Vietnam National University, Ho Chi Minh City
University of Technology
Faculty of Computer Science and Engineering

Assignment

Building machine learning model to detect fraudulent financial transactions

---

Course: Machine learning - CO3117

Instructor: Dr. Huỳnh Văn Thống

Sinh viên thực hiện: Nguyễn Lâm Huy 2311188
Nguyễn Tất Kiên 2311735
Nguyễn Thành Minh Khôi 2311687

21st November 2025

Duty Roster

No.	Name	StuID	Work	% Progress
1	Nguyễn Tất Kiên (L03)	2311735	- Algorithm research - Handle imbalance and split data - Write report section 3	100%
2	Nguyễn Thành Minh Khôi (L01)	2311687	- EDA - Feature Engineering - Data Preprocessing - Write report Section 2	100%
3	Nguyễn Lâm Huy (L03)	2311188	- Evaluation - Hyperparameters Tuning - Models Comparing - Write report Section 4	100%

Contents

1 Introduction

Financial fraud poses a significant and growing threat to individuals, businesses, and global economies, causing billions of dollars in losses each year. Fraudulent activities are becoming increasingly sophisticated, making them harder to detect with traditional rule-based systems. These conventional approaches are often rigid, fail to adapt to new attack strategies, and tend to generate high false alarm rates. To address these challenges, machine learning (ML) has emerged as a powerful tool for fraud detection, offering adaptability, scalability, and the ability to uncover complex patterns hidden in large datasets.

In this project, we investigate the effectiveness of three machine learning models in detecting fraudulent financial transactions. Our focus is on both supervised and unsupervised learning techniques, including Random Forests, Gradient Boosting (XGBoost), and Isolation Forest. The models are trained and evaluated on a dataset containing labeled financial transactions-Financial Transactions Dataset for Fraud Detection which is published on Kaggle, where fraudulent cases make up only a small fraction of the data (about 4%).

The performance of each model is rigorously compared using evaluation metrics tailored to fraud detection: precision, recall, F1-score, and the area under the ROC curve (AUC). Since accuracy alone can be misleading in highly imbalanced datasets, we prioritize recall to ensure that fraudulent cases are identified, and precision to minimize false positives.

By combining general insights on fraud detection with a comparative study of these ML models, this project illustrates both the opportunities and trade-offs in applying machine learning to financial security. Overall, the project underscores the potential of ML-driven fraud detection systems to complement existing security infrastructures, providing adaptive, accurate, and scalable solutions to combat financial crime.

Here is our Github link that contain the assignment: Financial-Transaction-Fraud-Detection

2 Data Processing

2.1 Introduction to Dataset

The dataset our group use is an open-sourced dataset from Kaggle, which is a community-friendly platform with several datasets and competitions relating to Machine Learning, Deep Learning, Data Analysis, etc. We can look it up at Financial-Transactions-Dataset-for-Fraud-Detection.

This dataset contains 5 million synthetically generated financial transactions designed to simulate real-world behavior for fraud detection research and machine learning applications. Each transaction record includes fields such as:

- Transaction Details: ID, timestamp, sender/receiver accounts, amount, type (deposit, transfer, etc.)
- Behavioral Features: time since last transaction, spending deviation score, velocity score, geo-anomaly score
- Metadata: location, device used, payment channel, IP address, device hash
- Fraud Indicators: binary fraud label (is_fraud) and type of fraud (e.g., money laundering, account takeover)

The dataset follows realistic fraud patterns and behavioral anomalies, making it suitable for:

- Binary and multiclass classification models
- Fraud detection systems
- Time-series anomaly detection
- Feature engineering and model explainability

The data set was generated programmatically using a custom Python script with NumPy, CSV, and datetime libraries. The following practices were used: Time-based simulation of transactions for one year (2023). Account behavior and geography-based anomalies to flag fraud. Transaction amounts sampled using normal, exponential and lognormal distributions according to transaction type. Behavioral features (spending_deviation_score, velocity_score, geo_anomaly_score) were calculated per account. Fraudulent transactions were injected using realistic conditional rules (e.g., large transfer + geo-shift = possible laundering).

How to import dataset in our Python/Jupyter Notebook:

```
import kagglehub

# Download latest version
path = kagglehub.dataset_download("aryan208/financial-transactions-
    dataset-for-fraud-detection")

print("Path to dataset files:", path)
```

2.2 Exploratory Data Analysis (EDA)

This dataset has 18 columns and is_fraud is the label column indicating fraudulent or legitimate transactions. In further analysis, we explore and draw charts based on the relevance of other features to is_fraud label.

Data Overview:

- Shape: (5000000, 18)
- Missing value:
  - fraud_type: 4,820,447
  - time_since_last_transaction: 896,513
- Class distribution:
  - Fraudulent transactions: 179,553 (3.59%)
  - Legitimate transactions: 4,820,447 (96.41%)

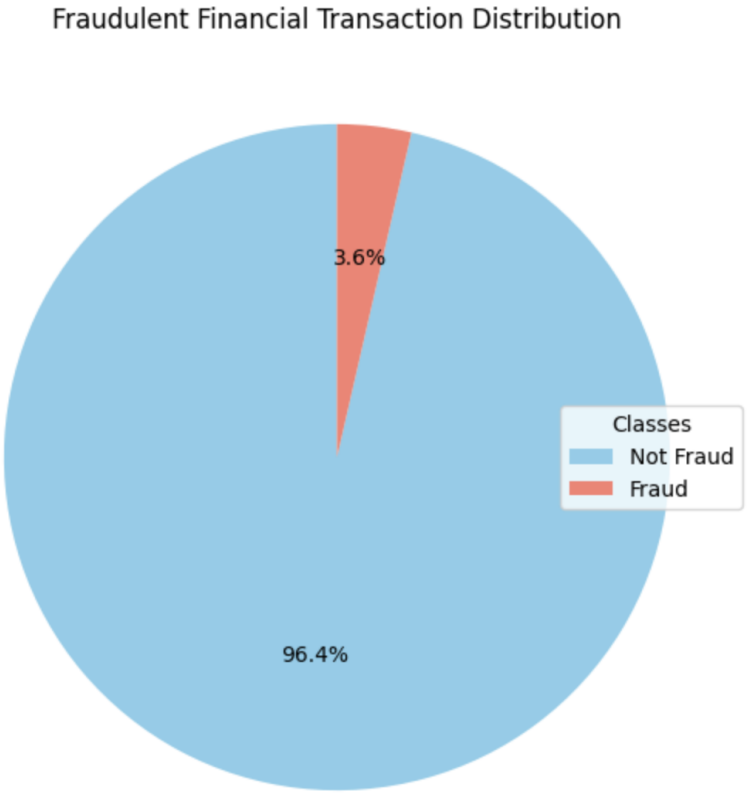

Figure 1: Class Distribution Overview

Timestamp feature is in ISO-format (for example, 2023-08-22T09:22:43.516168) and this may cause some confusion for further models training so we convert it into other feature columns such as hour, day, day_of_week and month. The plot below shows that the fraudulent transactions vary in different months, mostly in July with 15,395 transactions whereas February records the lowest with 13,777 in total. However, the numbers are not fluctuated to much because there are 11 months around 15,000 fraud transactions. If this data is real, it will cause a major damage for financial companies and organizations and mess up all measurement tools monthly.

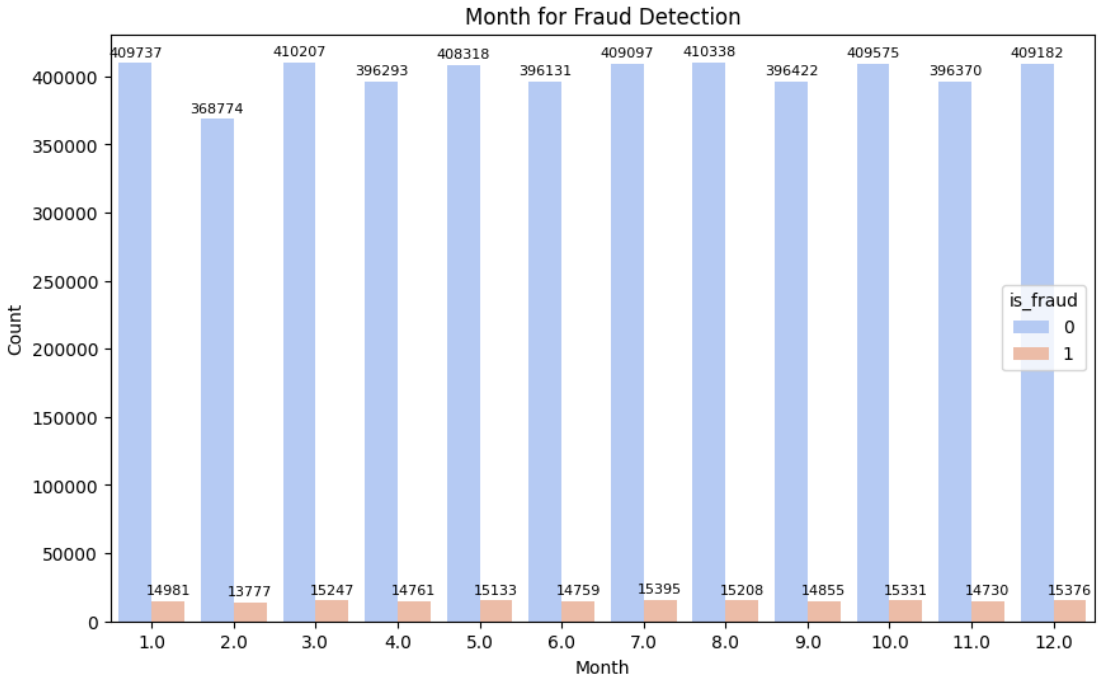

Figure 2: Fraud Detection by Month

According to day of month check for fraud transactions, we can observe that the numbers are distributed quite equally, mostly more than 5,000 transactions a day. The highest number of fraud transactions is on 9th day of the month, whilst the lowest is on the last day of the month. This can be explained that at some month, it does not have the 31st day or even the 30th day, for example, February, so the statistic is not even at the end of the month.

