In [118…

```
# Importing all the necessary Libraries

import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from sklearn.preprocessing import RobustScaler
from sklearn.metrics import confusion_matrix
from scipy.stats import f_oneway
from scipy.stats import chi2_contingency
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import cross_val_score
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import StratifiedKFold
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report
from sklearn.metrics import accuracy_score, precision_score, recall_score, f
from sklearn.metrics import roc_curve, roc_auc_score
```

In [62]:

```
# Reading the dataset from the system

dataset_path = "/Users/gaurang/Desktop/CIND_820/online_transactions.csv"

df = pd.read_csv(dataset_path)

df.head()
```

Out[62]:

	step	type	amount	nameOrig	oldbalanceOrg	newbalanceOrig	nam
0	1	PAYMENT	9839.64	C1231006815	170136.0	160296.36	M19797
1	1	PAYMENT	1864.28	C1666544295	21249.0	19384.72	M20442
2	1	TRANSFER	181.00	C1305486145	181.0	0.00	C5532
3	1	CASH_OUT	181.00	C840083671	181.0	0.00	C389
4	1	PAYMENT	11668.14	C2048537720	41554.0	29885.86	M12307

Initial Analysis

In [63]:
```python
# Calculating the number of rows and columns in the dataset

num_rows, num_columns = df.shape

print(f"Number of rows: {num_rows}")
print(f"Number of columns: {num_columns}")
```

```
Number of rows: 6362620
Number of columns: 11
```

In [64]:
```python
# Summary Statistics of the DataFrame

df.describe()
```

Out[64]:

	step	amount	oldbalanceOrg	newbalanceOrig	oldbalanceDest
count	6.362620e+06	6.362620e+06	6.362620e+06	6.362620e+06	6.362620e+06
mean	2.433972e+02	1.798619e+05	8.338831e+05	8.551137e+05	1.100702e+06
std	1.423320e+02	6.038582e+05	2.888243e+06	2.924049e+06	3.399180e+06
min	1.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
25%	1.560000e+02	1.338957e+04	0.000000e+00	0.000000e+00	0.000000e+00
50%	2.390000e+02	7.487194e+04	1.420800e+04	0.000000e+00	1.327057e+05
75%	3.350000e+02	2.087215e+05	1.073152e+05	1.442584e+05	9.430367e+05
max	7.430000e+02	9.244552e+07	5.958504e+07	4.958504e+07	3.560159e+08

In [65]:
```python
# Checking the datatypes of all the variables

print(df.dtypes)
```

```
step                int64
type               object
amount            float64
nameOrig           object
oldbalanceOrg     float64
newbalanceOrig    float64
nameDest           object
oldbalanceDest    float64
newbalanceDest    float64
isFraud             int64
isFlaggedFraud      int64
dtype: object
```

In [66]:
```
# Converting the 'isFraud' column into class variable 'Class'
df['Class'] = df['isFraud'].map({0: 'Non-Fraudulent', 1: 'Fraudulent'})

# Dropping the original 'isFraud' column if the 'Class' column is sufficient
df.drop('isFraud', axis=1, inplace=True)
```

The transformation of the 'isFraud' column from numeric (with values 0 and 1) to a class variable is essentially to establish its role as the target or dependent variable in a classification problem, particularly in the context of fraud detection.

In [67]:
```
# Printing the data types again to check the validity of the changes made ea

print(df.dtypes)
```

```
step                int64
type               object
amount            float64
nameOrig           object
oldbalanceOrg     float64
newbalanceOrig    float64
nameDest           object
oldbalanceDest    float64
newbalanceDest    float64
isFlaggedFraud      int64
Class              object
dtype: object
```

In [68]:
```
# Checking the dataset for any missing or null values (If the answer is True

df.isna().any().any()
```

Out[68]:
```
False
```

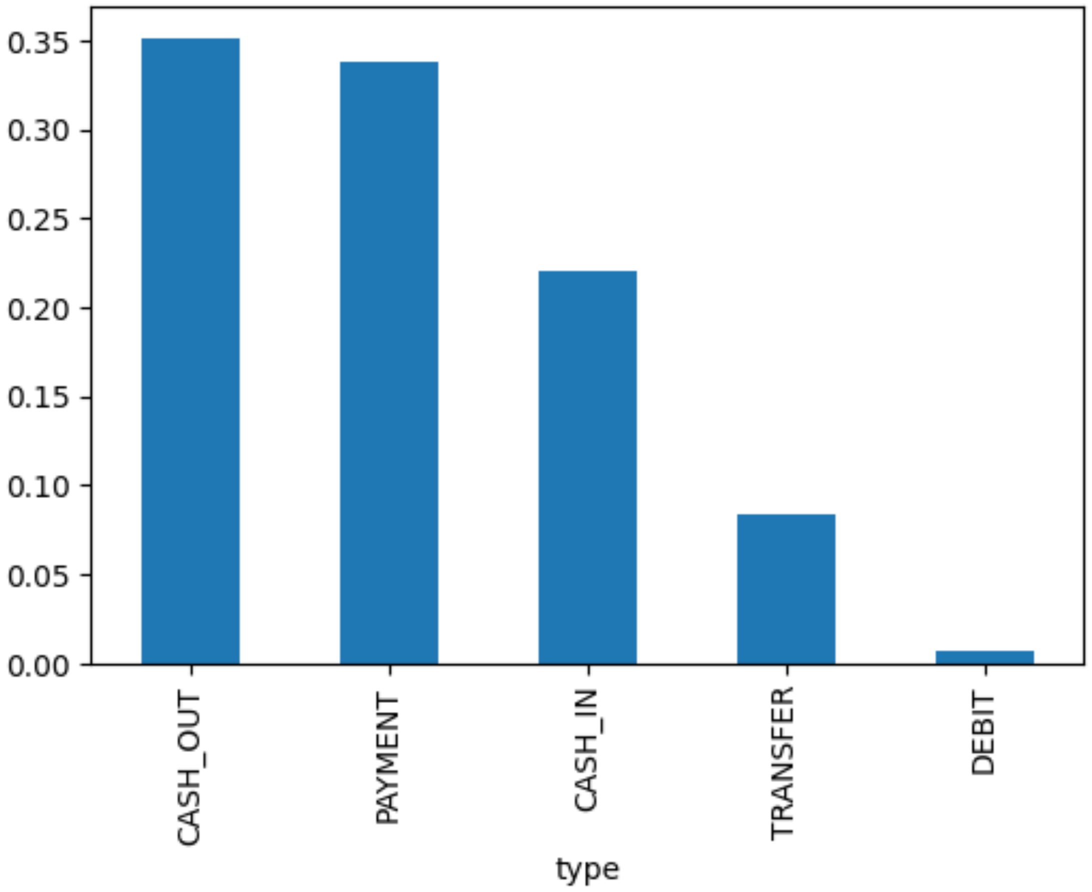
In [69]:
```
# Another way to check for any missing or null values (this will give us the

df.isnull().sum()
```

Out[69]:
```
step              0
type              0
amount            0
nameOrig          0
oldbalanceOrg     0
newbalanceOrig    0
nameDest          0
oldbalanceDest    0
newbalanceDest    0
isFlaggedFraud    0
Class             0
dtype: int64
```

In [70]:
```
# Checking for any duplicate rows and removing them, if any

df.drop_duplicates
```

