Both
Data Visualization	Both
Feature Engineering	Adelene Ng
Correlation	Adelene Ng
Feature Importance	Adelene Ng
Preparing the dataset for training & Imbalance dataset handling	Adelene Ng
Model Experimentation (LR, Naïve-Bayes)	Gareth Jonathan Halim
Model Experimentation (XBoost, DT, RFC, CatBoost, LightGBM, Bagging Classifier)	Adelene Ng
Grid Search (CatBoost, LightGBM)	Adelene Ng
Web-based Flask Application	Both

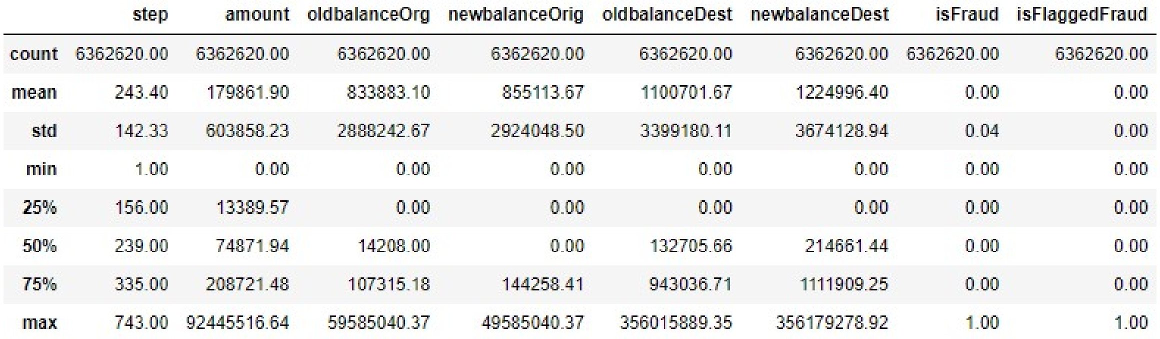
Table 2 – Task split amongst team members

5 Exploratory Data Analysis

We first determine the types associated with each of the attributes. There are a total of 11 attributes: 5 of which are floats, 3 are integers and there are 3 objects (Table 3).

Attribute	Type
step	Int64
type	object
amount	float64
nameOrig	object
oldbalanceOrg	float64
newbalanceOrig	float64
nameDest	object
oldbalanceDest	float64
newbalanceDest	float64
isFraud	int64
isFlaggedFraud	int64

Table 3 – Dataset attributes and their types



	step	amount	oldbalanceOrg	newbalanceOrig	oldbalanceDest	newbalanceDest	isFraud	isFlaggedFraud
count	6362620.00	6362620.00	6362620.00	6362620.00	6362620.00	6362620.00	6362620.00	6362620.00
mean	243.40	179861.90	833883.10	855113.67	1100701.67	1224996.40	0.00	0.00
std	142.33	603858.23	2888242.67	2924048.50	3399180.11	3674128.94	0.04	0.00
min	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
25%	156.00	13389.57	0.00	0.00	0.00	0.00	0.00	0.00
50%	239.00	74871.94	14208.00	0.00	132705.66	214661.44	0.00	0.00
75%	335.00	208721.48	107315.18	144258.41	943036.71	1111909.25	0.00	0.00
max	743.00	92445516.64	59585040.37	49585040.37	356015889.35	356179278.92	1.00	1.00

Figure 1 – Statistical Information for this dataset for numerical features

Figure 1 shows the statistical information (count, mean, standard deviation, minimum, maximum, median, 25th and 75th percentile) for each of the numerical features of this dataset.

5.1 Data Distribution

There are **NO** missing values in the dataset. There are a total of **6,362,620** rows in the dataset. In the '***isFraud'*** column, there are 2 unique values: 0 which represents a normal transaction and 1 a fraudulent transaction. The dataset is highly imbalanced: **6354407** representing normal transactions, **8213** representing fraudulent transactions. This is illustrated by the bar plot shown in Figure 2.

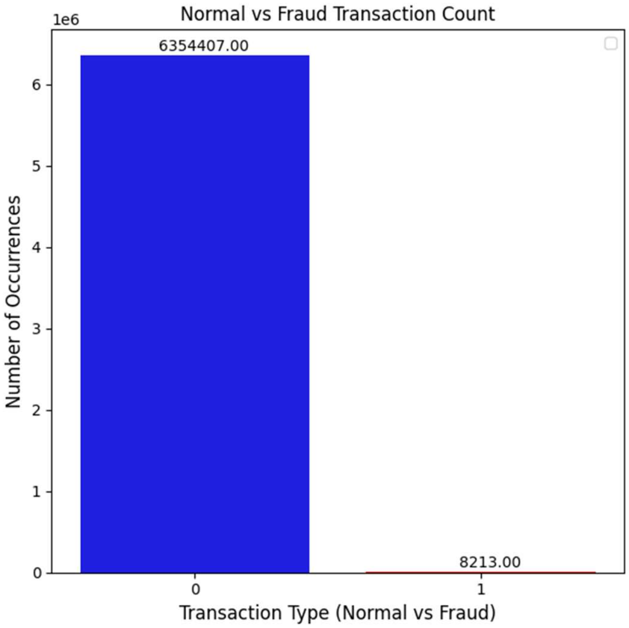


Figure 2 – Counts of Normal vs Fraudulent Transaction Types

Figure 3 shows that there are a total of 5 transaction types and they are:

- Cash Out
- Payment
- Cash In
- Transfer
- Debit

'Cash Out' transactions have the largest amount of transactions: **2,237,500**, whilst 'Debit' has the smallest amount of transactions at **41,432**.

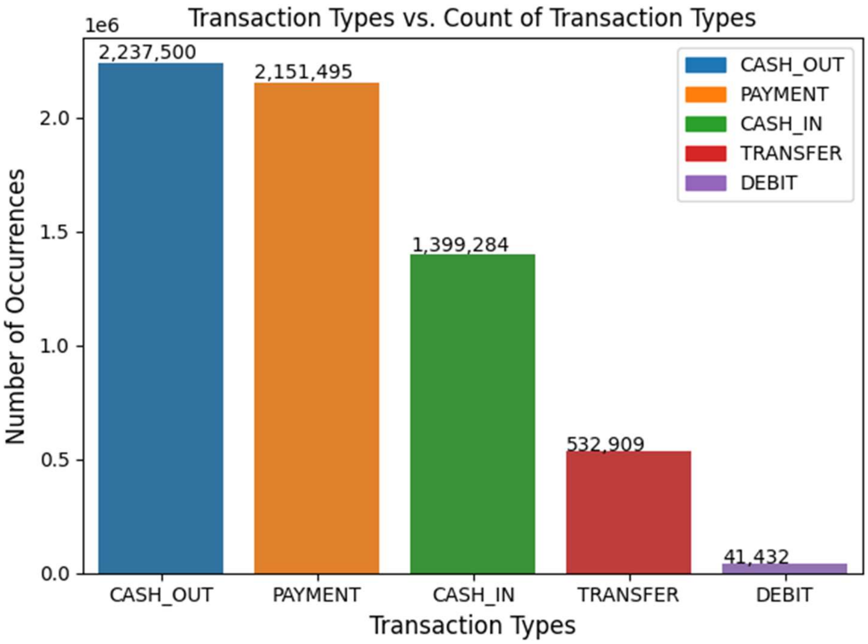


Figure 3 – Distribution of the different Transaction Types

Figure 4 shows a grouping of normal, fraudulent dealings by transaction types. 'Cash Out' and 'Transfer' have transactions that have been flagged as fraudulent i.e., **4,116** and **4,097** respectively. The remaining transaction types: 'Cash In', 'Debit' and 'Payment' are normal.

Figure 5 shows the fraudulent distribution percentages for 'Transfer' (**50.1%**) and 'Cash Out' (**49.9%**) as a pie chart. The figures show that their percentages are fairly evenly distributed.

We also examined the transaction originator and recipient of the transaction. There are 4 different types:

- Customer to Customer
- Customer to Merchant
- Merchant to Customer
- Merchant to Merchant

This is the breakdown after analysis (Table 4):

	Fraudulent	Normal
Customer to Customer	8213	4202912
Customer to Merchant	0	2151495

Table 4 – Originator to Recipient Transaction Breakdown flagged as Fraudulent, Normal

Table 4 shows that there were **8213** fraudulent transactions committed when the transactions were made from customer to customer (4,202,912 were normal). There were no transactions from merchant to customer and merchant to merchant in this dataset.

There were **287** fraudulent customer-to-customer transactions for amounts of **10,000,000**.

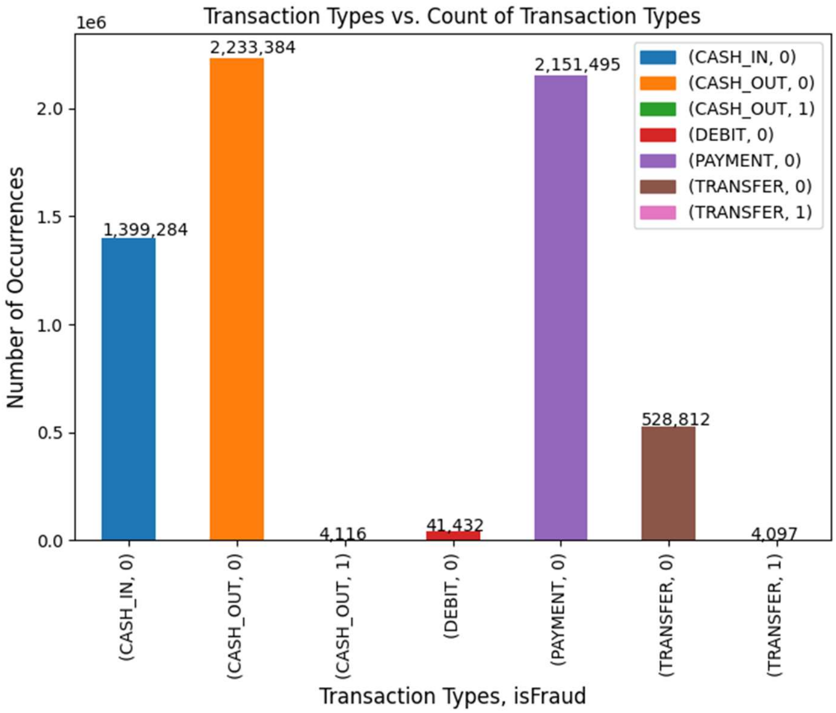


Figure 4 – Distribution of the various transaction types grouped by Normal vs. Fraudulent

In Figure 6, the right column displays the 'recipient balance amount mismatches' plot shows **25.6%** mismatch after subtracting the amount transacted from the original balance. The left column shows ‘recipient amount received exceeds available balance’ shows a value of **58.2%** (amount received does not exceed balance in account).

In Figure 7, the right column displays the ‘sender balance amount mismatches’ plot shows **85.1%** mismatch after subtracting the amount transacted from the original balance. The left column shows ‘sender amount transferred exceeds available balance’ shows a value of **64.1%**.

