Project Report on Credit Card Fraud Detection

December 4, 2024

1 Introduction

The Credit Card Transactions Dataset contains over 1.85 million rows of detailed records, including transaction times, amounts, and personal and merchant details. It can be used for fraud detection by identifying patterns in transaction amounts, locations, and user profiles. Customer segmentation can be achieved by analyzing spending patterns, location, and demographics to tailor marketing strategies. Transactions can be classified into categories like grocery or entertainment to understand spending behaviors and improve recommendation systems. Geospatial analysis can map spending patterns and detect regional trends or anomalies. Predictive modeling can forecast future spending behavior and potential fraudulent activities. Lastly, anomaly detection can identify unusual transaction patterns that deviate from normal behavior to detect potential fraud early.

In evaluating the dataset, various machine learning models such as Logistic Regression, Random Forest Classifier, Random Forest Classifier with PCA, Decision Tree Classifier, Gradient Boost Classifier, and K-fold Cross Validation with Random Forest Classification Model are used. Accuracy, Precision, Recall, and F1 scores are computed along with the Confusion Matrix to assess the performance of these models.

2 1. Exploratory Data Analysis

2.1 General Analysis

[ ]:
```
# All imports and utility functions
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import accuracy_score, f1_score, precision_score,␣
↪recall_score, classification_report, confusion_matrix, precision_recall_curve
from sklearn.model_selection import train_test_split

df = pd.read_csv(dataset_path)
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1296675 entries, 0 to 1296674
Data columns (total 24 columns):
 #   Column                 Non-Null Count    Dtype
---  ------                 --------------    -----
 0   Unnamed: 0             1296675 non-null  int64
```

```
 1   trans_date_trans_time  1296675 non-null  object
 2   cc_num                 1296675 non-null  int64
 3   merchant               1296675 non-null  object
 4   category               1296675 non-null  object
 5   amt                    1296675 non-null  float64
 6   first                  1296675 non-null  object
 7   last                   1296675 non-null  object
 8   gender                 1296675 non-null  object
 9   street                 1296675 non-null  object
 10  city                   1296675 non-null  object
 11  state                  1296675 non-null  object
 12  zip                    1296675 non-null  int64
 13  lat                    1296675 non-null  float64
 14  long                   1296675 non-null  float64
 15  city_pop               1296675 non-null  int64
 16  job                    1296675 non-null  object
 17  dob                    1296675 non-null  object
 18  trans_num              1296675 non-null  object
 19  unix_time              1296675 non-null  int64
 20  merch_lat              1296675 non-null  float64
 21  merch_long             1296675 non-null  float64
 22  is_fraud               1296675 non-null  int64
 23  merch_zipcode          1100702 non-null  float64
dtypes: float64(6), int64(6), object(12)
memory usage: 237.4+ MB
```

Observation:

1. Size and Structure: The dataset contains 1,296,675 entries with 24 columns, including transaction details and personal information.
2. Data Types: The columns include various data types such as integers, floats, and objects (strings).
3. Key Columns: Important columns include transaction time, credit card number, merchant, category, amount, and fraud indicator.
4. Missing Values: The merch_zipcode column has missing values, with only 1,100,702 non-null entries out of 1,296,675.

[ ]:
```
df.describe()
```

[ ]:
```
         Unnamed: 0        cc_num           amt           zip           lat  \
count  1.296675e+06  1.296675e+06  1.296675e+06  1.296675e+06  1.296675e+06
mean   6.483370e+05  4.171920e+17  7.035104e+01  4.880067e+04  3.853762e+01
std    3.743180e+05  1.308806e+18  1.603160e+02  2.689322e+04  5.075808e+00
min    0.000000e+00  6.041621e+10  1.000000e+00  1.257000e+03  2.002710e+01
25%    3.241685e+05  1.800429e+14  9.650000e+00  2.623700e+04  3.462050e+01
50%    6.483370e+05  3.521417e+15  4.752000e+01  4.817400e+04  3.935430e+01
75%    9.725055e+05  4.642255e+15  8.314000e+01  7.204200e+04  4.194040e+01
```

```
max    1.296674e+06  4.992346e+18  2.894890e+04  9.978300e+04  6.669330e+01

               long      city_pop     unix_time     merch_lat    merch_long  \
count  1.296675e+06  1.296675e+06  1.296675e+06  1.296675e+06  1.296675e+06
mean  -9.022634e+01  8.882444e+04  1.349244e+09  3.853734e+01 -9.022646e+01
std    1.375908e+01  3.019564e+05  1.284128e+07  5.109788e+00  1.377109e+01
min   -1.656723e+02  2.300000e+01  1.325376e+09  1.902779e+01 -1.666712e+02
25%   -9.679800e+01  7.430000e+02  1.338751e+09  3.473357e+01 -9.689728e+01
50%   -8.747690e+01  2.456000e+03  1.349250e+09  3.936568e+01 -8.743839e+01
75%   -8.015800e+01  2.032800e+04  1.359385e+09  4.195716e+01 -8.023680e+01
max   -6.795030e+01  2.906700e+06  1.371817e+09  6.751027e+01 -6.695090e+01

           is_fraud  merch_zipcode
count  1.296675e+06   1.100702e+06
mean   5.788652e-03   4.682575e+04
std    7.586269e-02   2.583400e+04
min    0.000000e+00   1.001000e+03
25%    0.000000e+00   2.511400e+04
50%    0.000000e+00   4.586000e+04
75%    0.000000e+00   6.831900e+04
max    1.000000e+00   9.940300e+04
```

Observations

1. Transaction Amounts: The transaction amounts range from 1 to 28,948.90 dollars, with a mean of approximately 70.35 dollars and a standard deviation of 160.32 dollars.
2. Geographical Data: Latitude and longitude values indicate the geographical spread of transactions, with means around 38.54 and -90.23 respectively.
3. Population Data: The city population associated with transactions varies widely, with a mean of 88,824 and a maximum of 2,906,700.
4. Fraud Indicator: The is_fraud column shows that fraudulent transactions are relatively rare, with a mean value of approximately 0.0058, indicating a small proportion of fraud cases in the dataset.

[ ]:
```python
df.head()
```

[ ]:
```
   Unnamed: 0 trans_date_trans_time            cc_num  \
0           0   2019-01-01 00:00:18  2703186189652095
1           1   2019-01-01 00:00:44      630423337322
2           2   2019-01-01 00:00:51    38859492057661
3           3   2019-01-01 00:01:16  3534093764340240
4           4   2019-01-01 00:03:06   375534208663984

                             merchant       category     amt      first  \
0          fraud_Rippin, Kub and Mann       misc_net    4.97   Jennifer
1     fraud_Heller, Gutmann and Zieme    grocery_pos  107.23  Stephanie
2                fraud_Lind-Buckridge  entertainment  220.11     Edward
```

```
3  fraud_Kutch, Hermiston and Farrell  gas_transport   45.00     Jeremy
4                 fraud_Keeling-Crist       misc_pos   41.96      Tyler

      last gender                          street  …       long city_pop  \
0    Banks      F                  561 Perry Cove  …   -81.1781     3495
1     Gill      F  43039 Riley Greens Suite 393  …  -118.2105      149
2  Sanchez      M        594 White Dale Suite 530  …  -112.2620     4154
3    White      M    9443 Cynthia Court Apt. 038  …  -112.1138     1939
4   Garcia      M                408 Bradley Rest  …   -79.4629       99

                                job         dob  \
0         Psychologist, counselling  1988-03-09
1  Special educational needs teacher  1978-06-21
2       Nature conservation officer  1962-01-19
3                   Patent attorney  1967-01-12
4     Dance movement psychotherapist  1986-03-28

                          trans_num   unix_time  merch_lat  merch_long  \
0  0b242abb623afc578575680df30655b9  1325376018  36.011293  -82.048315
1  1f76529f8574734946361c461b024d99  1325376044  49.159047 -118.186462
2  a1a22d70485983eac12b5b88dad1cf95  1325376051  43.150704 -112.154481
3  6b849c168bdad6f867558c3793159a81  1325376076  47.034331 -112.561071
4  a41d7549acf90789359a9aa5346dcb46  1325376186  38.674999  -78.632459

  is_fraud  merch_zipcode
0        0        28705.0
1        0            NaN
2        0        83236.0
3        0            NaN
4        0        22844.0

[5 rows x 24 columns]
```

[ ]:
```
# List all the columns available in the dataset
df.columns
```

[ ]:
```
Index(['Unnamed: 0', 'trans_date_trans_time', 'cc_num', 'merchant', 'category',
       'amt', 'first', 'last', 'gender', 'street', 'city', 'state', 'zip',
       'lat', 'long', 'city_pop', 'job', 'dob', 'trans_num', 'unix_time',
       'merch_lat', 'merch_long', 'is_fraud', 'merch_zipcode'],
      dtype='object')
```

[ ]:
```
# Study each attribute and its characteristics
df.dtypes
```

[ ]:
```
Unnamed: 0                 int64
trans_date_trans_time     object
```

```
cc_num                        int64
merchant                     object
category                     object
amt                         float64
first                        object
last                         object
gender                       object
street                       object
city                         object
state                        object
zip                           int64
lat                         float64
long                        float64
city_pop                      int64
job                          object
dob                          object
trans_num                    object
unix_time                     int64
merch_lat                   float64
merch_long                  float64
is_fraud                      int64
merch_zipcode               float64
dtype: object
```

[ ]:
```
# Check how many rows and columns are there in the dataset
df.shape
```

[ ]:
```
(1296675, 24)
```

Observations

1. Columns and Data Types: The dataset contains 24 columns with various data types, including integers, floats, and objects (strings). Key columns include transaction details, personal information, and geographical data.