Although, the total number of fraudulent transactions in some months and days is high, but surprisingly, if we look at fraud rate (or fraud percentage), it will peak at the end of the month and the start of the year. In month feature, the top 10 fraud percentage vary from the 2nd day to the 12th day with the percentage approximately at 3.6%. In day of month feature, the top 10 fraud gives us information that fraudulent acts are usually made at the beginning and the end of the month. The later average percentage is also 3.6%.

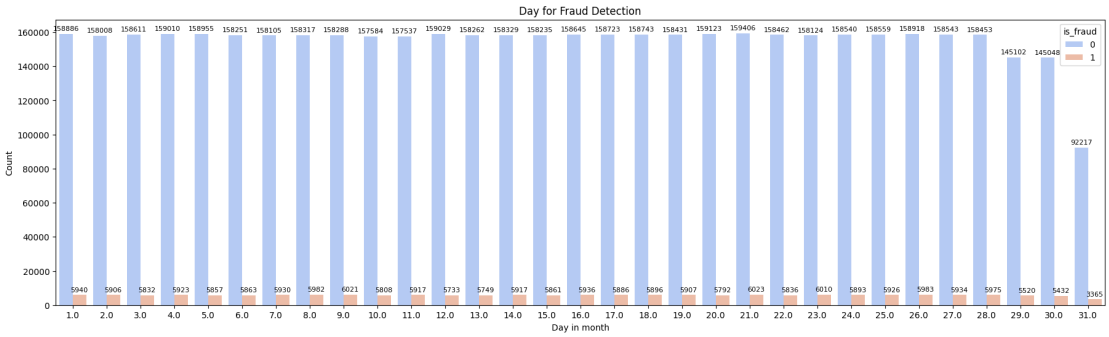

Figure 3: Fraud Detection by Day

Although, the dataset consists of 5 millions rows of data, the unique number of sender/receiver account (ID) is not totally the same. Through statistic, we get:

- Number of senders: 896513
- Number of receivers: 896639

In sender feature, user ACC983922 has the most active transaction history with 20 transactions in the dataset, whereas in the receiver feature, user ACC400278 has in total 24 transactions. This can give use some clue about fraudulent transactions when the total number of transaction is too suspicious.

sender_account	total_trans	fraud_trans	fraud_per (%)
ACC802400	4	3	75
ACC741428	4	3	75
ACC340345	4	3	75
ACC774428	4	3	75
ACC952479	4	3	75
ACC111101	4	3	75
ACC921829	4	3	75
ACC690487	4	3	75
ACC578240	4	3	75
ACC509812	4	3	75

Table 1: Sender Account Fraud Rate

Nevertheless, we should consider fraud percentage instead of the total number because some accounts may legitimately transact actively due to personal reason or job requirement. Table 1 shows the top 10 sender accounts that have the highest percentage of fraud and shows the same total transactions, fraud transactions, and fraud percentage of 4, 3 and 75%, respectively. These accounts are severely dangerous and must be banned in order to keep the transaction history integrated. This is also working with receiver accounts but the dataset is not so varied so we should not make the decision about it too early.

receiver_account	total_trans	fraud_trans	fraud_per (%)
ACC239502	2	2	100
ACC988121	1	1	100
ACC119203	1	1	100
ACC392984	1	1	100
ACC630181	1	1	100
ACC531033	1	1	100
ACC157216	1	1	100
ACC878661	1	1	100
ACC859588	1	1	100
ACC141744	1	1	100

Table 2: Receiver Account Fraud Rate

The amount of money that have been transacted is also a great feature for fraud detection. If an account transfers a very suspiciously large amount of money, it will give off some fraudulent features. However, we have scaled amount feature by log plus 1 in order to reduce skewed because some people transfer a little of money and some do severely many. Figure 4 is the log scale plot

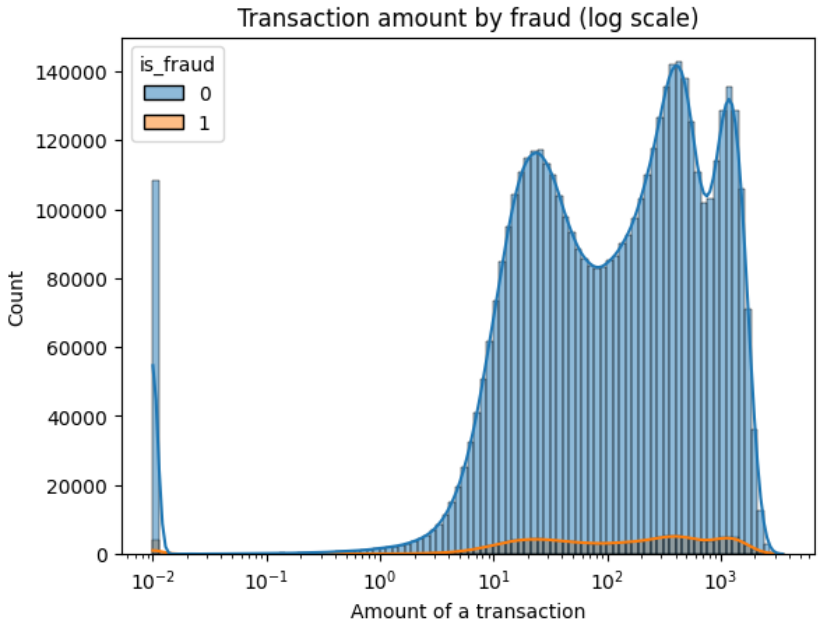

Figure 4: Fraud Detection by Amount of Transaction

that illustrates 2 opposite sides of the amount of money that varies from 2 heads of the chart. But mostly, it is from 10 to 1,000 log value.

We want to understand more about the dataset and merchant category can give a hint of fraudulence so we use a table and a chart to observe it.

merchant_category	total_trans	fraud_trans	fraud_rate (%)
entertainment	625332	22573	3.61
other	624589	22556	3.61
grocery	624954	22516	3.60
travel	625656	22503	3.60
online	623581	22324	3.58
restaurant	625483	22367	3.58
retail	626319	22453	3.58
utilities	624086	22261	3.57

Table 3: Fraud Detection by Merchant Category

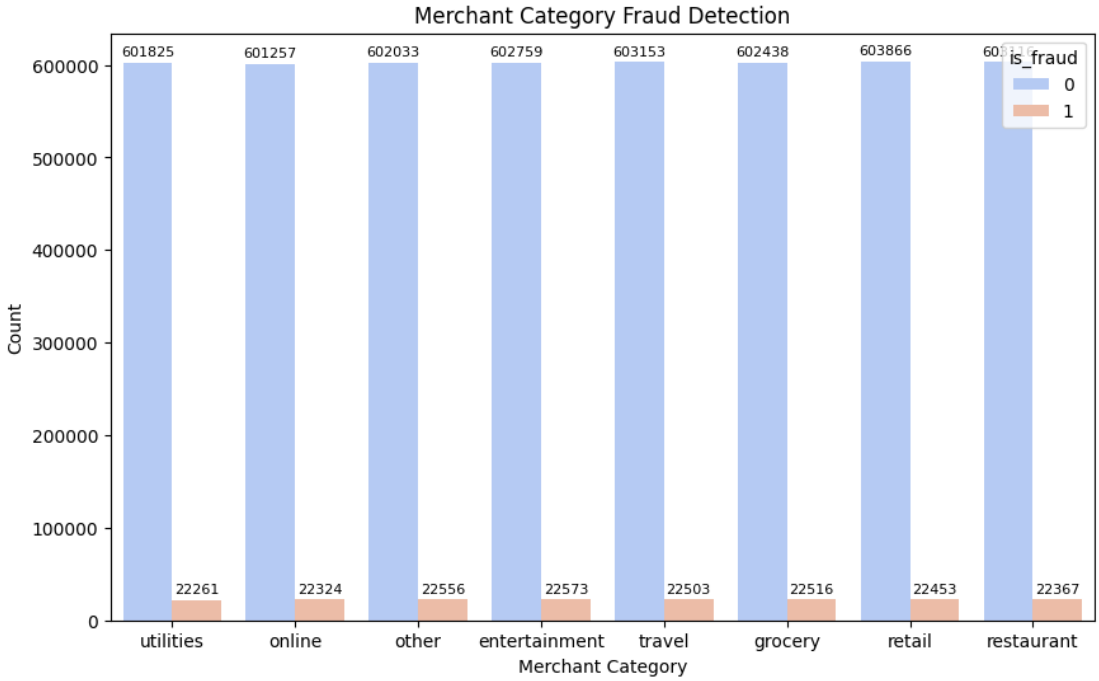

Figure 5: Fraud Detection by Merchant Category

We can see that the highest fraud rate is in entertainment and other categories at 3.61%, whilst the lowest is in the utilities category at 3.57%. However, the numbers in Table 3 and Figure 5 are approximately the same as those surround 3.6%. If the dataset is close to real world data, the frauds will choose the entertainment field as the dummy to misguide financial organization, and the least we can detect a fraud is from a utilities payment.

Likewise merchant category, transaction type can give us a hint of fraud activities.

