# Fraud Transaction Detection With Random Forest Classifier

By Ansh Kapil

# Project Overview:

- In recent years, there has been a significant rise in fraud attempts, highlighting the importance and challenge of fraud detection. Despite extensive efforts and human oversight, millions of dollars are lost to fraud annually. Fraud can occur through various methods such as stolen credit cards, deceptive accounting practices, phishing emails, and more. Detecting fraud is particularly challenging due to the small number of fraudulent cases in a large population.
- Data mining and machine learning play a crucial role in anticipating and swiftly identifying fraud, enabling quick action to minimize costs. With data mining tools, vast numbers of transactions can be analyzed to identify patterns and detect fraudulent transactions.

# Reading and exploring the dataset:

- We begin by loading the dataset using pandas' `read_csv()` function, which reads the dataset and converts it into a structured tabular format that we can easily analyze.

# Input Code:

```
In [1]: import pandas as pd
        import numpy as np
        import warnings
        warnings.filterwarnings("ignore", category=DeprecationWarning)
        import seaborn as sns
        import matplotlib.pyplot as plt
        data = pd.read_csv('PS_20174392719_1491204439457_log.csv')
        print('Data does not have any NULL value.')
        data.isnull().any()
```

# Output:

```
Data does not have any NULL value.

Out[1]: step                False
        type                False
        amount              False
        nameOrig            False
        oldbalanceOrg       False
        newbalanceOrig      False
        nameDest            False
        oldbalanceDest      False
        newbalanceDest      False
        isFraud             False
        isFlaggedFraud      False
        dtype: bool
```

# Displaying the Code with the help of head command:

```
In [2]: data.head()
```

Out[2]:

| | step | type | amount | nameOrig | oldbalanceOrg | newbalanceOrig | nameDest | oldbalanceDest | newbalanceDest | isFraud | isFlaggedFraud |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 1 | PAYMENT | 9839.64 | C1231006815 | 170136.0 | 160296.36 | M1979787155 | 0.0 | 0.0 | 0 | 0 |
| **1** | 1 | PAYMENT | 1864.28 | C1666544295 | 21249.0 | 19384.72 | M2044282225 | 0.0 | 0.0 | 0 | 0 |
| **2** | 1 | TRANSFER | 181.00 | C1305486145 | 181.0 | 0.00 | C553264065 | 0.0 | 0.0 | 1 | 0 |
| **3** | 1 | CASH_OUT | 181.00 | C840083671 | 181.0 | 0.00 | C38997010 | 21182.0 | 0.0 | 1 | 0 |
| **4** | 1 | PAYMENT | 11668.14 | C2048537720 | 41554.0 | 29885.86 | M1230701703 | 0.0 | 0.0 | 0 | 0 |

# The Problem of Dataset:

The provided data has the financial transaction data as well as the target variable **isFraud**, which is the actual fraud status of the transaction and **isFlaggedFraud** is the indicator which the simulation is used to flag the transaction using some **threshold value**.

# Input Code:

```
In [3]: data.rename(columns={'newbalanceOrig':'newbalanceOrg'},inplace=True)
        data.drop(labels=['nameOrig','nameDest'],axis=1,inplace=True)

In [4]: print('Minimum value of Amount, Old/New Balance of Origin/Destination:')
        data[[ 'amount','oldbalanceOrg', 'newbalanceOrg', 'oldbalanceDest', 'newbalanceDest']].min()
```

# Output:

```
       Minimum value of Amount, Old/New Balance of Origin/Destination:

Out[4]: amount            0.0
        oldbalanceOrg     0.0
        newbalanceOrg     0.0
        oldbalanceDest    0.0
        newbalanceDest    0.0
        dtype: float64
```

# Data Analysis:

- Since there is **no missing** and **garbage value**, there is no need for data cleaning, but we still need to perform **data analysis** as data contaion huge variation of the value in different columns. Normalization will also improve the overall accuracy of the machine learning model.

# Input Code:

```
In [5]: print('Maximum value of Amount, Old/New Balance of Origin/Destination:')
        data[[ 'amount','oldbalanceOrg', 'newbalanceOrg', 'oldbalanceDest', 'newbalanceDest']].max()
```

# Output:

```
Maximum value of Amount, Old/New Balance of Origin/Destination:

Out[5]: amount            9.244552e+07
        oldbalanceOrg     5.958504e+07
        newbalanceOrg     4.958504e+07
        oldbalanceDest    3.560159e+08
        newbalanceDest    3.561793e+08
        dtype: float64
```