2. Attributes: Each attribute has a specific data type, such as int64 for numerical values, float64 for decimal values, and object for categorical or string data.

3. Dataset Size: The dataset comprises 1,296,675 rows and 24 columns, indicating a substantial amount of transaction data to analyze.

[ ]:
```
# Check the missing values
df.isna().sum()
```

[ ]:
```
Unnamed: 0                  0
trans_date_trans_time       0
cc_num                      0
merchant                    0
category                    0
```

```
amt                           0
first                         0
last                          0
gender                        0
street                        0
city                          0
state                         0
zip                           0
lat                           0
long                          0
city_pop                      0
job                           0
dob                           0
trans_num                     0
unix_time                     0
merch_lat                     0
merch_long                    0
is_fraud                      0
merch_zipcode            195973
dtype: int64
```

[ ]:
```
# Check if the data has duplicate values
df.duplicated().sum()
```

[ ]:
```
0
```

Observations

1. Missing Values: The merch_zipcode column has 195,973 missing values, while all other columns have no missing values.
2. Duplicate Values: The dataset contains no duplicate rows, ensuring data integrity.

2.2 Univariate Analysis

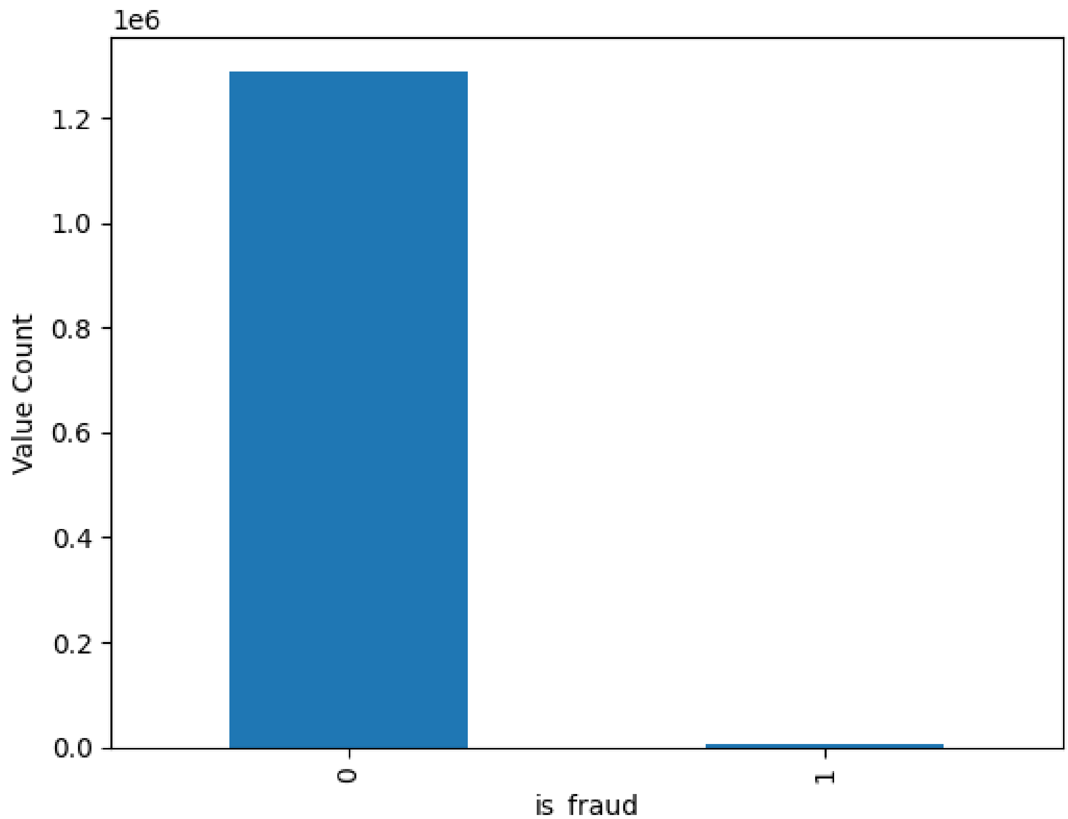
[ ]:
```
# Take a copy of subset of dataframe consisting only fraudulent transactions
df_fraud = df[df['is_fraud'] == 1].copy()
print(df_fraud.shape)
```

```
(7506, 24)
```

[ ]:
```
# How many total fraud transactions are there in the dataframe
print(df['is_fraud'].value_counts())
df['is_fraud'].value_counts().plot(kind='bar', xlabel="is_fraud", ylabel="Value␣
↪Count")
```

```
0    1289169
1       7506
Name: is_fraud, dtype: int64
```

[ ]: <Axes: xlabel='is_fraud', ylabel='Value Count'>



Observations

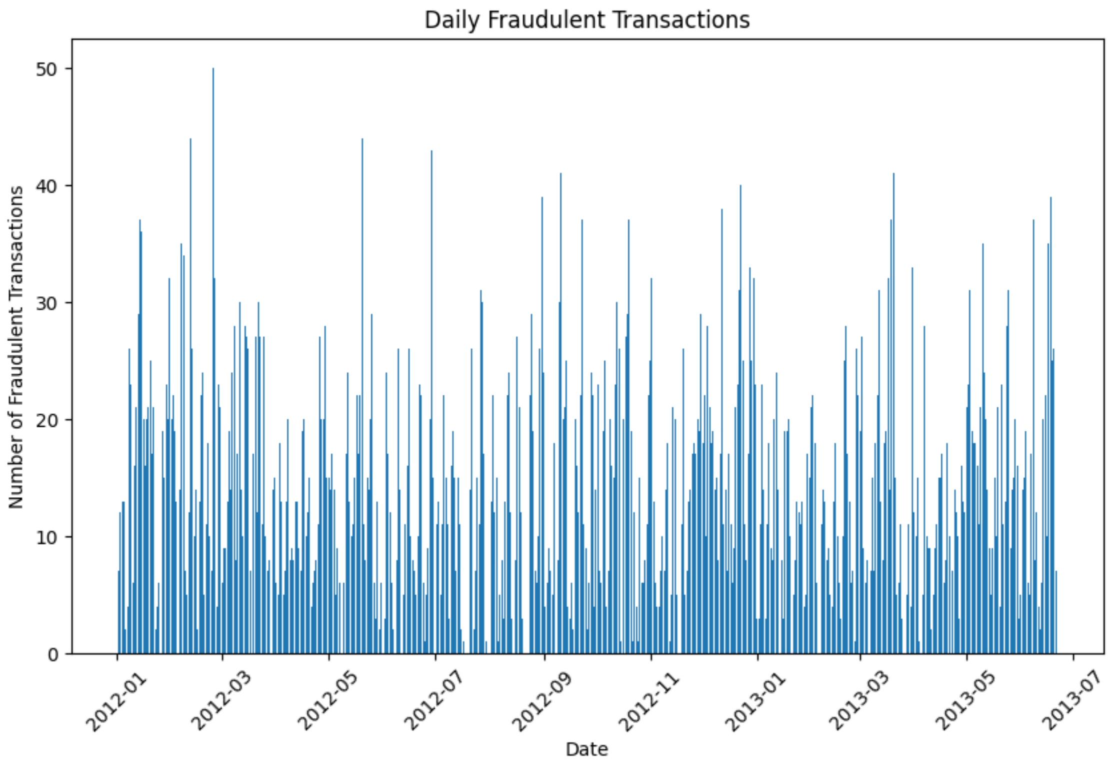
1. Subset of Fraudulent Transactions: A new dataframe df_fraud is created, containing only the rows where is_fraud equals.
2. Total Fraud Transactions: The total number of fraudulent transactions is displayed using df['is_fraud'].value_counts(), which shows the count of both fraudulent and non-fraudulent transactions.
3. If we seperate the dataset in genuine and fraudulent transaction classes, then we see that the two classes are highly imbalanced. As the dataset is skewed, so we did oversampling over Minority Data.