The highest fraud percentage of transaction type is transfer at 3.63%, while the lowest fraud percentage is payment at 3.56%. Those statistics are approximately equal to each other and

transaction_type	total_trans	fraud_trans	fraud_rate (%)
transfer	1250334	45328	3.63
withdrawal	1248635	44874	3.59
deposit	1250593	44786	3.58
payment	1250438	44565	3.56

Table 4: Fraud Detection by Transaction Type

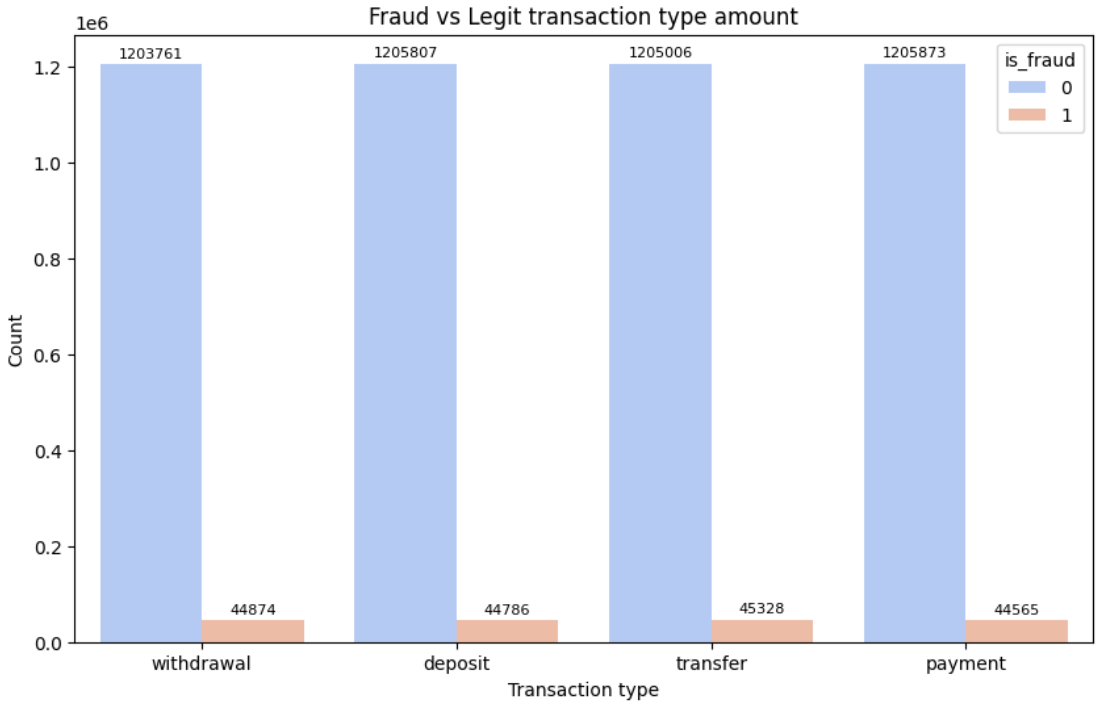

Figure 6: Fraud Detection by Transaction Type

surround at 3.6% fraud percentage. Transfer is shown to be less secured than other payment methods and this leads to lack of integrity.

Spending deviation is one of the most important features to detect fraud transactions due to abnormal over or under spending. The figure below shows the detail:

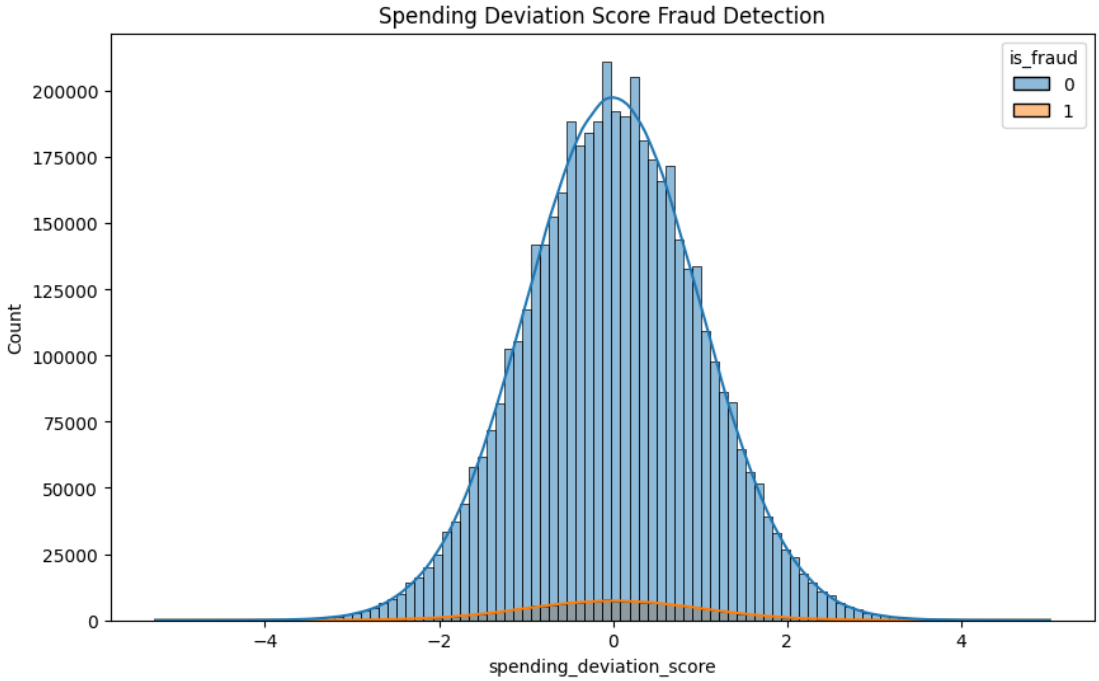

Figure 7: Fraud Detection by Spending Deviation Score

This feature's chart looks almost like a normal distribution so apparently, it is crucial for detection. It's statistics are: range from -5.2 to 5.02, mean $\approx$ 0, std $\approx$ 1. If a user suddenly increases the amount of money in a transaction from average amount of money transacted history, spending deviation score will also increase which is a possible fraud transaction.

Velocity score is also a quite good feature that it calculates the speed of an account. It ranges from 1 to 20, mean $\approx$10.5, std $\approx$5.77.

velocity_score	total_trans	fraud_trans	fraud_rate (%)
17	250353	9102	3.64
18	250777	9096	3.63
5	249956	9083	3.63
6	249818	9078	3.63
8	249065	9053	3.63

Table 5: Fraud Detection by Velocity Score

At the speed of 17, fraud rate and total fraud transaction are peaked at 9,102 transactions, while the bottom is at the speed of 15 which is 8,812 transactions. This feature can create plenty of useful features for detection because it has the speed factor in it. If an account spends money too fast, it can be fraud.

Geo anomaly score is one feature that is really crucial because it can indicate whether an account is fraud or not. It ranges from 0 to 1, mean $\approx$0.5, std $\approx$0.3.

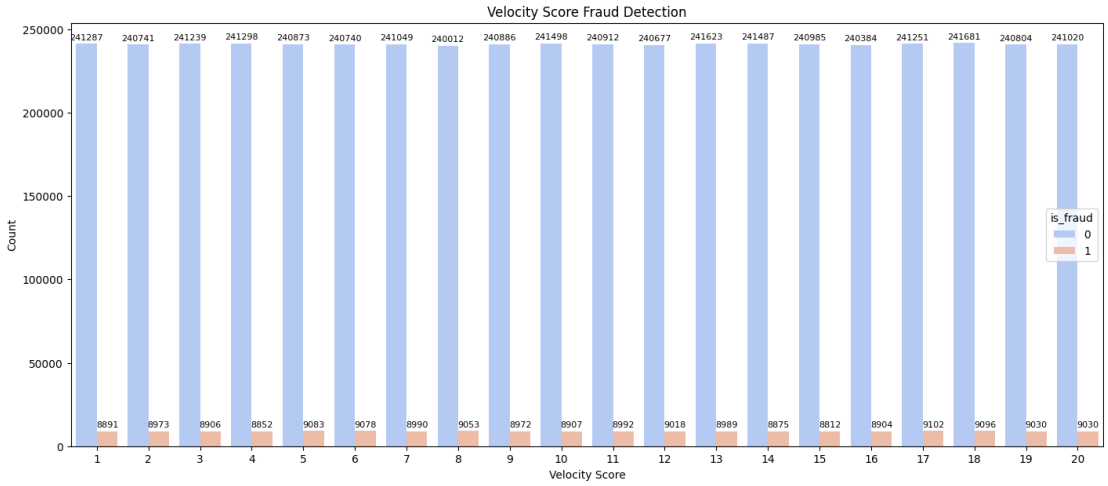

Figure 8: Fraud Detection by Velocity Score

geo_anomaly_score	total_trans	fraud_trans	fraud_rate (%)
0.04	50116	1680	3.35
0.68	49850	1692	3.39
0.15	49600	1697	3.42
0.83	49585	1694	3.42
0.67	50384	1734	3.44
0.18	49876	1721	3.45
0.47	49951	1731	3.47
0.43	50018	1738	3.47
0.08	50295	1749	3.48
0.94	50208	1746	3.48

Table 6: Fraud Detection by Geo Anomaly Score

Geo anomaly can tell a lot of an account. If an user usually transact money at a location but suddenly go thousand of kilometers away and make several transactions, this can consider fraudulent.

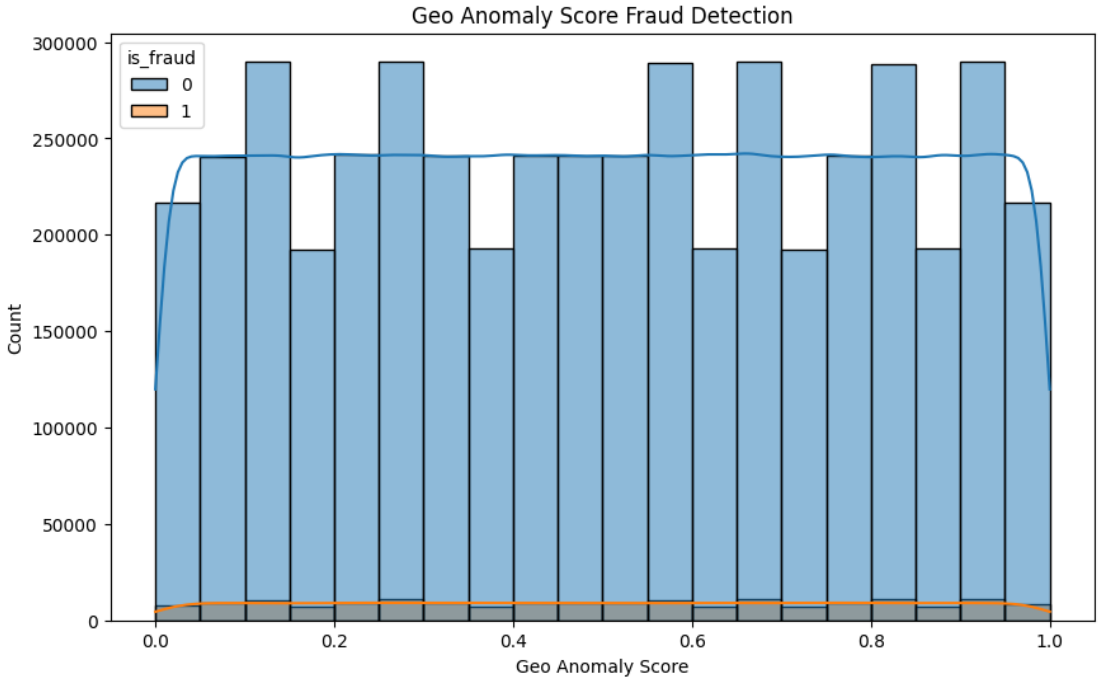

Figure 9: Fraud Detection by Geo Anomaly Score

Some cities are well-known to have a lot of fraudulent transaction due to weak financial policies or low fraud investigation rate. This dataset has 8 major cities in the world with prestigious financial policies. Figure 10 and Table 7 show the variation number of transaction in those cities.

location	total_trans	fraud_trans	fraud_rate (%)
London	624256	22478	3.60
Toronto	624349	22501	3.60
Berlin	625289	22435	3.59
New York	625354	22460	3.59
Singapore	625313	22461	3.59
Sydney	625125	22458	3.59
Dubai	624320	22340	3.58
Tokyo	625994	22420	3.58

Table 7: Fraud Detection by Location

Toronto has the highest fraud rate and total fraud transaction at 3.6% and 22,501 transactions, respectively. According to fraud rate, London also has the most at 3.6% with 22,478 transactions. The lowest fraud rate is Tokyo at 3.58% which is slightly lower than the highest 2%. Overall, those cities have equally total fraud transaction and fraud rate, surround at 3.6% and more than 22,000 transactions.

