Fraud Detection on Transactions

In [1]:
```
# import functions
import sys
import os
sys.path.append('../src')

from preprocessing import count_unique_numbers, generalize_categories
from model import *
```

In [2]:
```
# Import General libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Import libraries for preprocessing
from sklearn.model_selection import train_test_split
from sklearn.compose import make_column_transformer
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.pipeline import make_pipeline

# Import libraries for model building
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassif
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_auc_score, f1_score
from sklearn.model_selection import RandomizedSearchCV
```

Summary

Through this project, we attempted to build three classification models capable of distinguishing between fraud and non-fraud transactions, as indicated on customer accounts. The models we experimented with include logistic regression, random forest classifier, and gradient boost classifier. Due to an extreme imbalance in our data, we encountered challenges in developing an effective model in Milestone 1. The results of our experimented models are listed below.

	Logistic Regression	Random Forest Classifier	Gradient Boost Classifier
Train f1 Score	0.00623	0.0783	0.872
Test f1 Score	0.00612	0.0732	0.0386

To enhance the `f1` scores, we have provided suggestions in the discussion section. Potential improvements include incorporating frequency encoding through feature engineering, adjusting scoring metrics, and experimenting with more complex models.

Introduction

In recent times, credit card fraud has emerged as one of the most prevalent forms of fraudulent activities. According to The Ascent the incidence of credit card fraud has seen a significant rise, escalating from 371,000 reports in 2017 to 1.4 million in 2021. To combat this surge in credit card fraud, corporations have turned to machine learning algorithms, employing them to automatically detect and filter fraudulent transactions. These models are trained on historical data to identify potential fraud.

This project is driven by the goal of constructing an effective fraud detection model to mitigate the occurrence of credit card frauds. The Capital One Data Science Challenge dataset is utilized for building our model, consisting of 786,363 entries of synthetically generated data. Since all the data is synthetically generated, we ensure that customer confidentiality is maintained throughout the model-building process.

Methods & Results

Read Dataset (Web)

In [3]:
```
# Imports to download dataset from
# import json
# from zipfile import ZipFile
# from io import BytesIO
# from urllib.request import urlopen
```

In [4]:
```
url = "https://github.com/CapitalOneRecruiting/DS/blob/173ca4399629f1e4e7414
```

In [5]:
```
#resp = urlopen(url)
#myzip = ZipFile(BytesIO(resp.read()))

#data_list = []

#for line in myzip.open(myzip.namelist()[0]).readlines():
#    data_list.append(json.loads(line))
#df = pd.DataFrame(data_list)
```

In [6]:
```
# Saving raw_data to data folder
# raw_df.to_pickle('data/transactions.pkl.zip', compression='zip')
```

Read Dataset (Local)

In [7]:
```
df = pd.read_pickle('../data/transactions.pkl.zip', compression="infer")
```

EDA Analysis

1. Basic Information Dataset Overview

In [8]:
```
df.head()
```

Out[8]:

	accountNumber	customerId	creditLimit	availableMoney	transactionDateTime	tran
0	737265056	737265056	5000.0	5000.0	2016-08-13T14:27:32	
1	737265056	737265056	5000.0	5000.0	2016-10-11T05:05:54	
2	737265056	737265056	5000.0	5000.0	2016-11-08T09:18:39	
3	737265056	737265056	5000.0	5000.0	2016-12-10T02:14:50	
4	830329091	830329091	5000.0	5000.0	2016-03-24T21:04:46	

5 rows × 29 columns

In [9]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 786363 entries, 0 to 786362
Data columns (total 29 columns):
 #   Column                    Non-Null Count   Dtype  
---  ------                    --------------   -----  
 0   accountNumber             786363 non-null  object 
 1   customerId                786363 non-null  object 
 2   creditLimit               786363 non-null  float64
 3   availableMoney            786363 non-null  float64
 4   transactionDateTime       786363 non-null  object 
 5   transactionAmount         786363 non-null  float64
 6   merchantName              786363 non-null  object 
 7   acqCountry                786363 non-null  object 
 8   merchantCountryCode       786363 non-null  object 
 9   posEntryMode              786363 non-null  object 
 10  posConditionCode          786363 non-null  object 
 11  merchantCategoryCode      786363 non-null  object 
 12  currentExpDate            786363 non-null  object 
 13  accountOpenDate           786363 non-null  object 
 14  dateOfLastAddressChange   786363 non-null  object 
 15  cardCVV                   786363 non-null  object 
 16  enteredCVV                786363 non-null  object 
 17  cardLast4Digits           786363 non-null  object 
 18  transactionType           786363 non-null  object 
 19  echoBuffer                786363 non-null  object 
 20  currentBalance            786363 non-null  float64
 21  merchantCity              786363 non-null  object 
 22  merchantState             786363 non-null  object 
 23  merchantZip               786363 non-null  object 
 24  cardPresent               786363 non-null  bool   
 25  posOnPremises             786363 non-null  object 
 26  recurringAuthInd          786363 non-null  object 
 27  expirationDateKeyInMatch  786363 non-null  bool   
 28  isFraud                   786363 non-null  bool   
dtypes: bool(3), float64(4), object(22)
memory usage: 158.2+ MB
```

2. Descriptive Statistics

In [10]:
```
df.describe()
```

Out[10]:

	creditLimit	availableMoney	transactionAmount	currentBalance
count	786363.000000	786363.000000	786363.000000	786363.000000
mean	10759.464459	6250.725369	136.985791	4508.739089
std	11636.174890	8880.783989	147.725569	6457.442068
min	250.000000	-1005.630000	0.000000	0.000000
25%	5000.000000	1077.420000	33.650000	689.910000
50%	7500.000000	3184.860000	87.900000	2451.760000
75%	15000.000000	7500.000000	191.480000	5291.095000
max	50000.000000	50000.000000	2011.540000	47498.810000

3. Data Cleaning and Preprocessing Missing Values

In [11]:
```
missing_values = df.isnull().sum()
print(missing_values[missing_values > 0])
missing_values
```

```
Series([], dtype: int64)
```

Out[11]:
```
accountNumber               0
customerId                  0
creditLimit                 0
availableMoney              0
transactionDateTime         0
transactionAmount           0
merchantName                0
acqCountry                  0
merchantCountryCode         0
posEntryMode                0
posConditionCode            0
merchantCategoryCode        0
currentExpDate              0
accountOpenDate             0
dateOfLastAddressChange     0
cardCVV                     0
enteredCVV                  0
cardLast4Digits             0
transactionType             0
echoBuffer                  0
currentBalance              0
merchantCity                0
merchantState               0
merchantZip                 0
cardPresent                 0
posOnPremises               0
recurringAuthInd            0
expirationDateKeyInMatch    0
isFraud                     0
dtype: int64
```

In [12]:
```
def count_empty_strings(column):
    return (column == '').sum()