2.3 Bi-variate Analysis

[ ]:
```
df_fraud['datetime'] = pd.to_datetime(df_fraud['unix_time'], unit='s')
```

[ ]:
```
# Check the frequency of fraud transactions per day (using binning per day)
df_fraud['date_bin'] = df_fraud['datetime'].dt.floor('D')
```

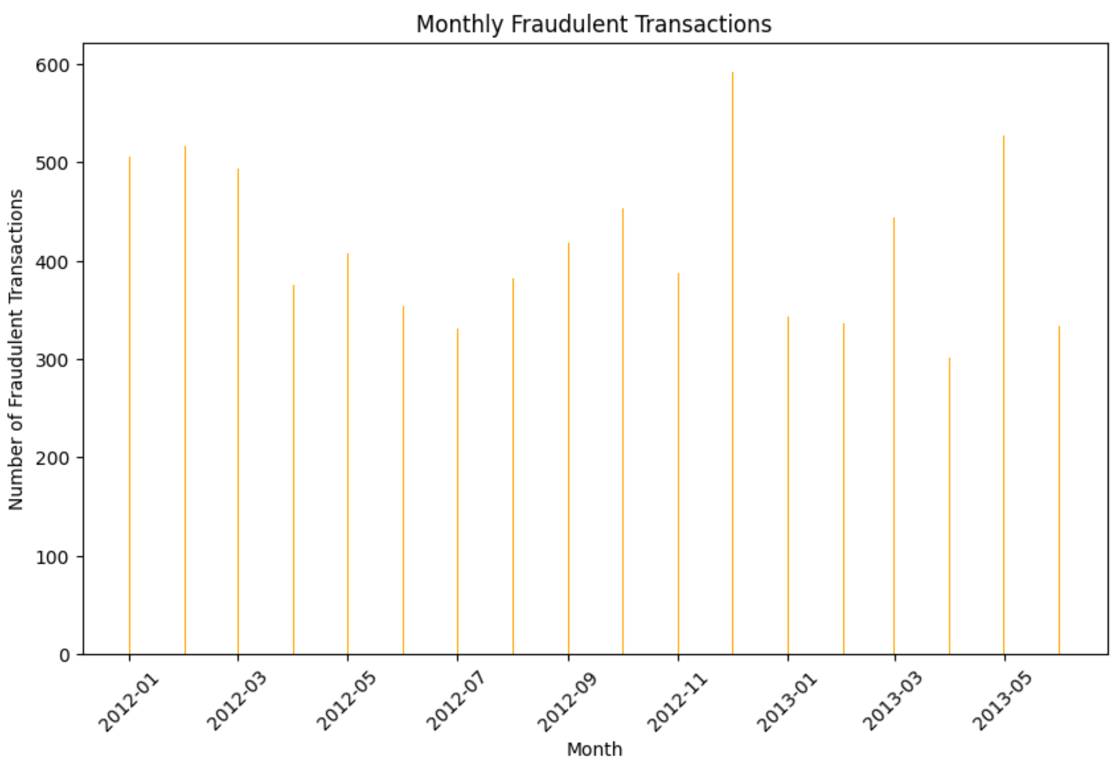
```
fraud_counts_by_day = df_fraud.groupby('date_bin').size().
↪reset_index(name='fraud_count_by_day')

plt.figure(figsize=(10, 6))
plt.bar(fraud_counts_by_day['date_bin'],␣
↪fraud_counts_by_day['fraud_count_by_day'])
plt.xlabel('Date')
plt.ylabel('Number of Fraudulent Transactions')
plt.title('Daily Fraudulent Transactions')
plt.xticks(rotation=45)
plt.show()
```



Observations

1. Daily Variability: The number of fraudulent transactions per day varies significantly, ranging from 0 to nearly 50 transactions. The highest peak is around 50 transactions, with multiple days showing counts above 40.
2. Noticeable Spikes: There are specific days where the number of fraudulent transactions spikes dramatically, such as on January 15th and February 20th, where the counts reach their highest levels. These spikes suggest occasional surges in fraudulent activity that deviate from the more consistent daily patterns.

[ ]:
```python
# Check the frequency of fraud transactions per month (using binning per month)
df_fraud['month_bin'] = df_fraud['datetime'].dt.to_period('M')
fraud_counts_by_month = df_fraud.groupby('month_bin').size().
    ↪reset_index(name='fraud_count')

# Convert 'month_bin' to datetime for plotting
fraud_counts_by_month['month_bin'] = fraud_counts_by_month['month_bin'].dt.
    ↪to_timestamp()

# Plotting the results
plt.figure(figsize=(10, 6))
plt.bar(fraud_counts_by_month['month_bin'],␣
    ↪fraud_counts_by_month['fraud_count'], color='orange')
plt.xlabel('Month')
plt.ylabel('Number of Fraudulent Transactions')
plt.title('Monthly Fraudulent Transactions')
plt.xticks(rotation=45)
plt.show()
```



Observations

1. Monthly Variability: The number of fraudulent transactions per month varies significantly, ranging from around 150 to approximately 550 transactions. The highest number of fraudulent

transactions occurred in November 2012, with a peak of approximately 550 transactions.

2. Noticeable Trends: There are noticeable peaks in fraudulent transactions in January 2012, March 2012, and November 2012, while significant drops are observed in May 2012, July 2012, and May 2013. This indicates fluctuating fraudulent activity over the observed period.

[ ]:
```
plt.figure(figsize=(10,4))
ft_by_gender = df_fraud.groupby('gender')['is_fraud'].sum()
sns.barplot(x=ft_by_gender.index, y=ft_by_gender.values, palette='viridis',␣
 ↪hue=ft_by_gender.values)
plt.xlabel('Gender')
plt.ylabel('Fraud Transactions')
plt.title('Fraud Transactions by Gender')

plt.show()
```

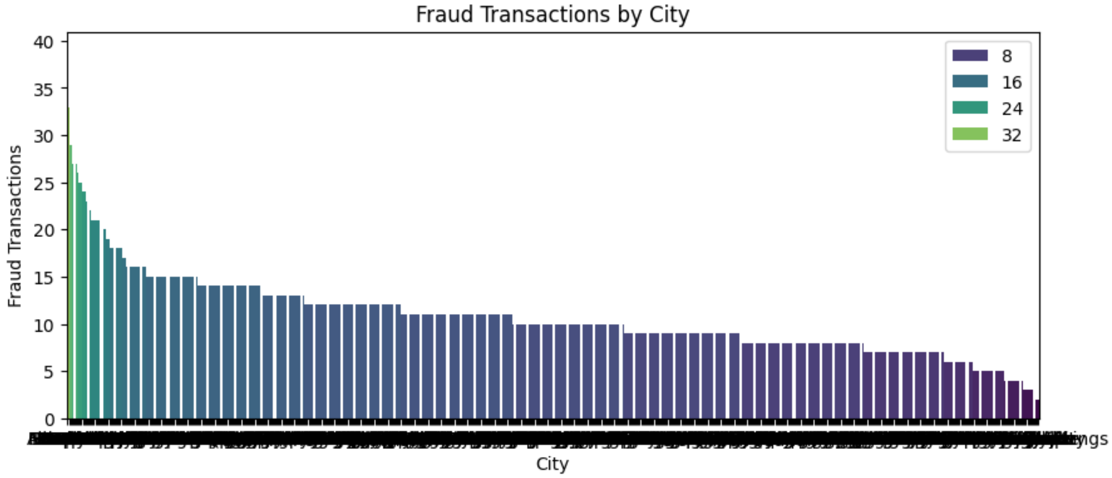
Observations

1. Fraud Transactions by Gender: The bar graph shows that females (F) have 3,735 fraud transactions, while males (M) have 3,771 fraud transactions.