This dataset has 4 payment channels and those are the major channels of the world wide for money transaction.

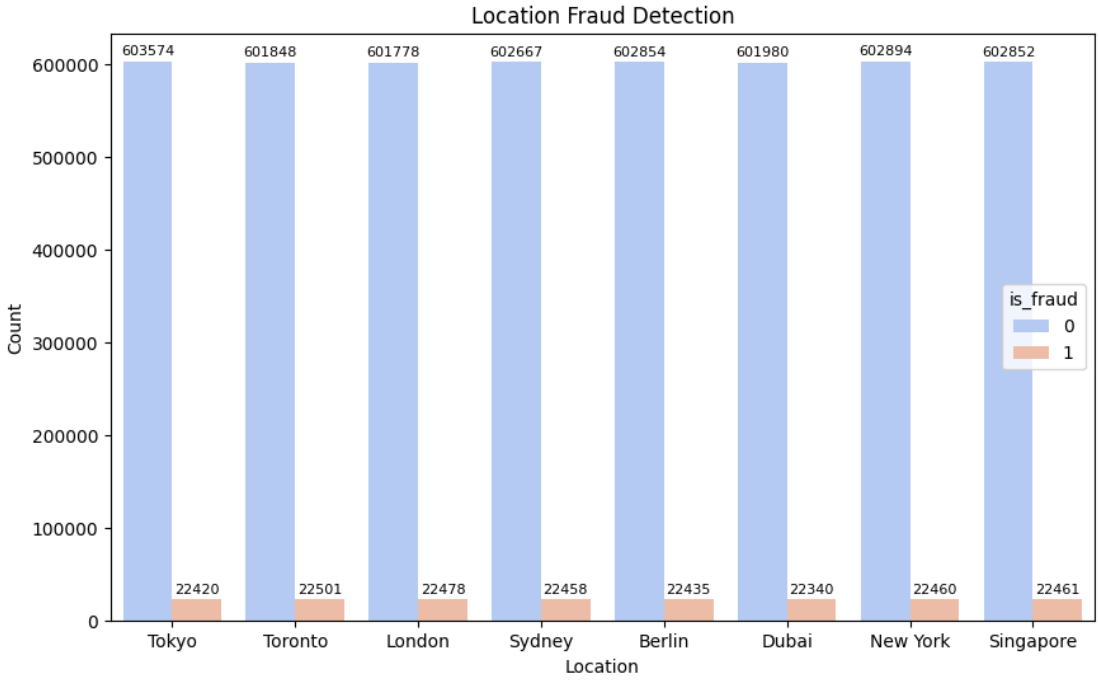

Figure 10: Fraud Detection by Location

payment_channel	total_trans	fraud_trans	fraud_rate (%)
wire_transfer	1251219	45034	3.60
UPI	1248847	44896	3.59
card	1249693	44885	3.59
AC	1250241	44738	3.58

Table 8: Fraud Detection by Payment Channel

Wire transfer is recently the most popular payment channel and this is the very easy way to make illegal transaction. It has the highest fraud rate at 3.6% and also, the highest number of fraud transaction with 45,034 transactions. AC is less popular than the other channels so it has the lowest fraud rate at 3.58 % and the lowest number of fraud transaction with only 44,738 transactions. However, those payment channel's fraud rates are not very fluctuated so a fraud can use different channels for misguide detection.

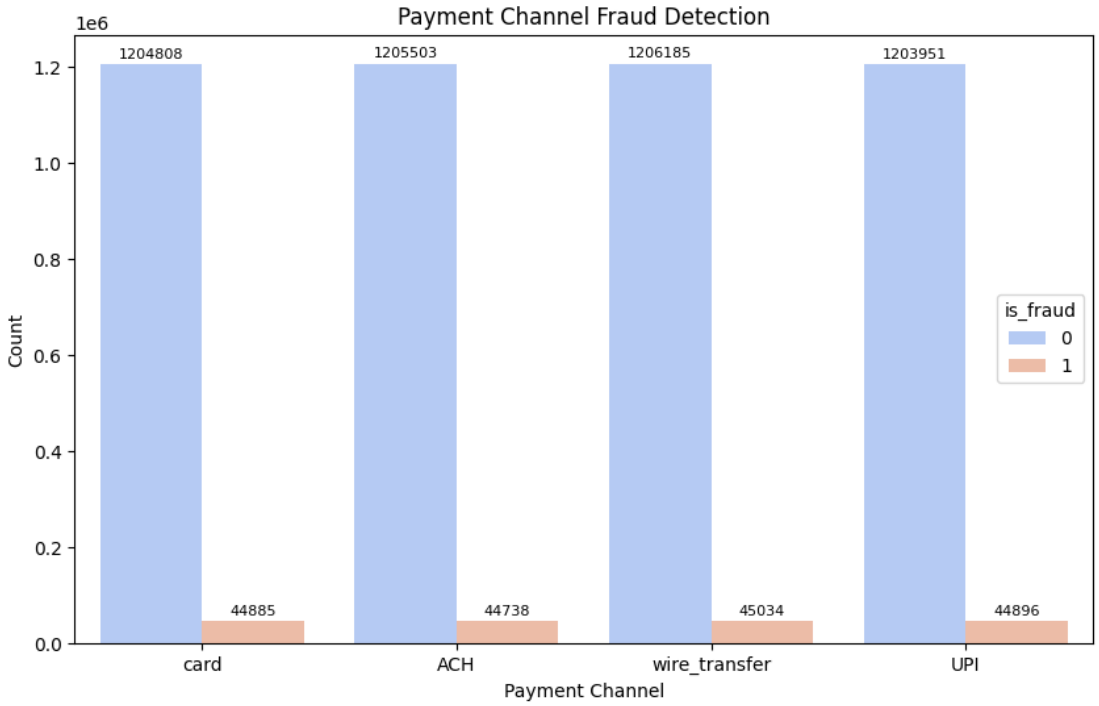

Figure 11: Fraud Detection by Payment Channel

Some features such as IP address, device hash and fraud type are not very important because those include a lot of unique values which make it hard to analyze so we do not consider it so much. For example, Table 9 shows very little information for analyzing which pattern IP address relating to fraud transaction.

ip_address	total_trans	fraud_trans	fraud_rate (%)
5.86.19.78	1	1	100.0
171.154.195.161	1	1	100.0
194.252.110.138	1	1	100.0
103.235.138.44	1	1	100.0
35.45.58.149	1	1	100.0
192.20.163.22	1	1	100.0
145.47.73.216	1	1	100.0
45.174.227.141	1	1	100.0
248.25.209.19	1	1	100.0

Table 9: Fraud Detection by IP Address

2.3 Data Preprocessing

This dataset has some faults that may avoid models to train and predict effectively so we have to make some change about the data.

Firstly, we handle missing value as we overview data and recognize there are 2 columns include missing value are fraud type and time since last spend. However, fraud type is a useless feature because it has the same function of is_fraud feature is to label transactions so we drop it and only handle missing value for time since last transaction. We use *SimpleImpute()* function from scikit-learn library and use "mean" value as a replacement for missing value. Table 10 and Table 11 show how data distribute in time since last transaction before and after transformation.

No.	time_since_last_spend
1	
2	
3	
4	
5	

Table 10: Time Since Last Spend Before (First 5)

No.	time_since_last_spend
1	1.525799
2	1.525799
3	1.525799
4	1.525799
5	1.525799

Table 11: Time Since Last Spend After (First 5)

Secondly, we handle categorical features because most of the models we used are numeric-friendly so we try to satisfy that by encoded categorical features.The tool we use to accomplish transformation is *LabelEncoder()*(LE). LE transforms string or object value into integer type range from 0 to n_classes - 1 so we do not have to worry about missing any value. Categorical features include transaction_id, sender_account, receiver_account, transaction_type, merchant_category, location, device_used, payment_channel, ip_address and device_hash. Table 12 and Table 13 illustrate first 6 features before and after transformation of those categorical features.

transaction_id	sender_account	receiver_account	transaction_type	merchant_category	location
T100000	ACC877572	ACC388389	withdrawal	utilities	Tokyo
T100001	ACC895667	ACC944962	withdrawal	online	Toronto
T100002	ACC733052	ACC377370	deposit	other	London
T100003	ACC996865	ACC344098	deposit	online	Sydney
T100004	ACC584714	ACC497887	transfer	utilities	Toronto

Table 12: Categorical Features Before Transformation (First 5)

transaction_id	sender_account	receiver_account	transaction_type	merchant_category	location
0	774605	287290	3	7	6
11	792638	841820	3	2	7
22	630602	276316	0	3	2
33	893387	243162	0	2	5
44	482844	396342	2	7	7

Table 13: Categorical Features After Transformation (First 5)

2.4 Feature Engineering

This data is a lack of correlation as Figure 12 shows nothing in relation.