Out[70]:
```
<bound method DataFrame.drop_duplicates of          step      type      amo
unt     nameOrig  oldbalanceOrg  \
0           1   PAYMENT     9839.64  C1231006815      170136.00
1           1   PAYMENT     1864.28  C1666544295       21249.00
2           1  TRANSFER      181.00  C1305486145         181.00
3           1  CASH_OUT      181.00   C840083671         181.00
4           1   PAYMENT    11668.14  C2048537720       41554.00
...       ...       ...         ...          ...            ...
6362615   743  CASH_OUT   339682.13   C786484425      339682.13
6362616   743  TRANSFER  6311409.28  C1529008245     6311409.28
6362617   743  CASH_OUT  6311409.28  C1162922333     6311409.28
6362618   743  TRANSFER   850002.52  C1685995037      850002.52
6362619   743  CASH_OUT   850002.52  C1280323807      850002.52

         newbalanceOrig     nameDest  oldbalanceDest  newbalanceDest  \
0             160296.36  M1979787155            0.00            0.00
1              19384.72  M2044282225            0.00            0.00
2                  0.00   C553264065            0.00            0.00
3                  0.00    C38997010        21182.00            0.00
4              29885.86  M1230701703            0.00            0.00
...                 ...          ...             ...             ...
6362615            0.00   C776919290            0.00       339682.13
6362616            0.00  C1881841831            0.00            0.00
6362617            0.00  C1365125890        68488.84      6379898.11
6362618            0.00  C2080388513            0.00            0.00
6362619            0.00   C873221189      6510099.11      7360101.63

         isFlaggedFraud            Class
0                     0  Non-Fraudulent
1                     0  Non-Fraudulent
2                     0      Fraudulent
3                     0      Fraudulent
4                     0  Non-Fraudulent
...                 ...             ...
6362615               0      Fraudulent
6362616               0      Fraudulent
6362617               0      Fraudulent
6362618               0      Fraudulent
6362619               0      Fraudulent

[6362620 rows x 11 columns]>
```

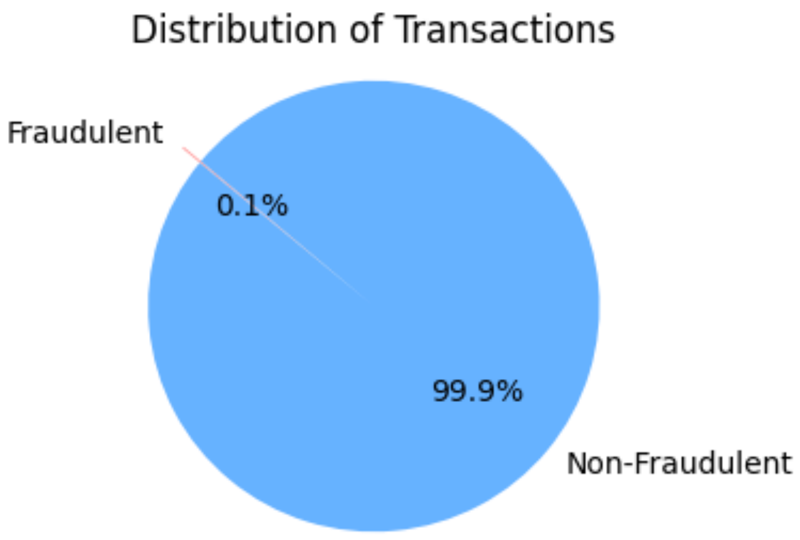
In [71]:
```
# Plotting a histogram to get the different categories and their frequency c

fig = plt.figure(figsize =(6, 4))
df['type'].value_counts(normalize=True).plot(kind='bar')
plt.show()
```



In [72]:
```
# Checking the number of transactions in each type

df['type'].value_counts()
```

Out[72]:
```
type
CASH_OUT    2237500
PAYMENT     2151495
CASH_IN     1399284
TRANSFER     532909
DEBIT         41432
Name: count, dtype: int64
```

In [73]:
```
# Creating a Pie Chart to display the distribution of Fradulent and Non-frac

# Counting the number of fraudulent and non-fraudulent transactions
fraudulent_count = (df['Class'] == 'Fraudulent').sum()
non_fraudulent_count = (df['Class'] == 'Non-Fraudulent').sum()

# Creating a list of counts
transaction_counts = [fraudulent_count, non_fraudulent_count]

# Labels for the two categories
labels = ['Fraudulent', 'Non-Fraudulent']

# Colors for the two categories
colors = ['#FF9999', '#66B2FF']

# Explode a slice if it is fraudulent
explode = (0.1, 0)

# Creating the pie chart
plt.figure(figsize=(3, 3))
plt.pie(transaction_counts, labels=labels, colors=colors, explode=explode, a
plt.title('Distribution of Transactions')
plt.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a circl

# Showing the pie chart
plt.show()
```



The Pie chart shows that 99.9% transactions are Non-Fradulent and only 0.1% of the transactions are Fradulent. This shows that the dataset is highly imbalanced. Therefore, we will have to deal with that in the Experimental Design phase.

In [74]:
```
# Checking the number of Fraud and notFraud transactions

df['Class'].value_counts()
```

Out[74]:
```
Class
Non-Fraudulent    6354407
Fraudulent           8213
Name: count, dtype: int64
```

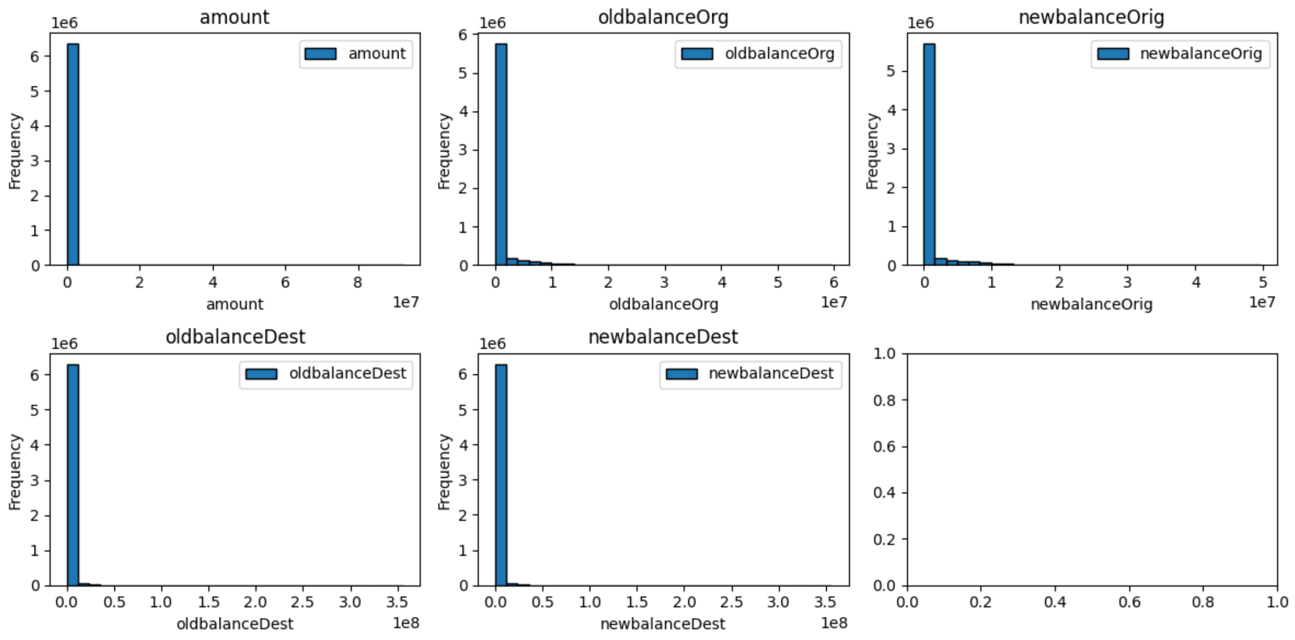
Exploratory Analysis

In [75]:
```
# Removing the irrelevant columns from the dataset (Sub-setting)

new_df = df.drop(['isFlaggedFraud','nameOrig','nameDest', 'step'], axis = 1)
```

The removal of these columns was based on their perceived lack of alignment with the primary objective of detecting fraudulent activities in credit card transactions. The decision was made to streamline the dataset, focusing on features more likely to aid in the accurate identification of fraudulent behavior.