2. Gender Comparison: Males have slightly more fraud transactions than females, with a difference of 36 transactions.

[ ]:
```
plt.figure(figsize=(10,4))
ft_by_city = df_fraud.groupby('city')['is_fraud'].sum()
sns.barplot(x=ft_by_city.index, y=ft_by_city.values,
            palette='viridis', hue=ft_by_city.values,
            order=ft_by_city.sort_values(ascending=False).index)
plt.xlabel('City')
plt.ylabel('Fraud Transactions')
plt.title('Fraud Transactions by City')
```

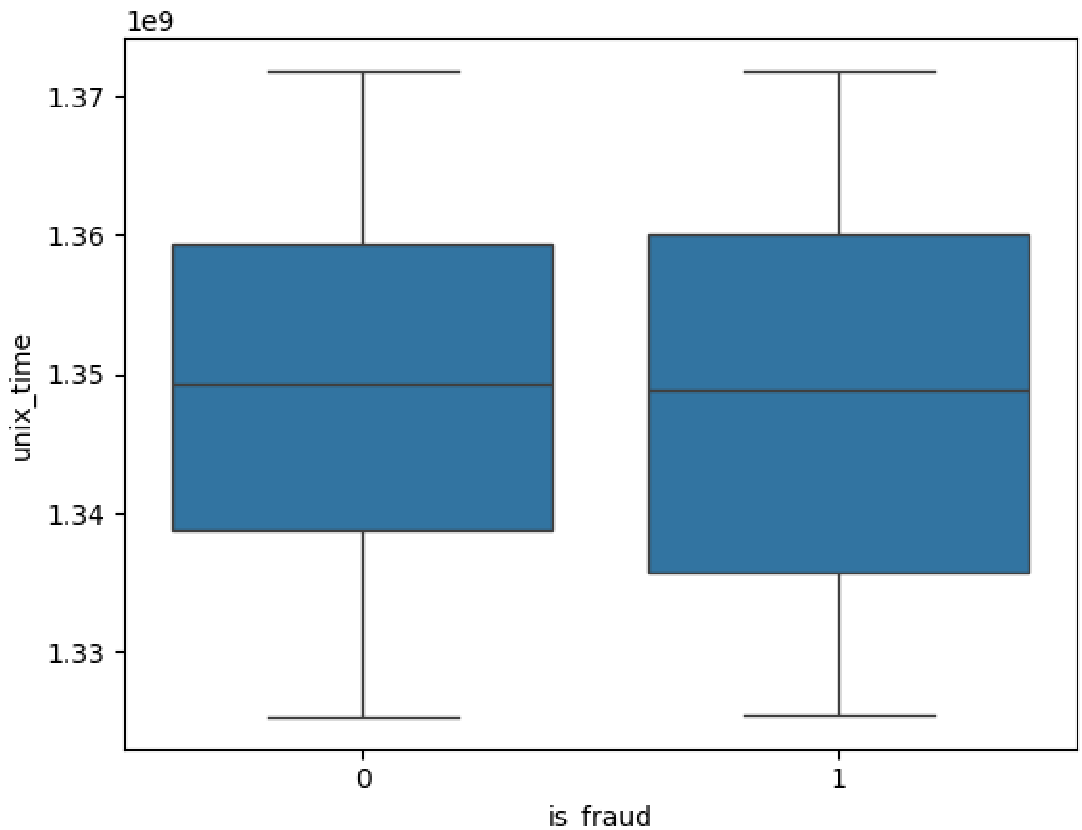
```
plt.show()
```



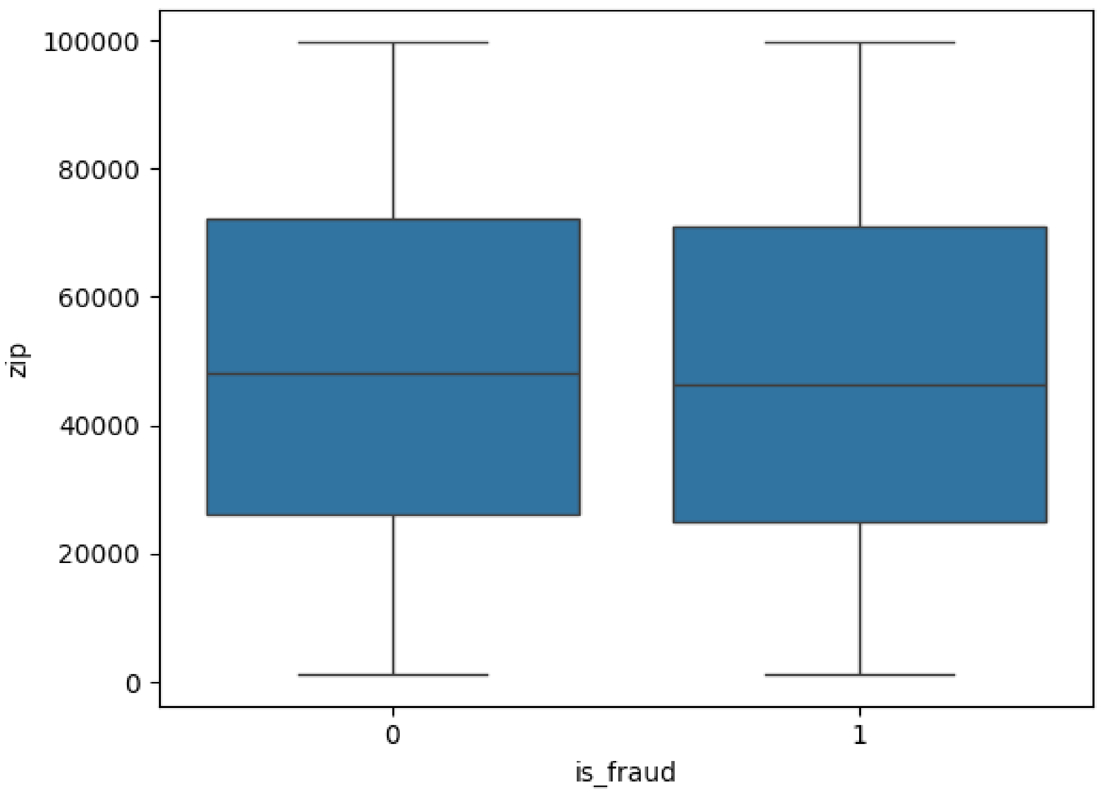
Observations

1. City with Highest Fraud Transactions: The city with the highest number of fraud transactions has slightly above 30 transactions.
2. Range of Fraud Transactions: The number of fraud transactions across various cities ranges from around 1 to slightly above 30, with a clear descending trend from the city with the highest fraud transactions to the city with the lowest.

[ ]:
```
sns.boxplot(data=df, x='is_fraud', y='unix_time')
plt.show()
```



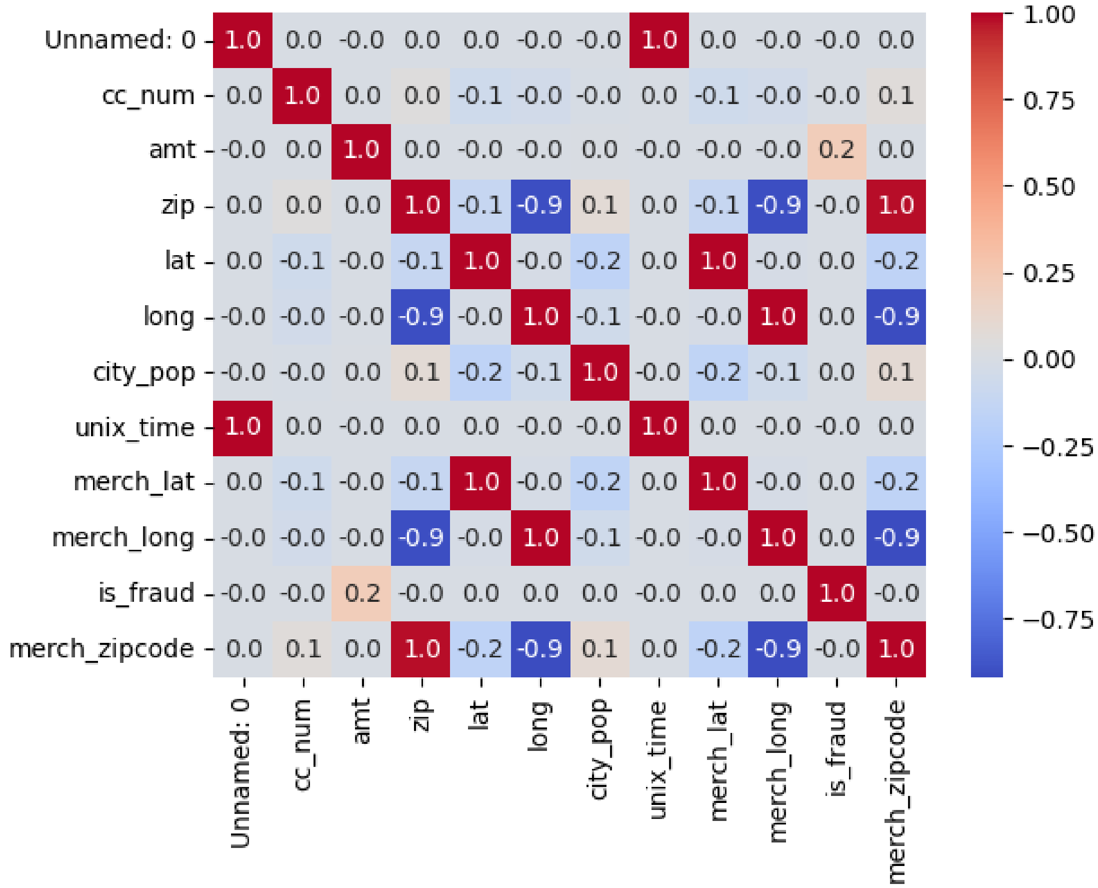
[ ]:
```
sns.boxplot(data=df, x='is_fraud', y='zip')
plt.show()
```



Observations

1. unix_time Box Plot: The median unix_time for both non-fraudulent (0) and fraudulent (1) transactions is approximately 1.35e9. The interquartile range (IQR) for both categories is similar, with the lower quartile around 1.34e9 and the upper quartile around 1.36e9. The whiskers extend from approximately 1.33e9 to 1.37e9 for both categories.
2. zip Box Plot: The median zip for both non-fraudulent (0) and fraudulent (1) transactions is around 50,000. The IQR for both categories is similar, with the lower quartile around 25,000 and the upper quartile around 75,000. The whiskers extend from approximately 0 to 100,000 for both categories.