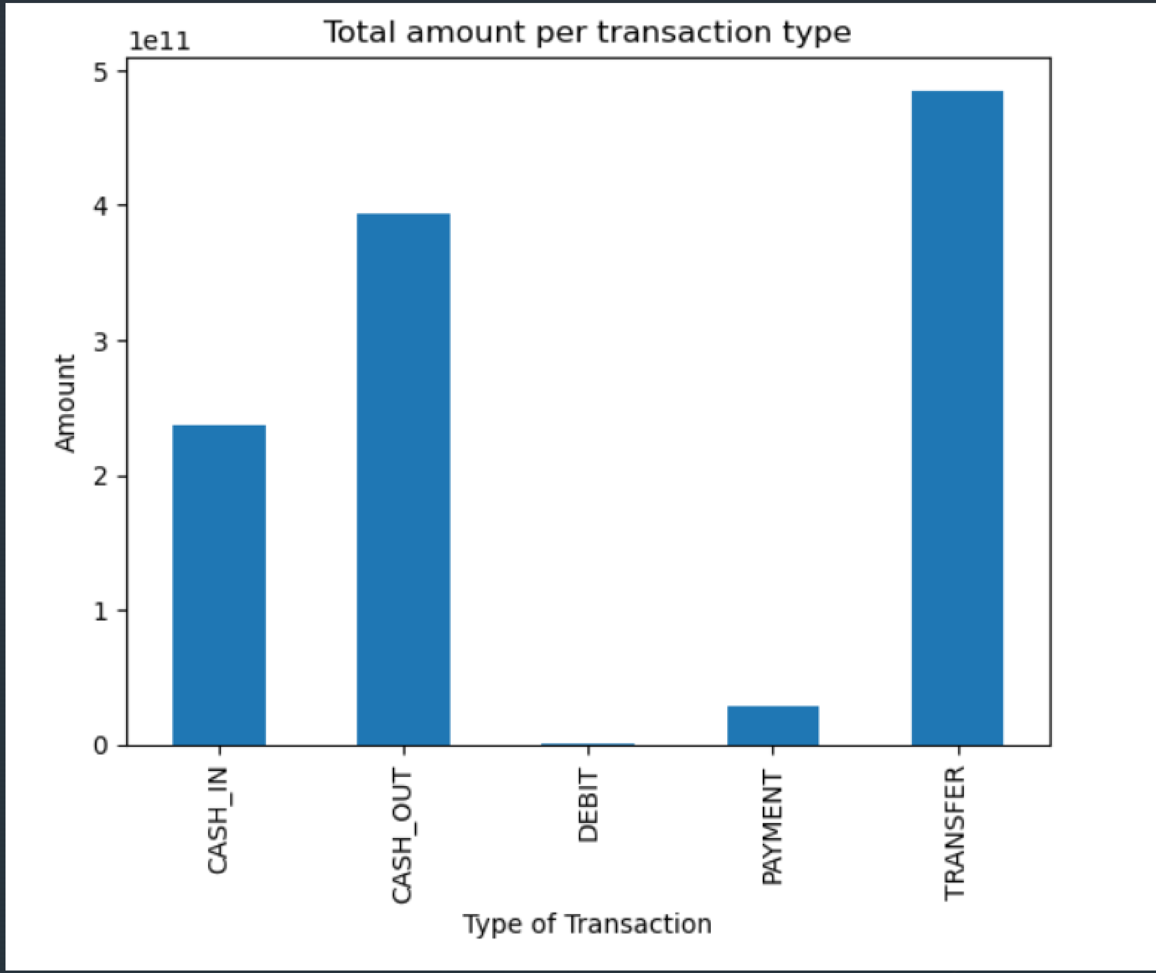
# Plotting the dataset

- The code groups the data by 'type', calculates the sum of 'amount' for each group, and then creates a bar chart to visualize the total amount per transaction type. The x-axis represents the different transaction types, and the y-axis represents the total amount. The bars in the chart will have heights corresponding to the total amount for each transaction type.
- This code allows you to quickly visualize and compare the total amounts associated with different categories or types within a dataset using a bar chart.

# Input Code:

```
In [6]: var = data.groupby('type').amount.sum()
        fig = plt.figure()
        ax1 = fig.add_subplot(1,1,1)
        var.plot(kind='bar')
        ax1.set_title("Total amount per transaction type")
        ax1.set_xlabel('Type of Transaction')
        ax1.set_ylabel('Amount');
```

# Output:



Total amount per transaction type
1e11
5
4
3
2
1
0
Amount
CASH_IN
CASH_OUT
DEBIT
PAYMENT
TRANSFER
Type of Transaction

# About the Bar Graph:

- The graph indicates that "TRANSFER" and "CASH_OUT" are the most common modes of transaction, and they are also the only methods through which fraud occurs. Therefore, our focus will be on these types of transactions.

# Input Code:

```
In [7]: data.loc[data.isFraud == 1].type.unique()

Out[7]: array(['TRANSFER', 'CASH_OUT'], dtype=object)
```

# Heat Map of the Data Attributes:

1. OldbalanceOrg and NewbalanceOrg are highly correlated.
2. OldbalanceDest and NewbalanceDest are highly correlated.
3. Amount is correlated with isFraud(Target Variable).
4. There is not much relation between the features, so we need to understand where the relationship between them depends on the type of transaction and amount. To do so, **we need to see the heat map of fraud and nonfraud transactions differently**.

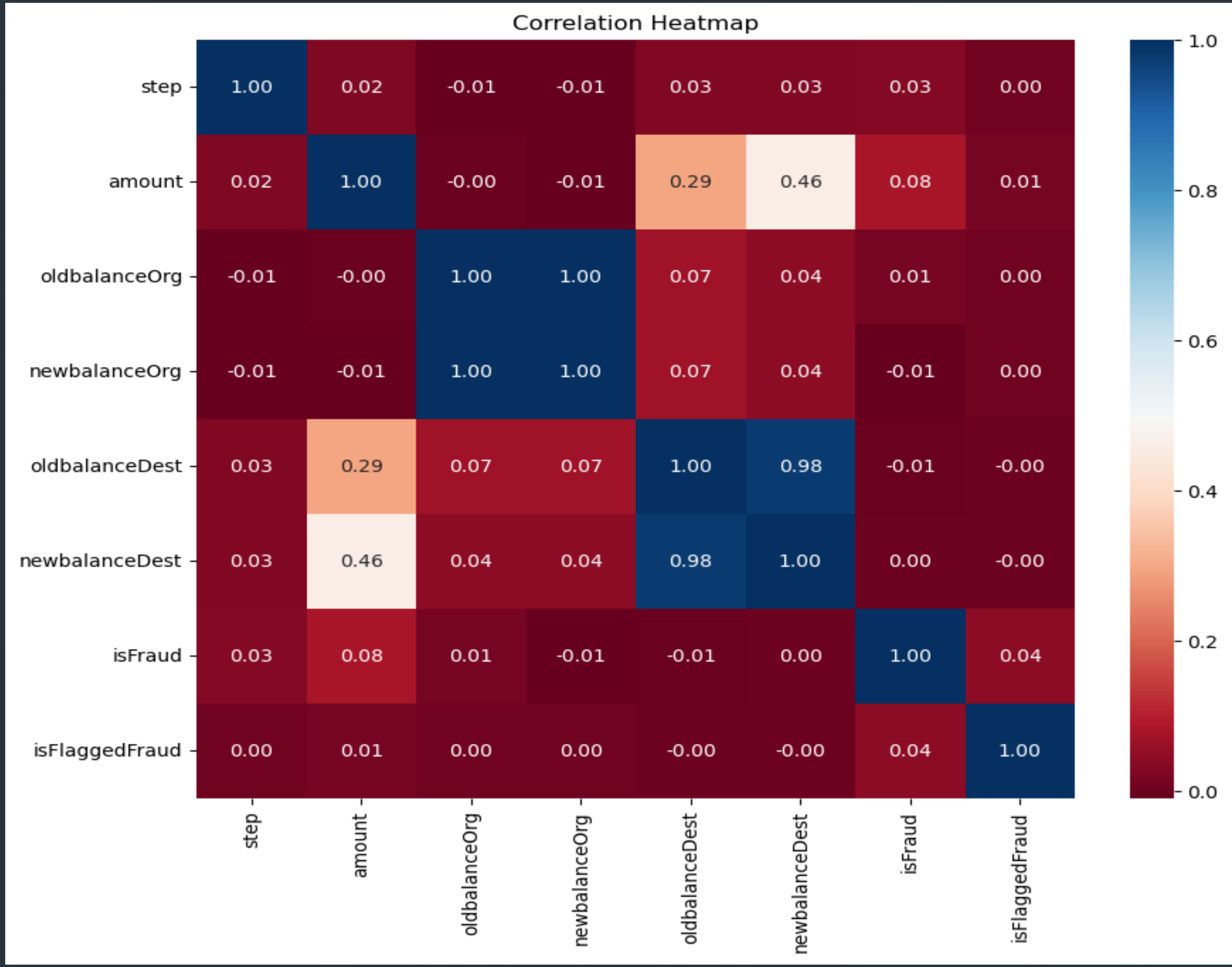
# Input Code:

```
In [8]:

# Assuming data is your DataFrame
numeric_data = data.select_dtypes(include=['number'])  # Select only numeric columns
correlation_matrix = numeric_data.corr()  # Compute correlation matrix

# Plot heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, cmap='RdBu', annot=True, fmt=".2f")
plt.title('Correlation Heatmap')
plt.show()
```