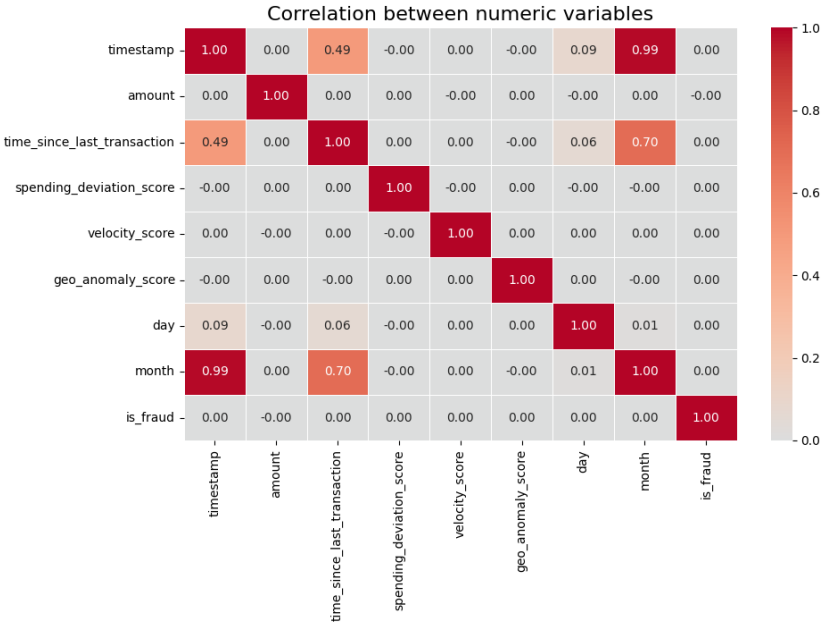

Figure 12: Correlation without Feature Engineering

As a consequence, we have to add more useful features and adjust existed features for better correlation. First, we work with amount of money by creating amount_log, amount_to_avg_ratio and amount_per_velocity. The first 2 features is only calculating log plus 1 and get the average ratio from amount but those have high impact on model training due to it's low skewed. The last is for detecting which account makes a large amount of transactions instantaneously can be very suspicious. Next, we add more features about frequency so that models can observe persistency of an account and easy to detect abnormalities. We also create some time feature to check whether the fraud activities take place at night or day, at the beginning of the week or at weekend, etc. Network between sender and receiver is an important clue to detect fraud by looking at degree, total transaction, total fraudulent transaction and fraud rate. By adding more features, the correlation matrix has become more colorful especially at is_fraud column which is good for supervised training.

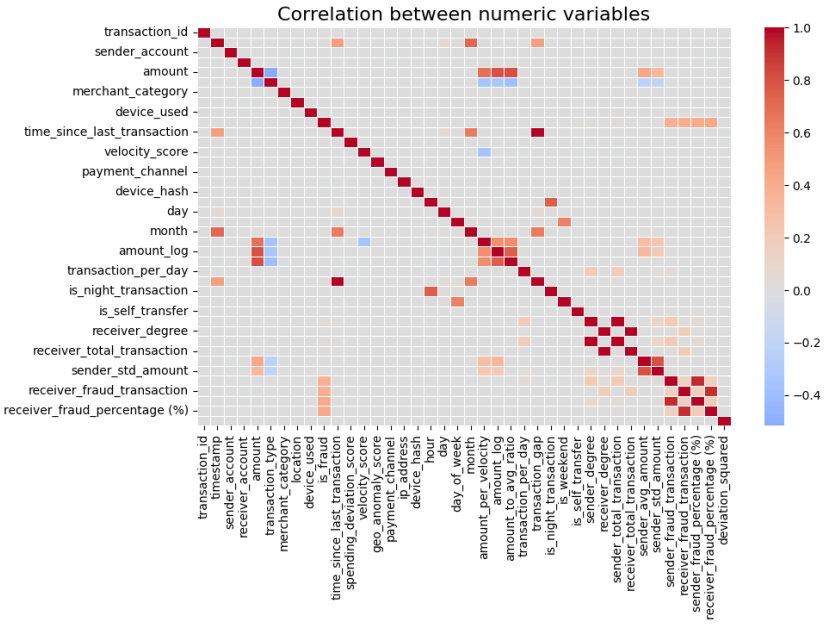

Figure 13: Correlation with Feature Engineering

3 Model Selection

Before introducing the algorithm and model, we addressed class imbalance and applied standard scaling as part of the data preparation process.

As our dataset is highly imbalance, we use undersampling method:

```
df_majority = df[df['is_fraud'] == 0]
df_minority = df[df['is_fraud'] == 1]
df_majority_downsampled = df_majority.sample(n=2*len(df_minority),
    random_state=36)
df_balanced = pd.concat([df_majority_downsampled, df_minority])
```

Split dataset to three types: train, test and validation

```
from sklearn.model_selection import train_test_split
y = df['is_fraud']
X = df.drop(columns=['is_fraud'])
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=36, stratify=y)
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,
    test_size=1/8, random_state=36, stratify=y_train)
X_train.head()
```

At this stage, the dataset has been fully preprocessed and is ready for modeling. The class imbalance issue was addressed through undersampling, and the data was split into training, validation, and test sets to enable reliable model evaluation. These steps ensure that the prepared dataset can now be utilized effectively in the subsequent modeling process.

3.1 Random Forest

3.1.1 Introduction

Random Forest is a supervised learning algorithm that can be used to solve two types of problems: regression and classification. Random Forest is an ensemble of decision trees, meaning the algorithm combines the predictions of the individual decision trees in the ensemble to make a final decision.

For the task of classifying risky transactions, the Random Forest algorithm performs the following steps:

- **Bootstrap Sampling:** A bootstrapped dataset is created using a sampling method with replacement. In scikit-learn, the size of the bootstrapped dataset is equal to the size of the original dataset by default.
- **Build Decision Trees:** A decision tree is created and trained using the bootstrapped dataset from the previous step. For each node split, the tree uses a randomly selected subset of features and finds the best feature based on the value of a loss function (scikit-learn supports three popular loss functions: gini, entropy, and log_loss).
- **Make Predictions:** Each individual decision tree in the forest makes its own prediction for a given transaction.
- **Majority Voting:** The final prediction for the transaction is determined by a majority vote among all the individual trees. For example, if 7 out of 10 trees predict a transaction is fraudulent, the final prediction for the Random Forest will be "fraudulent".

3.1.2 Workflow

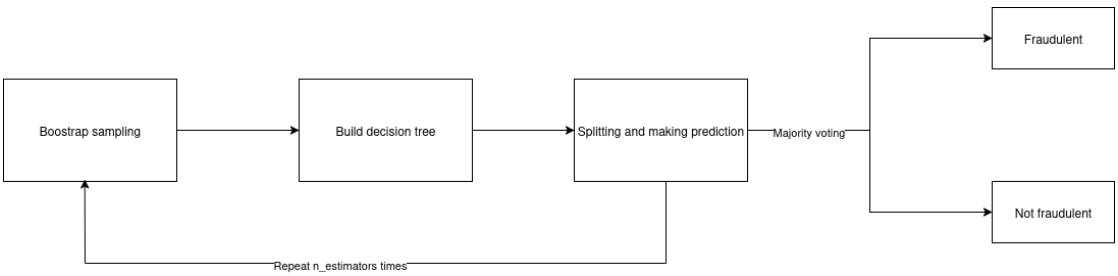

Figure 14: Random Forest Classifier workflow

3.1.3 Application

The Random Forest model is constructed using the RandomForestClassifier implementation from the Scikit-learn library, with consideration given to several key parameters below:

- **n_estimators:** This parameter defines the number of base estimators (trees) in the ensemble. A higher number of trees generally improves model stability and performance at the cost of increased computational time. Conversely, an insufficient number of estimators

can lead to poor generalization. Through experimentation, a value of 300 was selected, as it provided a suitable balance between model performance and training efficiency.

- **max_depth:** This parameter controls the maximum depth of each individual decision tree and is critical for managing model complexity. Deeper trees can capture more intricate patterns but are highly susceptible to overfitting, whereas shallower trees may result in underfitting. A depth of 12 was determined to be optimal, providing sufficient model capacity while mitigating the risk of overfitting and maintaining computational efficiency.
- **min_samples_split:** This parameter specifies the minimum number of samples an internal node must hold to be considered for a split. It functions as a regularization mechanism to prevent the model from learning from noise by creating splits on small, potentially unrepresentative sample groups. For this model, the value was set to 20.
- **class_weight:** This parameter is used to address class imbalance by adjusting the weight of samples during training. The 'balanced' option automatically adjusts weights to be inversely proportional to class frequencies, thereby assigning greater importance to the minority class. Given the slight class imbalance that persisted after data preprocessing, this setting was employed to ensure the model would not be biased towards the majority class.

3.2 Extreme Gradient Boosting (XGBoost)

3.2.1 Introduction

XGBoost, short for Extreme Gradient Boosting, is an optimized and highly efficient implementation of the gradient boosting framework. It is an ensemble learning method where multiple weak models, typically decision trees, are combined to create a strong predictive model. Unlike Random Forest, which builds trees independently (in parallel), XGBoost constructs them sequentially. Each new tree is trained to correct the errors made by the combination of all previous trees, a technique known as boosting. A key advantage of XGBoost is its incorporation of regularization terms within its objective function, which effectively reduces overfitting and enhances the model's ability to generalize to unseen data. Furthermore, XGBoost includes a sophisticated, built-in mechanism for intelligently handling missing values in the dataset.

For the task of fraudulent transaction detection, the XGBoost performs the following steps:

- **Initialize a Base Predictor:** The process begins with an initial, simple prediction (e.g., the prior probability of a transaction being fraudulent). The errors (residuals) between these initial predictions and the actual labels are calculated using a chosen loss function, such as logarithmic loss (log loss).
- **Train a Sequential Tree:** A decision tree is then trained not on the original labels, but on the residuals (errors) from the current model's predictions. The goal of this tree is to best predict these errors, thereby correcting the mistakes of the existing ensemble.
- **Add the Tree to the Ensemble:** The predictions from the new tree are scaled by a learning rate (a small value between 0 and 1) and then added to the previous ensemble's predictions. The learning rate controls the contribution of each tree, preventing overfitting by ensuring the model learns slowly and steadily.
- **Repeat:** Steps 2 and 3 are repeated iteratively. Each new tree focuses on the residuals that the current ensemble has not yet corrected.