[ ]:
```
# Check the correlations between attributes
sns.heatmap(df.corr(numeric_only=True), annot=True, fmt='.1f', cmap='coolwarm')
plt.show()
```



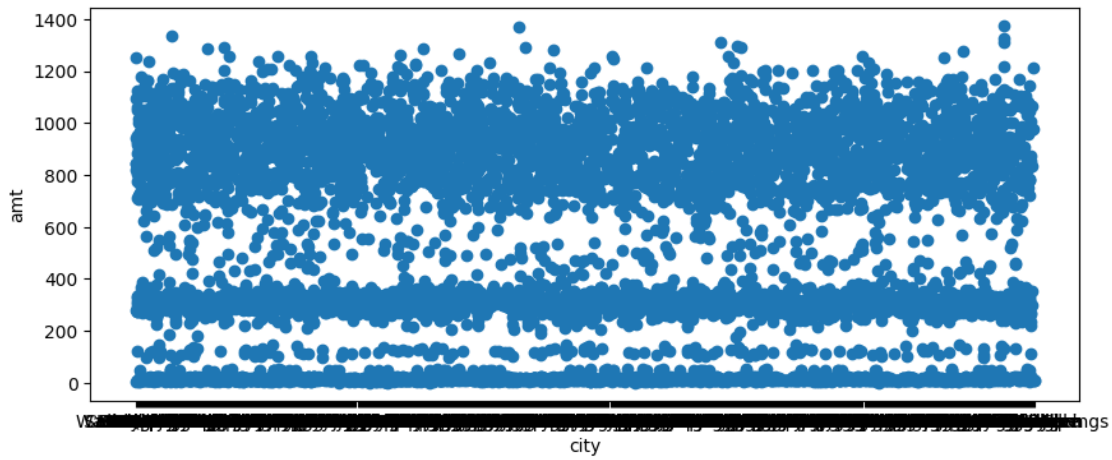
Observations

1. Perfect Correlations: The zip and merch_zipcode variables have a perfect positive correlation of 1.0, indicating they move together perfectly. Similarly, the long and merch_long variables have a perfect negative correlation of -1.0, indicating they move in exactly opposite directions.

2. Strong Correlations: Other notable correlations include a strong positive correlation between lat and merch_lat (0.9), and a strong negative correlation between long and merch_long (-0.9). These correlations suggest a strong relationship between the geographical coordinates of the cardholder and the merchant.

2.4 Multivariate Analysis

[ ]:
```
# Let's check the amount of fraud transaction per city
plt.figure(figsize=(10,4))
plt.scatter(df_fraud['city'], df_fraud['amt'])
plt.xlabel('city')
plt.ylabel('amt')
plt.show()
```



Observations

1. Transaction Amounts: The scatter plot shows that the transaction amounts (amt) for fraudulent transactions range from 0 dollars to approximately 1,400 dollars. The data points are densely packed, indicating a high volume of transactions across various cities.
2. Distribution Across Cities: The amt values are distributed across a wide range of cities, with no clear pattern or trend visible. The data points are scattered uniformly, suggesting that the amount of fraudulent transactions does not vary significantly between different cities.

3 Data Pre-processing

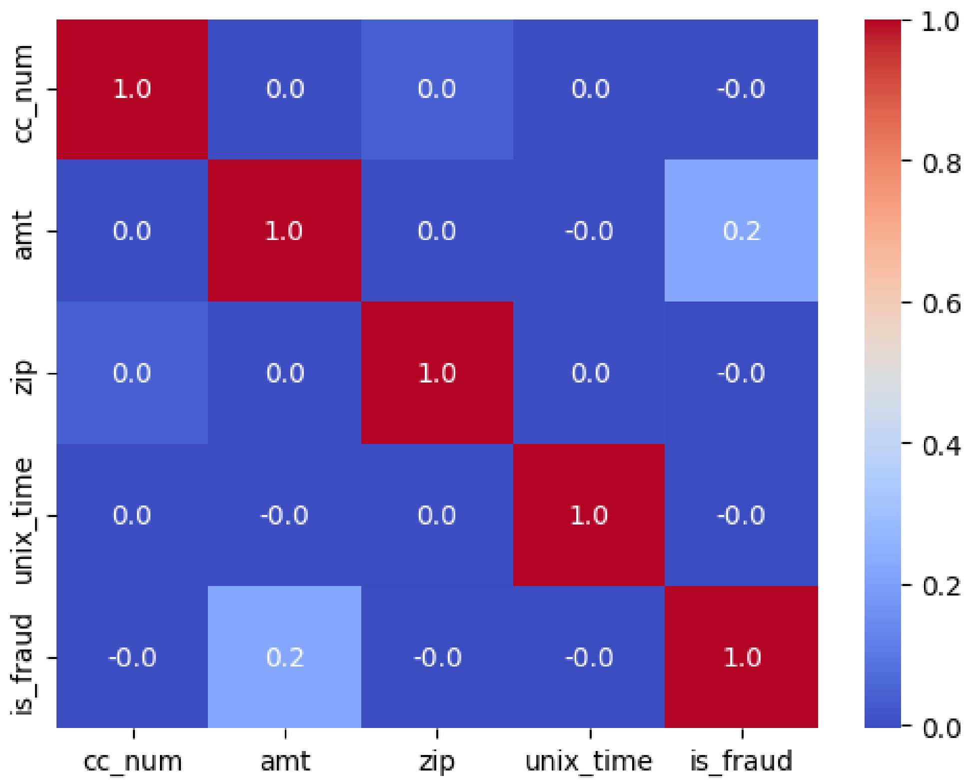
[ ]:
```
'''
Drop the columns
 1) which are not useful for fraud detection or
 2) have high correlations
'''
df.drop("Unnamed: 0", inplace=True, axis=1)
df.drop("gender", inplace=True, axis=1)
df.drop("first", inplace=True, axis=1)
df.drop("last", inplace=True, axis=1)
df.drop("lat", inplace=True, axis=1) # Almost same as merch_lat
df.drop("long", inplace=True, axis=1) # Almost same as merch_long
df.drop("dob", inplace=True, axis=1)
df.drop("job", inplace=True, axis=1)
df.drop("merch_zipcode", inplace=True, axis=1) # Because we have another zip␣
↪code also
df.drop("merch_lat", inplace=True, axis=1)
df.drop("merch_long", inplace=True, axis=1)
df.drop("street", inplace=True, axis=1)
df.drop("city_pop", inplace=True, axis=1)
```

```
df.drop("trans_date_trans_time", inplace=True, axis=1) # Becuase we have␣
↪unix_time
df.drop("trans_num", inplace=True, axis=1)
df.drop('city', inplace=True, axis=1)
df.drop('state', inplace=True, axis=1)

print(df.columns)
df.dtypes
```

```
Index(['cc_num', 'merchant', 'category', 'amt', 'zip', 'unix_time',
       'is_fraud'],
      dtype='object')
```

[ ]:
```
cc_num          int64
merchant       object
category       object
amt           float64
zip             int64
unix_time       int64
is_fraud        int64
dtype: object
```

[ ]:
```
# Check the correlations again among attributes
sns.heatmap(df.corr(numeric_only=True), annot=True, fmt='.1f', cmap='coolwarm')
plt.show()
```



Observations

1. Moderate Positive Correlation: The attribute amt (transaction amount) has a moderate positive correlation of 0.2 with the target variable is_fraud, indicating that higher transaction amounts might be somewhat associated with fraudulent transactions.

2. Low Correlations: The attributes cc_num, zip, and unix_time have very low or negligible correlations with is_fraud (correlation values close to 0), suggesting that these attributes might not be very useful for predicting fraud.