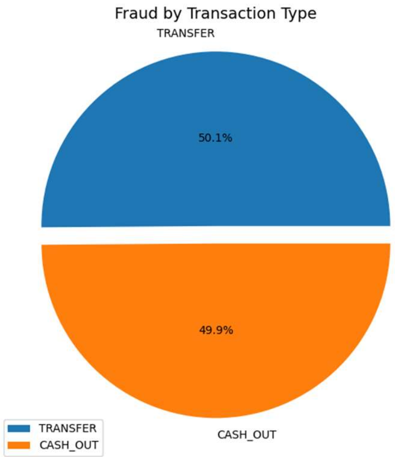


Figure 5 – Transaction Types ‘Transfer’ and ‘Cash Out’ Fraudulent Distributions

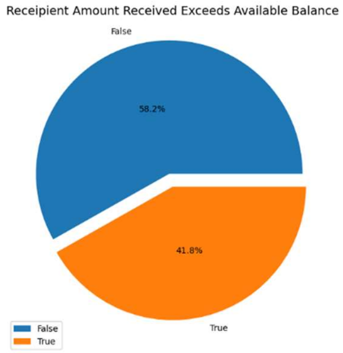
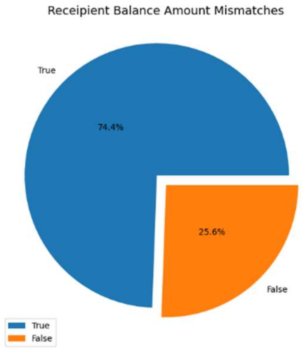
Recipient Amount Transferred Exceeds Available Balance	Recipient Balance Amount Mismatches
	

Figure 6 – Recipient Balances

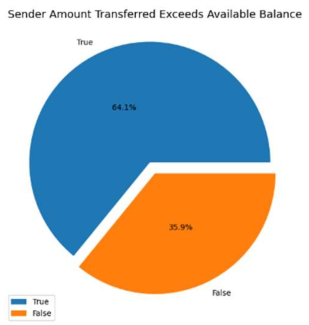
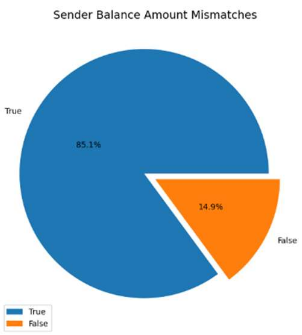
Sender Amount Transferred Exceeds Available Balance	Sender Balance Amount Mismatches
	

Figure 7 – Sender Balances

5.2 Analysis of the “Step” Feature

Here, we examine the ‘step’ attribute to determine if fraudulent transactions are happening at a particular time of the day or day of the week.

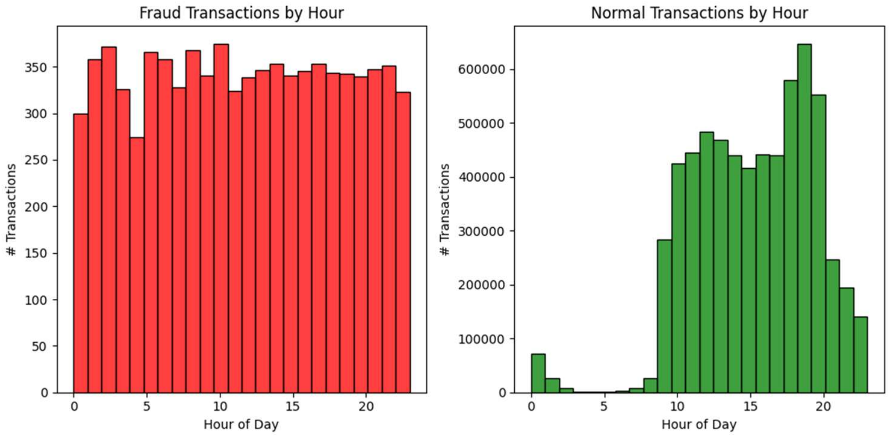


Figure 8 – Hour of day vs # Fraudulent Transactions (left), Hour of day vs # Normal Transactions (right)

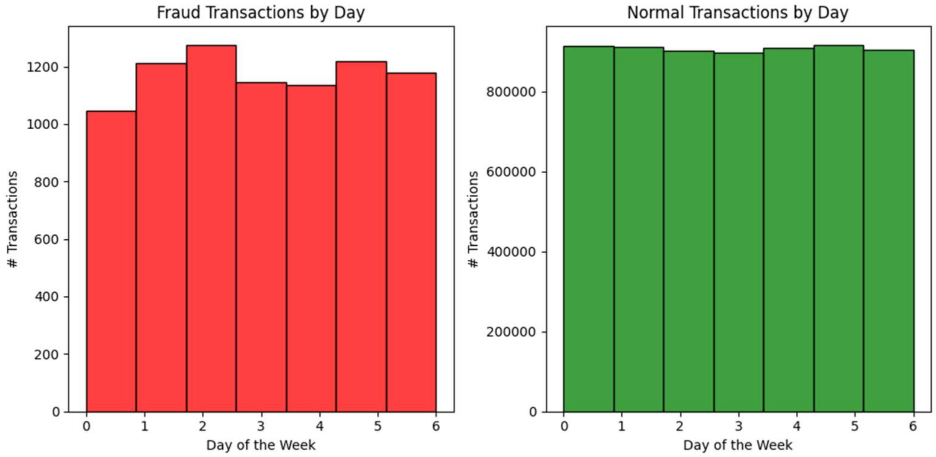


Figure 9 – Day of Week vs # Fraudulent Transactions (left), Day of Week vs # Normal Transactions (right)

Figure 8 shows a plot of hour of day vs fraudulent (in red) / normal transactions (in green). From the graph, fraudulent transactions occurred throughout the day, the minimum number of fraudulent transactions of 270 happening around the 6th hour and the maximum number of fraudulent transactions of around 370 happening at the 11th hour. The number of fraudulent transactions is fairly consistent throughout the day. Normal transactions mostly happen around the 10th hour and maxes out around the 20th hour before tapering off. Nearly no transactions happen between the 3rd to 7th hour.

Figure 9 shows a plot of day of the week vs fraudulent (in red) / normal transactions (in green). Normal transactions are fairly constant across the days of the week. Fraudulent transactions show a fairly consistent trend across the week.

5.3 Skew (Along Columns)

Next, we examine the skewness of this dataset.

Numerical Features	Skew	Skew (Log)
Step	0.38	-0.98
Amount	30.99	3.43
oldbalanceOrg	5.25	1.66
newbalanceOrig	5.18	1.64
oldbalanceDest	19.92	2.99
newbalanceDest	19.35	2.96
isFraud	27.78	3.32
isFlaggedFraud	630.60	6.45

Table 5 – Feature Skewness

Skewness measures the symmetry of a distribution.

If the absolute value of skew < 0.5 then the data is very symmetric.

If the absolute value of skew is between 0.5 and 1 then the data is slightly skewed

If the absolute value of skew is greater than 1 then the data is very skewed (to the right).

From the skewness values, we observe that the majority of the features have values that are bigger than 1 (except step). We can conclude that the dataset is highly skewed.

5.4 Kurtosis (Along Columns)

Here we explore data kurtosis.

Numerical Features	Kurtosis	Kurtosis (Log)
Step	0.33	-1.11
Amount	1797.96	7.49
oldbalanceOrg	32.96	3.50
newbalanceOrig	32.07	3.47
oldbalanceDest	948.67	6.86
newbalanceDest	862.16	6.76
isFraud	769.70	6.65
isFlaggedFraud	397659.06	12.89

Table 6 – Feature Kurtosis

Kurtosis is the measure of outliers present in the distribution. It determines the heaviness of the distribution tails.

High kurtosis in a data set is an indicator that data has heavy outliers.

Low kurtosis in a data set is an indicator that the data has a lack of outliers.

Since the majority of the kurtosis values are greater than 1 (except step), this indicates the distribution is very peaked i.e. is heavy tailed and hence has outliers.

6 Feature Engineering

Once exploratory data analysis is done, we drop the following columns:

- Step
- nameOrig
- nameDest
- isFlaggedFraud

and we add the following columns:

ErrorBalanceOrigin – Errors in the originating balances taking into the account the amount transacted

$$\boldsymbol{ErrorBalanceOrigin = newBalanceOrig - oldBalanceOrg + amount}$$

ErrorBalanceDest – Errors in the destination balances taking into the account the amount transacted

$$\boldsymbol{ErrorBalanceDest = newBalanceDest - oldBalanceDest + amount}$$

These 2 new features (account balance errors) were added to the dataset because errors in the originating and final destination could be a good indicator of fraudulent transactions. This was later verified when we computed the feature importance(s) (see section 6.2). '**ErrorBalanceOrigin**" came out tops and with '**ErrorBalanceDest**' coming in at number 4.

'**Step**' was removed because data analysis has shown that fraudulent transactions were not affected by time of day or even day of the week.

'**nameOrig**' and '**nameDest**' features were also dropped. Analysis have shown that all the fraudulent transactions occurred between 'Customer-to-Customer' (Table 4). The remaining transactions from customer to merchant were all normal. No transactions from merchant to customer and merchant to merchant were found in. the dataset.

Finally, the '**isFlaggedFraud**' attribute is also discarded. There are only **16** entries for this attribute, indicating that an illegal attempt was made to transfer more than 200.000 in a single transaction.

After applying feature engineering and throwing away the irrelevant attributes, the final features that we are left with are (Table 7):

Attribute	Type	
type	object	Will be one hot encoded since this is a categorical variable
amount	float64	
oldbalanceOrg	float64	
newbalanceOrig	float64	
oldbalanceDest	float64	
newbalanceDest	float64	
ErrorBalanceOrig	float64	
ErrorBalanceDest	float64	
isFraud	int64	Label

Table 7 – Results of Feature Engineering

The rows highlighted in yellow are the newly engineered features. Table 8 shows the results after one hot encoding has been applied to the categorical attribute, ‘type’. It has been expanded out to 5 ‘new’ attributes. The row highlighted in green will be used as the label.

Attribute	Type
type_CASH_IN	int64
type_CASH_OUT	int64
type_DEBIT	int64
type_PAYMENT	int64
type_TRANSFER	int64
amount	float64
oldbalanceOrg	float64
newbalanceOrig	float64
oldbalanceDest	float64
newbalanceDest	float64
ErrorBalanceOrig	float64
ErrorBalanceDest	float64
isFraud	int64

Table 8 – After applying One Hot Encoding

6.1 Correlation

Figure 10 shows the correlation map using the original feature attributes of the dataset. The ‘type’ attribute was one hot encoded before applying the correlation function.

Each square shows the correlation between the variables on each axis. Correlation ranges from -1 to +1. Values closer to zero means there is no linear trend between the two variables. The closer to 1 the correlation is the more positively correlated they are; that is as one increases so does the other and the closer to 1 the stronger this relationship is. A correlation closer to -1 is similar, but instead of both increasing, one variable will decrease as the other increases. The diagonals are all 1 because those squares are correlating each variable to itself (so it is a perfect correlation). For the rest, the larger the number, the higher the correlation between the two variables. The plot is also symmetrical about the diagonal since the same two variables are being paired together in those squares.

Figure 11 shows the correlation map with additional features: *ErrorBalanceOrigin* and *ErrorBalanceDest* added to the feature set.

From Figure 10, we observe that the following pairs are correlated:

y-axis	x-axis
newBalanceOrig	oldBalanceOrg
newBalanceDest	oldBalanceDest

Table 9 – Correlation Pairs based on Figure 10

We only look at the pale orange boxes, which show the highly correlated features.

From Figure 11, we observe that the following pairs are correlated:

y-axis	x-axis
newBalanceOrig	oldBalanceOrg
newBalanceDest	oldBalanceDest
ErrorBalanceOrig	amount
ErrorBalanceDest	amount
ErrorBalanceOrig	ErrorBalanceDest

Table 10 – Correlation Pairs based on Figure 11

Likewise, we look at the pale orange boxes, which show the highly correlated features.