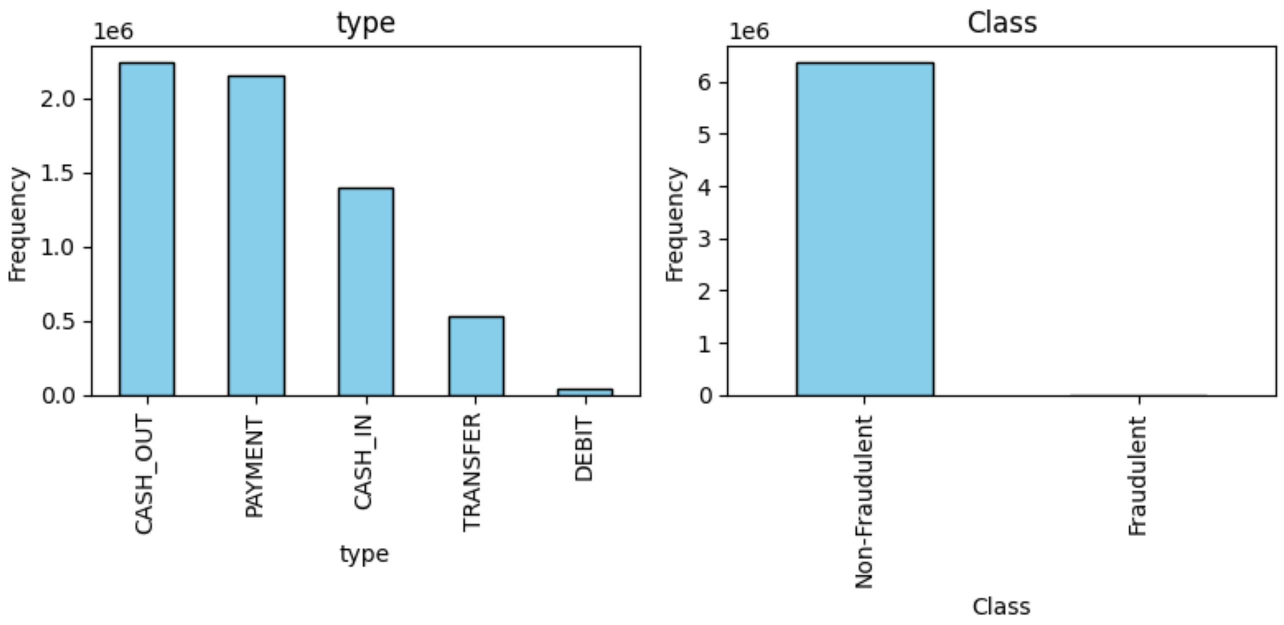
Note: This is not a part of Dimensionality Reduction. This is usually based of the domain knowledge and aim of the project.

In [76]:
```
# Plotting Histograms to better understand the numeric variables

fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(12, 6))

# List of numeric columns to create histograms for
numeric_columns = ["amount", "oldbalanceOrg", "newbalanceOrig", "oldbalanceD

# Creating histograms for each numeric variable
for i, col in enumerate(numeric_columns):
    ax = axes[i // 3, i % 3]
    df[col].plot(kind="hist", bins=30, ax=ax, edgecolor='k', legend=True)
    ax.set_title(col)
    ax.set_xlabel(col)
    ax.set_ylabel("Frequency")

# Adjusting the layout
plt.tight_layout()
plt.show()
```

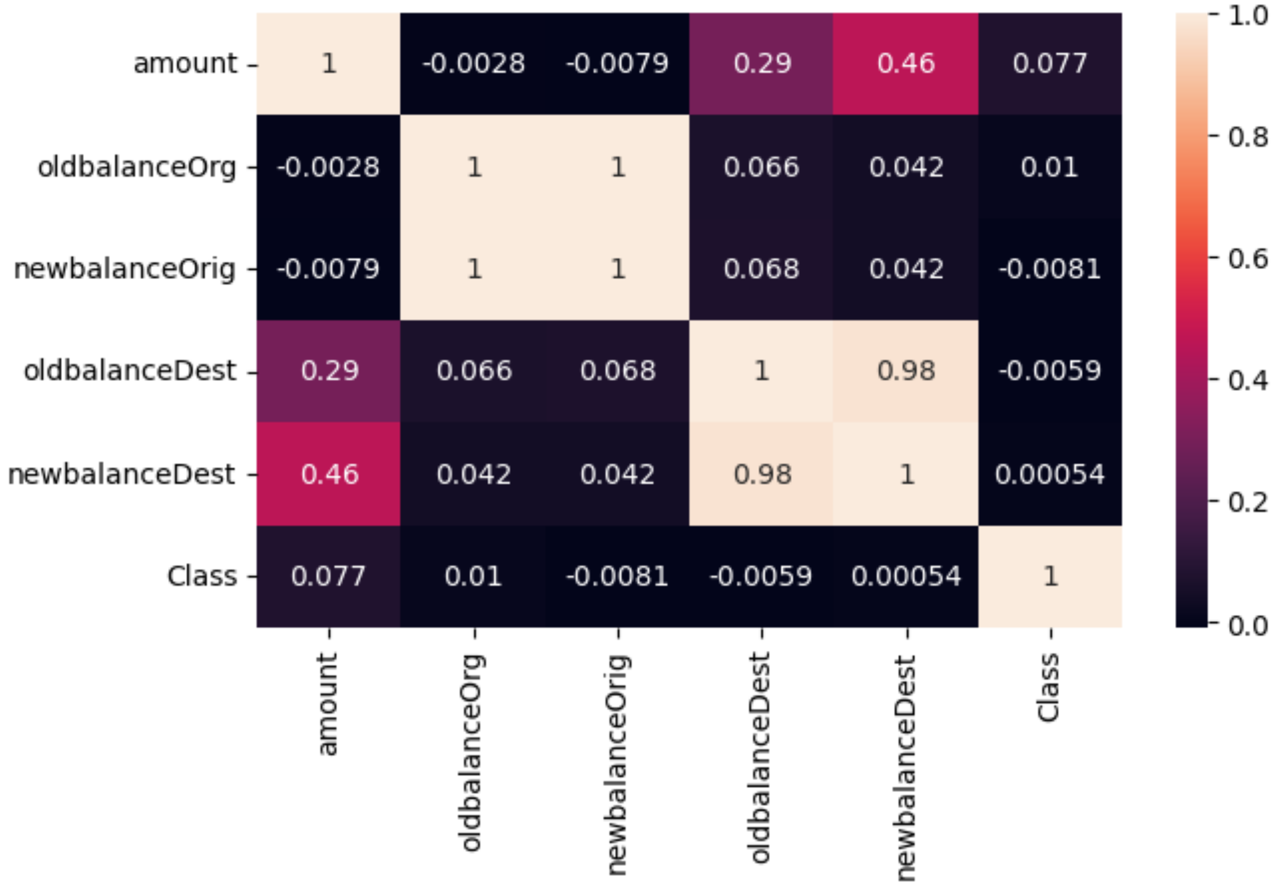


The histograms of the dataset reveal left-skewed data distributions. To ensure optimal performance, normalization will be necessary, especially for logistic regression, which benefits from normalized data due to optimization algorithms. While decision tree and random forest models aren't highly sensitive to data scale, using normalized data could potentially enhance their results. As a result, normalization will be conducted later to improve the overall modeling process.

```
In [77]: # Plotting Bar Charts to better understand the Categorical variables

         # Categorical columns to create bar charts for
         categorical_columns = ['type', 'Class']

         # Creating a figure with subplots
         fig, axes = plt.subplots(nrows=1, ncols=len(categorical_columns), figsize=(8

         # Creating bar charts for each categorical variable
         for i, col in enumerate(categorical_columns):
             ax = axes[i]
             counts = df[col].value_counts()
             counts.plot(kind="bar", ax=ax, color="skyblue", edgecolor='k', legend=Fa
             ax.set_title(col)
             ax.set_xlabel(col)
             ax.set_ylabel("Frequency")

         # Adjusting the layout
         plt.tight_layout()
         plt.show()
```