# Apply this function to each column
empty_string_counts = df.apply(count_empty_strings)
print(empty_string_counts)
```

```
accountNumber                  0
customerId                     0
creditLimit                    0
availableMoney                 0
transactionDateTime            0
transactionAmount              0
merchantName                   0
acqCountry                  4562
merchantCountryCode          724
posEntryMode                4054
posConditionCode             409
merchantCategoryCode           0
currentExpDate                 0
accountOpenDate                0
dateOfLastAddressChange        0
cardCVV                        0
enteredCVV                     0
cardLast4Digits                0
transactionType              698
echoBuffer                786363
currentBalance                 0
merchantCity              786363
merchantState             786363
merchantZip               786363
cardPresent                    0
posOnPremises             786363
recurringAuthInd          786363
expirationDateKeyInMatch       0
isFraud                        0
dtype: int64
```

3.1 Drop columns with mostly empty strings(count > 70,000)

In [13]:
```
df.drop(['echoBuffer', 'merchantCity', 'merchantZip', 'posOnPremises', 'recu
```

In [14]:
```
df.head(5)
```

Out[14]:

	accountNumber	customerId	creditLimit	availableMoney	transactionDateTime	tran
0	737265056	737265056	5000.0	5000.0	2016-08-13T14:27:32	
1	737265056	737265056	5000.0	5000.0	2016-10-11T05:05:54	
2	737265056	737265056	5000.0	5000.0	2016-11-08T09:18:39	
3	737265056	737265056	5000.0	5000.0	2016-12-10T02:14:50	
4	830329091	830329091	5000.0	5000.0	2016-03-24T21:04:46	

5 rows × 23 columns

4. Data Preprocessing, Data Type Conversion, Cat encoding, missing value imputing

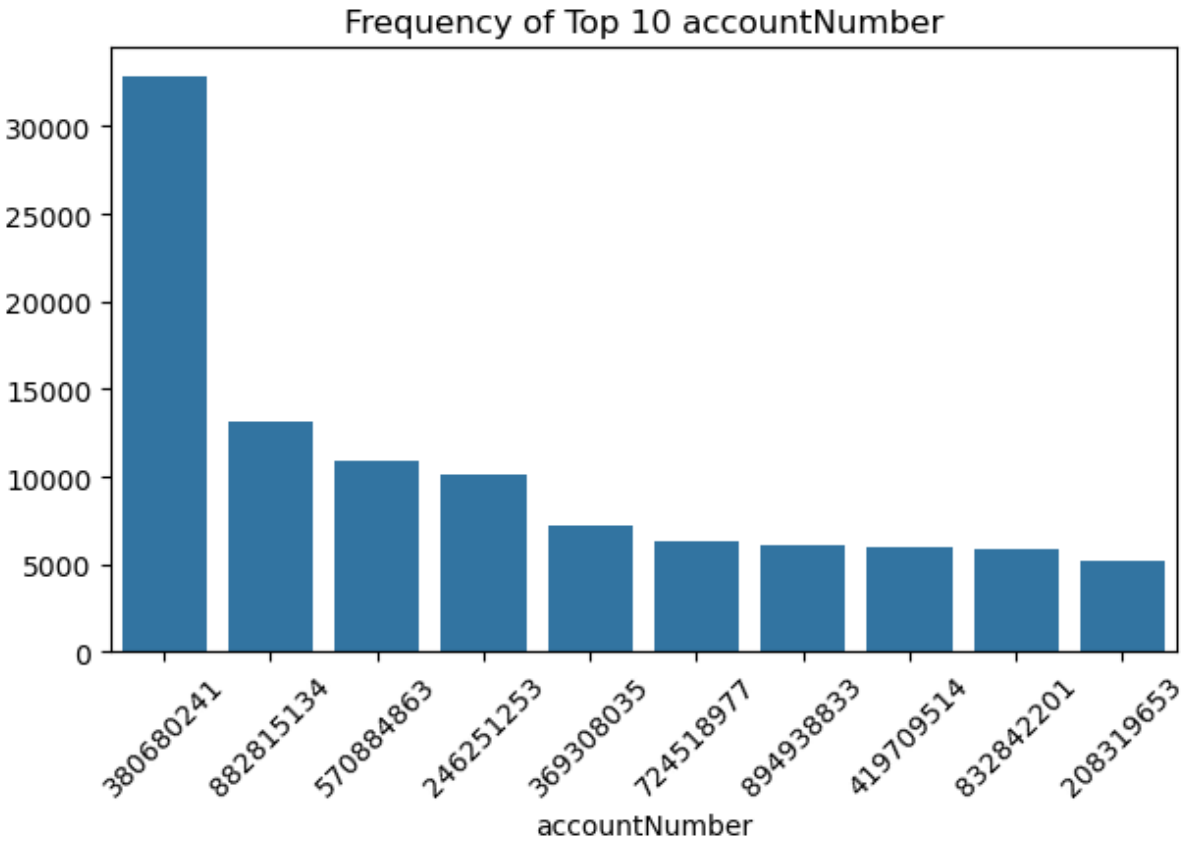
In [15]:
```
# Convert dates to datetime for plotting
df['transactionDateTime'] = pd.to_datetime(df['transactionDateTime'])
```

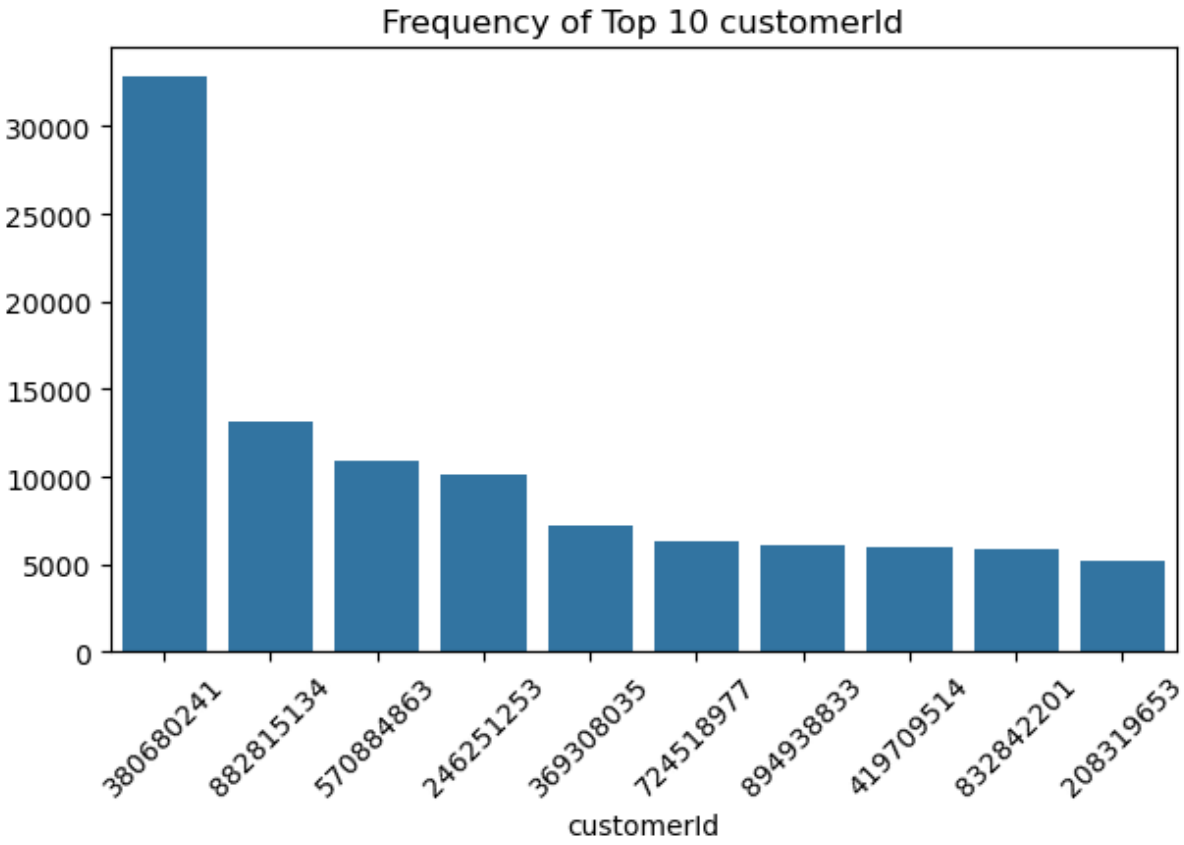
5. Data Visualization

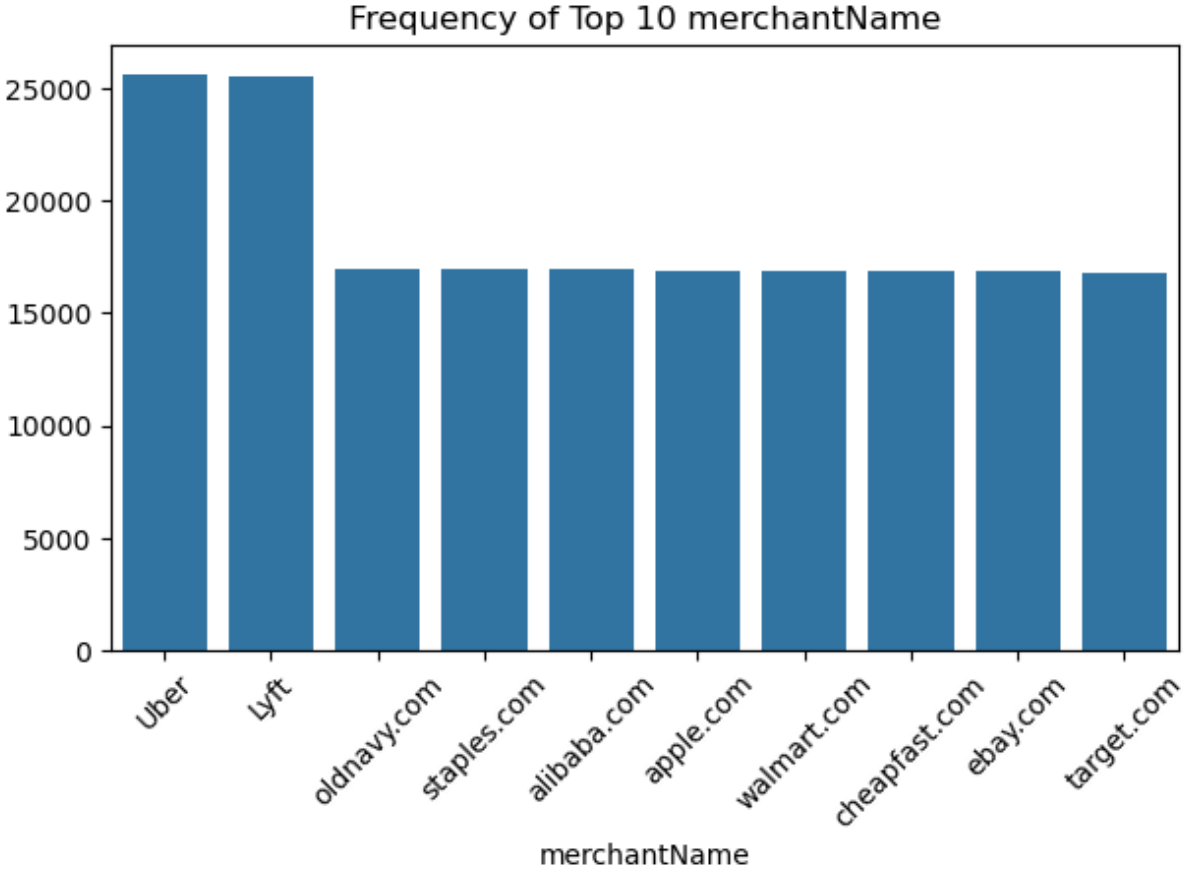
5.1 Numerical & Categorical Features visuals

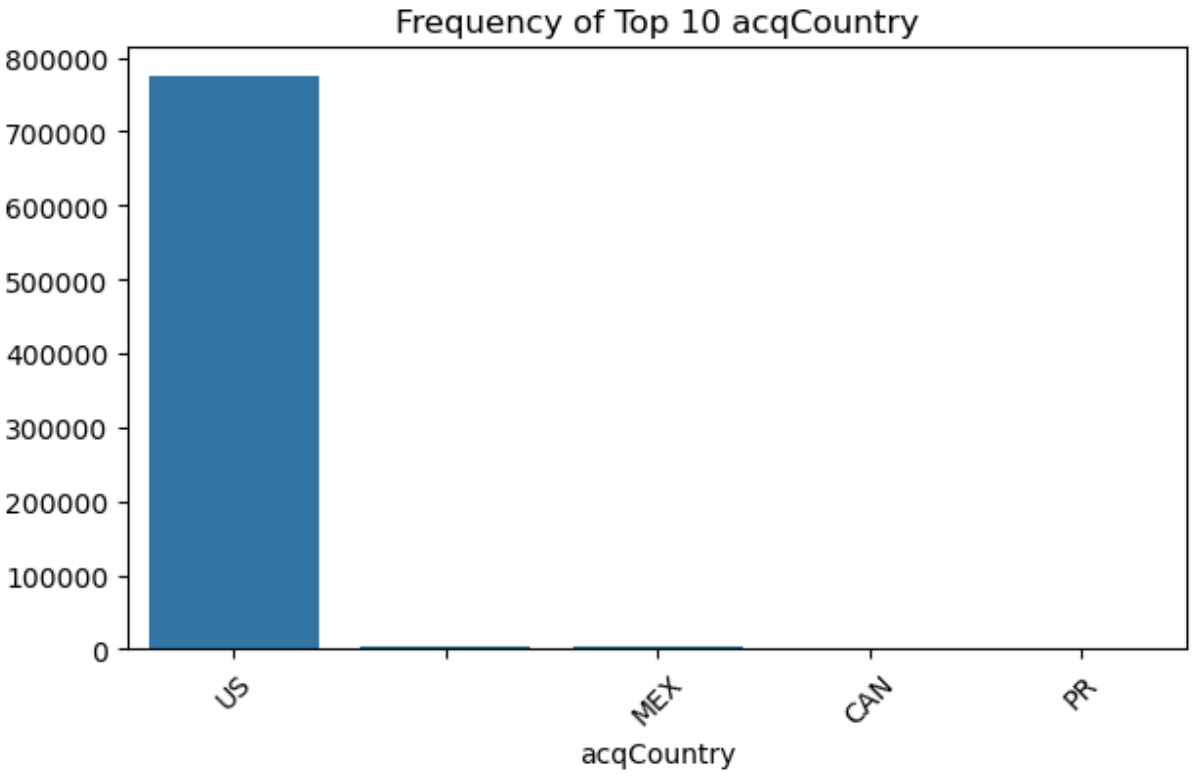
In [16]:
```
# 1. Histograms for Numerical Features
numerical_features = df.select_dtypes(include=['int64', 'float64']).columns
print('numerical features')
print(numerical_features)
categorical_features = df.select_dtypes(include=['object', 'bool']).columns
print('categorical_features')
print( categorical_features)