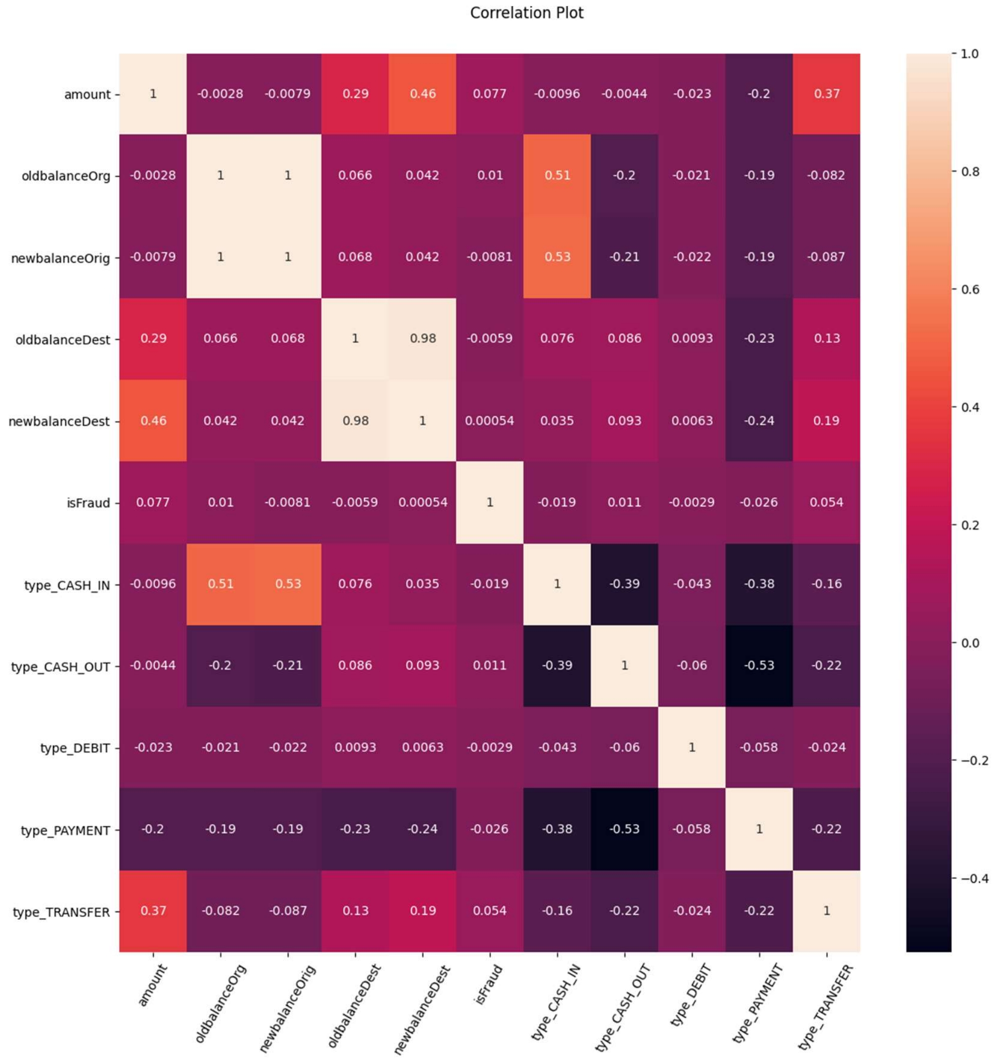


Figure 10 – Correlation Map

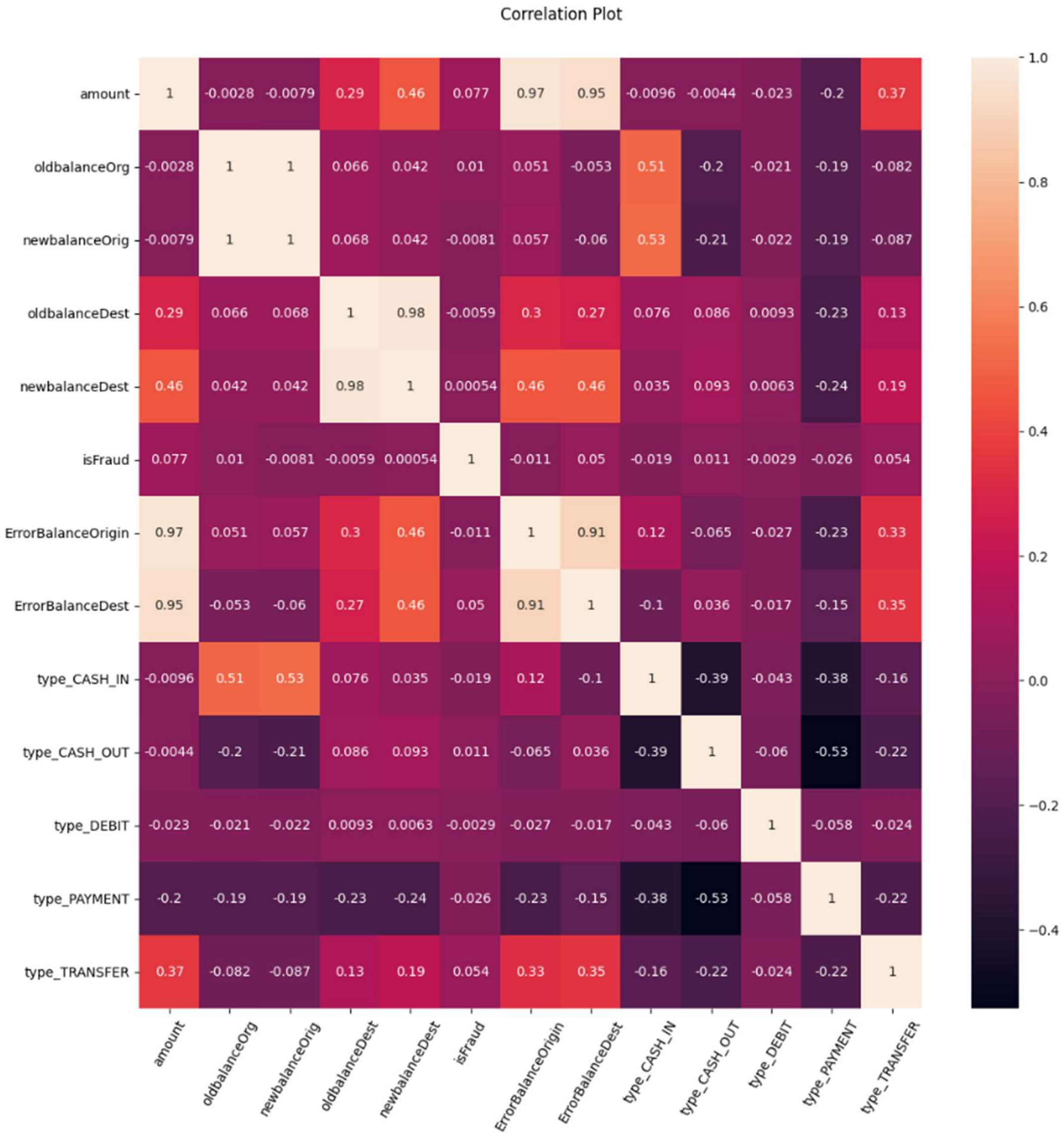


Figure 11 – Correlation Map (feature engineered with ‘ErrorBalanceOrigin’ and ‘ErrorBalanceDest’ added

6.2 Feature Importance

Next, we examine and visualize the important input features of the dataset.

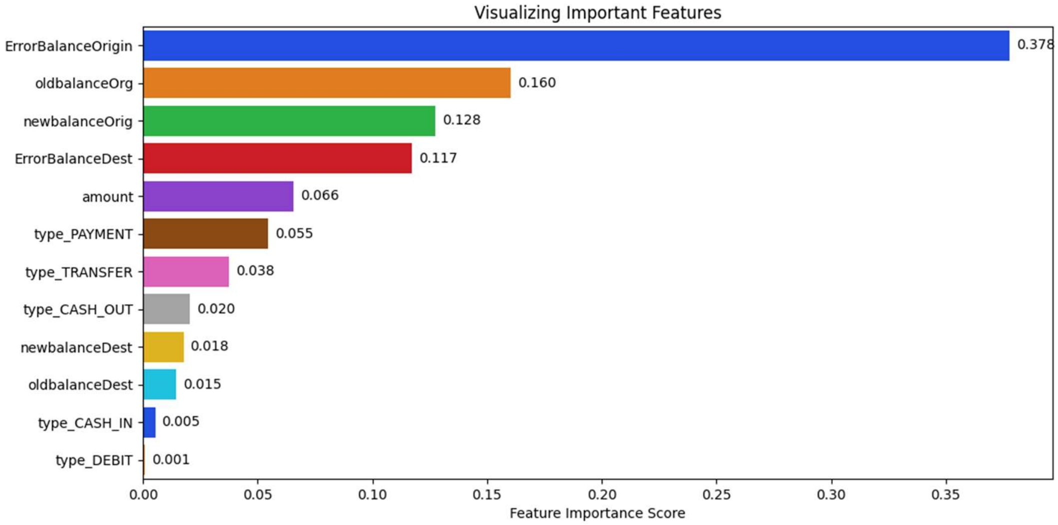


Figure 12 – Feature Importance Distribution

This uses a technique that assigns a score to the input features based on how useful they are at predicting a target variable. We use the *RandomForestClassifier()* algorithm to find out the important features associated with this dataset. After being fit, the model provides a *feature_importances_* property that can be accessed to retrieve the relative importance scores for each input feature. From Figure 12, the top 5 features are:

- ErrorBalanceOrigin
- OldbalanceOrg
- NewbalanceOrg
- ErrorBalanceDest
- Amount

6.3 Preparing the dataset for training & Imbalance dataset handling

We use *StratifiedShuffleSplit()* instead of *train_test_split()* to divide the dataset into train and test sets. *StratifiedShuffleSplit()* is a combination of both *ShuffleSplit()* and *StratifiedKFold()*. Using *StratifiedShuffleSplit()* the proportion of distribution of class labels is almost even between train and test dataset. The major difference between *train_test_split()* and *StratifiedShuffleSplit()* is that the former is pure random sampling without considering any distribution whilst the latter can be understood as a sampling, considering the distribution of categories. *StratifiedShuffleSplit()* is usually recommended for imbalanced datasets.

The dataset is split in the following proportion, 80:20, with “n_splits” set to 1.

In addition to applying *StratifiedShuffleSplit()*, we apply over-sampling techniques such as SMOTE to address the data imbalance problem.

7 Machine Learning Approaches to Fraud Detection

We will apply supervised learning approaches to the problem of detecting fraudulent financial transactions. We plan to use the following algorithms for building a fraud detection model:

a) Logistic Regression

b) Naïve-Bayes
c) XGBoost
d) Decision Tree Classifier
e) Random Forest Classifier
f) CatBoost
g) LightGBM
h) Bagging Classifier

Since this is a classification problem, we start with simple classifiers such as Logistic Regression, Naïve-Bayes and Decision Trees. We then go on to try ensemble methods: Bagging (Random Forest Classifier, Bagging Classifier) & Boosting in particular the Gradient Tree Boosting (XGBoost, LightGBM and CatBoost). The Bagging Classifier uses the Decision Tree Classifier as its base class.

8 Experiments and Evaluation Metrics

This section presents some of the preliminary results that were obtained when we ran the algorithms.

Accuracy alone is not a good metric for highly skewed datasets. We will have to look at other metrics such as:

- Confusion Matrix
- Precision
- Recall
- F1 Score
- Precision Recall curve

to get more insights into the performance of the model.

Using the above metrics, we find the best values for the different algorithms. The winning algorithm will be the final model that will be deployed in production.

Recall (sensitivity) is a measure of our model correctly identifying true positives. It is particularly pertinent when we need to correctly identify the positive scenarios, like in a cancer detection dataset or in this instance fraud detection. Accuracy or precision will not be that helpful in this case.

A good F1 score means that you have low false positives and low false negatives, so you're correctly identifying real threats (in this case real fraud cases), and you are not disturbed by false alarms. An F1 score is considered perfect (more accurate) when it is 1, while the model is a total failure when it is 0.

Precision-Recall curves are used in cases of imbalanced datasets (in the case of fraud detection).

AUC-ROC curves are unsuitable for imbalanced datasets because a small number of correct or incorrect predictions can result in a large change in the ROC Curve or ROC AUC score. The AUC-ROC curves are added here for completeness

The confusion matrix (or error matrix) is a summary of prediction results on a classification problem. It a measure of the effectiveness of our model.