# Output:


Correlation Heatmap
step
amount
oldbalanceOrg
newbalanceOrg
oldbalanceDest
newbalanceDest
isFraud
isFlaggedFraud
1.00 0.02 -0.01 -0.01 0.03 0.03 0.03 0.00
0.02 1.00 -0.00 -0.01 0.29 0.46 0.08 0.01
-0.01 -0.00 1.00 1.00 0.07 0.04 0.01 0.00
-0.01 -0.01 1.00 1.00 0.07 0.04 -0.01 0.00
0.03 0.29 0.07 0.07 1.00 0.98 -0.01 -0.00
0.03 0.46 0.04 0.04 0.98 1.00 0.00 -0.00
0.03 0.08 0.01 -0.01 -0.01 0.00 1.00 0.04
0.00 0.01 0.00 0.00 -0.00 -0.00 0.04 1.00
1.0
0.8
0.6
0.4
0.2
0.0

# Input Code:

```
In [9]:  fraud = data.loc[data.isFraud == 1]
         nonfraud = data.loc[data.isFraud == 0]

In [10]: fraudcount = fraud.isFraud.count()
         nonfraudcount = nonfraud.isFraud.count()
```

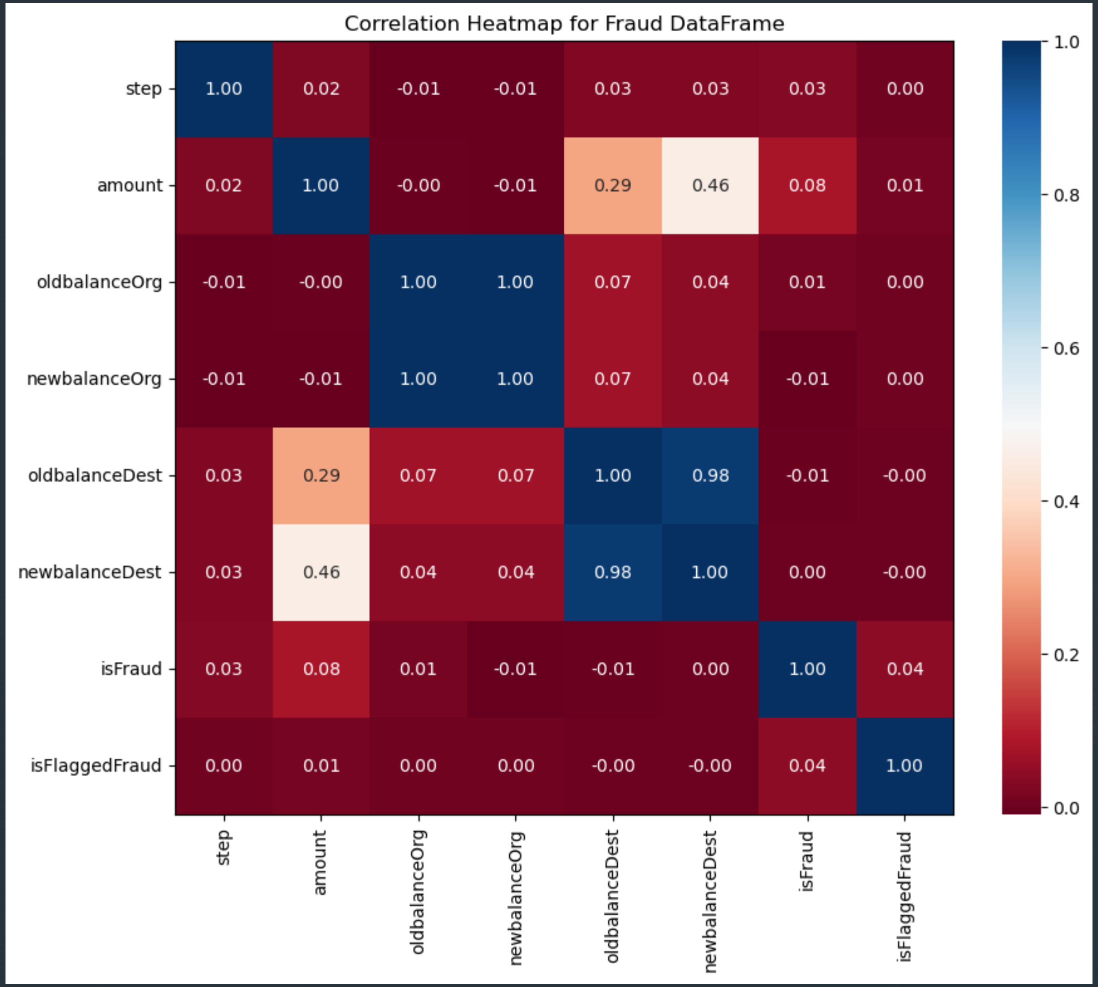
# Input Code:

```
In [11]: # Plot heatmap with manual annotation formatting
         plt.figure(figsize=(10, 8))
         heatmap = sns.heatmap(correlation_matrix, cmap='RdBu', annot=True)
         for _, spine in heatmap.spines.items():
             spine.set_visible(True)  # Ensure the heatmap borders are visible

         for text in heatmap.texts:
             try:
                 # Try converting the text to float and then format it
                 text.set_text("{:.2f}".format(float(text.get_text())))
             except ValueError:
                 # If conversion to float fails, leave the text unchanged
                 pass

         plt.title('Correlation Heatmap for Fraud DataFrame')
         plt.show()
```