# 2. Bar Charts for Categorical Features
for col in categorical_features:
    counts = df[col].value_counts().nlargest(10)  # Taking the top 10 for br
    plt.figure(figsize=(7,4))
    sns.barplot(x=counts.index, y=counts.values)
    plt.title(f'Frequency of Top 10 {col}')
    plt.xticks(rotation=45)
    plt.show()
```

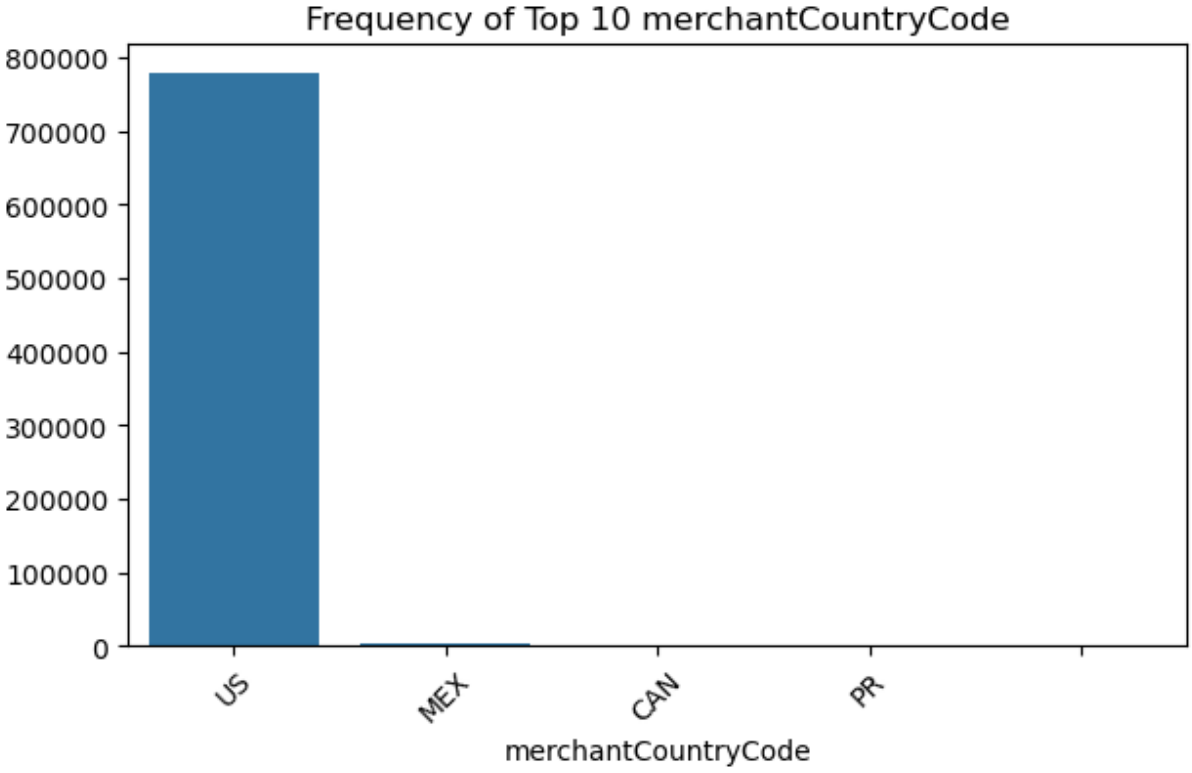
```
numerical features
Index(['creditLimit', 'availableMoney', 'transactionAmount', 'currentBalanc
e'], dtype='object')
categorical_features
Index(['accountNumber', 'customerId', 'merchantName', 'acqCountry',
       'merchantCountryCode', 'posEntryMode', 'posConditionCode',
       'merchantCategoryCode', 'currentExpDate', 'accountOpenDate',
       'dateOfLastAddressChange', 'cardCVV', 'enteredCVV', 'cardLast4Digit
s',
       'transactionType', 'cardPresent', 'expirationDateKeyInMatch',
       'isFraud'],
      dtype='object')
```

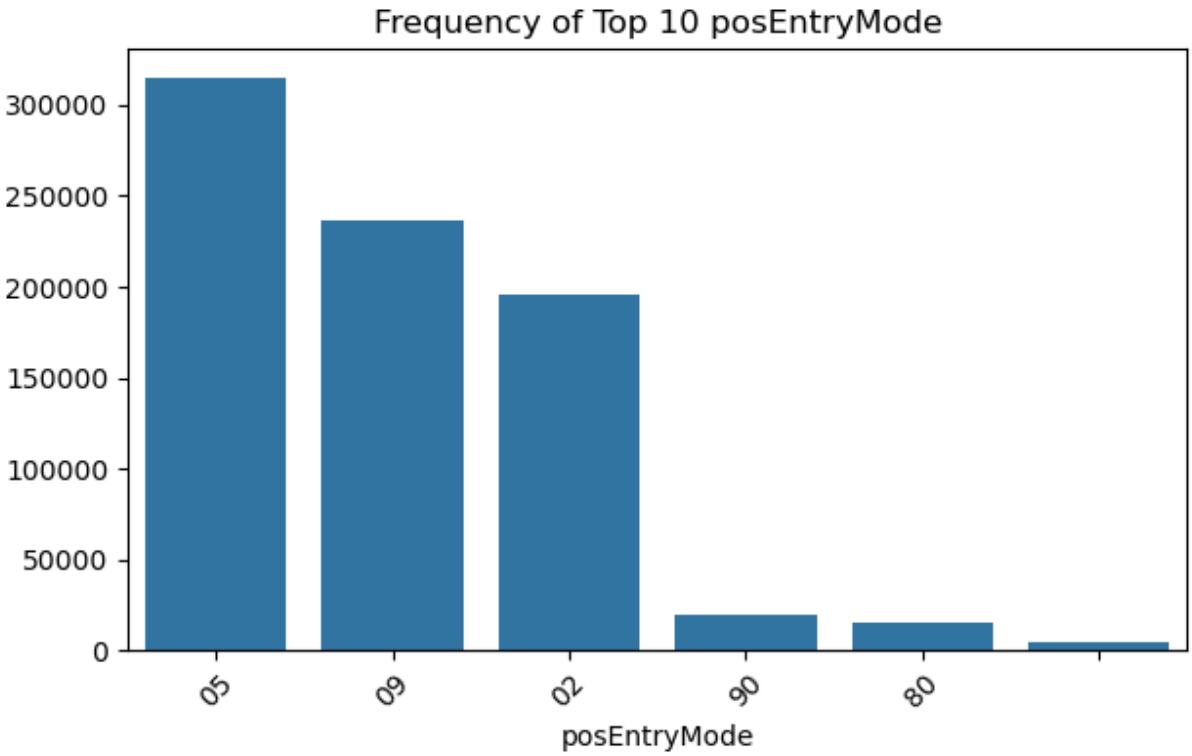


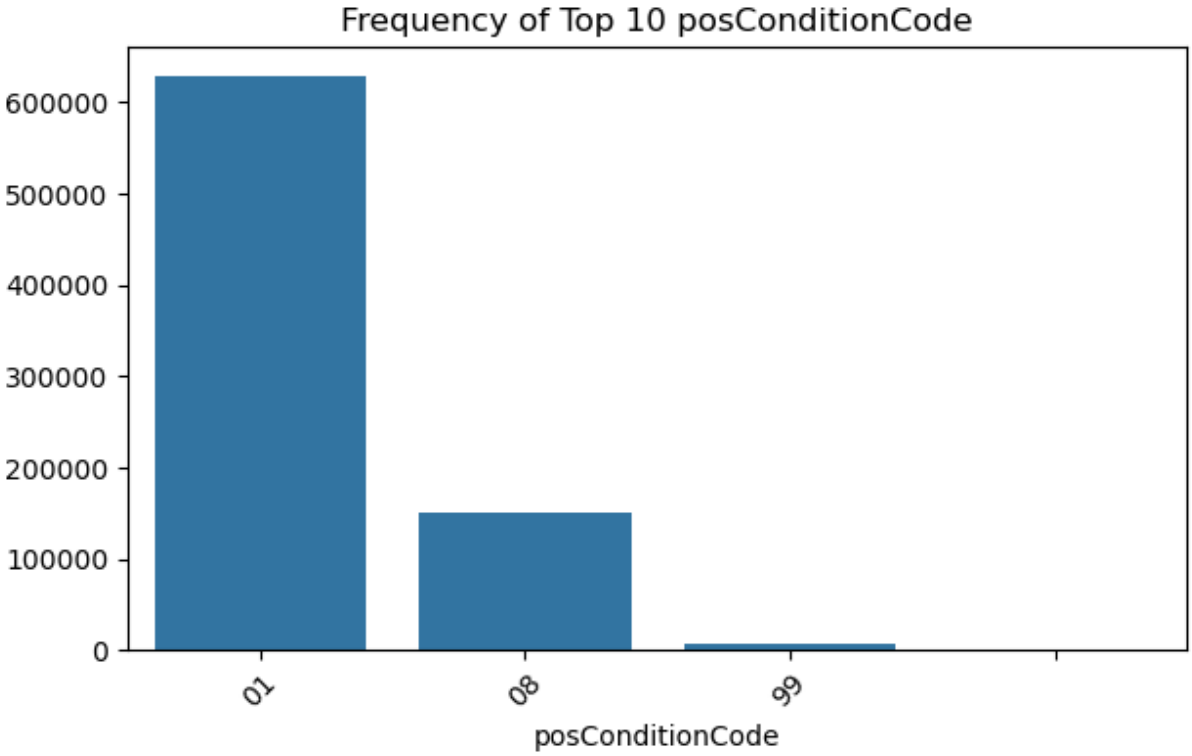

Frequency of Top 10 customerId
30000
25000
20000
15000
10000
5000
0
380680241
882815134
570884863
246251253
369308035
724518977
894938833
419709514
832842201
208319653
customerId