- **Form the Final Model:** The final model is the sum of the predictions from all sequential trees. For classification, this aggregated score is transformed via a function like the sigmoid function to output a probability that a transaction is fraudulent.

3.2.2 Workflow

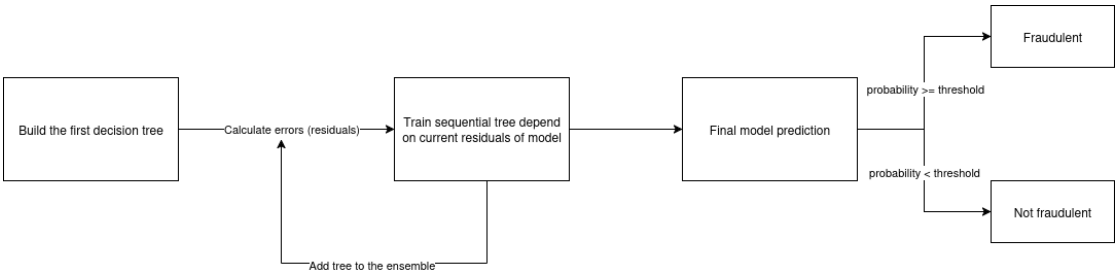

Figure 15: XGBoost Classifier workflow

3.2.3 Application

We employed the `XGBClassifier` implementation from the `xgboost` library to construct a model capable of distinguishing between normal and abnormal data points. The library offers a wide range of hyperparameters, which provide considerable flexibility in model configuration. In this study, we considered the following key parameters:

- **n_estimators:** Defines the number of trees in the ensemble. While a large number of trees may increase computational cost, an insufficient number may result in underfitting. We set this parameter to 300.
- **objective:** Determines the learning task and the corresponding objective function. Given the binary nature of our classification problem, we selected `binary:logistic`, which outputs probabilities after a logistic transformation and is well-suited for evaluation.
- **tree_method:** Specifies the algorithm for tree construction. Since the dataset is relatively large, we employed the efficient histogram-based method (`hist`).
- **max_depth:** Controls the maximum depth of each tree. Larger values increase model complexity and the risk of overfitting.
- **learning_rate:** Refers to the step size shrinkage used in updates to mitigate overfitting. Following each boosting step, the learning rate scales down feature weights, making the boosting process more conservative. We adopted a relatively small value of 0.1.
- **eval_metric:** Defines the evaluation criterion during training. Since the focus of this study is binary classification, we employed the area under the precision-recall curve (PR-AUC), which is well-suited for imbalanced data.
- **scale_pos_weight:** Balances the weights of positive and negative classes, which is particularly beneficial in the presence of class imbalance. The value was computed as follows:

```
scale_pos_weight = y_train[y_train == 1].count() / y_train[y_train == 0].count()
```

- **subsample:** Specifies the subsampling ratio of the training data to reduce overfitting and improve generalization. We set this parameter to 0.8.

3.3 Isolation Forest

3.3.1 Introduction

Isolation Forest is an efficient, unsupervised algorithm for anomaly detection. Unlike supervised learning algorithms such as Random Forest and XGBoost, Isolation Forest does not require labeled data. It identifies outliers in large datasets by isolating them through binary partitioning, a process that requires minimal computational overhead. This ability to find anomalies quickly is critical in applications where detecting unusual patterns is key to safeguarding against risks.

Isolation Forest is based on two fundamental concepts:

1. **Isolation Tree:** An isolation tree is composed of nodes. A node T is either an external node with no children or an internal node with a test and exactly two child nodes (T_l, T_r). A test consists of a randomly selected attribute q and a split value p, which divides the data points such that points with $q < p$ go to T_l and the rest go to T_r.

   Given a data sample $X = \{x_1, ..., x_n\}$, the algorithm recursively divides X by randomly selecting an attribute q and a split value p. This process continues until one of the following conditions is met: the tree reaches a predefined height limit, a node has only one data point ($|X| = 1$), or all data points in a node are identical.

2. **Path Length ($h(x)$):** The path length of a data point x, denoted as $h(x)$, is the number of edges it traverses in an isolation tree from the root node to an external node.

An anomaly score is necessary for any detection method. To derive a score from h(x), the algorithm normalizes it based on the sample size. Since an Isolation Tree has a structure similar to a Binary Search Tree, we can use the average path length of an unsuccessful search, c(n), for normalization.

The formula for the anomaly score is:

$$s(x, n) = 2^{-\frac{E(h(x))}{c(n)}}$$

where $E(h(x))$ is the average path length of x across a collection of isolation trees, and c(n) is the average path length for a given sample size n. Note that c(n) grows logarithmically with n (i.e., like ln(n)).

The interpretation of the score s is as follows:

- If the score is very close to 1 (i.e., $E(h(x)) \to 0$), the point is easily isolated and is therefore highly likely to be an *anomaly*.
- If the score is much smaller than 0.5 (i.e., $E(h(x)) \to n - 1$), the point is difficult to isolate and can be safely considered a *normal instance*.
- If the scores for all instances in a dataset are close to 0.5 (i.e., $E(h(x)) \to c(n)$), it suggests there are no distinct anomalies present.

To classify data points as normal or anomalous, the algorithm sets a threshold for the anomaly score. Points above the threshold are classified as anomalies while points below are considered normal.

3.3.2 Workflow

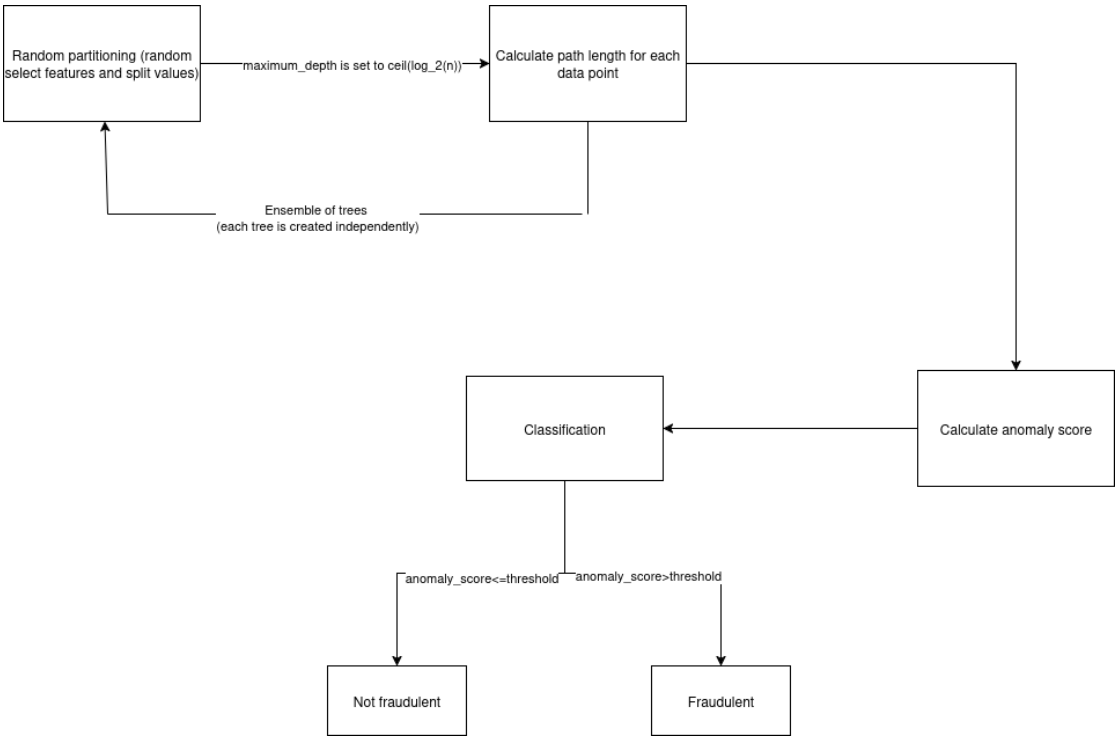

Figure 16: Isolation Forest workflow

3.3.3 Application

We use IsolationForest API from Scikit-learn to train model, some parameters we consider:

- **n_estimators:** The number of estimators in the ensemble. A very high value might lead to overfitting, while a very low value might lead to underfitting.
- **max_samples:** The number of samples to draw from X to train each base estimator. In the Isolation forest paper [4], a larger sample make algorithm hard to learn.
- **contaminations:** The amount of contamination of dataset, i.e the proportion of outliers in the dataset (it should be in the range (0,0.5]). This parameter is used to flag $contaminations * n\_samples$ data points as outliers after ranking anomaly score. A higher value caused the model to make more false positives, while a lower value caused the model to incorrectly detect actual fraudulent data points.

In scikit-learn's implementation, the anomaly score is defined in the opposite way compared to the original Isolation Forest paper: lower (more negative) scores indicate abnormal points.

Some challenges we meet when using Isolation Forest in this problem below:

1. **Train an unsupervised learning algorithm on a fully labeled dataset:** It's generally a disadvantage to train Isolation forest on a fully labeled dataset compared with supervised learning like Random forest or XGBoost, as we don't use all available information.

   Solution: Use mean of y_train (proportion of fraudulent data points in train dataset) for the contamination parameter in Isolation forest API in Scikit-learn.

2. **Curse of Dimensionality:** The performance of the Isolation Forest algorithm can be degraded by the curse of dimensionality. In high-dimensional feature spaces, data becomes increasingly sparse, which diminishes the efficacy of random splits in isolating anomalies. This phenomenon hinders the model's ability to distinguish outliers from normal instances, potentially leading to reduced detection accuracy and a higher false positive rate. The issue is particularly pronounced when the dataset contains a large proportion of non-informative or noisy features.

   To mitigate this effect, the max_features hyperparameter was configured. By setting this parameter to a small integer value, such as 3, each isolation tree is constructed using only a small, random subset of the total available features. This approach increases the probability that individual trees are built using relevant features, thereby reducing the impact of noise and preserving the model's isolation capability in a high-dimensional context.