[ ]:
```python
# Use onehot encoding for the categorical data (merchant, category)
df = pd.get_dummies(df, columns=['merchant', 'category'])

df.head()
```

[ ]:
```
             cc_num     amt    zip   unix_time  is_fraud  \
0  2703186189652095    4.97  28654  1325376018         0
1      630423337322  107.23  99160  1325376044         0
2    38859492057661  220.11  83252  1325376051         0
3  3534093764340240   45.00  59632  1325376076         0
4   375534208663984   41.96  24433  1325376186         0

   merchant_fraud_Abbott-Rogahn  merchant_fraud_Abbott-Steuber  \
0                             0                              0
1                             0                              0
2                             0                              0
3                             0                              0
4                             0                              0

   merchant_fraud_Abernathy and Sons  merchant_fraud_Abshire PLC  \
0                                  0                           0
1                                  0                           0
2                                  0                           0
3                                  0                           0
4                                  0                           0

   merchant_fraud_Adams, Kovacek and Kuhlman  …  category_grocery_pos  \
0                                          0  …                     0
1                                          0  …                     1
2                                          0  …                     0
3                                          0  …                     0
4                                          0  …                     0

   category_health_fitness  category_home  category_kids_pets  \
0                        0              0                   0
1                        0              0                   0
```

```
2                         0             0                   0
3                         0             0                   0
4                         0             0                   0

   category_misc_net  category_misc_pos  category_personal_care  \
0                  1                  0                       0
1                  0                  0                       0
2                  0                  0                       0
3                  0                  0                       0
4                  0                  1                       0

   category_shopping_net  category_shopping_pos  category_travel
0                      0                      0                0
1                      0                      0                0
2                      0                      0                0
3                      0                      0                0
4                      0                      0                0

[5 rows x 712 columns]
```

[ ]:
```
# See if no categorical attributes are left
df.dtypes
```

[ ]:
```
cc_num                      int64
amt                       float64
zip                         int64
unix_time                   int64
is_fraud                    int64
                           …
category_misc_pos           uint8
category_personal_care      uint8
category_shopping_net       uint8
category_shopping_pos       uint8
category_travel             uint8
Length: 712, dtype: object
```

[ ]:
```
# Let's fix the class imbalance first using SMOTE
!pip install imbalanced-learn
from imblearn.over_sampling import SMOTE

X = df.drop('is_fraud', axis=1)
y = df['is_fraud']

smote = SMOTE()
X,y = smote.fit_resample(X,y)
```

```
Requirement already satisfied: imbalanced-learn in
/usr/local/lib/python3.10/dist-packages (0.12.4)
```

```
Requirement already satisfied: numpy>=1.17.3 in /usr/local/lib/python3.10/dist-
packages (from imbalanced-learn) (1.26.4)
Requirement already satisfied: scipy>=1.5.0 in /usr/local/lib/python3.10/dist-
packages (from imbalanced-learn) (1.13.1)
Requirement already satisfied: scikit-learn>=1.0.2 in
/usr/local/lib/python3.10/dist-packages (from imbalanced-learn) (1.5.2)
Requirement already satisfied: joblib>=1.1.1 in /usr/local/lib/python3.10/dist-
packages (from imbalanced-learn) (1.4.2)
Requirement already satisfied: threadpoolctl>=2.0.0 in
/usr/local/lib/python3.10/dist-packages (from imbalanced-learn) (3.5.0)
```

[ ]:
```python
# Split the data for train and test

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    stratify=y, random_state=42)
print("Number of fraud transactions in the training datset:\n", y_train.
    value_counts())
print("\nNumber of fraud transactions in the testing dataset:\n", y_test.
    value_counts())
X.head()
```

```
Number of fraud transactions in the training datset:
 0    1031335
1    1031335
Name: is_fraud, dtype: int64

Number of fraud transactions in the testing dataset:
 1    257834
0    257834
Name: is_fraud, dtype: int64
```

[ ]:
```
             cc_num     amt    zip   unix_time  merchant_fraud_Abbott-Rogahn  \
0  2703186189652095    4.97  28654  1325376018                             0
1      630423337322  107.23  99160  1325376044                             0
2    38859492057661  220.11  83252  1325376051                             0
3  3534093764340240   45.00  59632  1325376076                             0
4   375534208663984   41.96  24433  1325376186                             0

   merchant_fraud_Abbott-Steuber  merchant_fraud_Abernathy and Sons  \
0                              0                                  0
1                              0                                  0
2                              0                                  0
3                              0                                  0
4                              0                                  0

   merchant_fraud_Abshire PLC  merchant_fraud_Adams, Kovacek and Kuhlman  \
0                           0                                          0
1                           0                                          0
```

```
2                           0                                          0
3                           0                                          0
4                           0                                          0

   merchant_fraud_Adams-Barrows  …  category_grocery_pos  \
0                             0  …                     0
1                             0  …                     1
2                             0  …                     0
3                             0  …                     0
4                             0  …                     0

   category_health_fitness  category_home  category_kids_pets  \
0                        0              0                   0
1                        0              0                   0
2                        0              0                   0
3                        0              0                   0
4                        0              0                   0

   category_misc_net  category_misc_pos  category_personal_care  \
0                  1                  0                       0
1                  0                  0                       0
2                  0                  0                       0
3                  0                  0                       0
4                  0                  1                       0

   category_shopping_net  category_shopping_pos  category_travel
0                      0                      0                0
1                      0                      0                0
2                      0                      0                0
3                      0                      0                0
4                      0                      0                0

[5 rows x 711 columns]
```

We can see that now both the classes are balanced.

4 Model Development and Validation

4.1 Logistic Regression

[ ]:
```python
from sklearn.linear_model import LogisticRegression

lr_model = LogisticRegression(random_state=0)
lr_model.fit(X_train, y_train)

lr_y_pred = lr_model.predict(X_test)
```

```
[ ]: print("===== Logistic Regression ======")
print("Accuracy:", accuracy_score(y_test, lr_y_pred))
print("Precision:", precision_score(y_test, lr_y_pred, average='weighted',␣
 ↪zero_division=1))
print("Recall:", recall_score(y_test, lr_y_pred, average='weighted'))
print("F1 Score:", f1_score(y_test, lr_y_pred, average='weighted'))

print("\nConfusion Matrix:")
print(confusion_matrix(y_test, lr_y_pred))

print("\nClassification Report:")
print(classification_report(y_test, lr_y_pred, zero_division=0))
```

```
===== Logistic Regression ======
Accuracy: 0.5
Precision: 0.75
Recall: 0.5
F1 Score: 0.3333333333333333

Confusion Matrix:
[[257834      0]
 [257834      0]]

Classification Report:
              precision    recall  f1-score   support

           0       0.50      1.00      0.67    257834
           1       0.00      0.00      0.00    257834

    accuracy                           0.50    515668
   macro avg       0.25      0.50      0.33    515668
weighted avg       0.25      0.50      0.33    515668
```

4.2 Random Forest Classifier

Increasing the n_estimators increses the validation scores, but that causes more execution time. So, to balance it, we are keeping it to 20

```
[ ]: from sklearn.ensemble import RandomForestClassifier

rf_model = RandomForestClassifier(n_estimators=20, min_samples_leaf=4,␣
 ↪max_depth=20, min_samples_split=4, random_state=0)
rf_model.fit(X_train, y_train)

rf_y_pred = rf_model.predict(X_test)
```

```
[ ]: print("===== RandomForest Classifier ======")
     print(f"Accuracy: {accuracy_score(y_test, rf_y_pred):.3f}")
     print(f"Precision: {precision_score(y_test, rf_y_pred, average='weighted',␣
      ↪zero_division=0):.3f}")
     print(f"Recall: {recall_score(y_test, rf_y_pred, average='weighted'):.3f}")
     print(f"F1 Score: {f1_score(y_test, rf_y_pred, average='weighted'):.3f}")

     print("\nConfusion Matrix:")
     print(confusion_matrix(y_test, rf_y_pred))

     print("\nClassification Report:")
     print(classification_report(y_test, rf_y_pred, zero_division=0))
```

```
===== RandomForest Classifier ======
Accuracy: 0.911
Precision: 0.917
Recall: 0.911
F1 Score: 0.910

Confusion Matrix:
[[251228   6606]
 [ 39434 218400]]

Classification Report:
              precision    recall  f1-score   support

           0       0.86      0.97      0.92    257834
           1       0.97      0.85      0.90    257834

    accuracy                           0.91    515668
   macro avg       0.92      0.91      0.91    515668
weighted avg       0.92      0.91      0.91    515668
```

The Type-I and Type-II error with RandomForestClassifier **is more** with n_estimators=20.

Also, the Precision, Recall, and f-1 scores are okay but not that good.