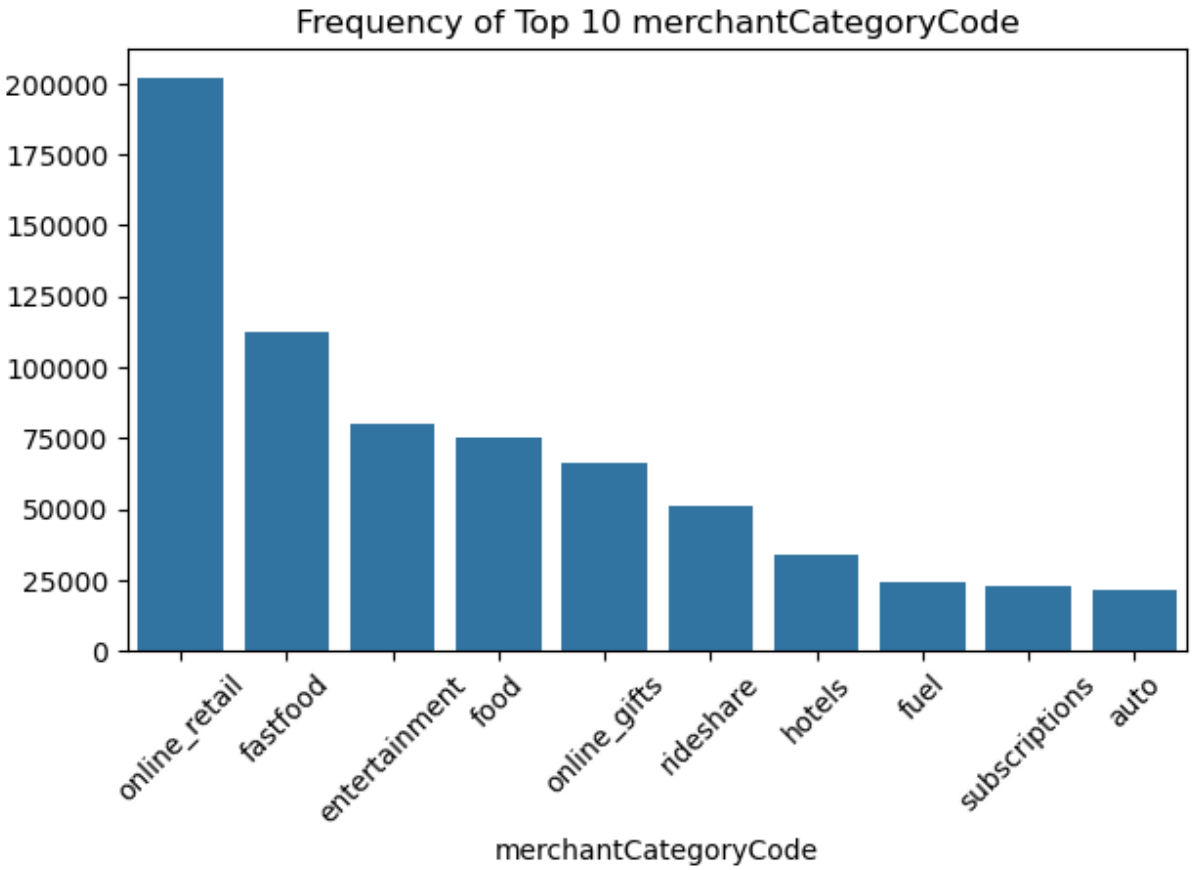

Frequency of Top 10 merchantName
25000
20000
15000
10000
5000
0
Uber
Lyft
oldnavy.com
staples.com
alibaba.com
apple.com
walmart.com
cheapfast.com
ebay.com
target.com
merchantName

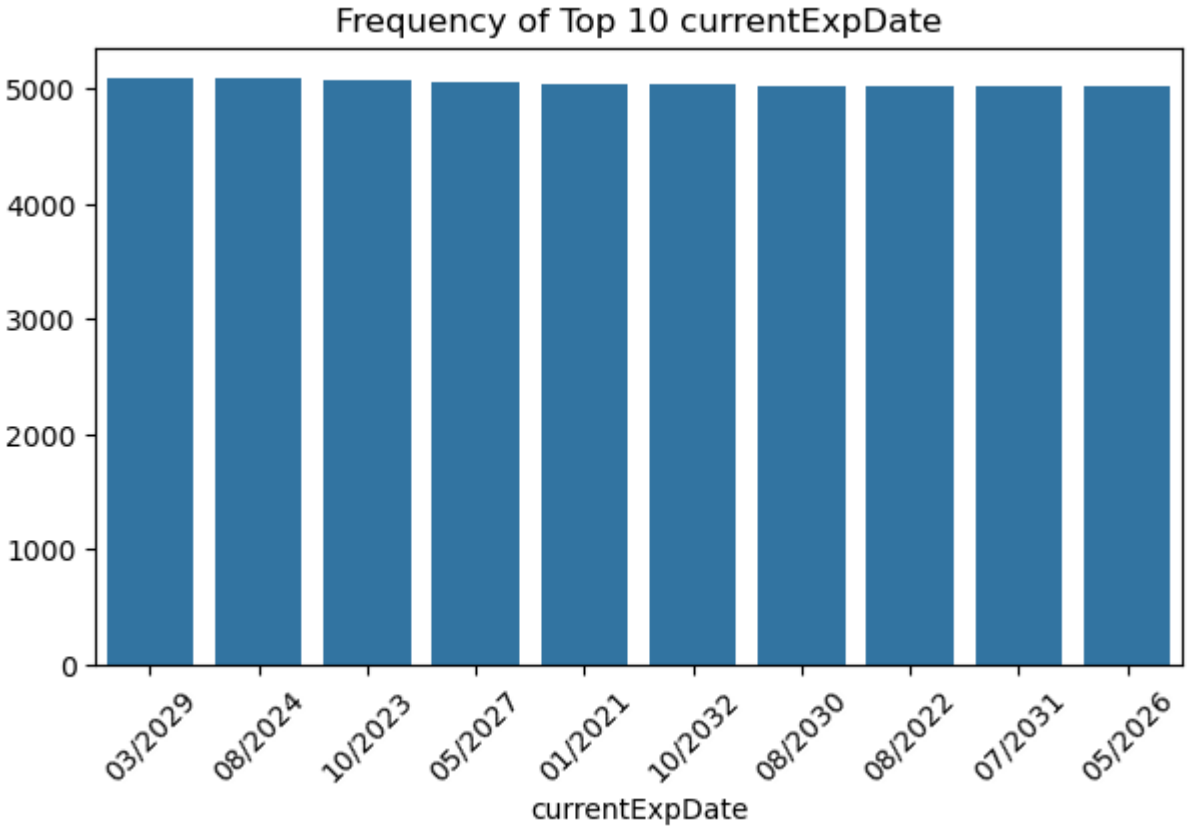

Frequency of Top 10 acqCountry
800000
700000
600000
500000
400000
300000
200000
100000
0
US
MEX
CAN
PR
acqCountry

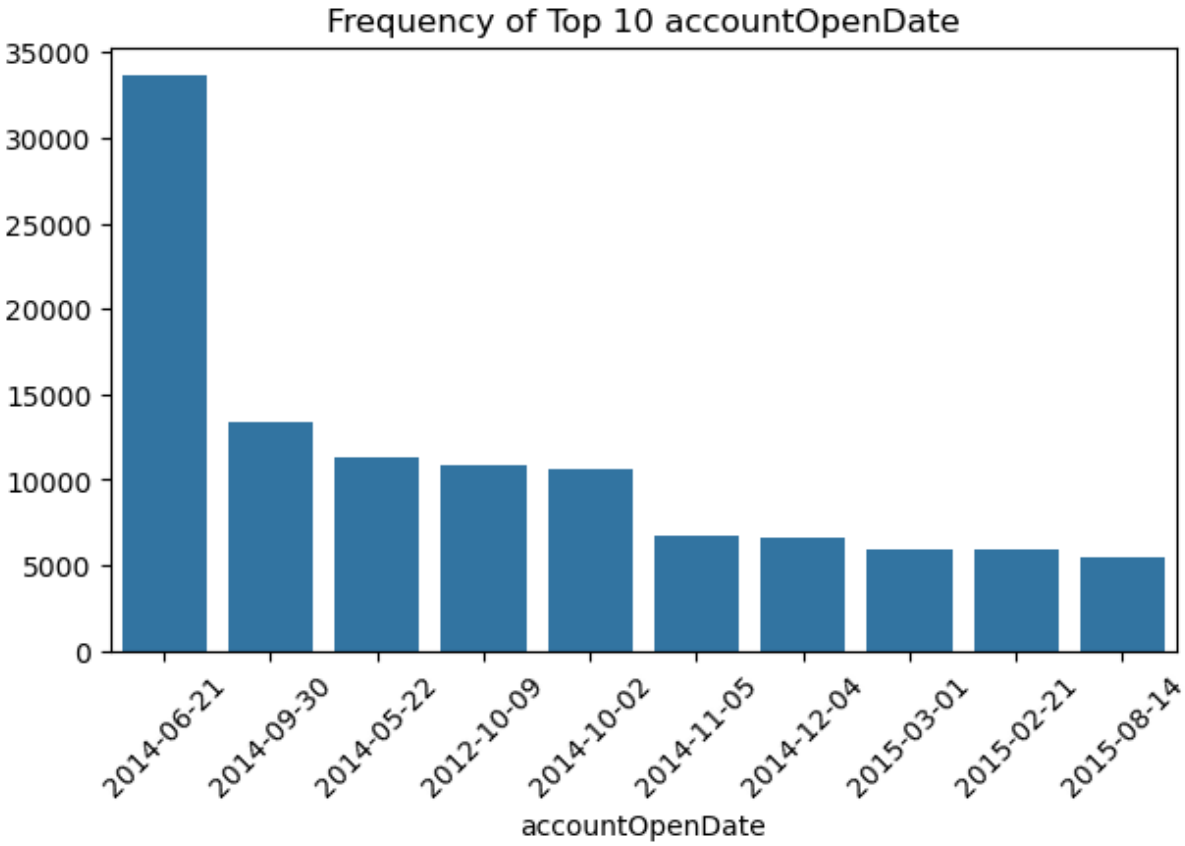

Frequency of Top 10 merchantCountryCode
800000
700000
600000
500000
400000
300000
200000
100000
0
US
MEX
CAN
PR
merchantCountryCode

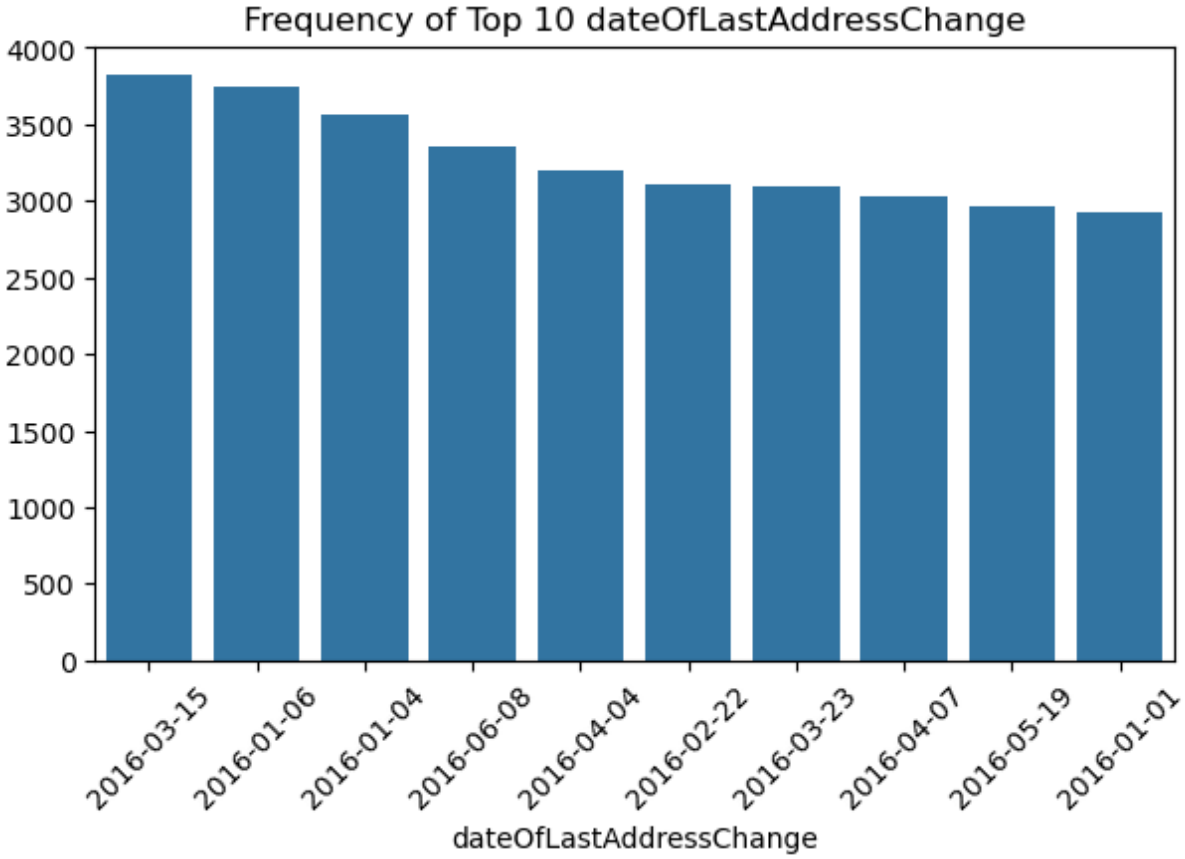

Frequency of Top 10 posEntryMode
300000
250000
200000
150000
100000
50000
0
05
09
02
90
80
posEntryMode

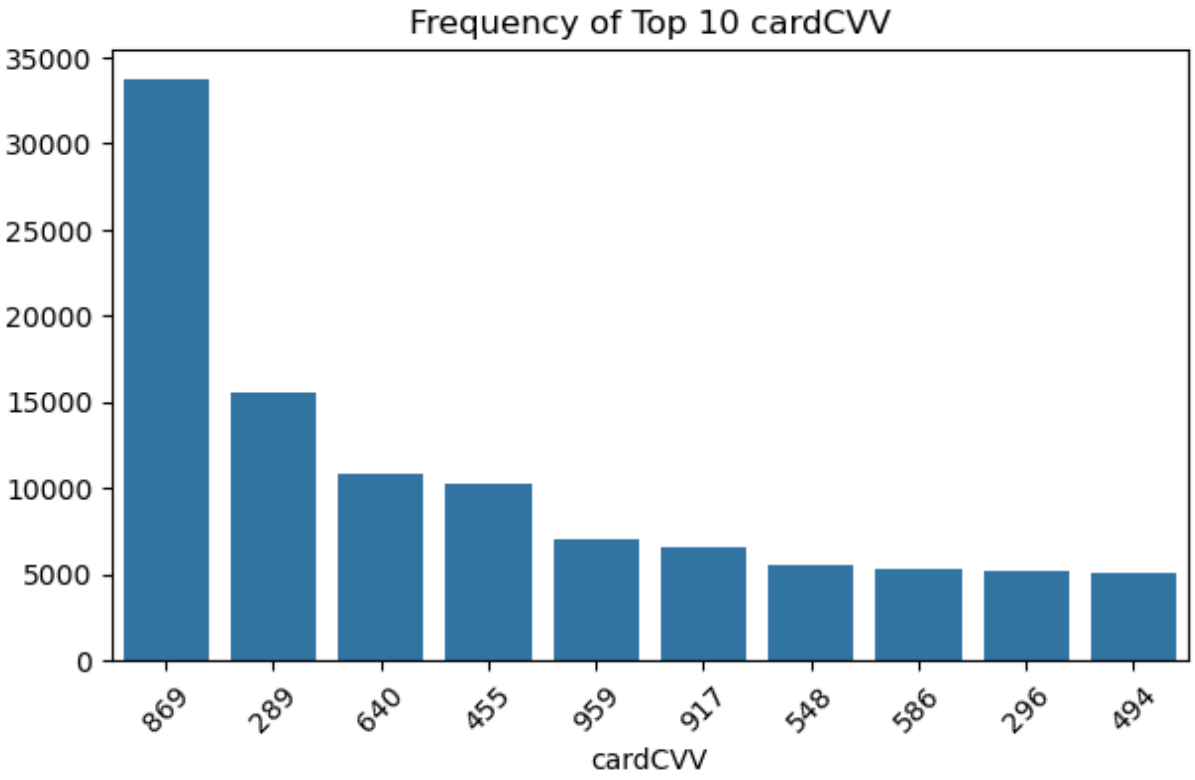

Frequency of Top 10 posConditionCode
600000
500000
400000
300000
200000
100000
0
01
08
99
posConditionCode

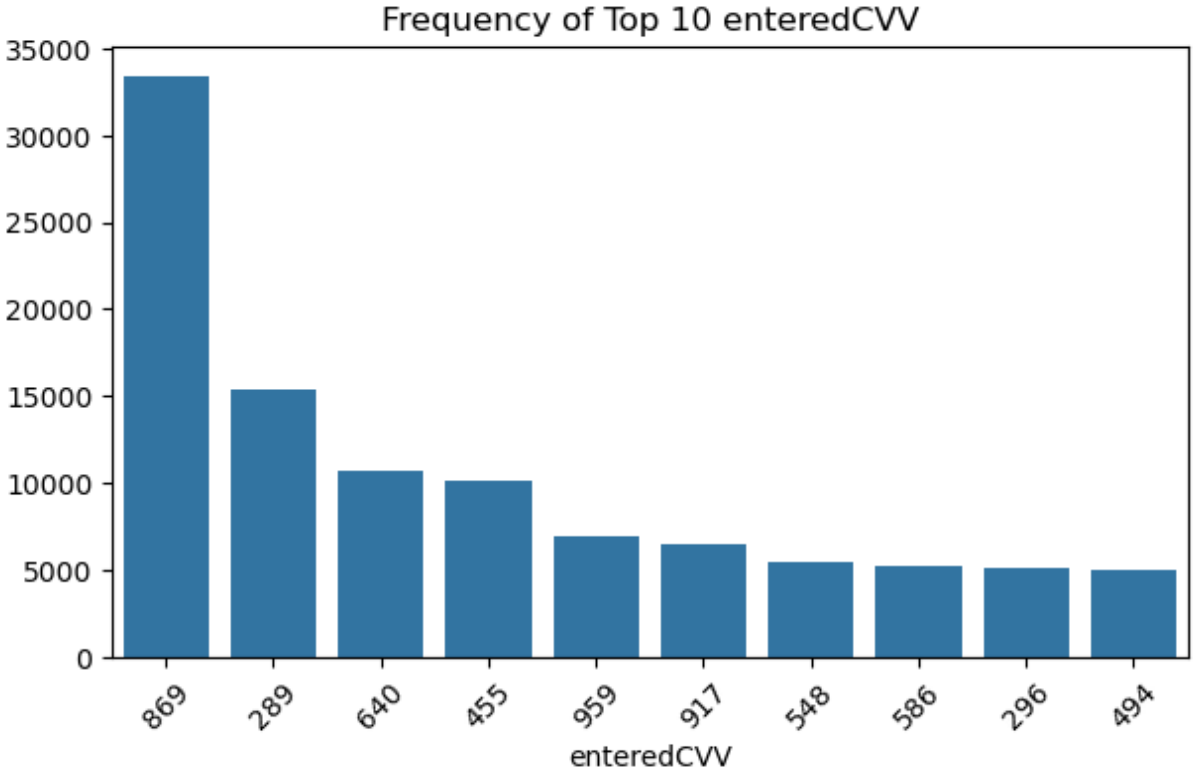

Frequency of Top 10 merchantCategoryCode
200000
175000
150000
125000
100000
75000
50000
25000
0
online_retail
fastfood
entertainment
food
online_gifts
rideshare
hotels
fuel
subscriptions
auto
merchantCategoryCode

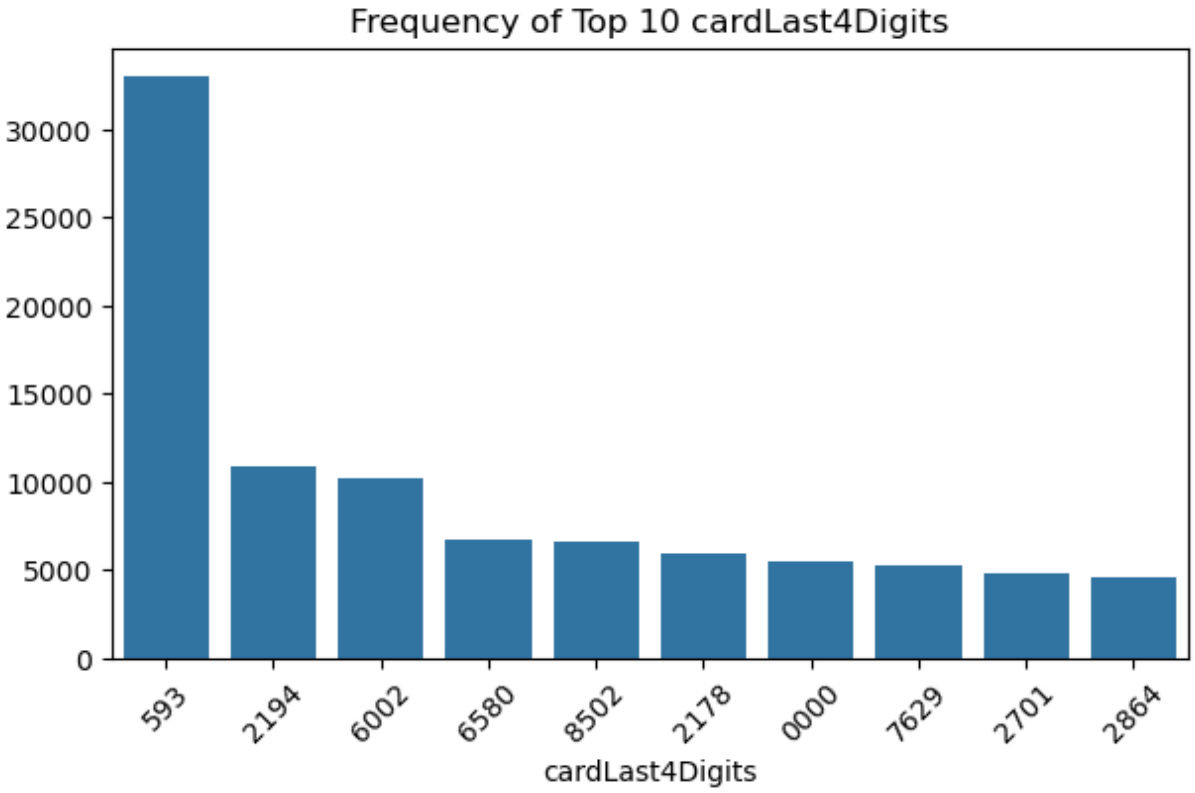

Frequency of Top 10 currentExpDate