# Output:



Correlation Heatmap for Fraud DataFrame
step
amount
oldbalanceOrg
newbalanceOrg
oldbalanceDest
newbalanceDest
isFraud
isFlaggedFraud
1.00 0.02 -0.01 -0.01 0.03 0.03 0.03 0.00
0.02 1.00 -0.00 -0.01 0.29 0.46 0.08 0.01
-0.01 -0.00 1.00 1.00 0.07 0.04 0.01 0.00
-0.01 -0.01 1.00 1.00 0.07 0.04 -0.01 0.00
0.03 0.29 0.07 0.07 1.00 0.98 -0.01 -0.00
0.03 0.46 0.04 0.04 0.98 1.00 0.00 -0.00
0.03 0.08 0.01 -0.01 -0.01 0.00 1.00 0.04
0.00 0.01 0.00 0.00 -0.00 -0.00 0.04 1.00
1.0
0.8
0.6
0.4
0.2
0.0

# Analyzing Fraud Flags:

There are 2 flags which stand out to me and it's interesting to look onto: isFraud and isFlaggedFraud column. From the hypothesis, **isFraud** is the indicator which indicates the **actual fraud transactions** whereas **isFlaggedFraud** is what the system prevents the transaction due to **some thresholds** being triggered. From the above heatmap we can see that there is some relation between other columns and isFlaggedFraud thus there must be relation between isFraud.

# Input Code:

```
In [12]: print('The total number of fraud transaction is {}.'.format(data.isFraud.sum()))
         print('The total number of fraud transaction which is marked as fraud {}.'.format(data.isFlaggedFraud.sum()))
         print('Ratio of fraud transaction vs non-fraud transaction is 1:{}.'.format(int(nonfraudcount//fraudcount)))
```

# Output:

```
The total number of fraud transaction is 8213.
The total number of fraud transaction which is marked as fraud 16.
Ratio of fraud transaction vs non-fraud transaction is 1:773.
```

# Input Code:

```
In [13]: print('Thus in every 773 transaction there is 1 fraud transaction happening.')
         print('Amount lost due to these fraud transaction is ${}.'.format(int(fraud.amount.sum())))
```

# Output:

Thus in every 773 transaction there is 1 fraud transaction happening.
Amount lost due to these fraud transaction is $12056415427.

# Addressing the Gap in Fraud Detection:

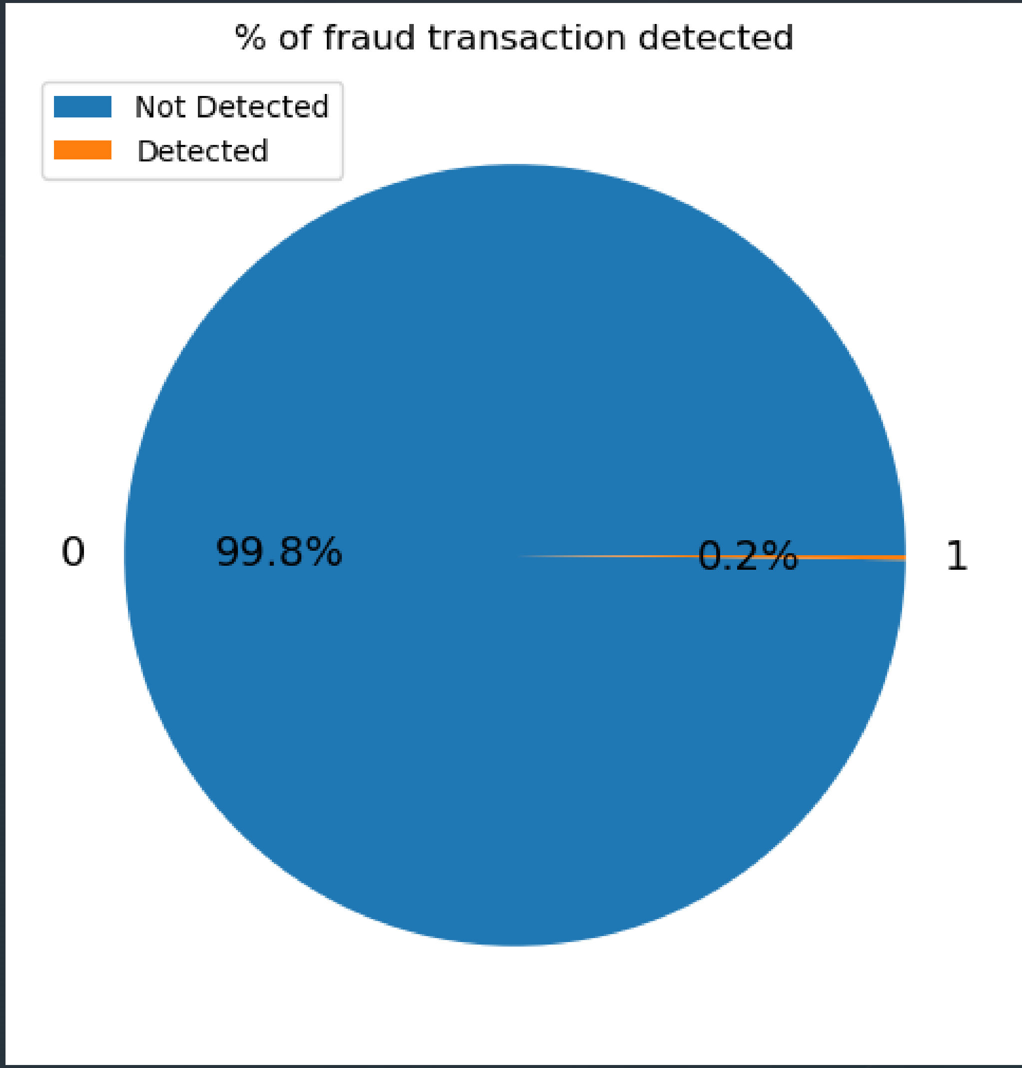
The plot indicates a clear requirement for a system that can swiftly and dependably identify fraudulent transactions. Currently, the system is allowing fraudulent transactions to go undetected, as they are not being flagged as such. Conducting thorough data exploration could be beneficial to uncover the relationships between various features and potentially identify patterns associated with fraud.