4.3 Random Forest Classifier with PCA

```
[ ]: from sklearn.preprocessing import StandardScaler
     from sklearn.decomposition import PCA
     from sklearn.ensemble import RandomForestClassifier

     '''
     Standardize the data becuase PCA is sensitive to the scale of features.
     Features should be on the same scale (mean of 0 and standard deviation of 1).

     X is alraedy after dropping some of the attributes and one-hot encoding on
```

```
two attributes becuase, categorical data were many and one-hot encoding on all
of them would have resulted in far too many columns.
'''
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

pca = PCA(n_components=0.95) # 95% of variance
X_pca = pca.fit_transform(X_scaled)
print("Number of components selected:", pca.n_components_)

X_train_pca, X_test_pca, y_train_pca, y_test_pca = train_test_split(X_pca, y,␣
↪test_size=0.2, random_state=1)

# Use RandomForest as ensemble method with PCA
rf_model_pca = RandomForestClassifier(n_estimators=20, min_samples_leaf=4,␣
↪max_depth=20, min_samples_split=4, random_state=0)
rf_model_pca.fit(X_train_pca, y_train_pca)

rf_y_pred_pca = rf_model_pca.predict(X_test_pca)
```

```
Number of components selected: 662
```

[ ]:
```
print("===== RandomForest Classifier with PCA ======")
print(f"Accuracy: {accuracy_score(y_test_pca, rf_y_pred_pca):.3f}")
print(f"Precision: {precision_score(y_test_pca, rf_y_pred_pca,␣
↪average='weighted', zero_division=0):.3f}")
print(f"Recall: {recall_score(y_test_pca, rf_y_pred_pca, average='weighted'):.
↪3f}")
print(f"F1 Score: {f1_score(y_test_pca, rf_y_pred_pca, average='weighted'):.
↪3f}")

print("\nConfusion Matrix:")
print(confusion_matrix(y_test_pca, rf_y_pred_pca))

print("\nClassification Report:")
print(classification_report(y_test_pca, rf_y_pred_pca, zero_division=0))
```

```
===== RandomForest Classifier with PCA ======
Accuracy: 0.992
Precision: 0.992
Recall: 0.992
F1 Score: 0.992

Confusion Matrix:
[[256652    545]
 [  3818 254653]]

Classification Report:
```

```
              precision    recall  f1-score   support

           0       0.99      1.00      0.99    257197
           1       1.00      0.99      0.99    258471

    accuracy                           0.99    515668
   macro avg       0.99      0.99      0.99    515668
weighted avg       0.99      0.99      0.99    515668
```

We can see that with same n_estimators=20 and max_depth=20, the RandomForestClassifier with PCA is giving better result compared to standalone RandomForestClassifier. - Type-I error is less - Type-II error is very less

4.4 Decision Tree Classifier

Increasing the max_depth increases the scores, so as execution time. With max_depth=20, the scores are already closed to 99%

[ ]:
```
from sklearn.tree import DecisionTreeClassifier

dt_model = DecisionTreeClassifier(max_depth=20, min_samples_split=4,␣
 ↪random_state=0)
dt_model.fit(X_train, y_train)

dc_y_pred = dt_model.predict(X_test)
```

[ ]:
```
print("===== DecisionTree Classifier ======")
print(f"Accuracy: {accuracy_score(y_test, dc_y_pred):.3f}")
print(f"Precision: {precision_score(y_test, dc_y_pred, average='weighted',␣
 ↪zero_division=0):.3f}")
print(f"Recall: {recall_score(y_test, dc_y_pred, average='weighted'):.3f}")
print(f"F1 Score: {f1_score(y_test, dc_y_pred, average='weighted'):.3f}")

print("\nConfusion Matrix:")
print(confusion_matrix(y_test, dc_y_pred))

print("\nClassification Report:")
print(classification_report(y_test, dc_y_pred, zero_division=0))
```

```
===== DecisionTree Classifier ======
Accuracy: 0.993
Precision: 0.993
Recall: 0.993
F1 Score: 0.993

Confusion Matrix:
[[255954   1880]
 [  1646 256188]]
```

```
Classification Report:
              precision    recall  f1-score   support

           0       0.99      0.99      0.99    257834
           1       0.99      0.99      0.99    257834

    accuracy                           0.99    515668
   macro avg       0.99      0.99      0.99    515668
weighted avg       0.99      0.99      0.99    515668
```

The Type-I and Type-II error with DecisonTreeClassifier is **very less** (~ 1%). - Type-I error (FP) is **less** than RandomForestClassifier with PCA - Type-II error (FN) is **more** than RandomForestClassifier with PCA

Also, the Precision, Recall, and f-1 **scores are very good**.

4.5 Gradient Boost Classifier

[ ]:
```
from sklearn.ensemble import GradientBoostingClassifier

gb_model = GradientBoostingClassifier(n_estimators=20, learning_rate=0.1,␣
 ↪max_depth=4, random_state=0)
gb_model.fit(X_train, y_train)

gb_y_pred = gb_model.predict(X_test)
```

[ ]:
```
print("===== GradientBoost Classifier ======")
print(f"Accuracy: {accuracy_score(y_test, gb_y_pred):.3f}")
print(f"Precision: {precision_score(y_test, gb_y_pred, average='weighted',␣
 ↪zero_division=0):.3f}")
print(f"Recall: {recall_score(y_test, gb_y_pred, average='weighted'):.3f}")
print(f"F1 Score: {f1_score(y_test, gb_y_pred, average='weighted'):.3f}")

print("\nConfusion Matrix:")
print(confusion_matrix(y_test, gb_y_pred))

print("\nClassification Report:")
print(classification_report(y_test, gb_y_pred, zero_division=0))
```

```
===== GradientBoost Classifier ======
Accuracy: 0.904
Precision: 0.911
Recall: 0.904
F1 Score: 0.903

Confusion Matrix:
[[250310   7524]
```

```
 [ 42216 215618]]

Classification Report:
              precision    recall  f1-score   support

           0       0.86      0.97      0.91    257834
           1       0.97      0.84      0.90    257834

    accuracy                           0.90    515668
   macro avg       0.91      0.90      0.90    515668
weighted avg       0.91      0.90      0.90    515668
```

4.6 k-fold Cross Validation with Random Forest Classification Model

[ ]:
```
from sklearn.model_selection import KFold, cross_val_score

k = 5 # Number of folds
kf = KFold(n_splits=k, shuffle=True, random_state=42)

scores = cross_val_score(rf_model, X, y, cv=kf, scoring='accuracy')
print(f'Cross Validation Accuracy: {scores.mean():.3f} (+/-{scores.std():0.
 ↪3f})')
```

```
Cross Validation Accuracy: 0.896 (+/-0.006)
```

4.6.1 Model Performance Summary

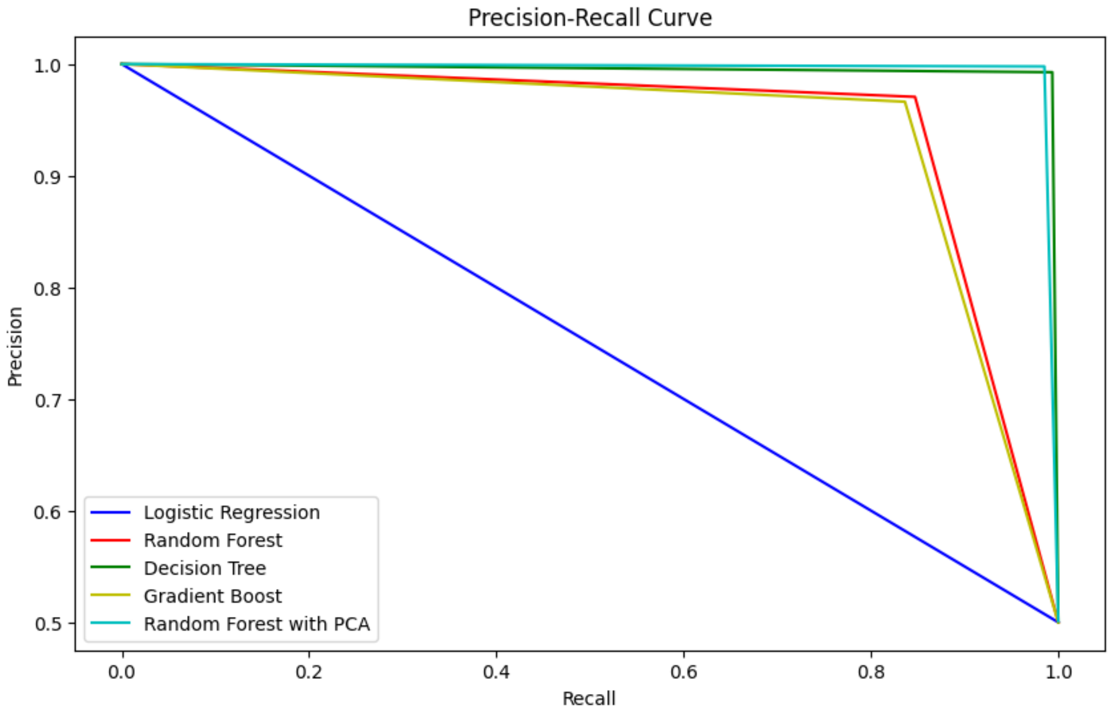
Model	Accuracy	Precision	Recall	F1 Score
Logistic Regression	0.50	0.75	0.50	0.33
RandomForest Classifier	0.911	0.917	0.911	0.910
RandomForest Classifier with PCA	0.992	0.992	0.992	0.992
DecisionTree Classifier	0.993	0.993	0.993	0.993
GradientBoost Classifier	0.904	0.911	0.904	0.903