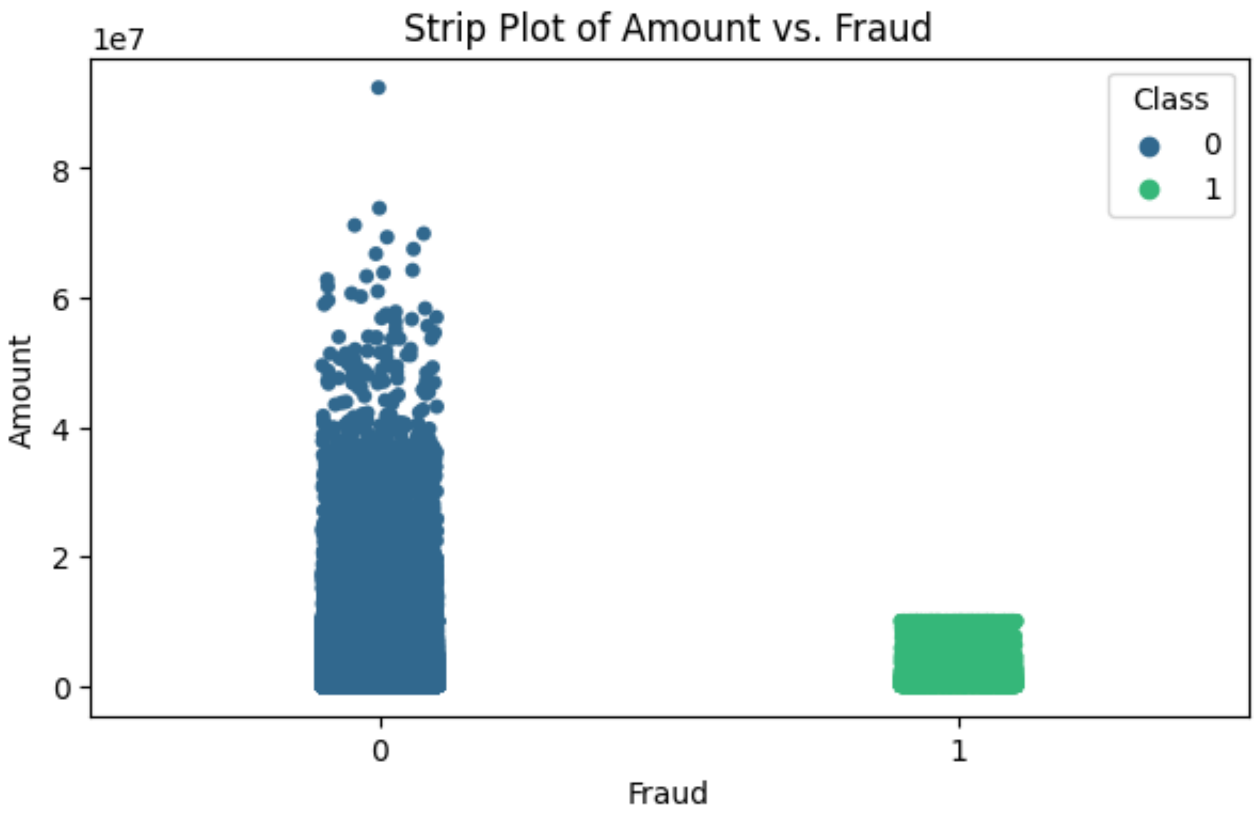
```
In [78]: # Checking the Correlation of each variable with the Class variable

         # Converting the 'Class' column to numeric
         new_df['Class'] = new_df['Class'].map({'Non-Fraudulent': 0, 'Fraudulent': 1}

         # Creating a copy of the DataFrame excluding non-numeric columns (if any)
         correlationdata = new_df.select_dtypes(include=[np.number])

         # Plotting the correlation heatmap
         fig = plt.figure(figsize=(7, 4))
         sns.heatmap(correlationdata.corr(), annot=True)
         plt.show()
```



	amount	oldbalanceOrg	newbalanceOrig	oldbalanceDest	newbalanceDest	Class
amount	1	-0.0028	-0.0079	0.29	0.46	0.077
oldbalanceOrg	-0.0028	1	1	0.066	0.042	0.01
newbalanceOrig	-0.0079	1	1	0.068	0.042	-0.0081
oldbalanceDest	0.29	0.066	0.068	1	0.98	-0.0059
newbalanceDest	0.46	0.042	0.042	0.98	1	0.00054
Class	0.077	0.01	-0.0081	-0.0059	0.00054	1

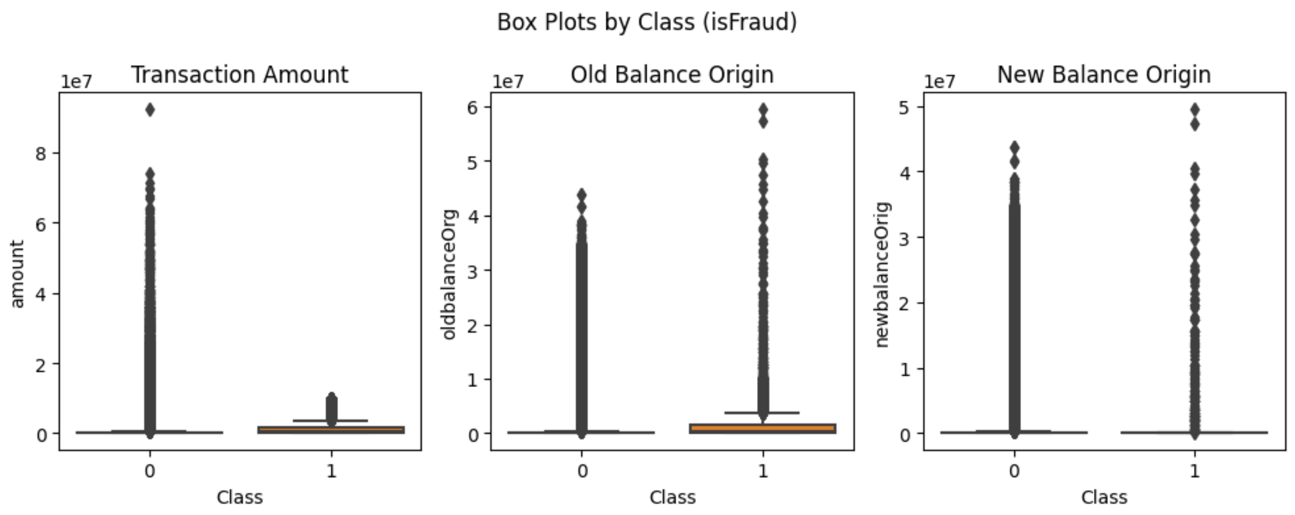
The absence of correlation in the heatmap between the 'Class' variable and other features suggests a lack of linear relationship. To address this, future steps will involve employing feature engineering techniques and anomaly detection methods. These strategies aim to uncover non-linear connections or hidden patterns between the 'Class' variable and the dataset's features that might not be captured by linear correlations.

```
In [79]: # Visualizing relationship between transaction amounts and whether they are
         # Using stripplot to visualize relationship between 'amount' and 'Class'

         plt.figure(figsize=(7, 4))
         sns.stripplot(data=new_df, x='Class', y='amount', hue='Class', palette='viri
         plt.title('Strip Plot of Amount vs. Fraud')
         plt.xlabel('Fraud')
         plt.ylabel('Amount')
         plt.show()
```



The strip plot visualization reveals a clear trend: all fraudulent transactions in the dataset are linked to smaller transaction amounts. This observation underscores the potential significance of lower transaction amounts as a key feature in identifying fraudulent activities within the dataset, offering valuable insights for fraud detection.

In [80]:
```python
# Generating box plots for the 'amount,' 'oldbalanceOrg,' and 'newbalanceOri

# Setting the figure size
plt.figure(figsize=(10, 4))

# Creating box plots for 'amount', 'oldbalanceOrg', and 'newbalanceOrig' by
plt.subplot(1, 3, 1)
sns.boxplot(x='Class', y='amount', data=new_df)
plt.title('Transaction Amount')

plt.subplot(1, 3, 2)
sns.boxplot(x='Class', y='oldbalanceOrg', data=new_df)
plt.title('Old Balance Origin')

plt.subplot(1, 3, 3)
sns.boxplot(x='Class', y='newbalanceOrig', data=new_df)
plt.title('New Balance Origin')

# Adding a title to the overall figure
plt.suptitle('Box Plots by Class (isFraud)')

# Adjusting the layout
plt.tight_layout()

# Showing the plots
plt.show()
```



The box plot highlights potential outliers in the data, suggesting an unbalanced distribution between fraudulent and non-fraudulent transactions. Although the presence of outliers is identified, due to the scarcity of fraud cases, no direct outlier removal is proposed, as it could further limit the already sparse number of fraud transactions. Subsequent outlier handling should consider this class imbalance and is planned for a later stage in the analysis.