# Input Code:

```
In [14]: piedata = fraud.groupby(['isFlaggedFraud']).sum()
```

```
In [15]: f, axes = plt.subplots(1,1, figsize=(6,6))
         axes.set_title("% of fraud transaction detected")
         piedata.plot(kind='pie',y='isFraud',ax=axes, fontsize=14,shadow=False,autopct='%1.1f%%');
         axes.set_ylabel('');
         plt.legend(loc='upper left',labels=['Not Detected','Detected'])
         plt.show()
```

# Output:


% of fraud transaction detected
Not Detected
Detected
0
99.8%
0.2%
1

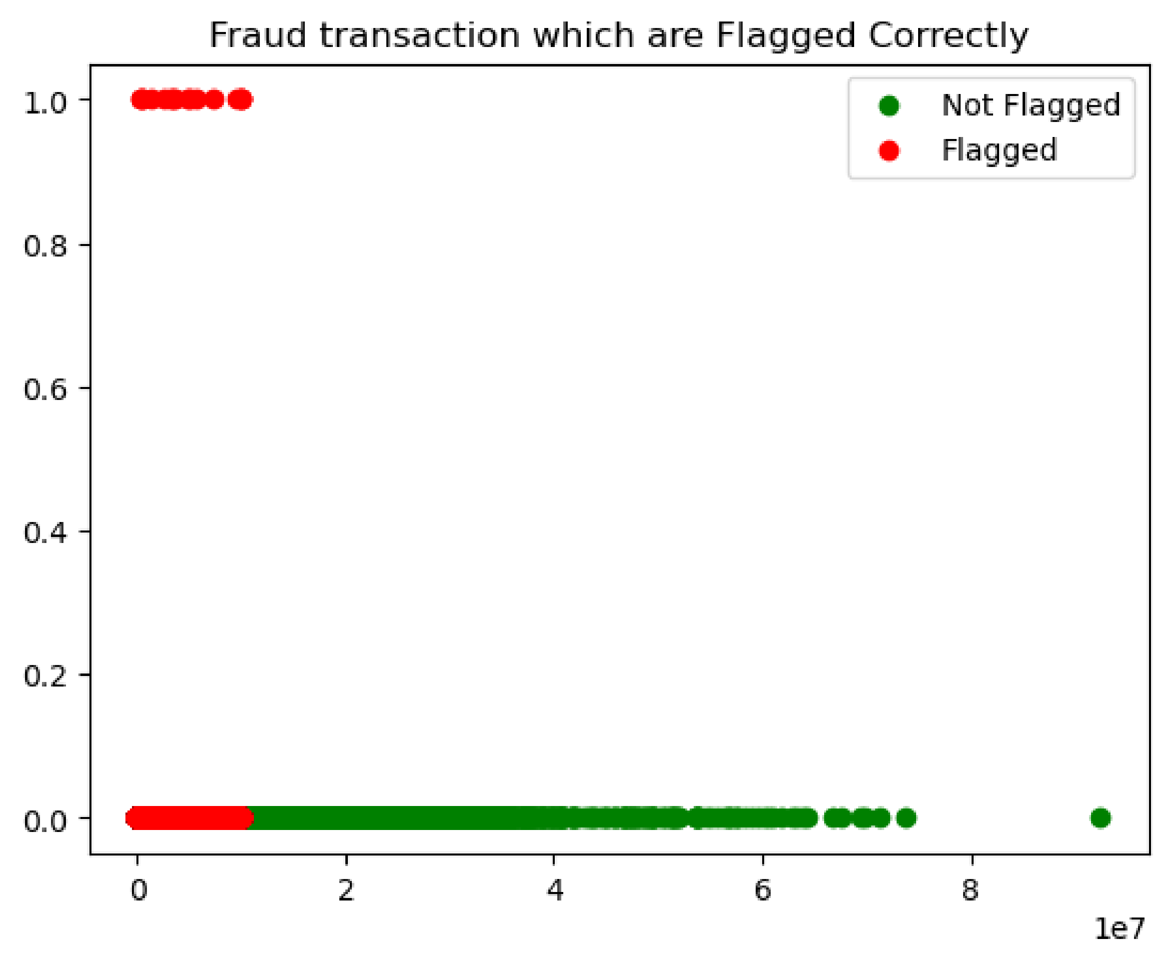
# Data Exploration:

Data exploration is a crucial process in data analysis where analysts examine and summarize data to understand its characteristics. It involves tasks such as calculating descriptive statistics, visualizing data, cleaning data, and identifying relationships between variables. The goal is to gain insights and identify patterns or anomalies in the data, which can guide further analysis and decision-making.

# Input Code:

```
In [16]: fig = plt.figure()
         axes = fig.add_subplot(1,1,1)
         axes.set_title("Fraud transaction which are Flagged Correctly")
         axes.scatter(nonfraud['amount'],nonfraud['isFlaggedFraud'],c='g')
         axes.scatter(fraud['amount'],fraud['isFlaggedFraud'],c='r')
         plt.legend(loc='upper right',labels=['Not Flagged','Flagged'])
         plt.show()
```