5 Plotting graphs

[ ]:
```
# Plot Precision-Recall Curve
lr_precision, lr_recall, _ = precision_recall_curve(y_test, lr_y_pred)
rf_precision, rf_recall, _ = precision_recall_curve(y_test, rf_y_pred)
dc_precision, dc_recall, _ = precision_recall_curve(y_test, dc_y_pred)
gb_precision, gb_recall, _ = precision_recall_curve(y_test, gb_y_pred)
rf_precision_pca, rf_recall_pca, _ = precision_recall_curve(y_test_pca,␣
 ↪rf_y_pred_pca)

plt.figure(figsize=(10,6))
```

```python
plt.plot(lr_recall, lr_precision, color = 'b', label='Logistic Regression')
plt.plot(rf_recall, rf_precision, color = 'r', label='Random Forest')
plt.plot(dc_recall, dc_precision, color = 'g', label='Decision Tree')
plt.plot(gb_recall, gb_precision, color = 'y', label='Gradient Boost')
plt.plot(rf_recall_pca, rf_precision_pca, color = 'c', label='Random Forest␣
↪with PCA')

plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve')
plt.legend()
```

[ ]: <matplotlib.legend.Legend at 0x78a653e84a00>

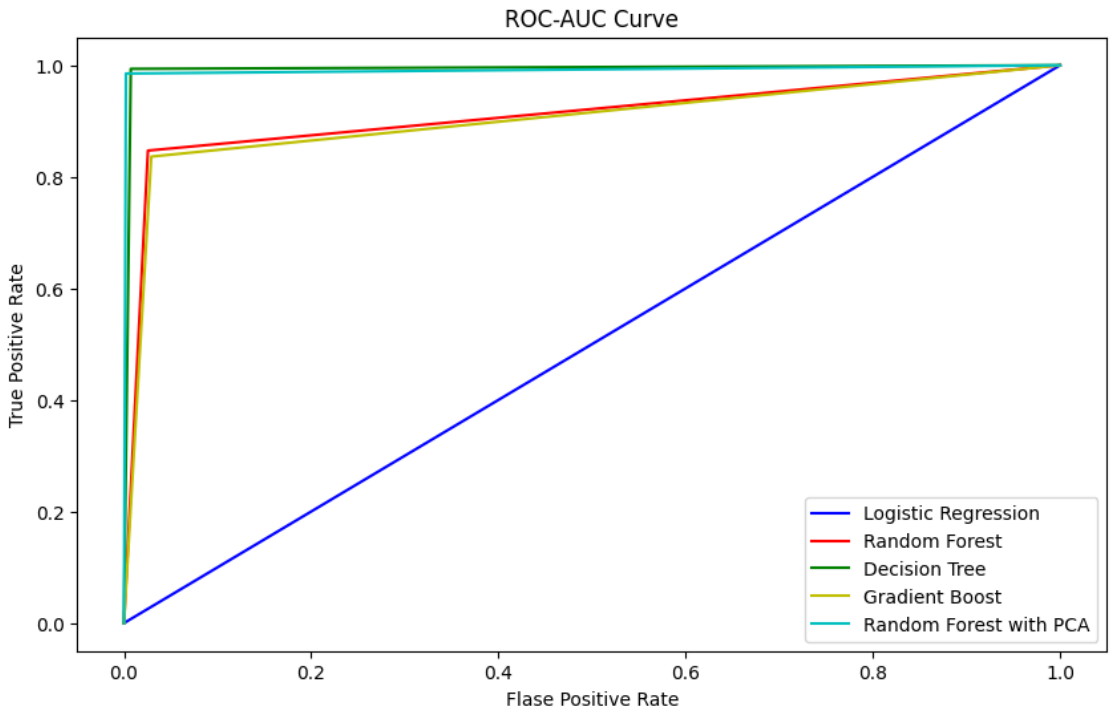


[ ]:
```python
# Plot the ROC-AUC Curve
from sklearn.metrics import roc_auc_score, roc_curve
lr_roc_auc = roc_auc_score(y_test, lr_y_pred)
rf_roc_auc = roc_auc_score(y_test, rf_y_pred)
dc_roc_auc = roc_auc_score(y_test, dc_y_pred)
gb_roc_auc = roc_auc_score(y_test, gb_y_pred)
rf_roc_auc_pca = roc_auc_score(y_test_pca, rf_y_pred_pca)

print("LogisticRegression ROC-AUC Score:", lr_roc_auc)
print("RandomForest ROC-AUC Score:", rf_roc_auc)
```

```
print("DecisionTree ROC-AUC Score:", dc_roc_auc)
print("GradientBoost ROC-AUC Score:", gb_roc_auc)
print("RandomForest ROC-AUC Score:", rf_roc_auc_pca)

lr_fpr, lr_tpr, lr_thresholds = roc_curve(y_test, lr_y_pred)
rf_fpr, rf_tpr, rf_thresholds = roc_curve(y_test, rf_y_pred)
dc_fpr, dc_tpr, dc_thresholds = roc_curve(y_test, dc_y_pred)
gb_fpr, gb_tpr, gb_thresholds = roc_curve(y_test, gb_y_pred)
rf_fpr_pca, rf_tpr_pca, rf_thresholds_pca = roc_curve(y_test_pca, rf_y_pred_pca)

plt.figure(figsize=(10,6))
plt.plot(lr_fpr, lr_tpr, color='b', label='Logistic Regression')
plt.plot(rf_fpr, rf_tpr, color='r', label='Random Forest')
plt.plot(dc_fpr, dc_tpr, color='g', label='Decision Tree')
plt.plot(gb_fpr, gb_tpr, color='y', label='Gradient Boost')
plt.plot(rf_fpr_pca, rf_tpr_pca, color='c', label='Random Forest with PCA')

plt.xlabel('Flase Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC-AUC Curve')
plt.legend()
```

```
LogisticRegression ROC-AUC Score: 0.5
RandomForest ROC-AUC Score: 0.9107177486289627
DecisionTree ROC-AUC Score: 0.9931622671951721
GradientBoost ROC-AUC Score: 0.9035425894179977
RandomForest ROC-AUC Score: 0.991554759327341
```

```
[ ]: <matplotlib.legend.Legend at 0x78a65c120610>
```



Observations

1. Highest Performance: The DecisionTree Classifier has the highest ROC-AUC score of 0.993, indicating excellent classification ability. The RandomForest Classifier with PCA also performs exceptionally well with a ROC-AUC score of 0.992.

2. Lowest Performance: The Logistic Regression model has the lowest ROC-AUC score of 0.5, indicating it performs no better than random guessing. The other models (RandomForest, GradientBoost) have significantly higher ROC-AUC scores, demonstrating better classification performance.

6 Conclusion

1. The **Logistic Regression** model has an accuracy of 0.50, indicating it performs no better than random guessing. Its precision is 0.75, but the recall is only 0.50, resulting in a low F1 score of 0.33.
2. The **RandomForest Classifier** shows a significant improvement with an accuracy of 0.911. It has a precision of 0.917 and a recall of 0.911, leading to a high F1 score of 0.910.
3. The **RandomForest Classifier with PCA** achieves an impressive accuracy of 0.992. Both its precision and recall are 0.992, resulting in an excellent F1 score of 0.992.
4. The **DecisionTree Classifier** performs exceptionally well with an accuracy of 0.993. It has a precision and recall of 0.993, leading to a near-perfect F1 score of 0.993.
5. The **GradientBoost Classifier** also performs well with an accuracy of 0.904. It has a precision of 0.911 and a recall of 0.904, resulting in a solid F1 score of 0.903.

6. The **Logistic Regression** model's performance is hindered by its inability to correctly classify fraudulent transactions, as indicated by its low recall.
7. The **RandomForest Classifier** balances precision and recall well, making it a reliable model for fraud detection.
8. The **RandomForest Classifier with PCA** and **DecisionTree Classifier** both demonstrate excellent performance, with very high accuracy, precision, recall, and F1 scores.
9. The **GradientBoost Classifier** provides a good balance between precision and recall, making it a strong contender for fraud detection tasks.
10. Overall, the **DecisionTree Classifier** and **RandomForest Classifier with PCA** are the top-performing models, offering the best combination of accuracy, precision, recall, and F1 score.