Dimensionality Reduction

In [81]:
```
# Encoding 'type' column using one-hot encoding method

data = pd.get_dummies(data = new_df,columns = ['type'], drop_first = True)
data.head()
```

Out[81]:

	amount	oldbalanceOrg	newbalanceOrig	oldbalanceDest	newbalanceDest	Class
0	9839.64	170136.0	160296.36	0.0	0.0	0
1	1864.28	21249.0	19384.72	0.0	0.0	0
2	181.00	181.0	0.00	0.0	0.0	1
3	181.00	181.0	0.00	21182.0	0.0	1
4	11668.14	41554.0	29885.86	0.0	0.0	0

One-hot encoding the 'type' variable was done to ensure compatibility with logistic regression, which requires numerical inputs. While decision trees and random forests can handle categorical data directly, logistic regression specifically needs numerical features. The true/false values resulting from one-hot encoding are treated as binary (0 and 1), which suits logistic regression's requirements while allowing the data to be used across all three models effectively.

In [82]:
```
# Robust Scaling of the Entire Dataset using RobustScaler (Normalizing)

rscaler = RobustScaler()
scaled_data = rscaler.fit_transform(data)
data_sc = pd.DataFrame(scaled_data, columns = data.columns)

data_sc.head()
```

Out[82]:

	amount	oldbalanceOrg	newbalanceOrig	oldbalanceDest	newbalanceDest	Class
0	-0.332932	1.452991	1.111175	-0.140722	-0.193057	0.
1	-0.373762	0.065610	0.134375	-0.140722	-0.193057	0.
2	-0.382380	-0.130708	0.000000	-0.140722	-0.193057	1.
3	-0.382380	-0.130708	0.000000	-0.118260	-0.193057	1.
4	-0.323571	0.254820	0.207169	-0.140722	-0.193057	0.

In handling this dataset, robust scaling was chosen for normalization. Given the limited amount of fraud data, traditional outlier removal methods might lead to the loss of important information in the already sparse fraudulent transactions. Robust scaling was preferred as it's less sensitive to outliers, enabling normalization while preserving the integrity of the fraud-related data.

In [83]:
```
# Checking unique values in the 'Class' column

unique_classes = data_sc['Class'].unique()
print(unique_classes)
```

```
[0. 1.]
```

We can see that our Class variable and a few other variables are converted from Binary Integer format to Float format. This is because of Robust Scaling. The RobustScaler, used for scaling data, standardizes features, and is robust to outliers. When applied, it converts integer variables like '0' and '1' to floating-point numbers ('0.0' and '1.0'). Despite the change in data type, these float representations maintain the original categorical information—the values '0.0' and '1.0' still represent the same categories as '0' and '1', ensuring the integrity of the binary classification.

Even though the float numbes represent the binary class, we will still change the float numbers to integers to avoid any confusion since the original dataset was also in Binary Integer format.

In [84]:
```
# Converting the columns from float to integer

data_sc['Class'] = data_sc['Class'].astype(int)
data_sc['type_CASH_OUT'] = data_sc['type_CASH_OUT'].astype(int)
data_sc['type_DEBIT'] = data_sc['type_DEBIT'].astype(int)
data_sc['type_PAYMENT'] = data_sc['type_PAYMENT'].astype(int)
data_sc['type_TRANSFER'] = data_sc['type_TRANSFER'].astype(int)
```

In [85]:
```
# Verifying the change

data_sc.head()
```

Out[85]:

	amount	oldbalanceOrg	newbalanceOrig	oldbalanceDest	newbalanceDest	Class
0	-0.332932	1.452991	1.111175	-0.140722	-0.193057	0
1	-0.373762	0.065610	0.134375	-0.140722	-0.193057	0
2	-0.382380	-0.130708	0.000000	-0.140722	-0.193057	1
3	-0.382380	-0.130708	0.000000	-0.118260	-0.193057	1
4	-0.323571	0.254820	0.207169	-0.140722	-0.193057	0

In [86]:
```
# Counting the number of Fradulent transactions in each transaction type

# List of the newly created type variables after one-hot encoding
new_type_columns = ['type_CASH_OUT', 'type_DEBIT', 'type_PAYMENT', 'type_TRA

# Counting the number of fraud (1) and non-fraud (0) transactions for each r
for column in new_type_columns:
    fraud_count = data_sc[data_sc[column] == 1]['Class'].sum()  # Number of
    non_fraud_count = len(data_sc[data_sc[column] == 0]) - fraud_count  # Nu
    print(f"Variable: {column}")
    print(f"Fraudulent transactions: {fraud_count}")
    print(f"Non-fraudulent transactions: {non_fraud_count}\n")
```

```
Variable: type_CASH_OUT
Fraudulent transactions: 4116
Non-fraudulent transactions: 4121004

Variable: type_DEBIT
Fraudulent transactions: 0
Non-fraudulent transactions: 6321188

Variable: type_PAYMENT
Fraudulent transactions: 0
Non-fraudulent transactions: 4211125

Variable: type_TRANSFER
Fraudulent transactions: 4097
Non-fraudulent transactions: 5825614
```

The analysis conducted involved counting the number of fraudulent and non-fraudulent transactions within each transaction type, breaking down the fraud occurrences based on different transaction categories. This analysis aimed to explore how fraud incidents are distributed across various transaction types.

This analysis showcases the distribution of fraud across various transaction types. It suggests that fraud is more prevalent in 'type_CASH_OUT' and 'type_TRANSFER' transactions, while 'type_DEBIT' and 'type_PAYMENT' categories have no fraudulent activities.

In [87]:
```
# No-Model Prediction

# Defining a simple rule to predict all transactions as non-fraudulent (0)
predicted_labels = [0] * len(data_sc)

# True labels from the 'Class' column
true_labels = data_sc['Class']

# Creating a confusion matrix
cm = confusion_matrix(true_labels, predicted_labels)
print("Confusion Matrix:")
print(cm)

from sklearn.metrics import accuracy_score

# Calculate accuracy
accuracy = accuracy_score(true_labels, predicted_labels)
print("Accuracy:", accuracy)
```

```
Confusion Matrix:
[[6354407       0]
 [   8213       0]]
Accuracy: 0.9987091795518198
```

The no-model evaluation was conducted to establish a baseline performance and to gain insights into the inherent distribution of the 'Class' variable within the dataset. A simple rule was applied to predict all transactions as non-fraudulent (0) without leveraging any model or feature analysis.

The resulting confusion matrix displayed a high accuracy of 99.87%. This accuracy signifies that the no-model prediction aligned perfectly with the existing distribution of the 'Class' variable, correctly assigning the majority class label to the instances.