Sections 8.1 - 8.8 will first provide a short description of the algorithms used and show the graphs of the performance metrics and individual performance scores. The results from all the different algorithms used will be tabulated and consolidated in Section 9 where we present the combined results in a tabular form for easy visualization. An analysis of the results will also be presented in Section 9.

8.1 Logistic Regression

In Logistic Regression, input values (x) are combined linearly using weights or coefficient values to predict an output value (y). A key difference from linear regression is that the output value being modelled is a binary value (0 or 1) rather than a numeric value. The idea is to find a relationship between features and the probability of a particular outcome. The linear regression equation is shown below:

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

where:

$\hat{y} = predicted\ value$

n = number of features

x_i = $feature\ i\ value$

θ_0 = bias term

θ_j= jth model parameter

In order to constrain the linear regression term to outputs between 0 and 1, we apply a logistic or sigmoid function to $\hat{y}$ as follows:

$$\hat{p} = \sigma(\hat{y})$$

Expanding the above term, we get:

$$\hat{p} = \sigma(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n)$$

$\hat{p}$ is the estimated probability that y = 1 given input x.

In the model that is generated after training, it will be the coefficients of the equations that are stored.

The experimental results after running the 'Logistic Regression' are shown in Table 11. The confusion matrix and the precision recall curves are shown in Figure 13 and Figure 14 respectively. Table 12 shows the hyper-parameters using for the training of the Logistic Regression model.

A dump of the classification report is shown below:

```
precision    recall  f1-score   support


        0       1.00      0.95      0.97   1270881
        1       0.02      0.94      0.04      1643


   accuracy                          0.95   1272524
  macro avg       0.51      0.95      0.51   1272524
weighted avg       1.00      0.95      0.97   1272524
```

Score Type	Value
Precision Score	0.023
Recall Score	0.943
F1 Score	0.044
Balanced Accuracy	0.945
AUC ROC Score	0.945

Table 11 – Logistic Regression Performance Scores

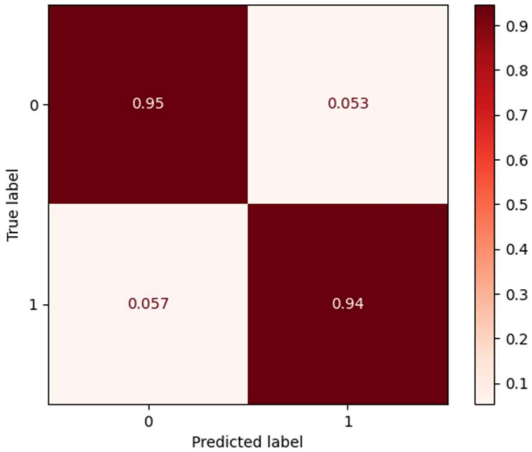


Figure 13 – Logistic Regression Confusion Matrix

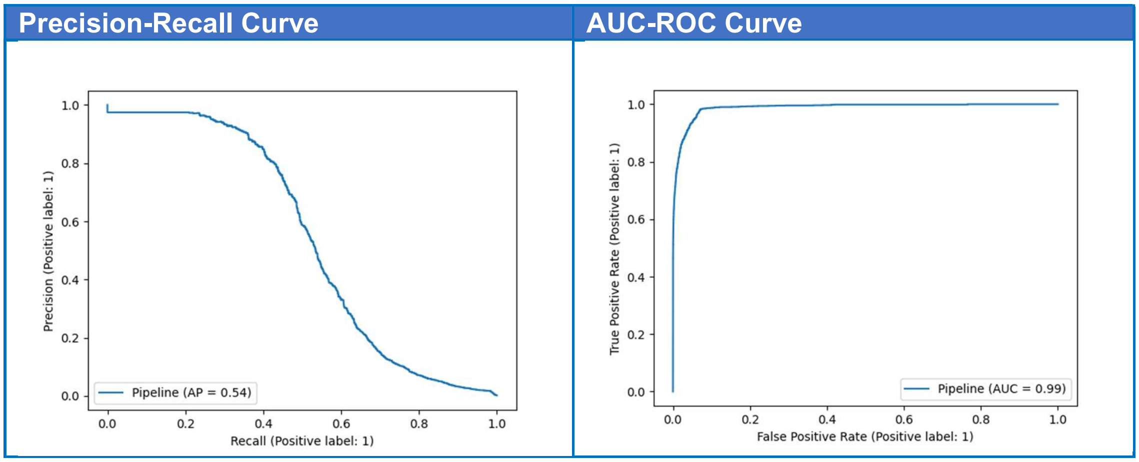


Figure 14 – Logistic Regression: Precision-Recall and AUC-ROC Curves

Hyper-parameter	Value
max_iter	10000
random_state	88

Table 12 – Logistic Regression Hyper-parameters

8.2 Naive-Bayes Classifier

This is a classification technique based on Bayes Theorem. It assumes that all the features are independent of each other. This technique is based on conditional probability. This is the probability that something will happen, ***given that something else has already occurred***.

So given that we have two possible output classes, fraud or a normal transaction and 12 features, we calculate:

$$P(fraud\,|x) = P(x_{amount}\,|fraud) \times P\left(x_{oldBalanceOrg}\middle|fraud\right) \times P(x_{oldBalanceDe}\ \ |fraud) \times \cdots \times P\left(x_{type\_PAYMNET}\middle|fraud\right)$$

$$P(normal\,|x) = P(x_{amount}\,|normal) \times P\left(x_{oldBalanceOrg}\middle|normal\right) \times P(x_{oldBalanceDest}|normal) \times \cdots \times P\left(x_{type\_PAYMNET}\middle|normal\right)$$

where x is the sample data (observation). Then,

if

$$P(fraud\,|x) > P(normal\,|x)$$

then the transaction is classified as fraudulent, else the transaction is normal

The experimental results after running the ‘Naïve-Bayes Classifier’ are shown in Table 13. The confusion matrix and the precision recall curves are shown in Figure 15 and Figure 16 respectively.

The default hyper-parameters were used for Naïve-Bayes Classifier.

A dump of the classification report is shown below:

```
precision    recall  f1-score   support


         0       1.00      0.57      0.73   1270881
         1       0.00      0.99      0.01      1643


   accuracy                           0.57   1272524
  macro avg       0.50      0.78      0.37   1272524
weighted avg       1.00      0.57      0.73   1272524
```

Score Type	Value
Precision Score	0.003
Recall Score	0.993
F1 Score	0.006
Balanced Accuracy	0.783
AUC ROC Score	0.783

Table 13 –Naïve-Bayes Classifier Performance Scores

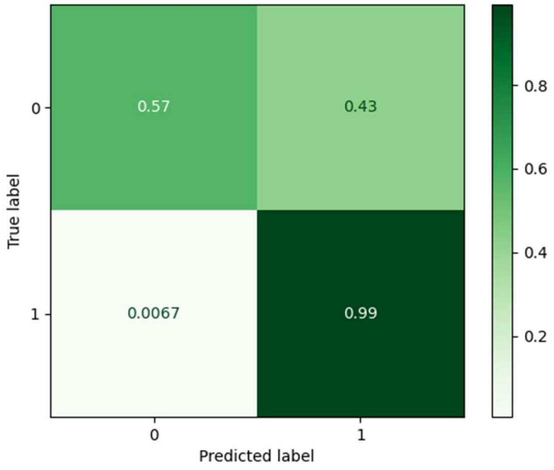


Figure 15 – Naïve-Bayes Classifier Confusion Matrix

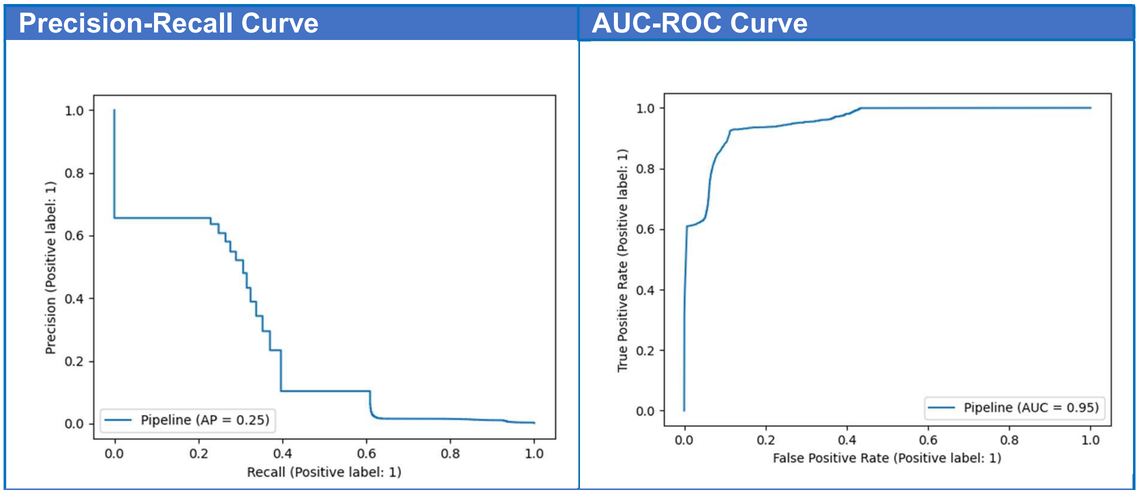


Figure 16 - Naïve-Bayes Classifier: Precision-Recall and AUC-ROC Curves

8.3 XGBoost

XGBoost (e**X**treme **G**radient **B**oosting) belong to the family of Gradient Boosting models in particular the Gradient Tree Boosting (GTB) class. It is an ensemble method that attempts to accurately predict a target variable by combining an ensemble of estimates from a set of simpler and weaker models.

XGBoost is an ensemble of decision trees. Individually, these trees are poor performers but when combined they will be high-performing. It starts out by creating a simple tree, which by itself is a poor performer. It then builds another tree which is trained to predict what the first tree was unable to, and is itself a weak learner. The algorithm continues by sequentially building more weak learners, each one correcting the previous tree until a stopping condition is reached, such as the number of trees (estimators) to build.

To decide which attribute or feature we will use to split the tree, XGBoost uses a pre-sorted and histogram-based method. Pre-sorting splitting works by:

- First, enumerating over all features for each node
- For each feature, sort the instances by the values of the features

- We then have to decide the best split along that feature
- Finally, we take the best split along all the features

Histogram-based method splits all the data points for a feature into discrete bins and uses these to find the split value of histogram.

The experimental results after running the 'XGBoost' are shown in Table 14. The confusion matrix and the precision recall curves are shown in Figure 17 and Figure 18 respectively. Table 15 shows the default parameters used for XGBoost.