4 Evaluation

In the evaluation phase, we measured model performance using not only accuracy but also Precision, Recall, and F1-score, which are crucial for fraud detection due to data imbalance. Precision shows how many predicted fraud cases are correct, while Recall measures how many frauds are actually detected. F1-score balances both. We also used confusion matrices, ROC-AUC curves, and comparison charts to clearly show each model's strengths and weaknesses.

4.1 Metrics

To evaluate the performance of fraud detection models, several metrics were applied:

- **Accuracy**: Measures the overall proportion of correct predictions. However, it can be misleading in imbalanced datasets.
- **Precision**: Indicates how many of the transactions predicted as fraud were actually fraud, helping to reduce false alarms.
- **Recall**: Measures how many actual fraud cases were correctly detected, which is critical in fraud detection.
- **F1-Score**: The harmonic mean of Precision and Recall, providing a balanced measure when both false positives and false negatives matter.
- **ROC-AUC**: Evaluates the model's ability to distinguish between fraud and non-fraud across thresholds. A higher AUC indicates better discrimination.

4.2 Training process

4.2.1 Machine Learning Models preparation

In this stage, three machine learning models were prepared for the fraud detection task: **Random Forest**, **XGBoost**, and **Isolation Forest**. Each model was configured with hyperparameters tailored to balance performance, efficiency, and robustness.

- **Random Forest**:
  - `n_estimators=200` – number of decision trees in the forest, which improves stability.
  - `max_depth=8` – limits tree depth to prevent overfitting.
  - `min_samples_split=20` – minimum samples required to split an internal node, adding regularization.
  - `class_weight="balanced"` – automatically adjusts weights inversely proportional to class frequencies to handle imbalance.
  - `n_jobs=-1` – uses all available CPU cores for parallel training.
- **XGBoost**:
  - `n_estimators=200` – number of boosting rounds.
  - `max_depth=8` – controls complexity of each tree.

  - `learning_rate=0.1` – shrinks contribution of each tree, improving generalization.
  - `scale_pos_weight = scale_pos_weight` – ratio of negative to positive classes, helping balance fraud vs. non-fraud cases.
  - `subsample=0.8` – fraction of samples used per tree, reducing overfitting.
  - `tree_method='gpu_hist'` and `device='cuda'` – enables GPU acceleration.
  - `eval_metric=['aucpr']` – evaluates performance using Precision-Recall AUC, which is more informative for imbalanced datasets.
  - `random_state=42` – ensures reproducibility.

- **Isolation Forest**:
  - `n_estimators=200` – number of isolation trees used for anomaly detection.
  - `max_samples=128` – number of samples used per tree, making training efficient.
  - `max_features=3` – number of features considered when splitting nodes.
  - `contamination=y_train.mean()` – expected proportion of fraud cases in the data.
  - `bootstrap=False` – disables bootstrap sampling, ensuring full sample use.
  - `random_state=42` – fixes randomness for reproducibility.
  - `n_jobs=-1` – parallel execution for faster training.

4.2.2 Traning Models

The training and evaluation pipeline for all selected models are what we going to do in this stage . Two dictionaries are used throughout the process:

- results: stores the accuracy scores of each model.
- predictions: saves the predicted labels for later evaluation and visualization.

This approach ensures that all models can be compared under the same framework, making the evaluation both consistent and systematic. In this training section, we need to do a special handling for the Isolation Forest models. The Isolation Forest is fundamentally different from the supervised models because it is an unsupervised anomaly detection method. This means:

- It does not require labeled outcomes (y_train) during training.
- Instead, it only learns patterns from the training features (X_train) and isolates points that appear unusual or rare.

In addition, the default predictions of the Isolation Forest's API is:

- -1 $\rightarrow$ anomaly (treated as Fraud)
- +1 $\rightarrow$ normal (treated as Not fraud)

In contrast, for the supervised models (Random Forest and XGBoost), both the feature matrix (X_train) and the corresponding labels (y_train) were provided during training. Once fitted, the models generated predictions on the test set, which were then compared against the ground truth labels to compute accuracy.

Finally, to ensure that no other names than the three models have been discussed, the entire training process was wrapped in an error-handling mechanism. If any exception occurred during training, the model's accuracy was set to zero and its predictions were replaced with default values. This approach guaranteed that all models could be compared fairly without the process being disrupted by runtime errors.

4.2.3 Hyperparameter Tuning

In this project, no hyperparameter tuning approach (such as Grid Search or Randomized Search) was applied. Instead, the models were trained using carefully chosen default or manually selected parameters based on prior knowledge and common practices for fraud detection.

The decision not to perform hyperparameter tuning was made to keep the training process computationally efficient and focused on evaluating baseline model performance. Future work could include applying Randomized Search or Bayesian Optimization to further optimize parameters and potentially improve predictive performance.

For consistency and fair comparison across models, we fixed some hyperparameters the number of estimators (`n_estimators`) and the random seed (`random_state`) to the same values:

- The number of estimators (`n_estimators`):
  - At **100 estimators**, the models achieved reasonable accuracy with faster training times. However, variance was slightly higher.
  - At **200 estimators**, the models reached a stable performance and provided a balance between accuracy and computational efficiency.
  - At **300 estimators**, the accuracy improved marginally compared to 200, but training time increased significantly, leading to diminishing returns.
- The random seed (`random_state = 42`). This ensured that differences in results were due to the algorithms themselves rather than randomness or unequal settings.

For each models, we usually focus on a few key parameters that really affect performance:

- **Random Forest**:
  - **max_depth = 8:** This parameter controls model complexity, with deeper trees capturing more detail but risking overfitting. Through experimentation and common practice in fraud detection tasks, we selected the value **8** as a balanced value. This depth is sufficiently large to model complex relationships between transaction features and fraud patterns, while still limiting tree complexity to reduce the risk of overfitting. Moreover, a moderate tree depth also reduces training time and improves generalization on unseen data.
- **XGBoost**:
  - **max_depth = 8:** For the tree depth parameter (**max_depth**), we use the same value as in the Random Forest model in order to provide a fair comparison between the two supervised approaches.
  - **learning_rate = 0.1:** Regarding the learning rate, setting this value too low makes the model more robust but also requires many more boosting rounds, which increases training time. Therefore, we choose a value of 0.1 as a balance between robustness and computational efficiency.
- **Isolation Forest**: In the Isolation Forest model, the parameters `max_samples` and `max_features` play an important role in determining how each tree is built and how well the anomalies are separated.

- **max_samples = 128:** This parameter defines the number of samples drawn to train each tree. By limiting it to 128, we ensure that each tree is trained on a small subset of the dataset, which increases randomness and helps the ensemble isolate anomalies more effectively. Using a smaller sample size also reduces computational cost and prevents the model from being dominated by majority (non-fraud) samples.
- **max_features = 3:** This parameter defines how many features are randomly selected for each split. Choosing only 3 features forces the model to explore different random subspaces of the data, which increases diversity among the trees. This randomness improves the ability to detect rare fraudulent patterns, especially in high-dimensional transaction data.
- **contamination = y_train.mean():** This hyperparameter is quite special because we don't have an exact probability of Fraud. Since the dataset is so large, we have to take a smaller sample for training. This causes some variability in the original dataset. Therefore, we use the **y_train.mean()** to get an accurate value to get the best results for comparison.

4.3 Training results

After training the models on the prepared dataset, we evaluated their performance on the test set. The evaluation focused on common metrics for fraud detection, including *Accuracy*, *Precision*, *Recall*, and *F1-score*. Additionally, confusion matrix, and the ROC-AUC score were considered to provide insight into the overall discriminative ability of each model.

- **Random Forest model:**

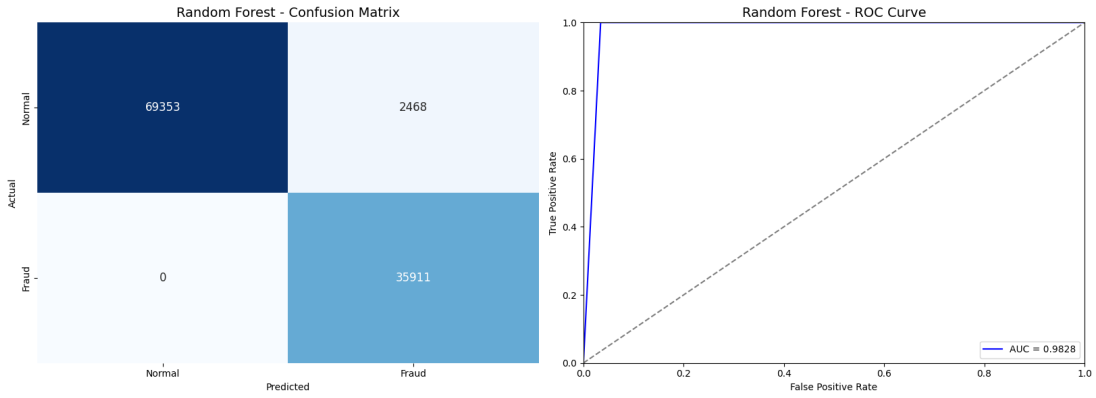

Figure 17: Random Forest training results: Confusion matrix, and ROC curve.

Result Analyzing:

- **Normal transactions:** Precision is 1.00 and recall is 0.97, meaning that almost all normal cases are classified correctly, with a very small number of false positives.
- **Fraudulent transactions:** Recall reaches 1.00, indicating that the model successfully detects all fraudulent cases. Precision is 0.94, suggesting that there are some false alarms, but these are relatively limited.

– **Overall performance:** The F1-score for fraud detection is 0.97, reflecting a good balance between precision and recall. The ROC curve confirms the strong discriminative ability, with an AUC of 0.9828.