5000
4000
3000
2000
1000
0
03/2029
08/2024
10/2023
05/2027
01/2021
10/2032
08/2030
08/2022
07/2031
05/2026
currentExpDate

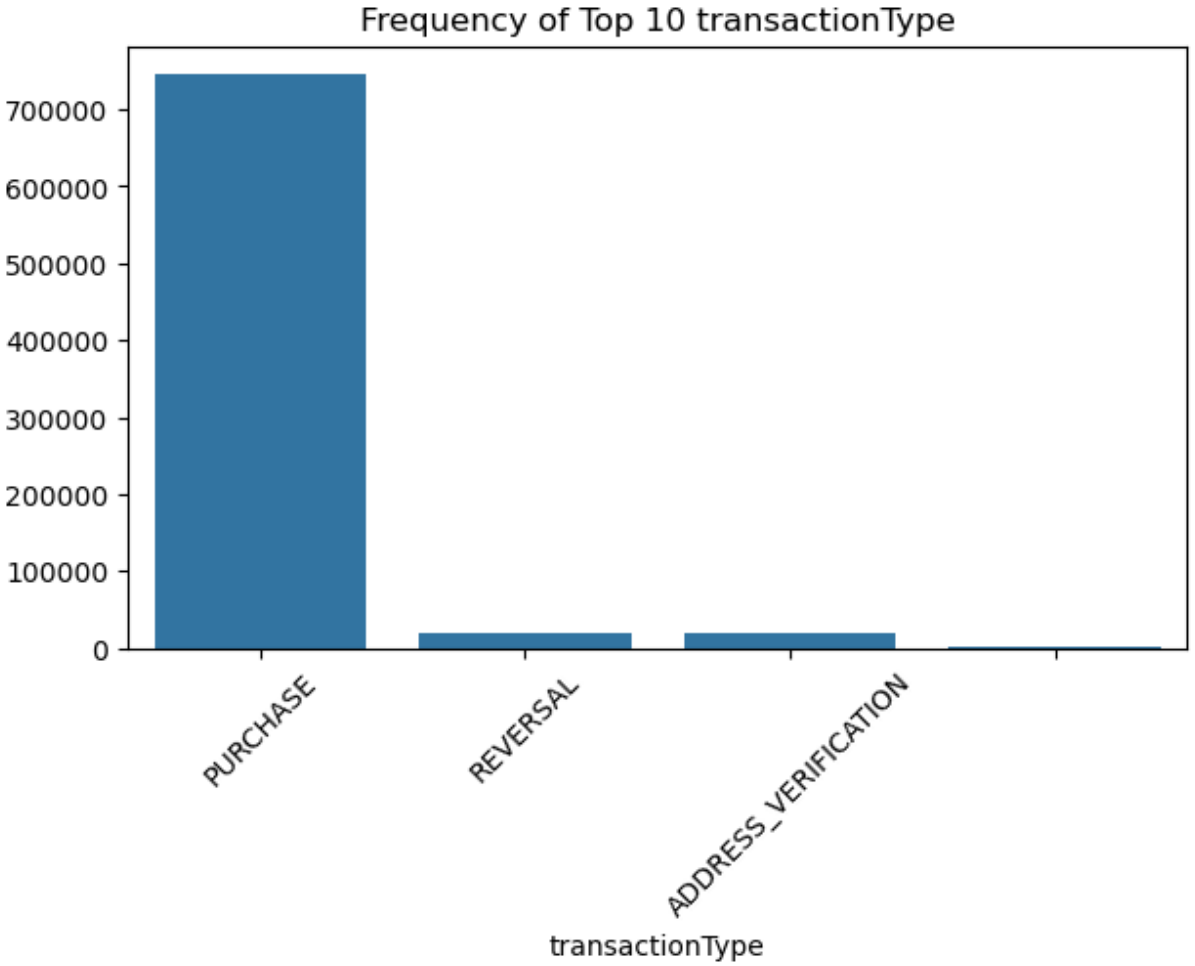

Frequency of Top 10 accountOpenDate
35000
30000
25000
20000
15000
10000
5000
0
2014-06-21
2014-09-30
2014-05-22
2012-10-09
2014-10-02
2014-11-05
2014-12-04
2015-03-01
2015-02-21
2015-08-14
accountOpenDate

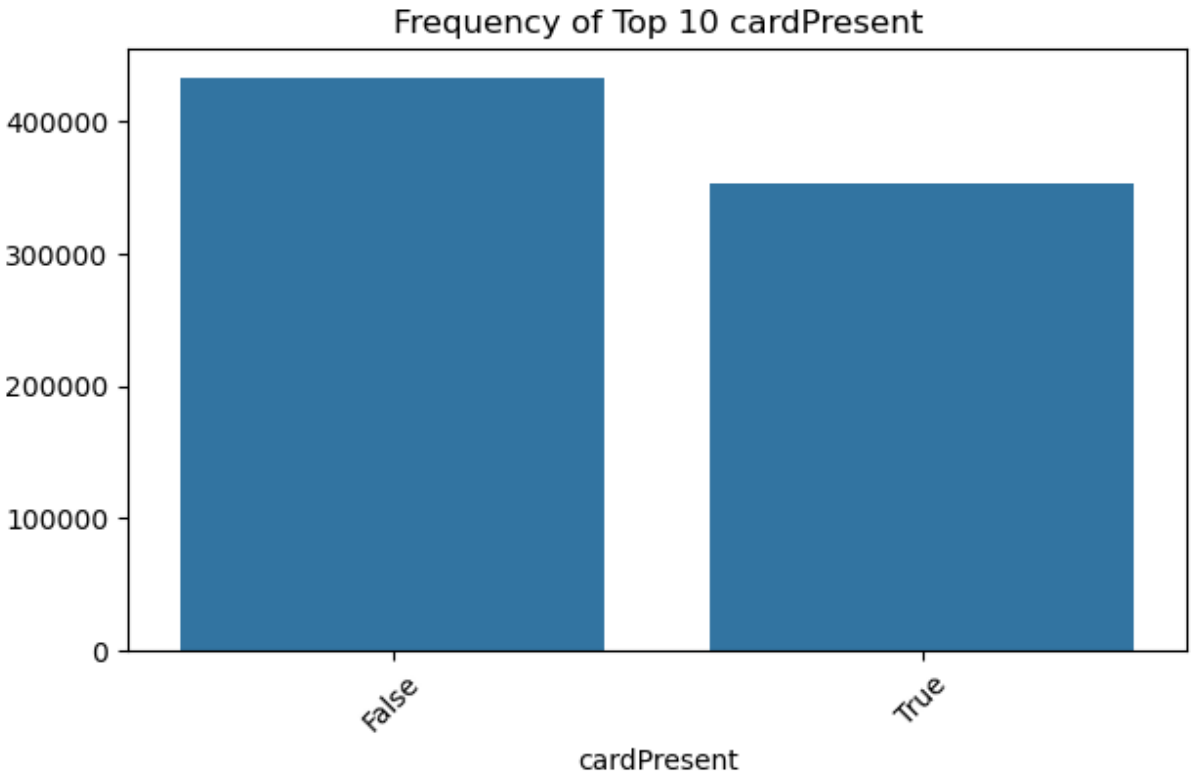

Frequency of Top 10 dateOfLastAddressChange
4000
3500
3000
2500
2000
1500
1000
500
0
2016-03-15
2016-01-06
2016-01-04
2016-06-08
2016-04-04
2016-02-22
2016-03-23
2016-04-07
2016-05-19
2016-01-01
dateOfLastAddressChange

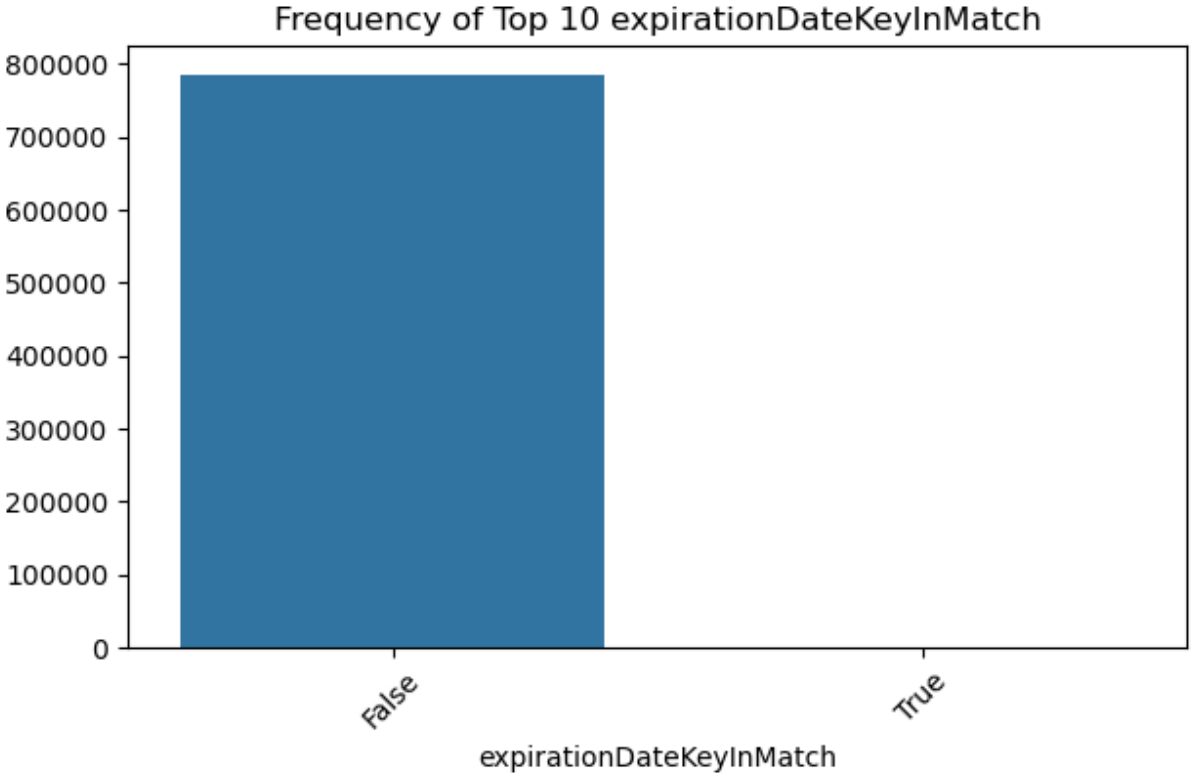

Frequency of Top 10 cardCVV
35000
30000
25000
20000
15000
10000
5000
0
869
289
640
455
959
917
548
586
296
494
cardCVV

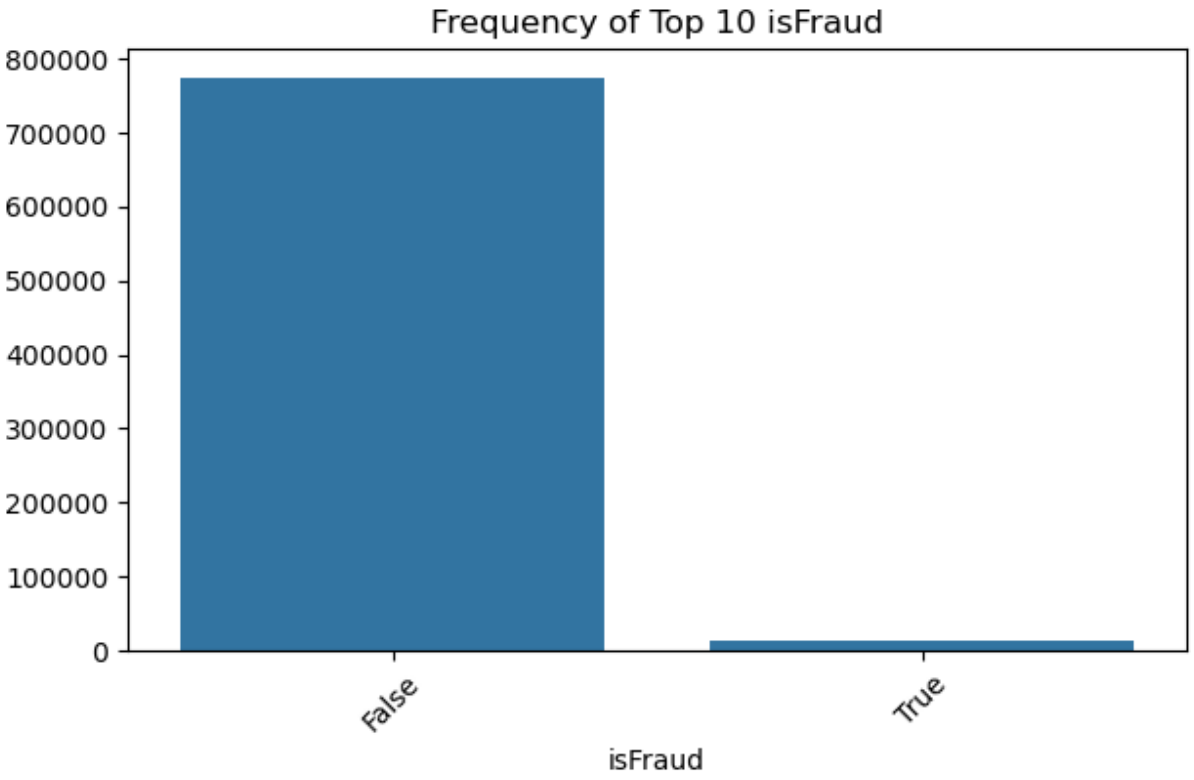

Frequency of Top 10 enteredCVV
35000
30000
25000
20000
15000
10000
5000
0
869
289
640
455
959
917
548
586
296
494
enteredCVV


Frequency of Top 10 cardLast4Digits
30000
25000
20000
15000
10000
5000
0
593
2194
6002
6580
8502
2178
0000
7629
2701
2864
cardLast4Digits


Frequency of Top 10 transactionType
700000
600000
500000
400000