A dump of the classification report is shown below:

```
precision    recall  f1-score   support


         0       1.00      1.00      1.00   1270881
         1       0.83      1.00      0.90      1643


    accuracy                          1.00   1272524
   macro avg       0.91      1.00      0.95   1272524
weighted avg       1.00      1.00      1.00   1272524
```

Score Type	Value
Precision Score	0.826
Recall Score	0.996
F1 Score	0.903
Balanced Accuracy	0.998
AUC ROC Score	0.998

Table 14 – XGBoost Performance Scores

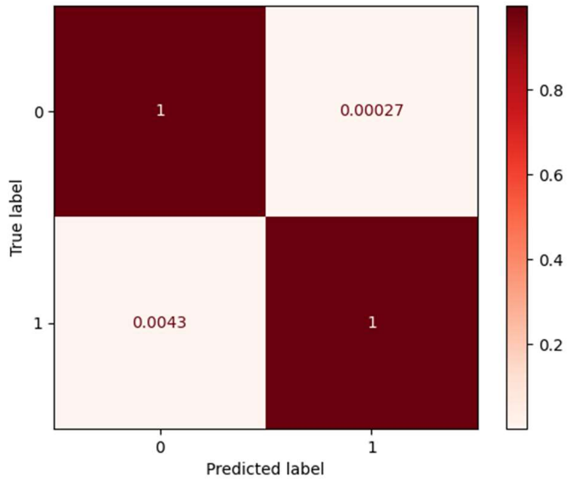


Figure 17 – XGBoost Confusion Matrix

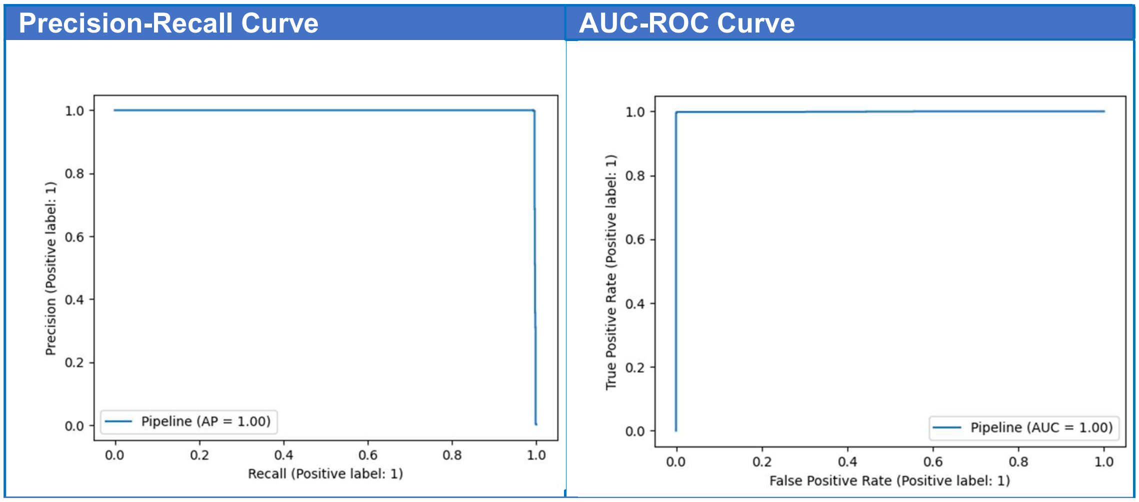


Figure 18 – XGBoost: Precision-Recall and AUC-ROC Curves

Hyper-parameter	Value
tree_method	exact
max_depth	3
n_estimators	50
random_state	88

Table 15 – XGBoost Hyper-parameters

8.4 Decision Tree Classifier

Decision Trees are used to create models that can be used to predict the target class. This is achieved by splitting the training dataset into successively “purer” subsets until each subset contains instances belonging to only one class.

A decision tree consists of root, decision and leaf nodes. Each node has some attribute or feature associated with it. These contain test conditions or decision rules that can be used to separate the training samples into subsets that have similar characteristics.

In order to decide which attribute or feature we will use to split the tree; we can use entropy or the Gini index. The attribute that has the smallest entropy or Gini value will be used to split the tree. Gini is slighter faster to compute whereas entropy results in more balanced trees.

The Gini score of a node is given by

$$G = 1 - \sum_{i=1}^{n} P_i^2$$

where P_i is the fraction of the ith instance of the node.

Entropy of a node is given by

$$H = -\sum_{i=1}^{n} P_i \log_2(P_i)$$

where P_i is the fraction of the ith instance of the node.

The impurity of the decision node G is computed based on the weighted average of the impurity scores of the child nodes and is given by:

$$G = \frac{m_1}{m}G_1 + \frac{m_2}{m}G_2$$

G_1, G_2 are the impurity scores of the left and right child nodes

m_1, m_2 are the number of instances of the child nodes

$m = m_1 + m_2$ and is the total number of training instances associated with the parent node

The tree is built recursively and stops when the subset at a node all have the same value as the target variable, or when splitting no longer adds value to the predictions

Once the tree is built, we compare the values of the root attribute with the record's attribute. On the basis of comparison, we follow the branch corresponding to that value and jump to the next node. We continue in this manner until we reach the leaf nodes.

The experimental results after running the 'Decision Tree Classifier' are shown in Table 16. The confusion matrix and the precision recall curves are shown in Figure 19 and Figure 20 respectively. Table 17 shows the default hyper-parameters used for the Decision Tree Classifier.

A dump of the classification report is shown below:

```
precision    recall  f1-score   support


       0      1.00      1.00      1.00   1270881

       1      0.97      0.99      0.98      1643


   accuracy                        1.00   1272524

  macro avg      0.99      1.00      0.99   1272524

weighted avg      1.00      1.00      1.00   1272524
```

Score Type	Value
Precision Score	0.975
Recall Score	0.995
F1 Score	0.985
Balanced Accuracy	0.997
AUC ROC Score	0.997

Table 16 – Decision Tree Classifier Performance Scores

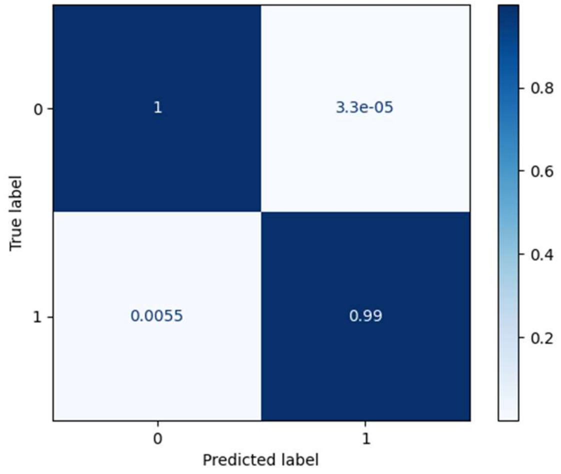
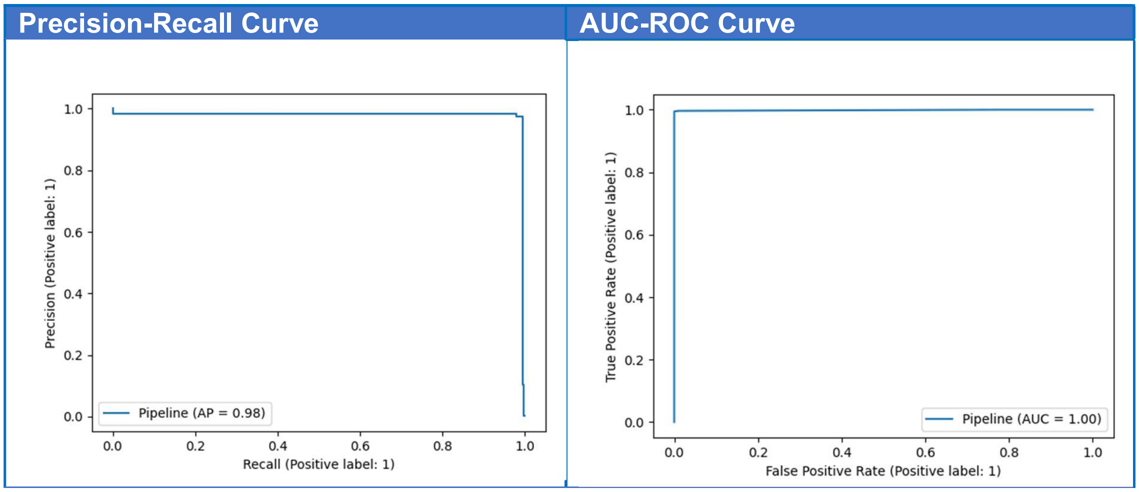


Figure 19 – Decision Tree Classifier Confusion Matrix


Precision-Recall Curve

AUC-ROC Curve

Figure 20 – Decision Tree Classifier: Precision-Recall and AUC-ROC Curves

Hyper-parameter	Value
max_depth	3
random_state	88

Table 17 – Decision Tree Classifier Hyper-parameters

8.5 Random Forest Classifier

Random Forests is a container containing **X** decision trees each having a different set of hyper-parameters and trained on different data subsets. These training samples (typically the same size for each subset) are normally drawn from the training dataset using the bagging method (sampling with replacement or is random). So, if I have 1000 decision trees in my basket each with different sets of hyper-parameters and different subsets of training data, the predictions given by these 1000 decision trees will vary considerably. We need only take one decision on the sample test data and this can be obtained by average or majority vote.

The Random Forest Classifier is an ensemble method meaning that it makes predictions based on a number of different models. Combining the individual models results in a model that is more flexible (less bias) and less data-sensitive (less variance). The random forest classifier uses bagging as the ensemble method and the decision tree as the individual model. Bagging here means that the individual models are trained in parallel, each with a random subset of the data.

The experimental results after running the 'Random Forest Classifier' are shown in Table 18. The confusion matrix and the precision recall curves are shown in Figure 21 and Figure 22 respectively. Table 19 shows the default hyper-parameters used for training the random forest classifier.

A dump of the classification report is shown below:

```
precision    recall  f1-score   support


        0       1.00      1.00      1.00   1270881
        1       0.87      0.99      0.93      1643


   accuracy                         1.00   1272524
  macro avg       0.93      1.00      0.96   1272524
weighted avg       1.00      1.00      1.00   1272524
```

Score Type	Value
Precision Score	0.867
Recall Score	0.995
F1 Score	0.927
Balanced Accuracy	0.997
AUC ROC Score	0.997

Table 18 – Random Forest Classifier Performance Scores

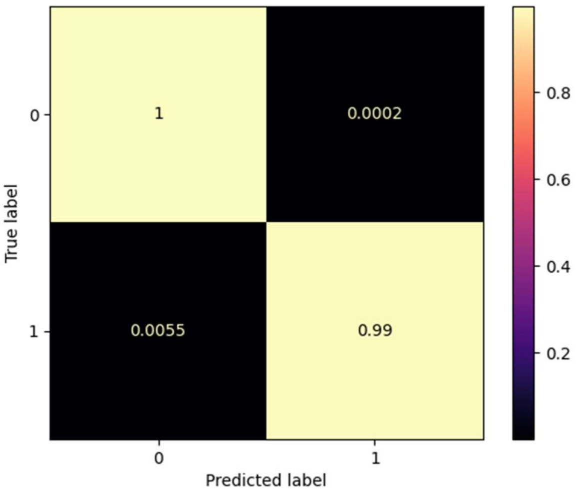
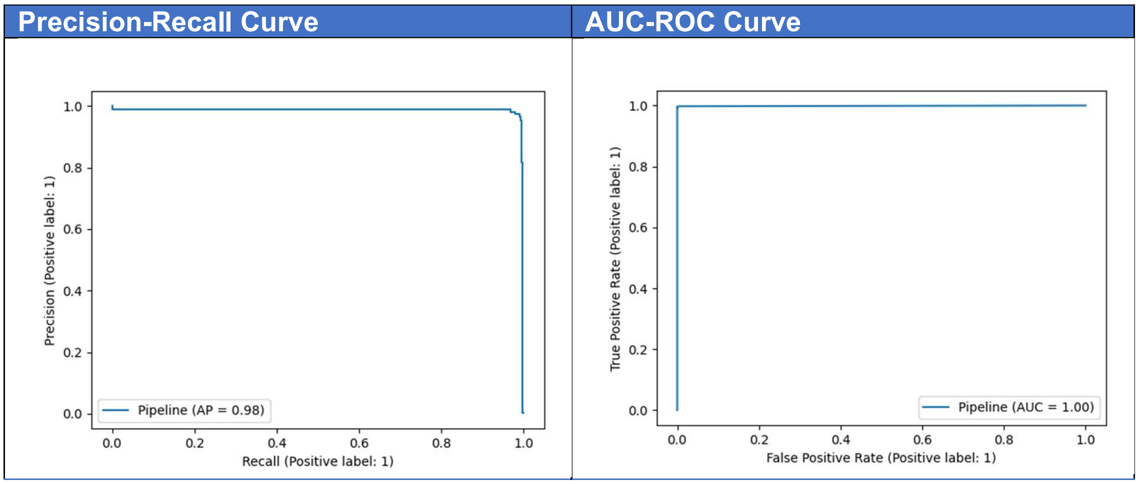


Figure 21 – Random Forest Classifier Confusion Matrix



Precision-Recall Curve

AUC-ROC Curve

Figure 22 – Random Forest Classifier Precision-Recall and AUC-ROC Curve

Hyper-parameter	Value
n_estimators	50
n_jobs	-1
oob_score	True
random_state	88

Table 19 – Random Forest Classifier Hyper-parameters

8.6 CatBoost

CatBoost (Categorical Boosting) also belongs to the family of Gradient Boosting algorithms. It works in the same way as other Gradient Boosting algorithms, for example, XGBoost. It has built-in support for categorical variables and has a higher level of accuracy without tuning parameters and also offers GPU support to speed up the training process.