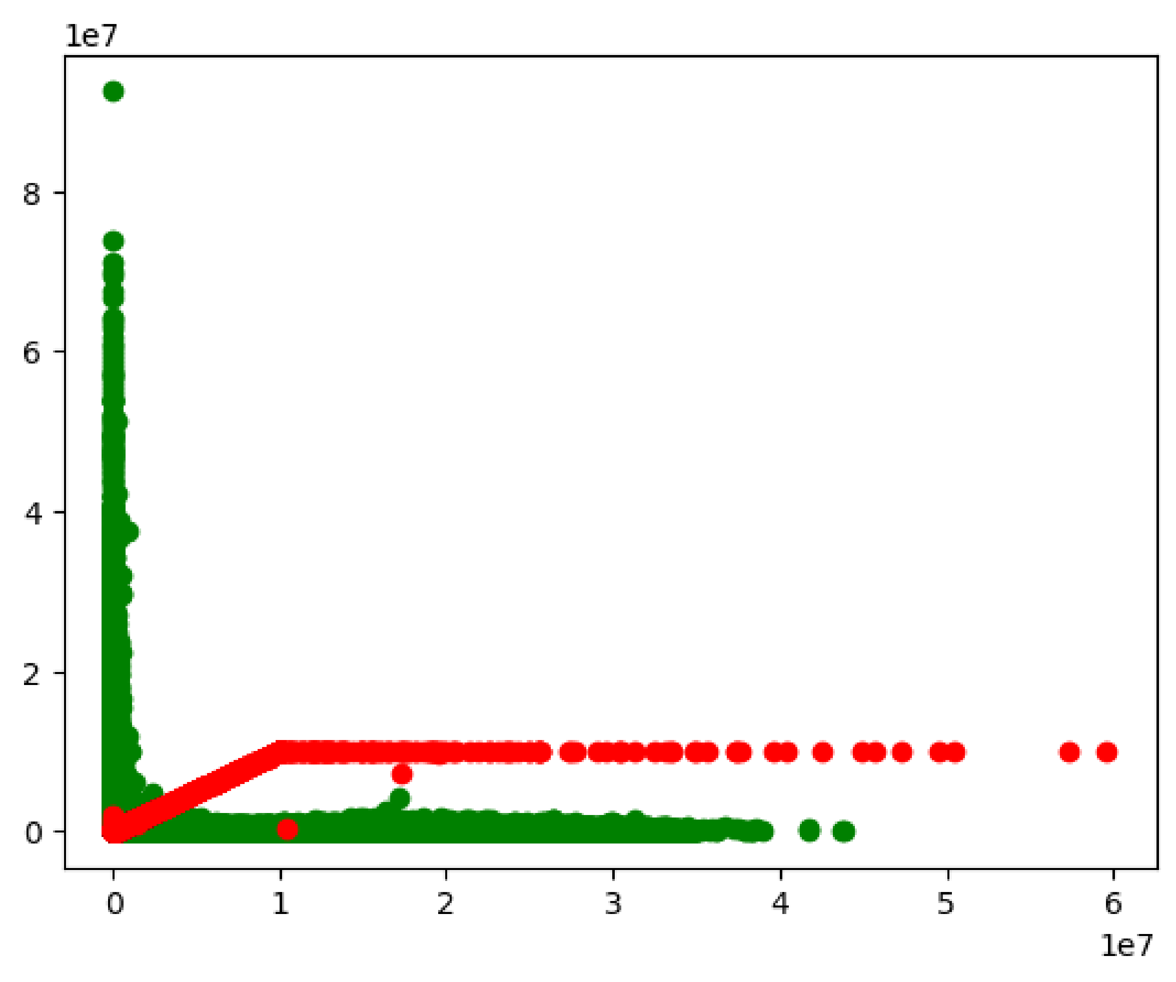
# Output:


Fraud transaction which are Flagged Correctly
Not Flagged
Flagged
1.0
0.8
0.6
0.4
0.2
0.0
0
2
4
6
8
1e7

# Input Code:

```
In [17]: fig = plt.figure()
         ax = fig.add_subplot(1,1,1)
         ax.scatter(nonfraud['oldbalanceOrg'],nonfraud['amount'],c='g')
         ax.scatter(fraud['oldbalanceOrg'],fraud['amount'],c='r')
         plt.show()
```

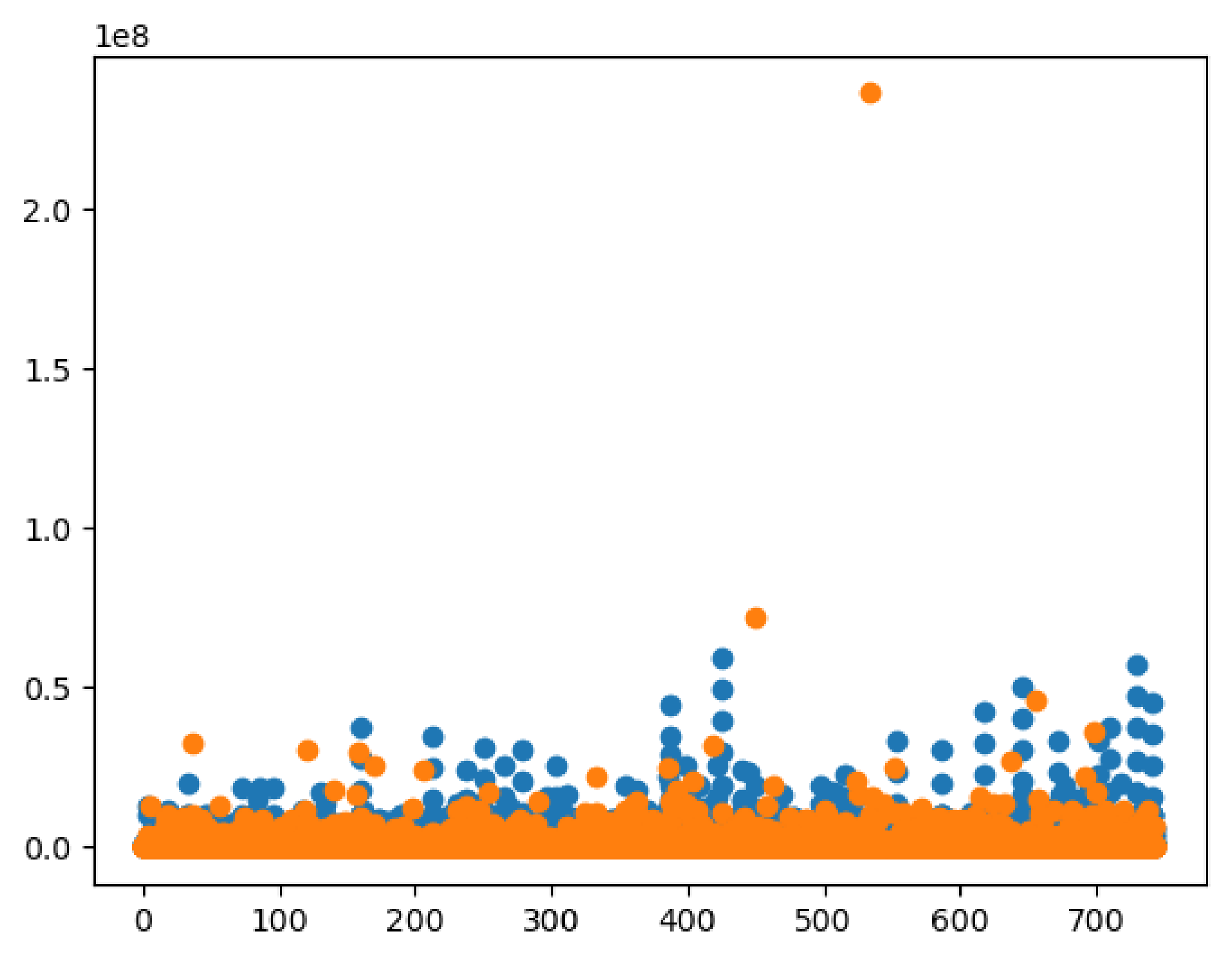
# Output:



# Input Code:

```
In [18]: fig = plt.figure()
         ax = fig.add_subplot(1,1,1)
         ax.scatter(fraud['step'],fraud['oldbalanceOrg'])
         ax.scatter(fraud['step'],fraud['oldbalanceDest'])
         plt.show()
```

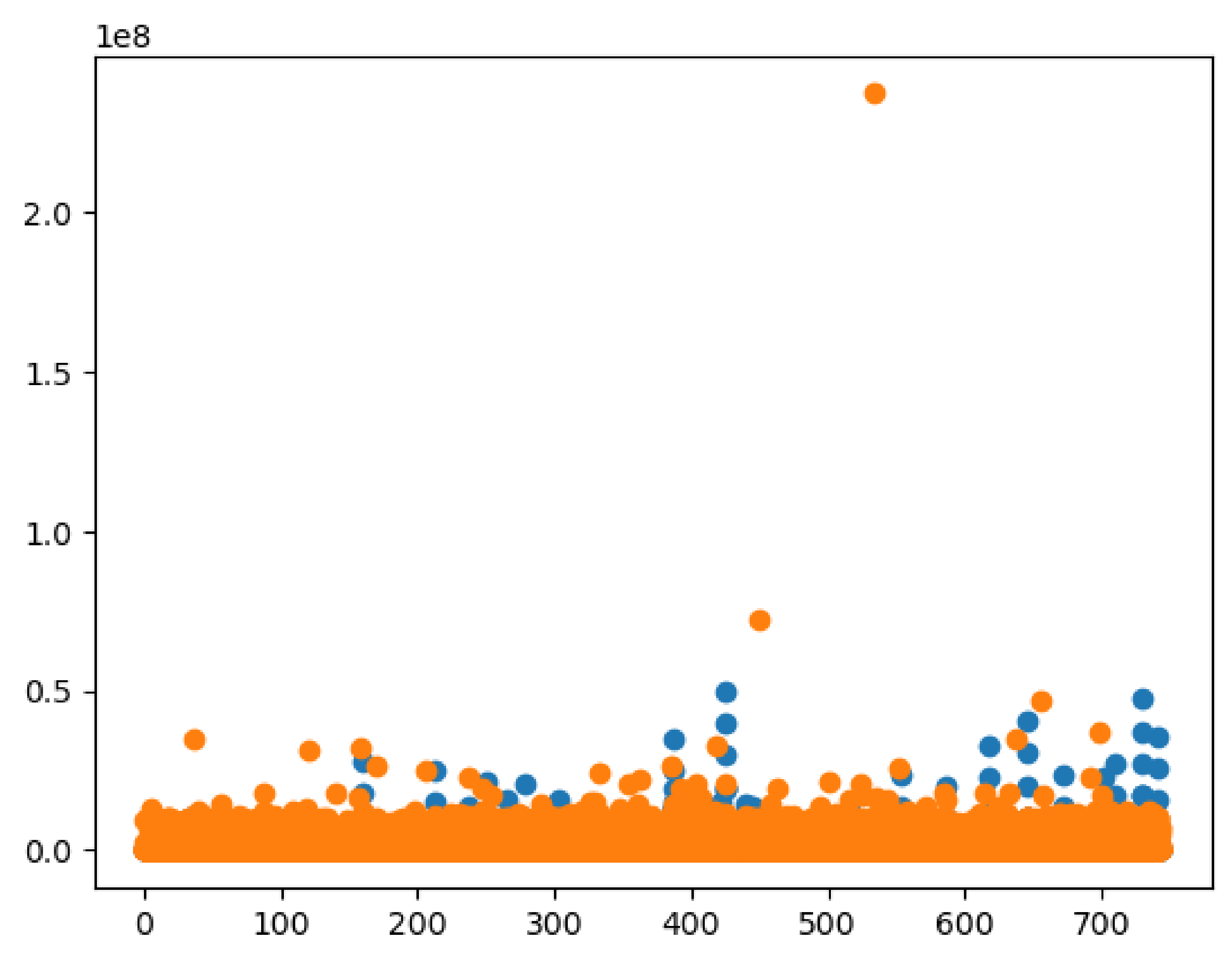
# Output:



# Input Code:

```
In [19]: fig = plt.figure()
         ax = fig.add_subplot(1,1,1)
         ax.scatter(fraud['step'],fraud['newbalanceOrg'])
         ax.scatter(fraud['step'],fraud['newbalanceDest'])
         plt.show()
```