300000
200000
100000
0
PURCHASE
REVERSAL
ADDRESS_VERIFICATION
transactionType

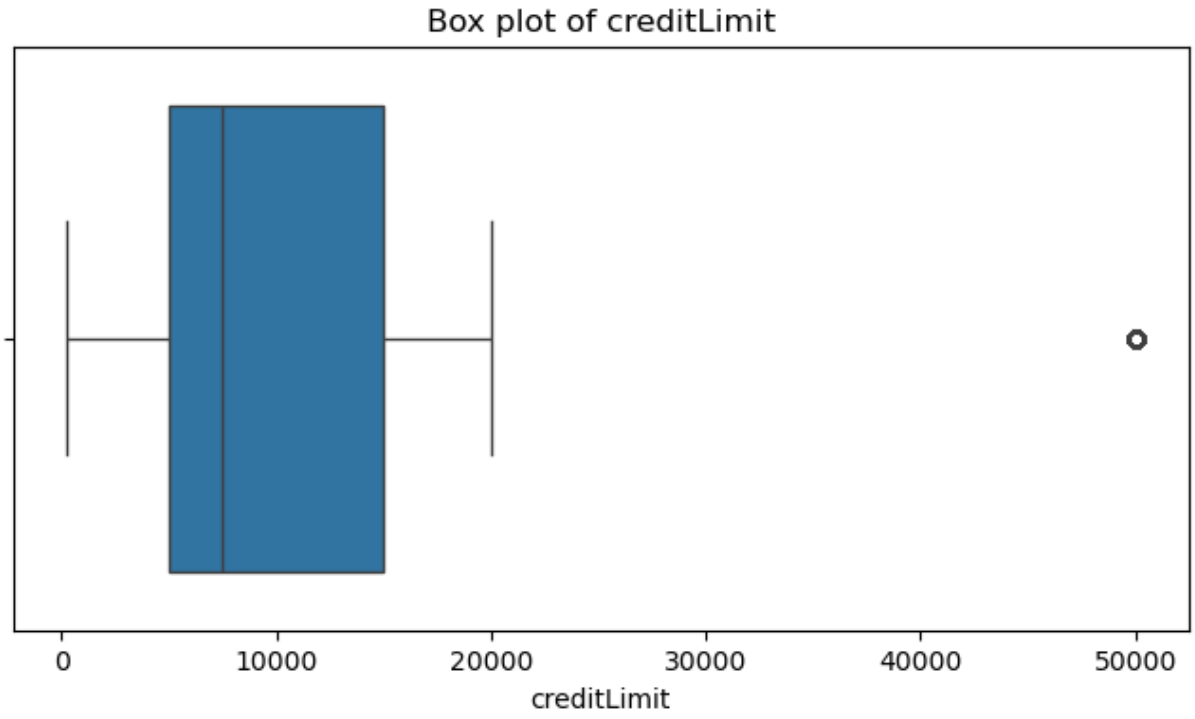

Frequency of Top 10 cardPresent
400000
300000
200000
100000
0
False
True
cardPresent

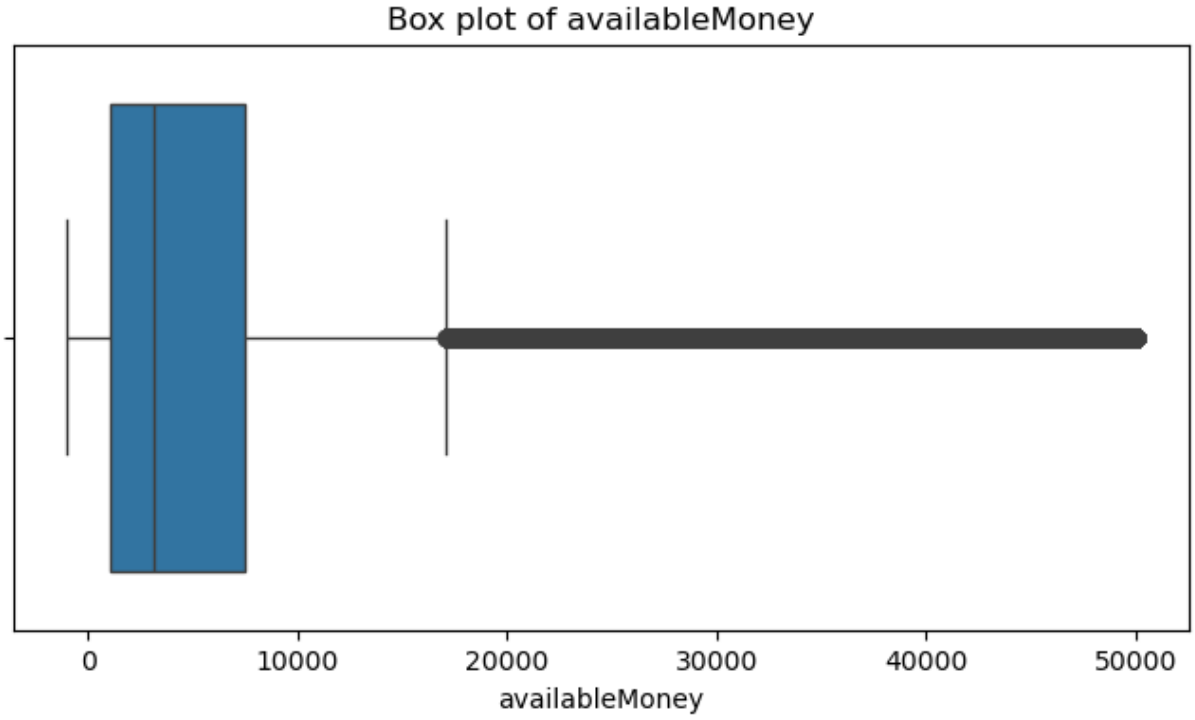

Frequency of Top 10 expirationDateKeyInMatch
800000
700000
600000
500000
400000
300000
200000
100000
0
False
True
expirationDateKeyInMatch

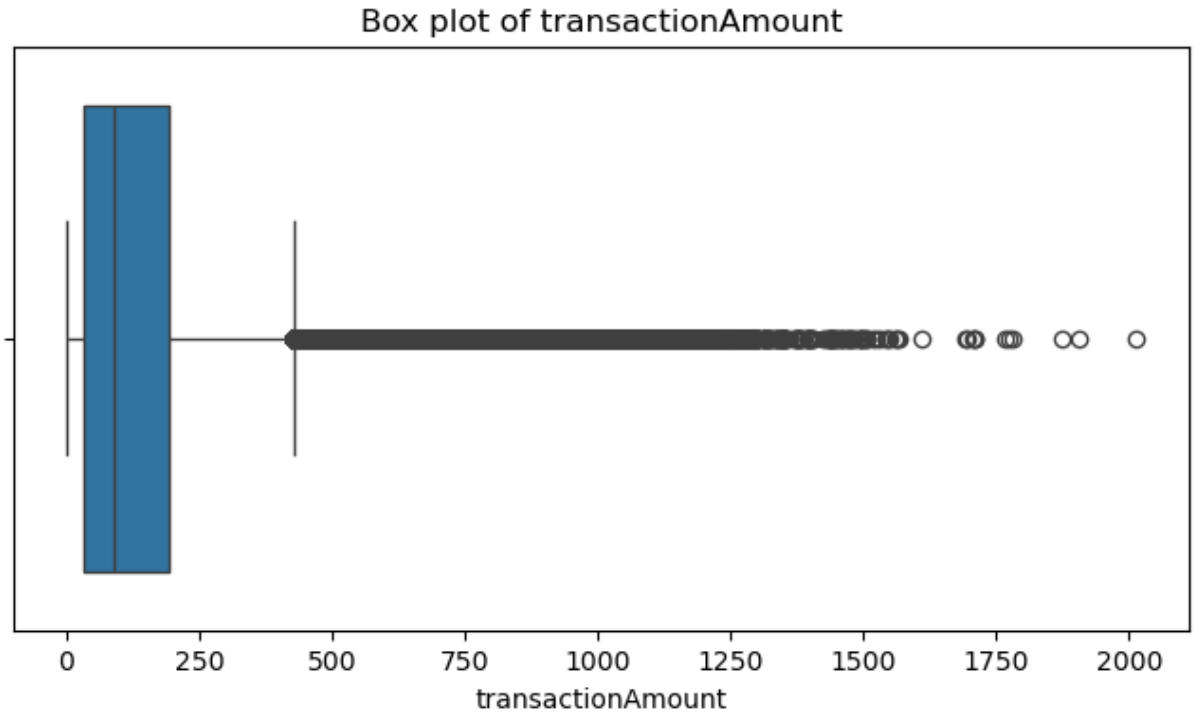


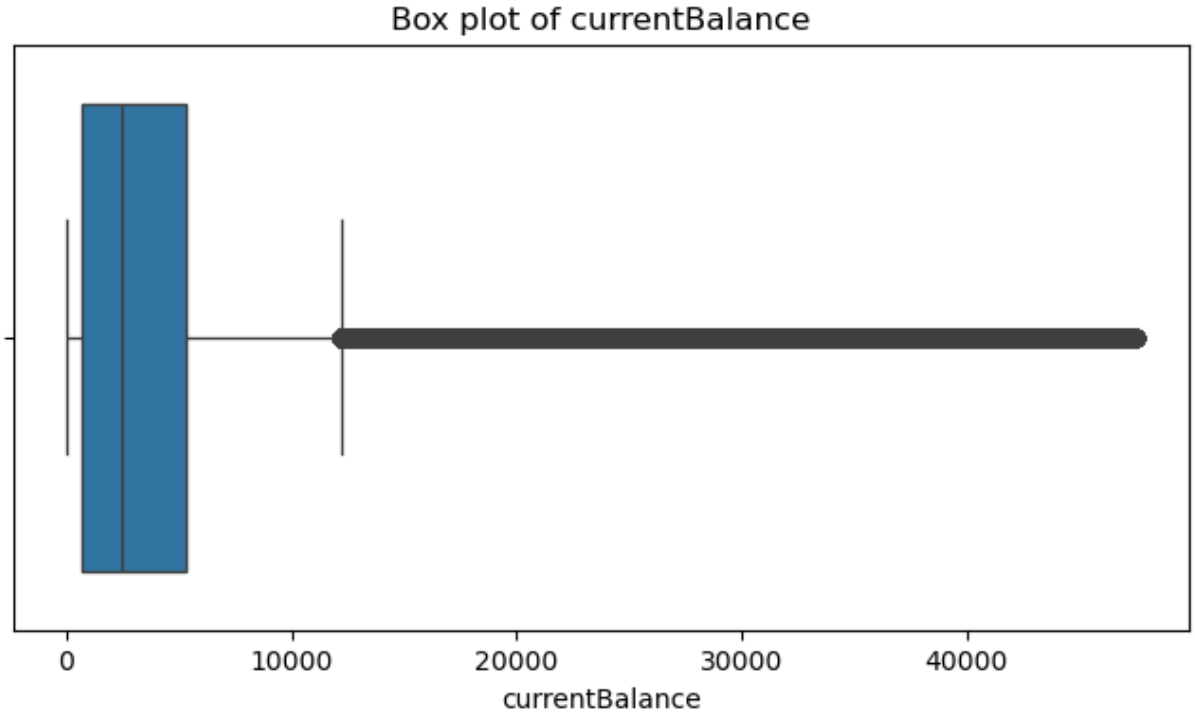
5.2 Outlier Detection with boxplots

In [17]:
```
# 3. Box Plots for Numerical Features
for col in numerical_features:
    plt.figure(figsize=(8, 4))
    sns.boxplot(data=df, x=col)
    plt.title(f'Box plot of {col}')
    plt.show()
```


Box plot of creditLimit
0
10000
20000
30000
40000
50000
creditLimit

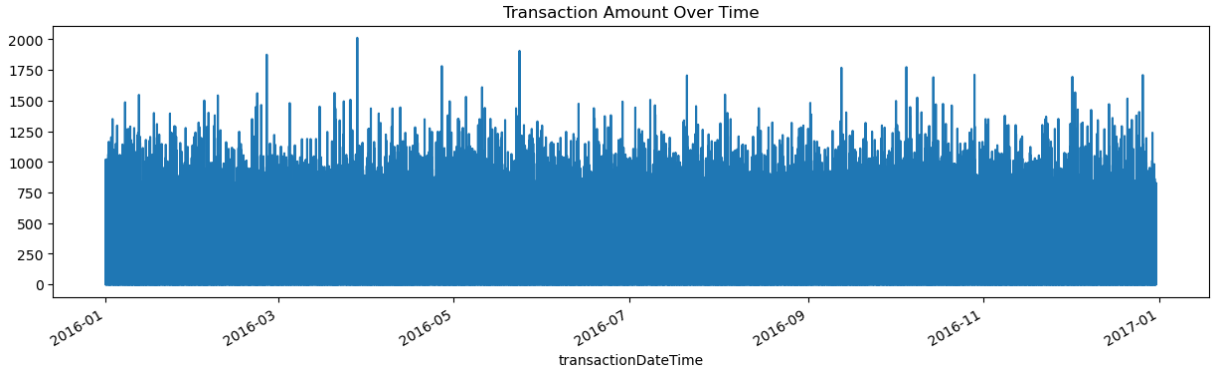

Box plot of availableMoney
0
10000
20000
30000
40000
50000
availableMoney





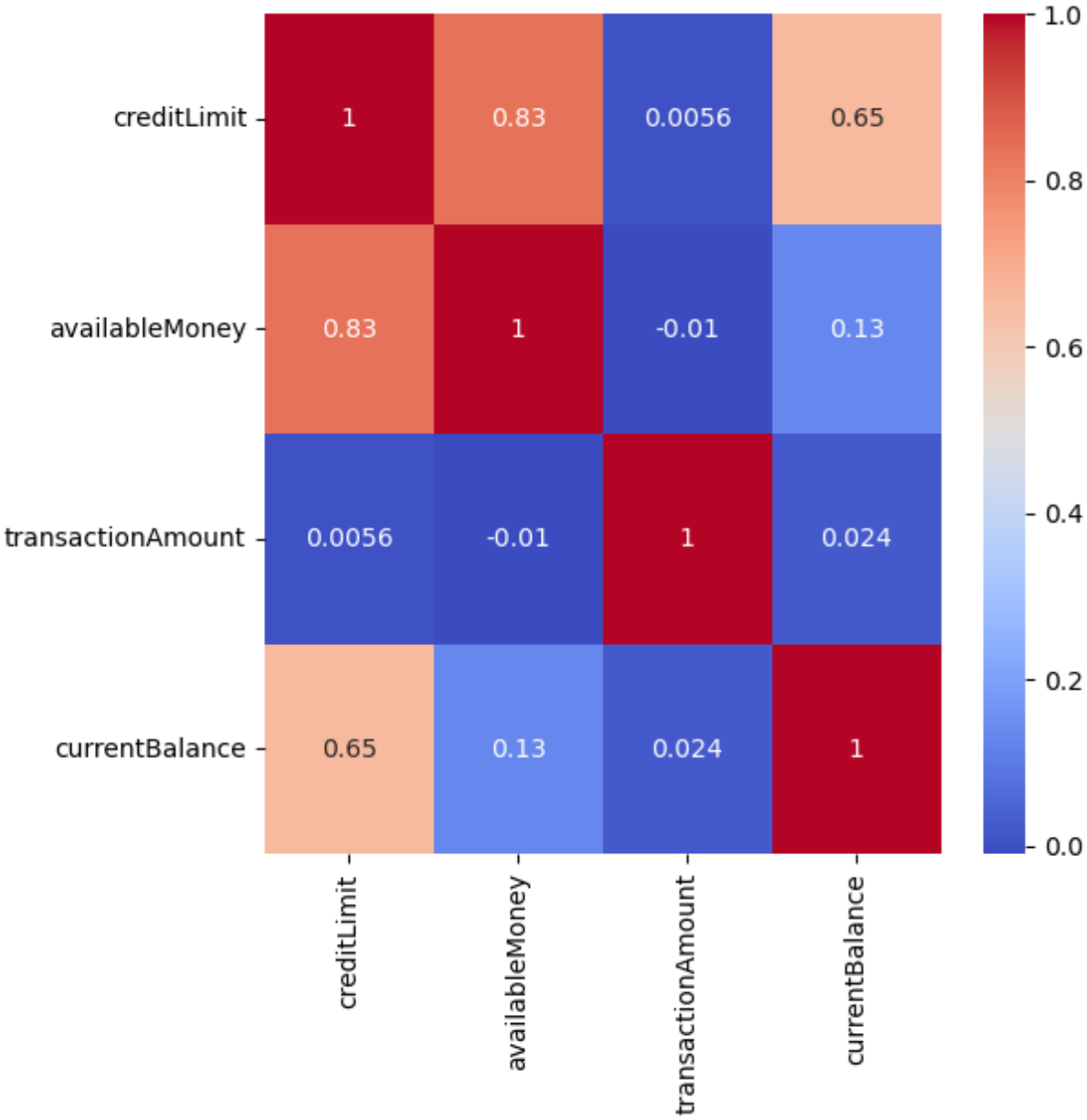
5.3 Time Series Analysis

In [18]:
```
# 4. Time Series Plot
plt.figure(figsize=(15, 4))
df.set_index('transactionDateTime')['transactionAmount'].plot()
plt.title('Transaction Amount Over Time')
plt.show()
```