It is based on gradient decision trees and when training this model, a set of decision trees is built consecutively. As training progresses, each successive tree is built with a reduced loss compared to the previous tree. One main difference between CatBoost and other boosting algorithms is that it implements symmetric trees.

The experimental results after running the ‘CatBoost’ are shown in Table 20. The confusion matrix and the precision recall curves are shown in Figure 23 and Figure 24 respectively. Table 21 shows the hyper-parameters for training the CatBoost model.

A dump of the classification report is shown below:

```
precision    recall  f1-score   support


           0       1.00      1.00      1.00   1270881
           1       0.98      1.00      0.99      1643


    accuracy                           1.00   1272524
   macro avg       0.99      1.00      1.00   1272524
weighted avg       1.00      1.00      1.00   1272524
```

Score Type	Value
Precision Score	0.984
Recall Score	0.996
F1 Score	0.99
Balanced Accuracy	0.998
AUC ROC Score	0.998

Table 20 – CatBoost Classifier Performance Scores

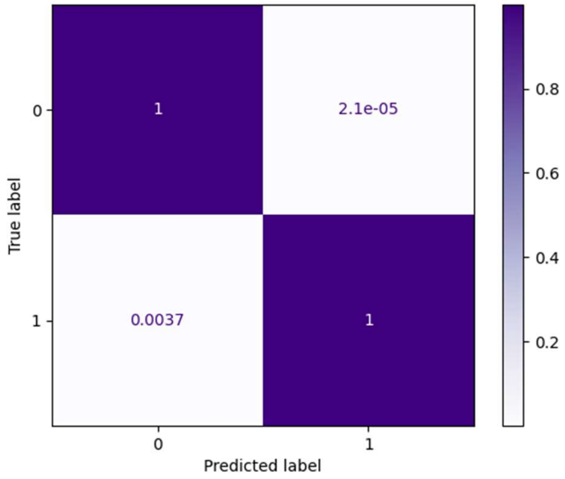
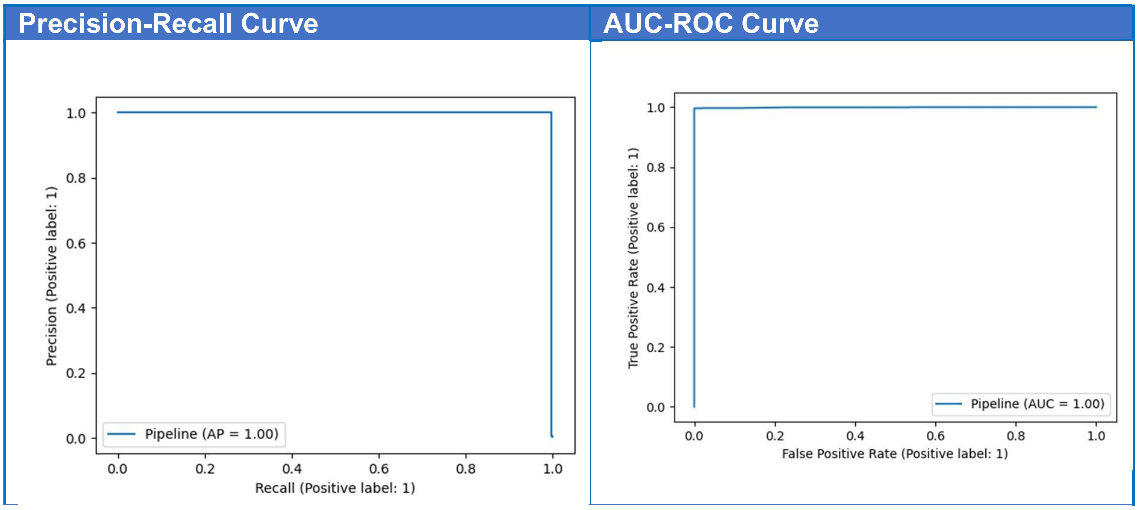


Figure 23 – CatBoost Confusion Matrix


Precision-Recall Curve

AUC-ROC Curve

Figure 24 – CatBoost: Precision-Recall and AUC-ROC Curves

Hyper-parameter	Value
iterations	5
learning_rate	0.1
random_state	88

Table 21 – CatBoost Hyper-parameters

8.7 LightGBM

LightGBM also belongs to the family of Gradient Boosting algorithms. It utilizes the patterns in the residual errors and strengthens a model with weak predictions. This classifier uses Gradient-based One-Side Sampling (GOSS) to filter out the data instances to decide which attribute or feature we will use to split the tree and Exclusive Feature Bundling (EFB).

In GOSS, the instances with larger gradients (i.e., under-trained instances) will contribute more to the information gain. GOSS will keep those instances with large gradients (e.g., larger than a predefined threshold, or among the top percentiles), and randomly drop those instances with smaller gradients to retain the accuracy of information gain estimation.

EFB is a feature reduction technique for sparse, high dimensional data. An exclusive feature bundle is where features that are mutually exclusive can be safely bundled into a single feature

LightGBM grows a tree vertically (leaf-wise) whilst other algorithms (XGBoost) grow trees horizontally (level-wise).

The experimental results after running the 'LightGBM' are shown in Table 22. The confusion matrix and the precision recall curves are shown in Figure 25 and Figure 26 respectively. Table 23 shows the hyper-parameters used for training the LightGBM model.

A dump of the classification report is shown below:

```
precision    recall  f1-score   support


           0       1.00      1.00      1.00   1270881
           1       0.81      1.00      0.89      1643


    accuracy                           1.00   1272524
   macro avg       0.90      1.00      0.95   1272524
weighted avg       1.00      1.00      1.00   1272524
```

Score Type	Value
Precision Score	0.808
Recall Score	0.998
F1 Score	0.893
Balanced Accuracy	0.999
AUC ROC Score	0.999

Table 22 – LightGBM Performance Scores

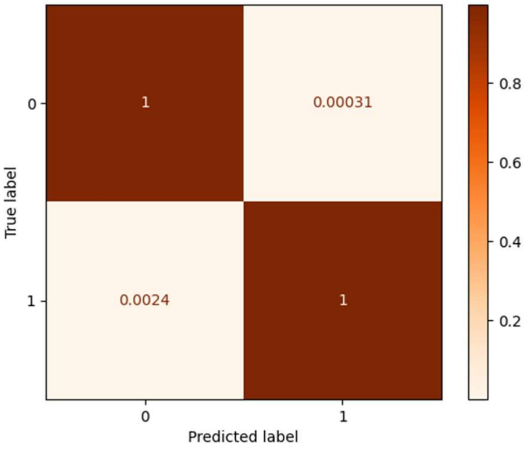


Figure 25 – LightGBM Confusion Matrix

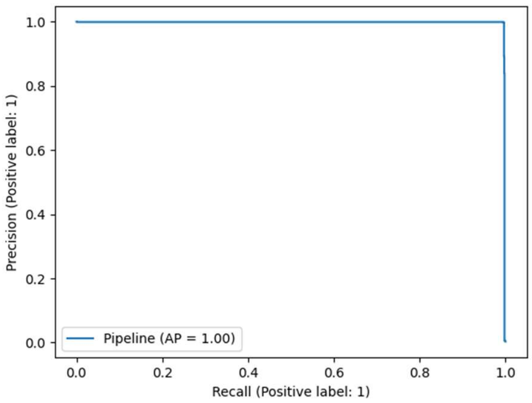
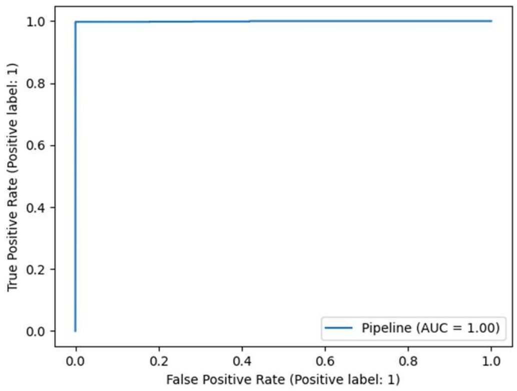
Precision-Recall Curve	AUC-ROC Curve
	

Figure 26 – LightGBM: Precision-Recall and AUC-ROC Curves

Hyper-parameter	Value
boosting_type	gbdt
learning_rate	0.1
random_state	88

Table 23 – LightGBM Hyper-parameters

8.8 Bagging Classifier

This is an ensemble method and the term “bagging” comes from bootstrap + aggregating. A bootstrap is a sample of a dataset with replacement.

A classifier is created for each of the data sample sets. The same learning algorithm is used for each of these classifiers.

Each of these classifiers will produce an output.

The final result is then obtained by averaging or majority voting.

The difference between this and the random forest classifier is that in addition to taking a random subset of data, the random forest classifier also takes a random selection of features rather than using all features to grow trees.

The experimental results after running the ‘Bagging Classifier’ are shown in Table 24. The confusion matrix and the precision recall curves are shown in Figure 27 and Figure 28 respectively. Table 25 shows the default parameters used training the bagging classifier model.

A dump of the classification report is shown below:

```
precision    recall  f1-score   support


        0       1.00      1.00      1.00   1270881
        1       0.86      0.99      0.92      1643


   accuracy                         1.00   1272524
  macro avg       0.93      1.00      0.96   1272524
weighted avg       1.00      1.00      1.00   1272524
```

Score Type	Value
Precision Score	0.858
Recall Score	0.995
F1 Score	0.921
Balanced Accuracy	0.997
AUC ROC Score	0.997

Table 24 – Bagging Classifier Performance Scores

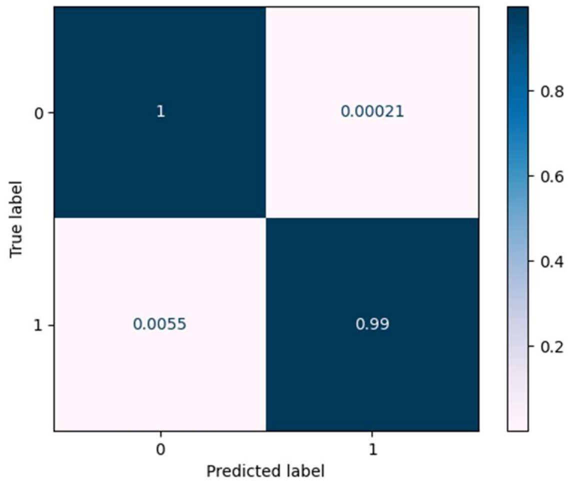


Figure 27 – Bagging Classifier Confusion Matrix

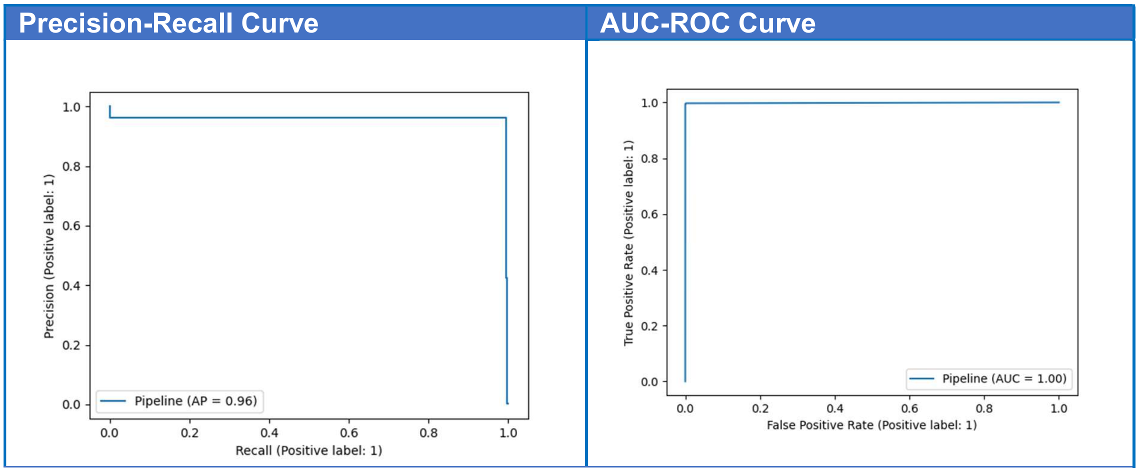


Figure 28 – Bagging Classifier: Precision-Recall and AUC-ROC Curves

Hyper-parameter	Value
base_estimtator	DecisionTreeClassifier(random_state=88)
random_state	88

Table 25 – Bagging Classifier Hyper-parameters

9 Results Summary

Table 26 shows the summary scores for all the experiments we carried out on this dataset.