The purpose of this evaluation was to provide a preliminary understanding of the class distribution and to set a baseline for subsequent model assessments. The high accuracy in the no-model prediction suggests that the dataset predominantly consists of non-fraudulent transactions. However, it does not imply the effectiveness of the model or the impact of feature engineering; rather, it confirms that the original class distribution was replicated through a simple, rule-based prediction method. This baseline helps in comparing and evaluating the performance of subsequent models and analyses in the context of fraud detection.

```
In [88]: # Performing Chi-squared Test on categorical variables

# Subset the relevant columns for the contingency table
subset = data_sc[['Class', 'type_CASH_OUT', 'type_DEBIT', 'type_PAYMENT', 't

# Create a contingency table
contingency_table = pd.crosstab(subset['Class'], [subset['type_CASH_OUT'], s

# Perform the chi-squared test
chi2, p, dof, expected = chi2_contingency(contingency_table)

# Output the results
print(f"Chi-squared statistic: {chi2}")
print(f"P-value: {p}")
print(f"Degrees of freedom: {dof}")
print("Expected frequencies table:")
print(expected)
```

```
Chi-squared statistic: 22082.535713191082
P-value: 0.0
Degrees of freedom: 4
Expected frequencies table:
[[1.39747778e+06 5.32221110e+05 2.14871781e+06 4.13785187e+04
  2.23461179e+06]
 [1.80622440e+03 6.87889834e+02 2.77719374e+03 5.34812728e+01
  2.88821075e+03]]
```

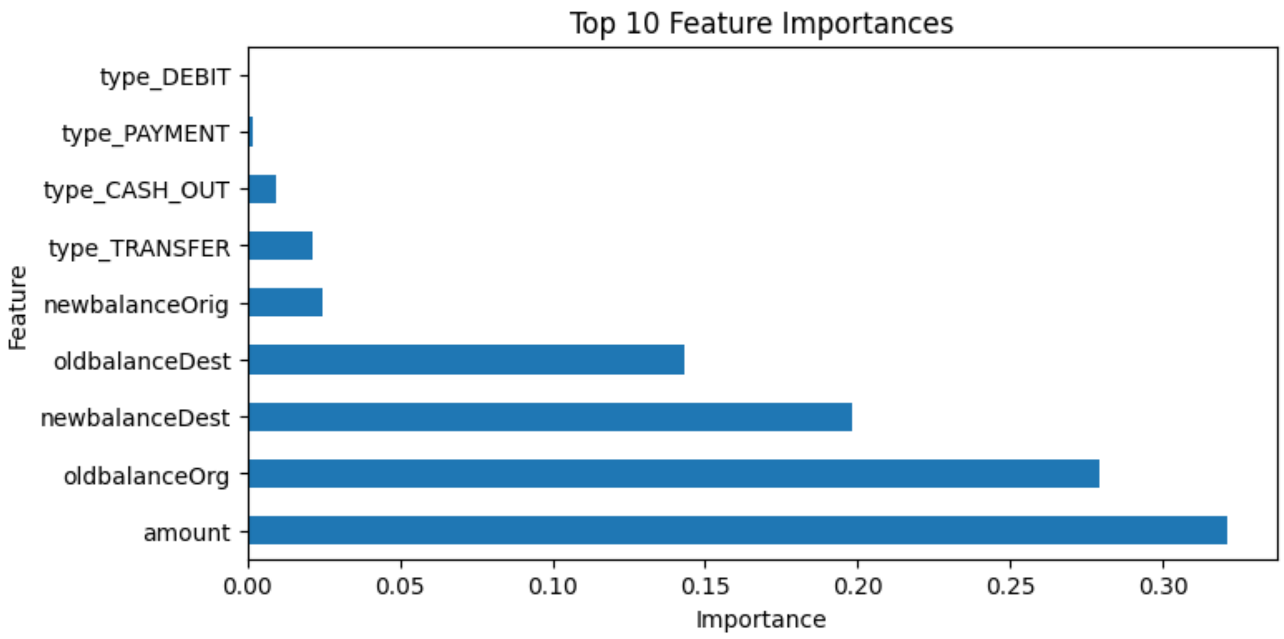
The low p-value and the significant chi-squared statistic suggest a strong association between the 'Class' variable and the 'type_' variables. It indicates that the 'type_' variables are not independent of the 'Class' variable, meaning that the transaction type ('type_') is associated with the occurrence of fraud ('Class').

In [89]:
```
# Performing ANVOVA Test on numeric variables

# Select the columns for the ANOVA test
selected_columns = ['amount', 'oldbalanceOrg', 'newbalanceOrig', 'oldbalance

# Perform ANOVA
f_statistic, p_value = f_oneway(*[data_sc[column] for column in selected_col

print(f'ANOVA F-Statistic: {f_statistic}')
print(f'ANOVA p-value: {p_value}')
```

```
ANOVA F-Statistic: 300794.9628888012
ANOVA p-value: 0.0
```

The ANOVA output indicates a highly significant difference among the means of the variables being analyzed. The extremely low p-value and the large F-statistic value suggest that at least one of the means among the selected variables is significantly different from the others. This indicates a substantial statistical difference in the group means, implying that these variables have a strong association with the class distribution (fraudulent vs. non-fraudulent transactions)

In [90]:
```
# Using Random Forest Classifier to determine feature importances.
# Assuming 'Class' is the target variable and the rest are features
X = data_sc.drop('Class', axis=1).head(100000)
y = data_sc['Class'].head(100000)

# Creating and fitting a Random Forest model
model = RandomForestClassifier()
model.fit(X, y)

# Getting feature importances
feature_importances = model.feature_importances_
features = X.columns

# Creating a Series with feature importances
feature_importance_series = pd.Series(feature_importances, index=features)

# Sorting feature importances in descending order
feature_importance_series = feature_importance_series.sort_values(ascending=

# Plotting the top 'n' most important features
n = 10  # Setting the number of features to display
plt.figure(figsize=(8, 4))
feature_importance_series.head(n).plot(kind='barh')
plt.title(f'Top {n} Feature Importances')
plt.xlabel('Importance')
plt.ylabel('Feature')
plt.show()
```



The ANOVA test revealed the insignificance of 'type_PAYMENT' and 'type_DEBIT' due to their lack of association with fraudulent transactions. As a result, these features were removed to refine the dataset for better modeling. The feature importance chart further indicated lower relevance for 'type_CASH_OUT' and 'type_TRANSFER'. To ensure more accurate predictions for all transaction types and considering the challenges posed by the dataset's size, these features were also excluded to streamline the dataset for improved modeling accuracy and to avoid potential compilation issues.

```
In [91]: data_sc.drop(['type_TRANSFER', 'type_CASH_OUT', 'type_PAYMENT', 'type_DEBIT'
         data_sc.head(10)
```

Out[91]:

	amount	oldbalanceOrg	newbalanceOrig	oldbalanceDest	newbalanceDest	Clas
0	-0.332932	1.452991	1.111175	-0.140722	-0.193057	(
1	-0.373762	0.065610	0.134375	-0.140722	-0.193057	(
2	-0.382380	-0.130708	0.000000	-0.140722	-0.193057	'
3	-0.382380	-0.130708	0.000000	-0.118260	-0.193057	'
4	-0.323571	0.254820	0.207169	-0.140722	-0.193057	(
5	-0.343284	0.369491	0.319165	-0.140722	-0.193057	(
6	-0.346918	1.574679	1.220638	-0.140722	-0.193057	(
7	-0.343059	1.508447	1.166141	-0.140722	-0.193057	(
8	-0.362704	-0.107506	0.000000	-0.140722	-0.193057	(
9	-0.355980	0.256366	0.252202	-0.096293	-0.156769	(

Re-introducing the 'step' variable for feature engineering can provide valuable insights into transaction timing. The time at which a transaction occurs could hold significant relevance, allowing for the creation of a new feature related to transaction timing or frequency, potentially contributing to a more comprehensive understanding of fraudulent activities.

In [92]:
```
# Extracting 'step' variable from 'df' dataset and adding it to 'data_sc'

# Converting 'step' variable to datetime
df['step'] = pd.to_datetime(df['step'], unit='h')

data_sc['step'] = df['step']
```

In [93]:
```
# Extracting information about the occurrence of fraud at different times of

# Filtering fraudulent transactions
fraudulent_data = data_sc[data_sc['Class'] == 1]

# Grouping by hour and counting fraudulent transactions
fraud_by_hour = fraudulent_data['step'].dt.hour.value_counts()

# Identifying the hour with the most fraud occurrences
peak_fraud_hour = fraud_by_hour.idxmax()
max_fraud_count = fraud_by_hour.max()

print(f"The peak hour for fraud occurrences is at {peak_fraud_hour}:00 with
```

```
The peak hour for fraud occurrences is at 10:00 with 375 fraud transactions.
```

In [94]:
```
# Creating a new feature 'isPeakFraudHour' to flag transactions occurring at

data_sc['isPeakFraudHour'] = (data_sc['step'].dt.hour == 10).astype(int)

# This feature flags transactions at the specific peak fraud hour (10:00) as
```

The 'isPeakFraudHour' feature flags transactions occurring at the specific peak hour for fraud (10:00). It's a binary column: 1 denotes transactions at 10:00, and 0 represents all others. Adding this feature to 'data_sc' aids models in identifying potential risk during the critical peak hour. It helps distinguish fraudulent activities at this time from others, enhancing the models' ability to recognize correlations between transactions and fraud occurrences.