Parameter	Value
n_estimators	200
random_state	42
class_weight	balanced
max_depth	8
min_samples_split	20
n_jobs	-1

Table 14: Parameters of Randơm Forest

Metric	Value
Accuracy	0.98
Precision (Fraud)	0.94
Recall (Fraud)	1.00
F1-score (Fraud)	0.97
AUC	0.9828

Table 15: Evaluation Metrics of Random Forest

- **XGBoost model:**

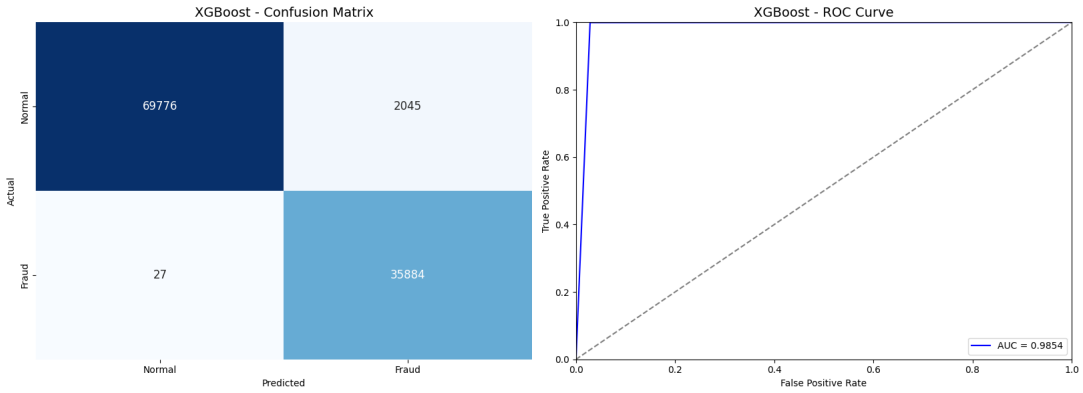

Figure 18: XGBoost training results: Confusion matrix, and ROC curve.

Result Analyzing:

– **Normal transactions:** Precision is 1.00 and recall is 0.97, showing that almost all normal cases are classified correctly with only a small portion misclassified as fraud.

– **Fraudulent transactions:** Recall reaches 1.00, which means the model successfully detects nearly all fraudulent cases. Precision is 0.95, indicating that there are some false positives, but they remain limited.

– **Overall performance:** The F1-score for fraud detection is 0.97, demonstrating a strong balance between precision and recall. The ROC curve further confirms the discriminative power of the model with an AUC of 0.9854.

Parameter	Value
n_estimators	200
objective	binary:logistic
tree_method	gpu_hist
max_depth	8
learning_rate	0.1
scale_pos_weight	ratio $\frac{n_{neg}}{n_{pos}}$
subsample	0.8
random_state	42
device	cuda
n_jobs	-1

Table 16: Parameters of XGBoost

Metric	Value
Accuracy	0.97
Precision (Fraud)	0.95
Recall (Fraud)	1.00
F1-score (Fraud)	0.97
AUC	0.9854

Table 17: Evaluation Metrics of XGBoost

- **Isolation Forest model:**

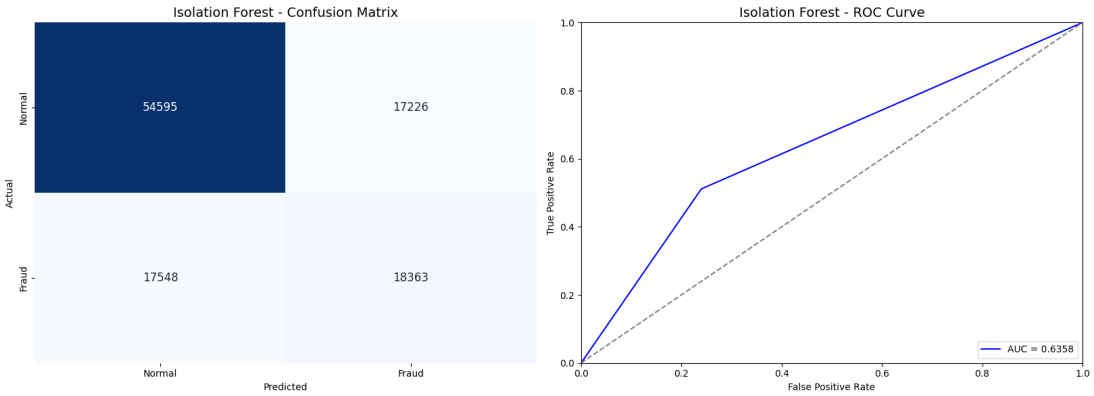

Figure 19: Isolation Forest training results: Confusion matrix, and ROC curve.

Result Analyzing:

- **Normal transactions:** Precision and recall are both 0.76, showing that the model correctly identifies most normal cases but misclassifies a significant portion as fraud.
- **Fraudulent transactions:** Precision is 0.52 and recall is 0.51, indicating that the model struggles to accurately detect fraudulent cases and generates many false positives and false negatives.
- **Overall performance:** The F1-score for fraud detection is 0.51, reflecting weak balance between precision and recall. The ROC curve also demonstrates limited discriminative ability, with an AUC of 0.6358.

Parameter	Value
n_estimators	200
max_samples	128
max_features	3
contamination	$y\_train.mean()$
random_state	42
n_jobs	-1
bootstrap	False

Table 18: Parameters of Isolation Forest

Metric	Value
Accuracy	0.68
Precision (Fraud)	0.52
Recall (Fraud)	0.51
F1-score (Fraud)	0.51
AUC	0.6358

Table 19: Evaluation Metrics of Isolation Forest

4.4 Models Comparison

When comparing the performance of the three models, we can observe clear differences in their ability to detect fraudulent transactions:

- **Random Forest:** Achieves strong performance with an accuracy of 0.98. It shows high recall (1.00) for fraud detection, meaning that nearly all fraudulent cases are identified. Precision for fraud is 0.94, indicating some false alarms but still a reliable performance. The AUC score of 0.9828 further confirms its robust discriminative ability.
- **XGBoost:** Delivers comparable results to Random Forest, also reaching an accuracy of 0.98. Fraud recall is 1.00 and precision is slightly higher at 0.95, which reduces false positives compared to Random Forest. Its AUC score of 0.9854 indicates slightly stronger overall classification capability.
- **Isolation Forest:** As an unsupervised method, its performance is significantly lower. The model reaches only 0.68 accuracy, with a fraud F1-score of 0.51. Both fraud precision (0.52) and recall (0.51) suggest weak detection ability. The ROC curve confirms this limitation, with an AUC of 0.6358.

Model	Accuracy	Precision	Recall	F1-Score
XGBoost	0.980767	0.981770	0.980767	0.980893
Random Forest	0.977091	0.978564	0.977091	0.977272
Isolation Forest	0.677218	0.676498	0.677218	0.676852

Table 20: Comparison of Models on Evaluation Metrics

Overall, the supervised models (Random Forest and XGBoost) clearly outperform the unsupervised Isolation Forest. Between the two supervised approaches, XGBoost achieves a slight advantage in precision and AUC, making it the most effective model for fraud detection in this experiment.

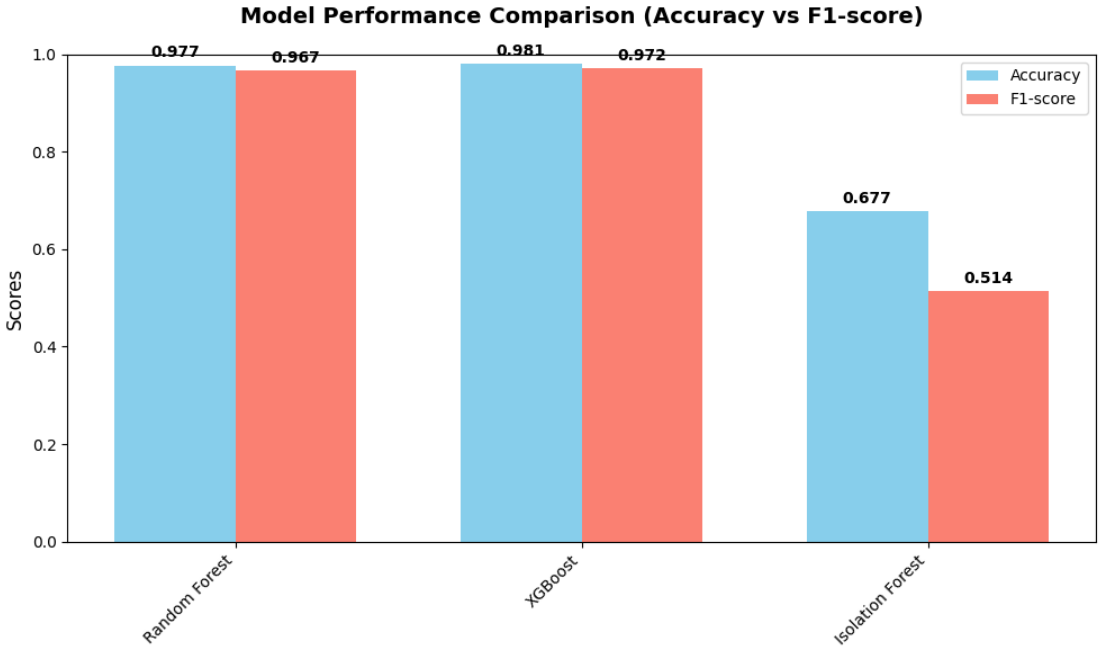

Figure 20: Models's metrics comparison

5 Summary

In summary, the evaluation highlights clear differences among the three models. Both Random Forest and XGBoost demonstrate excellent capability in detecting fraudulent transactions, with accuracy around 0.98 and AUC scores above 0.98. XGBoost shows slightly stronger discriminative power, achieving a higher precision and F1-score for fraudulent cases, which suggests it is more effective at balancing false positives and false negatives. Random Forest also performs very well, with perfect recall for fraud detection, ensuring that almost no fraudulent cases are missed. On the other hand, the unsupervised Isolation Forest model achieves significantly lower performance, with accuracy around 0.68 and an AUC of only 0.64, indicating limited ability to separate fraud from normal transactions in this dataset. These results suggest that supervised learning methods, when labeled data is available, provide much stronger and more reliable performance for fraud detection compared to unsupervised anomaly detection approaches.

References

[1] Scikit-learn, *RandomForestClassifier*

[2] GeeksforGeeks, *Random Forest Classifier using Scikit-learn*

[3] Scikit-learn, *IsolationForestClassifier*

[4] Isolation Forest `https://www.researchgate.net/publication/224384174_Isolation_Forest`