	Recall	Precision	F1 Score	Balanced Accuracy
Logistic Regression	0.943	0.023	0.044	0.945
Gaussian Naïve-Bayes	0.993	0.003	0.006	0.783
XGBoost	0.996	0.826	0.903	0.998
Decision Tree Classifier	0.995	0.975	0.985	0.997
Random Forest Classifier	0.995	0.867	0.927	0.997
CatBoost	0.996	0.984	0.99	0.998
LightGBM	0.998	0.808	0.893	0.999
Bagging Classifier	0.995	0.858	0.921	0.997

Table 26 – Summary Scores

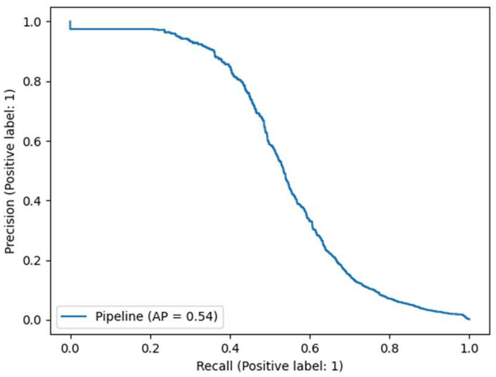
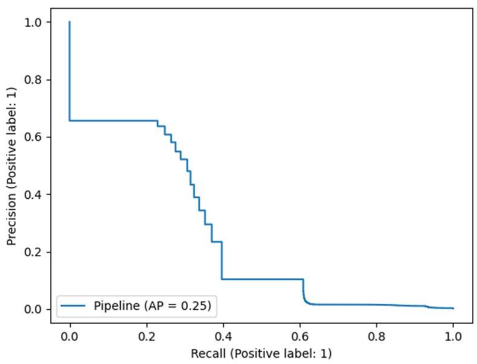
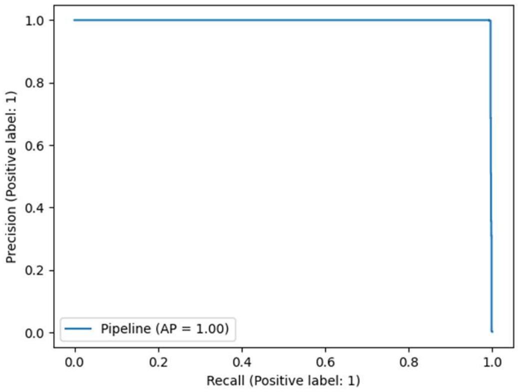
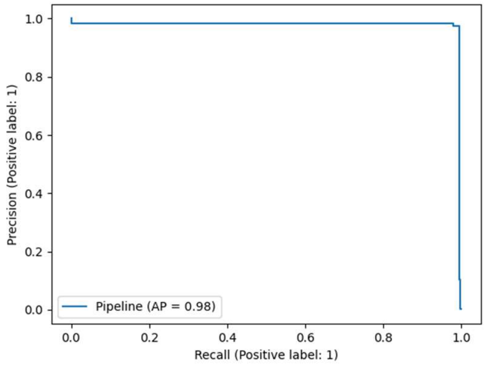
The results show a close race between *CatBoost* and *LightGBM*.

Recall is a better measure to use compared to precision or accuracy in cases when there is a high cost associated with false negatives (e.g., fraud detection or sick patient detection).

Accuracy does not work in in imbalanced datasets (as in this case). Here we have 99% more non-fraudulent data and less than 1% fraud data. If we predict all data points as a normal class, we will at least get 99% correct. This is called accuracy paradox.

Precision is a good measure to determine when the cost of false positives is high e.g., a movie recommender system which drives value by consistently delivering suggestions the user will enjoy. Precision is the priority in this scenario because every suggestion must be a correct prediction to maintain usability. For fraud detection, precision will not be a good measure.

The F1-score can be used primarily to compare the performance of two classifiers. LightGBM has a recall of **0.998** compared to CatBoost, whose recall is **0.996**. CatBoost has a higher precision **0.984** compared to LightGBM **0.808**. In this case, the F1-scores for both the classifiers can be used to determine which one produces better results. The F1-score for CatBoost is **0.99** whilst that of LightGBM is **0.893**. We can conclude that **CatBoost is the better classifier**.

Precision-Recall Curves for the different classifiers	
Logistic Regression PR-Curve	**Gaussian Naïve-Bayes PR-Curve**
	
XBGoost PR-Curve	**Decision Tree Classifier PR-Curve**
	
Random Forest Classifier PR-Curve	**CatBoost PR-Curve**

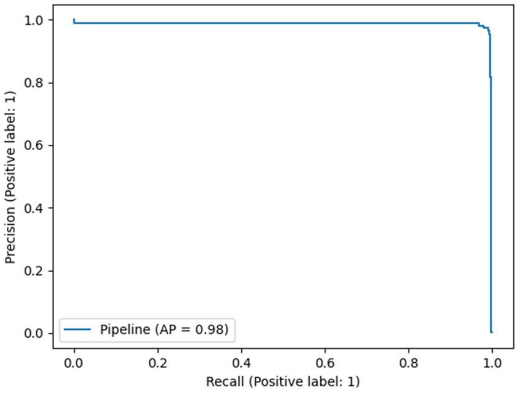
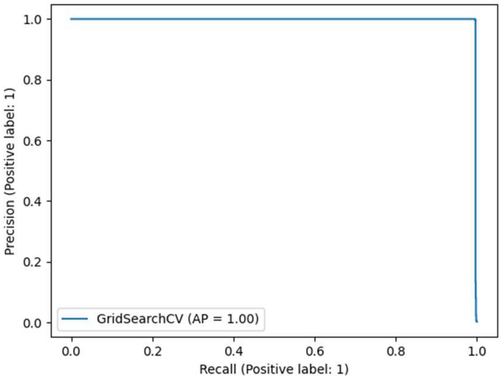
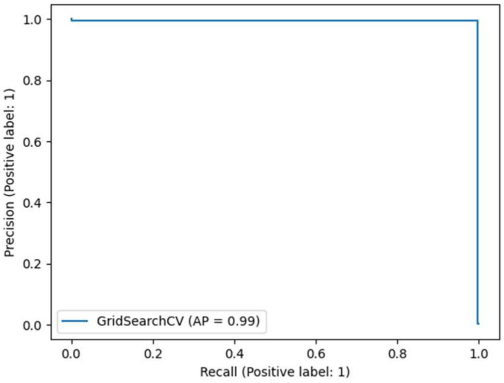
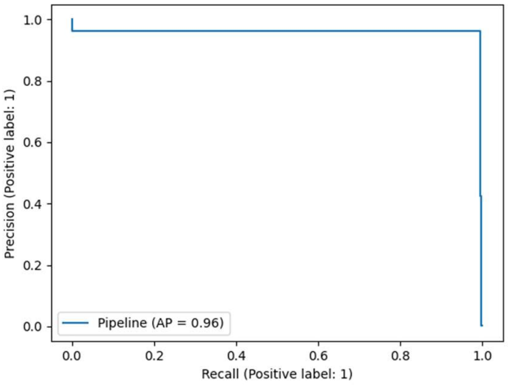
Precision-Recall Curves for the different classifiers	
	
LightGBM PR-Curve 	**Bagging Classifier PR-Curve** 

Table 27 – Precision- Recall (PR) Curves for the different classifiers

Table 27 shows the PR curves for the different classifiers used in our experiments. We observe that all the ensemble methods (random forest, CatBoost, LightGBM and Bagging Classifiers) exhibit near perfect precision recall curves. The Decision Tree Classifier also exhibits a near perfect precision recall curve. The PR-curves for Logistic Regression and Naïve-Bayes Classifier display fairly poor results when compared to the ensemble and decision tree classifiers.

10 Grid Searching to find best parameters

Given the close race between the CatBoost and LightGBM classifiers, the next steps would be to apply GridSearch to determine if we can narrow down and find the best parameters to use. Once the best parameters have been found, we will need to run the experiments to evaluate the resulting metrics from the experimental runs to determine the final model we will use in our deployment.

The parameters that we selected to apply grid searching on are the important parameters that control overfitting [2].

The maximum number of iterations used for both was 25.

For CatBoost, the parameters we used to tune were:

- Learning rate
- Depth
- L2 leaf regularization

Likewise, for LightGBM, the following parameters were chosen for tuning:

- Learning rate
- Maximum depth
- Minimum data in leaf

Table 28 show the parameter values that we use as a basis to start the parameter search and the results of grid search are displayed under the 'best parameters' column (highlighted in yellow).

CatBoost	**Values**	**Best Parameters**	**LightGBM**	**Values**	**Best Parameters**
Learning Rate (This determines how fast or slow the model will learn. The default is usually 0.03)	0.001, 0.01, 0.5	0.01	Learning Rate (Default: 0.1)	0.001, 0.01, 0.5	0.01
Depth (up to 16) (tree depth)	4, 6, 10	10	Max depth (Controls the max depth of each trained tree; If you use a large value of **max_depth**, your model will likely be **over fit** to the train set)	10, 50	10
L2-leaf-reg (Coefficient at the L2 regularization term of the cost function. The default is 3.0)	10, 15, 25	10	Min Data in Leaf: the minimum number of data/sample/count per leaf (default is 20; lower min_data_in_leaf means less conservative/control, potentially overfitting).	500, 1000	1000

Table 28 – Best Parameters after GridSearch applied

Using the best parameters that were returned by GridSearch, we use these to retrain our model. Table 29 shows the improvement in the precision and F1 scores for both CatBoost and LightGBM when using the new parameter set (items highlighted in yellow).

Score Type	CatBoost (Old Hyper Parameters)	CatBoost	Light GBM (Old Hyper Parameters)	Light GBM
Precision Score	0.984	0.993	0.808	0.888
Recall Score	0.996	0.996	0.998	0.997
F1 Score	0.99	0.995	0.893	0.939
Balanced Accuracy	0.998	0.998	0.999	0.998
AUC ROC Score	0.998	0.998	0.999	0.998

Table 29 – Performance Scores using best parameters found by GridSearch

We again use the F1-score to compare the performance of two classifiers. LightGBM has a recall of **0.997** compared to CatBoost, whose recall is **0.996**. CatBoost has a new higher precision **0.993** compared to LightGBM's new precision score of **0.888**. In this case, the F1-scores for both the classifiers can be used to determine which one produces better results. The new F1-score for CatBoost is **0.995** whilst the new F1-score for LightGBM is **0.939**. From this, we again conclude that **CatBoost is the better classifier**.