In [95]:
```
# Dropping the 'step' variable and keeping the 'isPeakFraudHour'

data_sc.drop('step', axis=1, inplace=True)
data_sc.head()
```

Out[95]:

	amount	oldbalanceOrg	newbalanceOrig	oldbalanceDest	newbalanceDest	Clas
0	-0.332932	1.452991	1.111175	-0.140722	-0.193057	(
1	-0.373762	0.065610	0.134375	-0.140722	-0.193057	(
2	-0.382380	-0.130708	0.000000	-0.140722	-0.193057	'
3	-0.382380	-0.130708	0.000000	-0.118260	-0.193057	'
4	-0.323571	0.254820	0.207169	-0.140722	-0.193057	(

Dropping the 'step' variable and retaining 'isPeakFraudHour' helps streamline the dataset. The 'isPeakFraudHour' feature encapsulates the critical time for potential fraud occurrences, providing valuable information for predicting fraudulent activities. This streamlined dataset focuses on the most relevant temporal aspect, enhancing the model's ability to identify fraud patterns during peak hours.

Experimental Design

In [96]:

```
# Splitting data into features (X) and target variable (y)

X = data_sc.drop('Class', axis=1)  # Features
y = data_sc['Class']  # Target variable

# Using stratified sampling to split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, str
```

Stratified sampling ensures that the class distribution is maintained in both the training and test datasets, which is crucial for imbalanced classification problems. It helps in preserving the proportion of the target classes across the train/test splits.

In [97]:
```
# Checking the class distribution in the original data
original_class_distribution = data_sc['Class'].value_counts(normalize=True)
print("Original Class Distribution:")
print(original_class_distribution)

# Checking the class distribution in the training set
training_class_distribution = y_train.value_counts(normalize=True)
print("\nTraining Set Class Distribution:")
print(training_class_distribution)

# Checking the class distribution in the test set
test_class_distribution = y_test.value_counts(normalize=True)
print("\nTest Set Class Distribution:")
print(test_class_distribution)
```

```
Original Class Distribution:
Class
0    0.998709
1    0.001291
Name: proportion, dtype: float64

Training Set Class Distribution:
Class
0    0.998709
1    0.001291
Name: proportion, dtype: float64

Test Set Class Distribution:
Class
0    0.998709
1    0.001291
Name: proportion, dtype: float64
```

Verifying the stratified split's effectiveness by checking the class distribution in the training and test sets.

In [98]:
```
# Applying SMOTE to the training set

smote = SMOTE(random_state=42)
X_train_resampled, y_train_resampled = smote.fit_resample(X_train, y_train)
```

SMOTE is a method used to address imbalanced datasets, especially in cases like credit card fraud detection, where instances of fraud are infrequent compared to non-fraudulent transactions. By generating synthetic instances for the minority class (fraudulent transactions), it rebalances the dataset, helping machine learning models to better learn from a more balanced dataset. This step is performed on the training set to prevent the model from learning synthetic patterns from the test set, ensuring its generalization ability.

In [99]:
```
# Limiting the number of rows to 1 million
X_train_subset = X_train_resampled[:1000000]
y_train_subset = y_train_resampled[:1000000]

# Initializing the Stratified K-Folds
stratified_kf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# Random Forest model
rf_model = RandomForestClassifier()
rf_scores = cross_val_score(rf_model, X_train_subset, y_train_subset, cv=str
print("Random Forest Cross-validation scores:")
print(rf_scores)
print(f"Average Accuracy: {rf_scores.mean()}\n")

# Decision Tree model
dt_model = DecisionTreeClassifier()
dt_scores = cross_val_score(dt_model, X_train_subset, y_train_subset, cv=str
print("Decision Tree Cross-validation scores:")
print(dt_scores)
print(f"Average Accuracy: {dt_scores.mean()}\n")

# Logistic Regression model
lr_model = LogisticRegression()
lr_scores = cross_val_score(lr_model, X_train_subset, y_train_subset, cv=str
print("Logistic Regression Cross-validation scores:")
print(lr_scores)
print(f"Average Accuracy: {lr_scores.mean()}\n")
```

```
Random Forest Cross-validation scores:
[0.99947  0.99957  0.9995   0.99957  0.999515]
Average Accuracy: 0.999525

Decision Tree Cross-validation scores:
[0.99947  0.999505 0.99953  0.999455 0.999445]
Average Accuracy: 0.999481

Logistic Regression Cross-validation scores:
[0.99919  0.99921  0.999215 0.999315 0.99918 ]
Average Accuracy: 0.9992219999999999
```

The cross-validation results demonstrate the average accuracy of different models. Each model's accuracy was assessed using various subgroups of the training data to provide a more robust evaluation. The outcomes show that Random Forest and Decision Tree models exhibit higher accuracy than the Logistic Regression model. However, all models demonstrate extremely high accuracy levels, suggesting strong predictive capabilities in identifying the target variable, possibly indicating a good fit to the training data.

Cross-validation, specifically stratified k-fold, is conducted before the actual modeling to evaluate and choose the best-suited model or technique. This method helps prevent overfitting and assesses the model's ability to generalize to unseen data by rigorously testing it on various dataset partitions. However, due to the computational challenges posed by a very large dataset, the initial dataset was constrained to 1 million rows for verification purposes. The intention was to overcome issues related to compilation times. In the forthcoming research paper, the models will be established and evaluated across various parameters to determine the most suitable one for the credit card fraud detection task.

Modelling

In the project, the approach involves building three models—Random Forest, Decision Tree, and Logistic Regression—followed by their evaluation using untouched test data to ensure an unbiased performance assessment. Given the high accuracy observed for all models, it's beneficial to construct each model and thoroughly evaluate them.

In [100…
```
# Initializing the Random Forest model
random_forest_model = RandomForestClassifier()

# Training the model using the training data
random_forest_model.fit(X_train_resampled, y_train_resampled)
```

Out[100]:
```
▾ RandomForestClassifier
RandomForestClassifier()
```

In [101…
```
# Initializing the Decision Tree model
decision_tree_model = DecisionTreeClassifier(random_state=42)

# Fitting the model to the training data
decision_tree_model.fit(X_train_resampled, y_train_resampled)
```

Out[101]:
```
▾          DecisionTreeClassifier
DecisionTreeClassifier(random_state=42)
```

In [102…
```
# Creating the Logistic Regression model
logistic_model = LogisticRegression()

# Fitting the model on the training data
logistic_model.fit(X_train_resampled, y_train_resampled)
```

Out[102]:
```
▾ LogisticRegression
LogisticRegression()
```

Evaluation

In [111…
```python
# Predicting the target values using the Random Forest model

# Evaluating on test data
y_pred = random_forest_model.predict(X_test)

# Accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy}")

# Precision
precision = precision_score(y_test, y_pred)
print(f"Precision: {precision}")

# Recall
recall = recall_score(y_test, y_pred)
print(f"Recall: {recall}")

# F1 Score
f1 = f1_score(y_test, y_pred)
print(f"F1 Score: {f1}")
```

```
Accuracy: 0.9989147552423373
Precision: 0.5459004905395936
Recall: 0.9482653682288497
F1 Score: 0.692906382032466
```

The model has high accuracy at 99.89%. The precision for identifying fraud is around 54.59%, and it can correctly detect about 94.83% of the actual fraud cases, resulting in an overall balanced performance of around 69.29%.

In [112…
```python
# Random Forest Confusion Matrix
conf_matrix = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:")
print(conf_matrix)
```

```
Confusion Matrix:
[[1269585    1296]
 [     85    1558]]
```

True Positives (TP): 1558 - Correctly predicted fraudulent transactions.

True Negatives (TN): 1269585 - Accurately predicted non-fraudulent transactions.

False Positives (FP): 1296 - Incorrectly predicted non-fraudulent when the actual is fraudulent.

False Negatives (FN): 85 - Incorrectly predicted fraudulent when the actual is non-fraudulent.