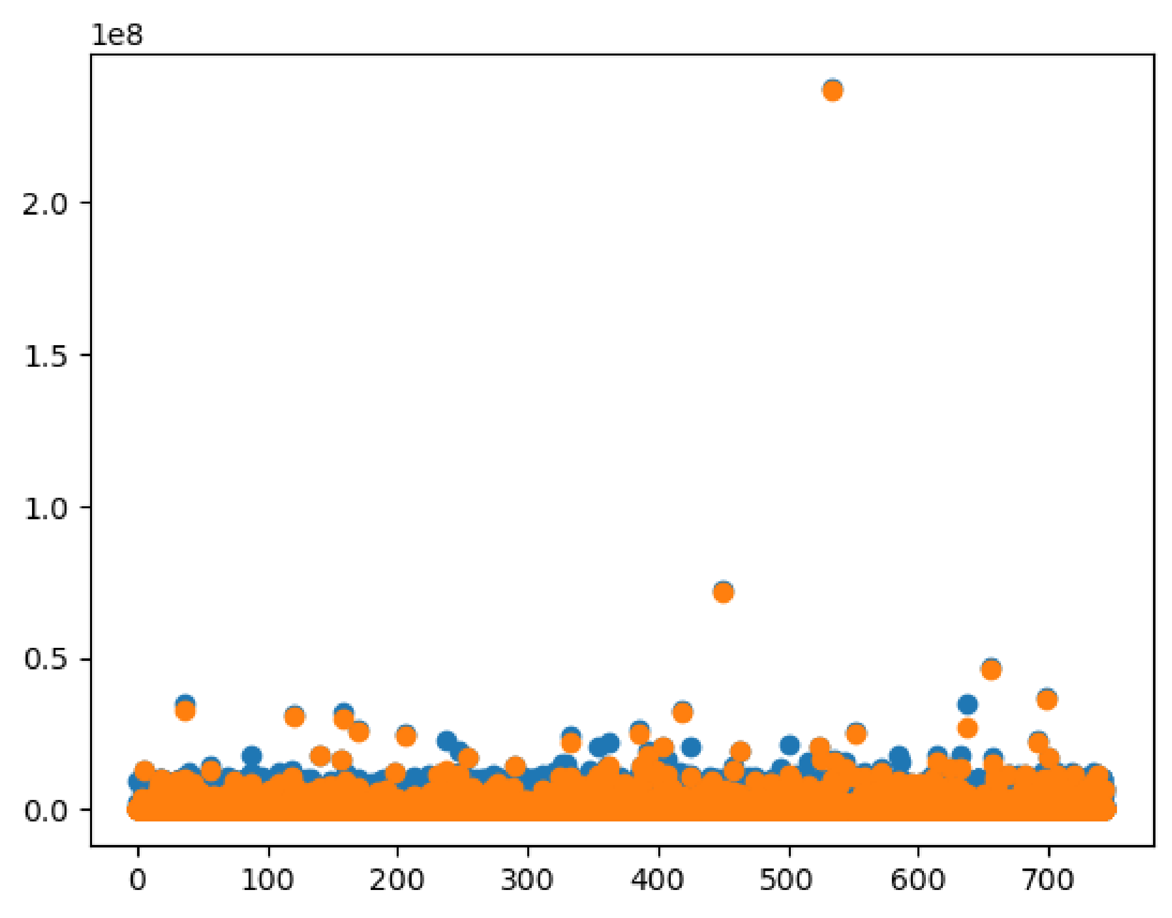
# Output:



# Input Code:

```
In [20]: fig = plt.figure()
         ax = fig.add_subplot(1,1,1)
         ax.scatter(fraud['step'],fraud['newbalanceDest'])
         ax.scatter(fraud['step'],fraud['oldbalanceDest'])
         plt.show()
```

# Output:



# Data Cleaning:

Data cleaning is a crucial step in data analysis that involves identifying and correcting errors, inconsistencies, and missing values in the dataset. It ensures that the data is accurate, reliable, and ready for analysis. Tasks include handling missing data, removing duplicates, standardizing data formats, handling outliers, correcting errors, and ensuring data consistency. Data cleaning improves the quality of the data, leading to more accurate analysis and reliable insights.

# Input Code:

```
In [32]: import pandas as pd
         import numpy as np
         import pickle
         import warnings
         warnings.filterwarnings("ignore", category=DeprecationWarning)
         data_fraud = pd.read_csv('PS_20174392719_1491204439457_log.csv')

         data_fraud = data_fraud.replace(to_replace={'PAYMENT':1,'TRANSFER':2,'CASH_OUT':3,
                                                     'CASH_IN':4,'DEBIT':5,'No':0,'Yes':1})
         data_fraud.drop(['nameOrig','nameDest','isFlaggedFraud'],axis=1,inplace=True)
         data_fraud.head()
```

# Output:

Out[32]:

| | step | type | amount | oldbalanceOrg | newbalanceOrig | oldbalanceDest | newbalanceDest | isFraud |
|---|---|---|---|---|---|---|---|---|
| **0** | 1 | 1 | 9839.64 | 170136.0 | 160296.36 | 0.0 | 0.0 | 0 |
| **1** | 1 | 1 | 1864.28 | 21249.0 | 19384.72 | 0.0 | 0.0 | 0 |
| **2** | 1 | 2 | 181.00 | 181.0 | 0.00 | 0.0 | 0.0 | 1 |
| **3** | 1 | 3 | 181.00 | 181.0 | 0.00 | 21182.0 | 0.0 | 1 |
| **4** | 1 | 1 | 11668.14 | 41554.0 | 29885.86 | 0.0 | 0.0 | 0 |

# Random Forest Classifier:

Using a Random Forest Classifier for fraud transaction detection involves preparing the dataset by handling missing values, encoding categorical variables, and balancing the dataset if needed. Relevant features are selected, and the dataset is split into training and testing sets. The Random Forest Classifier is then trained on the training set and tuned for optimal performance. The model is evaluated using metrics like accuracy, precision, recall, and F1-score. Once the model is satisfactory, it can be deployed for real-time or batch processing, with regular monitoring and retraining as necessary. The Random Forest Classifier is chosen for its ability to handle large, high-dimensional, and imbalanced datasets, making it suitable for fraud detection.

# Input Code:

```
In [27]: X = data_fraud.drop(['isFraud'],axis=1)
         y = data_fraud[['isFraud']]

In [28]: from sklearn.model_selection import train_test_split
         train_X, test_X, train_y, test_y = train_test_split(X, y, test_size = 0.2, random_state = 121)

In [29]: from sklearn.ensemble import RandomForestClassifier
         clf = RandomForestClassifier(n_estimators=15)
```

# Input Code:

```
In [30]: import warnings
         from sklearn.exceptions import DataConversionWarning

         # Suppress all warnings by default
         warnings.simplefilter("ignore")

         # Assuming clf is a classifier object from scikit-learn
         # and train_X, train_y, and test_X are properly defined

         # Fit the classifier to the training data and predict probabilities for test data
         with warnings.catch_warnings():
             # Suppress warnings related to data conversion
             warnings.filterwarnings("ignore", category=FutureWarning)
             warnings.filterwarnings("ignore", category=DataConversionWarning)

             probabilities = clf.fit(train_X, train_y.values.ravel()).predict(test_X)
```

# Accuracy of Random Forest Output:

```
In [31]: from sklearn.metrics import average_precision_score
         if True:
             print(average_precision_score(test_y,probabilities))

         0.765657410753324
```

# Thank You