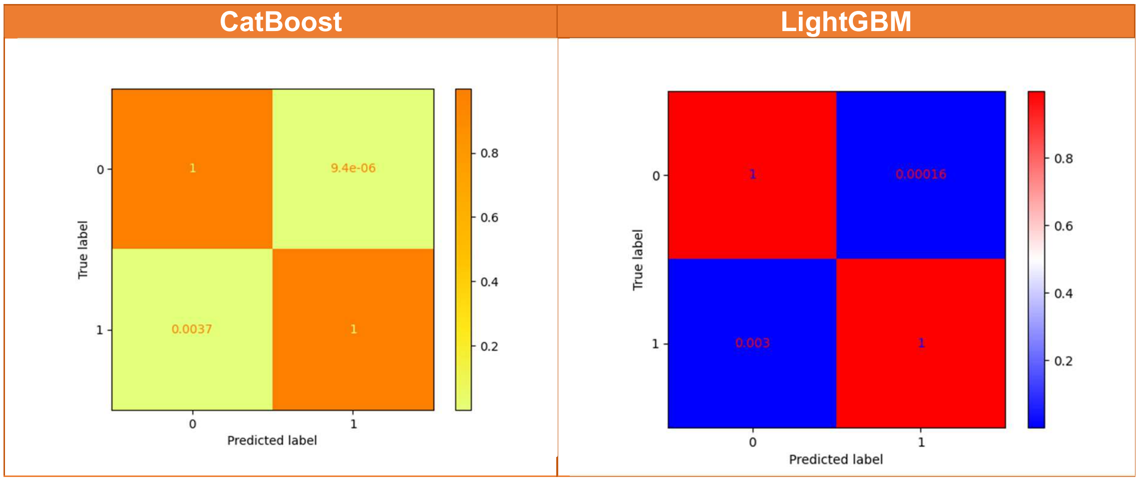


Figure 29 – Confusion Matrix CatBoost and LightGBM using best hyper-parameters found by GridSearch

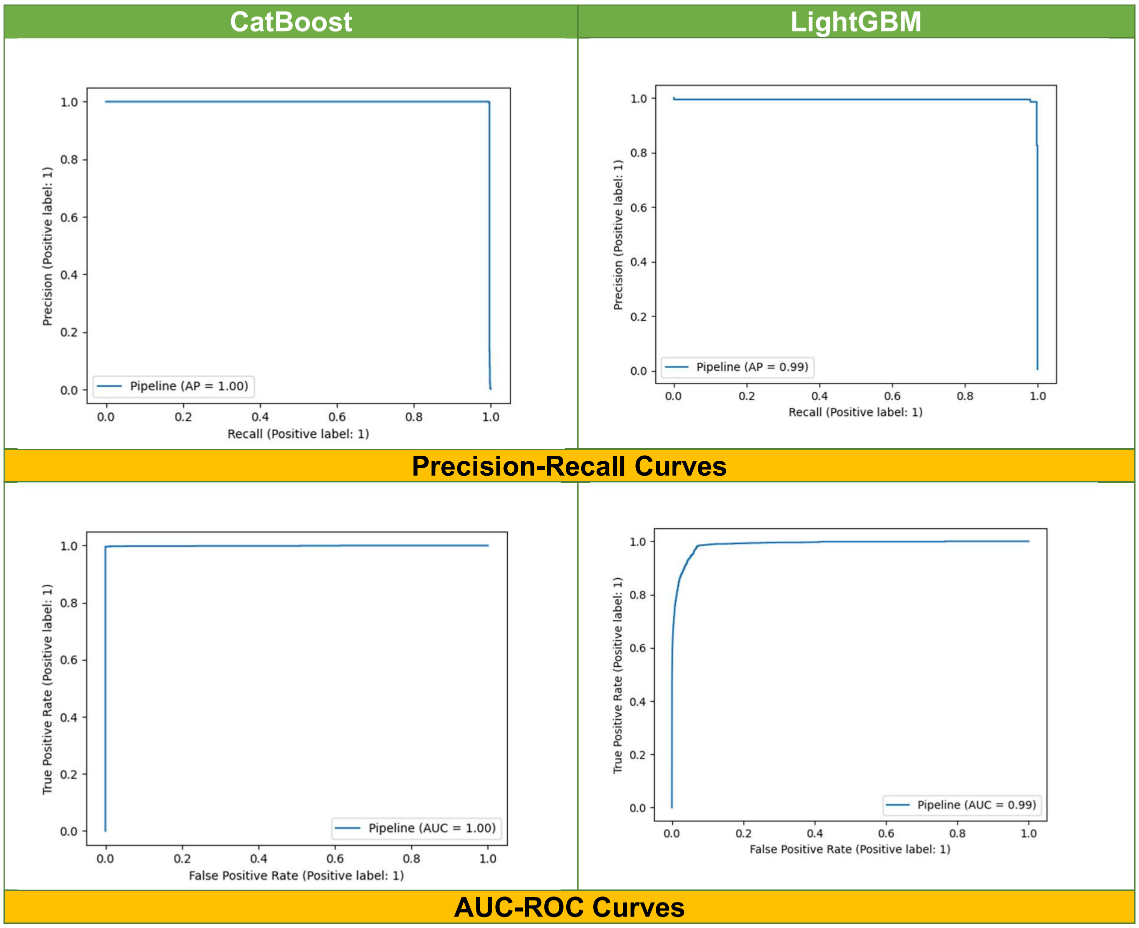


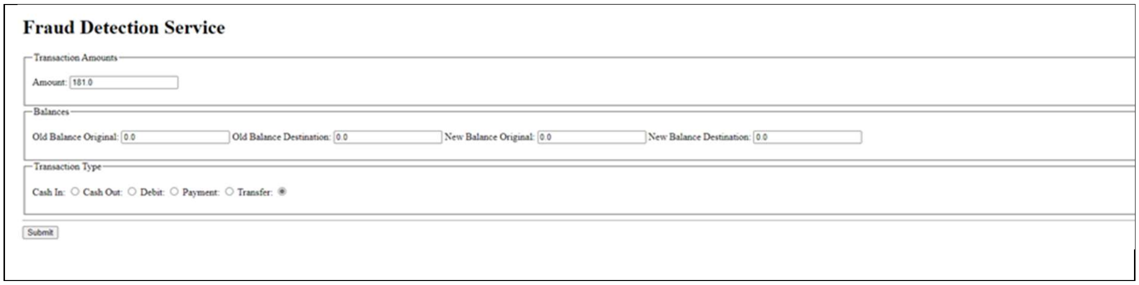
Figure 30 – CatBoost and LightGBM Precision-Recall and AUC-ROC Curves using best hyper-parameters found by GridSearch

Again, the PR-curves for CatBoost and LightGBM show similar characteristics – near perfect precision-recall curves (Figure 30), even after applying Grid Search to determine the best parameters.

11 Deployment

We have created a simple Flask-based web application where users are able to input a set of features, (Amount, Initial Sender Balance, New Sender Balance, Initial Recipient Balance, New Recipient Balance, Sender Amount Difference, Recipient Amount Difference & Transfer type), as parameters to query our model and it will generate an output on whether the transaction is a fraudulent or not (1 for fraudulent, 0 for genuine).

The screen shots for our web-based flask application are shown Figure 31. Results returned as fraudulent or normal are shown in Figure 33 and Figure 32 respectively.



Fraud Detection Service

Transaction Amounts

Amount: 181.0

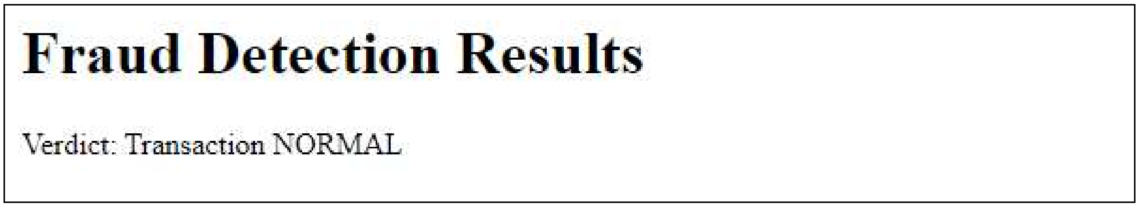
Balances

Old Balance Original: 0.0 Old Balance Destination: 0.0 New Balance Original: 0.0 New Balance Destination: 0.0

Transaction Type

Cash In: ○ Cash Out: ○ Debit: ○ Payment: ○ Transfer: ●

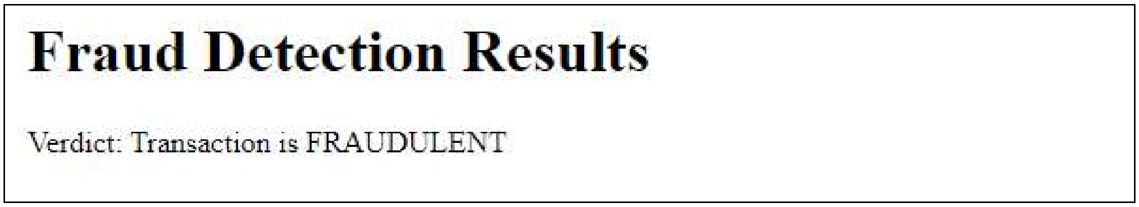
Figure 31 – Fraud Detector Web Front End



Fraud Detection Results

Verdict: Transaction NORMAL

Figure 32 – Results Returned – OK



Fraud Detection Results

Verdict: Transaction is FRAUDULENT

Figure 33 – Results Returned – FRAUD

12 Conclusion and Future Work

This project investigated the problem of detecting fraud in financial transactions. As financial transactional data is hard to come by, we used the dataset from Kaggle. This has been generated by the PaySim mobile generator [1]. It simulates mobile money transactions based on a sample of real transactions extracted from one month of financial logs from a mobile money service implemented in an African country.

The work is broken down into several parts and they are listed as follows:

- Exploratory Data Analysis
- Feature Engineering
- Experiments and analysis of results
- Applying Grid Search to narrow down the best parameters on the top 2 models
- Deploying the solution as a Web Application

A number of different algorithms were tried. These ranged from the simple like Logistic Regression to more complex models (ensemble techniques).

Results from the experimental runs have demonstrated that “LightGBM” gave the highest recall score of **0.997** followed closely by “CatBoost”, with a recall score of **0.996**. The difference between the 2 scores is only 0.001. Only the precision and F1 scores showed an improvement. Precision scores are not as useful compared to recall in the case of fraud detection. The F1 score for “CatBoost” is higher (**0.995**) compared to “LightGBM” (**0.939**) – a difference of 0.056. This means that CatBoost is more accurate compared to LightGBM on the basis of the F1 score values.

From this, we conclude that CatBoost is the better classifier compared to LightGBM.

Finally, as next steps, we should consider using Bayesian Optimization [3]**Error! Reference source not found.** for hyper-parameter tuning. Given that hyper-parameter tuning is a very time-consuming task, we have only tried a small subset of hyper-parameters. Bayesian optimization promises to be faster way to tune hyper-parameters. It is an approach that uses Bayes Theorem to direct the search in order to find the minimum or maximum of an objective function. This is most useful for objective functions that are complex, noisy, and/or expensive to evaluate and can be potentially used to help in hyper-parameter tuning.

We can also study how we can apply deep learning and neural networks to this problem set.

13 Reference

[1]. Edgar Lopez-Rojas, Synthetic Financial Datasets For Fraud Detection

[2]. Alvira Swalin, CatBoost vs. Light GBM vs. XGBoost

[3]. Somang Han, Tuning Hyperparameters Under 10 Minutes (LGBM)