In [113…

```
# Predicting the target values using the Decision Tree model

# Evaluating on test data
y_pred = decision_tree_model.predict(X_test)

# Accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy}")

# Precision
precision = precision_score(y_test, y_pred)
print(f"Precision: {precision}")

# Recall
recall = recall_score(y_test, y_pred)
print(f"Recall: {recall}")

# F1 Score
f1 = f1_score(y_test, y_pred)
print(f"F1 Score: {f1}")
```

```
Accuracy: 0.9989634773096617
Precision: 0.5558620689655173
Recall: 0.9811320754716981
F1 Score: 0.7096632181377943
```

The model demonstrates substantial accuracy at approximately 99.90%. In identifying instances of fraud, the precision stands at around 55.59%. The recall, or the ability to correctly detect actual fraud cases, is about 98.11%. This leads to an overall balanced performance of roughly 70.97%.

In [114…

```
# Decision Tree Confusion Matrix
conf_matrix = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:")
print(conf_matrix)
```

```
Confusion Matrix:
[[1269593    1288]
 [     31    1612]]
```

True Positives (TP): 1612 - Correctly predicted fraudulent transactions.

True Negatives (TN): 1269593 - Accurately predicted non-fraudulent transactions.

False Positives (FP): 1288 - Incorrectly predicted non-fraudulent when the actual is fraudulent.

False Negatives (FN): 31 - Incorrectly predicted fraudulent when the actual is non-fraudulent.

In [115…

```
# Predicting the target values using the Logistic Regression model

# Evaluating on test data
y_pred = logistic_model.predict(X_test)

# Accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy}")

# Precision
precision = precision_score(y_test, y_pred)
print(f"Precision: {precision}")

# Recall
recall = recall_score(y_test, y_pred)
print(f"Recall: {recall}")

# F1 Score
f1 = f1_score(y_test, y_pred)
print(f"F1 Score: {f1}")
```

```
Accuracy: 0.963153543665974
Precision: 0.029433177327093083
Recall: 0.8612294583079733
F1 Score: 0.056921034635343336
```

The model demonstrates an accuracy of 96.32%. The precision of approximately 2.94% suggests a low rate of correctly identifying fraud cases, but the model displays a moderate recall of about 86.12%, indicating the ability to capture most actual fraud instances. Additionally, the F1 score, which balances precision and recall, stands at around 5.69%.

In [116…
```
# Logistic Regression Confusion Matrix
conf_matrix = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:")
print(conf_matrix)
```

```
Confusion Matrix:
[[1224221   46660]
 [    228    1415]]
```

True Positives (TP): 1415 - Correctly predicted fraudulent transactions.

True Negatives (TN): 1224221 - Accurately predicted non-fraudulent transactions.

False Positives (FP): 46660 - Incorrectly predicted non-fraudulent when the actual is fraudulent.

False Negatives (FN): 228 - Incorrectly predicted fraudulent when the actual is non-fraudulent.

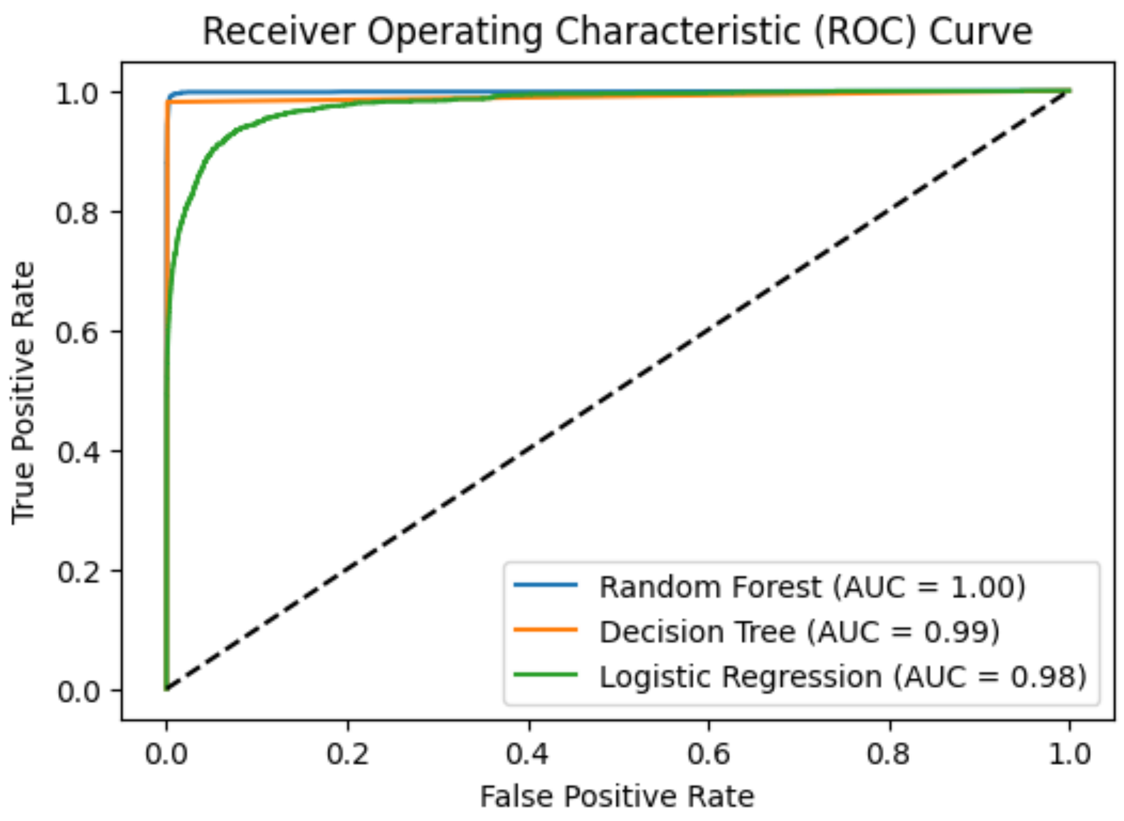
In [120…
```
# Calculating AUC and plotting ROC curve

# Random Forest model
rf_probs = random_forest_model.predict_proba(X_test)[:, 1]
rf_fpr, rf_tpr, _ = roc_curve(y_test, rf_probs)
rf_auc = roc_auc_score(y_test, rf_probs)

# Decision Tree model
dt_probs = decision_tree_model.predict_proba(X_test)[:, 1]
dt_fpr, dt_tpr, _ = roc_curve(y_test, dt_probs)
dt_auc = roc_auc_score(y_test, dt_probs)

# Logistic Regression model
lr_probs = logistic_model.predict_proba(X_test)[:, 1]
lr_fpr, lr_tpr, _ = roc_curve(y_test, lr_probs)
lr_auc = roc_auc_score(y_test, lr_probs)

# Print AUC for each model
print(f"Random Forest AUC: {rf_auc}")
print(f"Decision Tree AUC: {dt_auc}")
print(f"Logistic Regression AUC: {lr_auc}")
```

```
Random Forest AUC: 0.9983661517808896
Decision Tree AUC: 0.9900593802282789
Logistic Regression AUC: 0.9780811489278334
```

The AUC (Area Under the Curve) for the Random Forest model is 0.998, reflecting a high ability to distinguish between classes. The Decision Tree model follows with an AUC of 0.990, showing strong performance in classification. Lastly, the Logistic Regression model, with an AUC of 0.978, exhibits a slightly lower but still commendable capability to separate classes in this context.

In [122…

```
# Plotting ROC curves for all three models
plt.figure(figsize=(6, 4))

# Random Forest
plt.plot(rf_fpr, rf_tpr, label=f'Random Forest (AUC = {rf_auc:.2f})')

# Decision Tree
plt.plot(dt_fpr, dt_tpr, label=f'Decision Tree (AUC = {dt_auc:.2f})')

# Logistic Regression
plt.plot(lr_fpr, lr_tpr, label=f'Logistic Regression (AUC = {lr_auc:.2f})')

# Plotting the ROC curve for a random classifier
plt.plot([0, 1], [0, 1], linestyle='--', color='black')

# Customizing the plot
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend()
plt.show()
```



```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```