5.4 Correlation Analysis for Numerical Features

In [19]:
```
plt.figure(figsize=(6, 6))
sns.heatmap(df[numerical_features].corr(), annot=True, cmap='coolwarm')
plt.show()
```



6 Feature Engineering

We created a cvv matched feature to check if the entered cvv for the purchase is the same as card cvv

In [20]:
```
numerical_features = numerical_features.tolist()
categorical_features = categorical_features.tolist()
```

In [21]:
```
df["CVVmatched"] = df.cardCVV == df.enteredCVV
df = df.drop(["cardCVV", "enteredCVV"], axis=1)
categorical_features.remove("cardCVV")
categorical_features.remove("enteredCVV")
categorical_features.append("CVVmatched")
```

7 Preprocessing

In [22]:
```
display(df.head())
print("Number of columns:",len(df.columns.tolist()))
print("Number of numeric feature columns:",len(numerical_features))
print("Number of categorical feature columns:",len(categorical_features))
```

	accountNumber	customerId	creditLimit	availableMoney	transactionDateTime	trans
0	737265056	737265056	5000.0	5000.0	2016-08-13 14:27:32	
1	737265056	737265056	5000.0	5000.0	2016-10-11 05:05:54	
2	737265056	737265056	5000.0	5000.0	2016-11-08 09:18:39	
3	737265056	737265056	5000.0	5000.0	2016-12-10 02:14:50	
4	830329091	830329091	5000.0	5000.0	2016-03-24 21:04:46	

5 rows × 22 columns

```
Number of columns: 22
Number of numeric feature columns: 4
Number of categorical feature columns: 17
```

In [23]:
```
len(df["transactionDateTime"].unique().tolist())
```

Out[23]: 776637

We have included all columns as our features except for `transactionDateTime`, which is a `datetime` column. As seen from the above, the number of unique values in `transactionDateTime` is 776637; thus, this will be dropped.

In [24]:
```
drop_features = ["transactionDateTime"]
```

In [25]:
```
categorical_features.remove("isFraud")
```

In [26]:
```
# Account number and customer id columns are redundant.
print((df.accountNumber == df.customerId).all())
df = df.drop("customerId", axis=1)
categorical_features.remove("customerId")
```

```
True
```

In [27]:
```
# We decided to drop merchantName column because it vastly increased our fea
# Without adding much information. merchantCategoryCode will be used to cate
df = df.drop("merchantName", axis=1)
categorical_features.remove("merchantName")
```

In [28]:
```
df = df.drop(["accountOpenDate", 'cardLast4Digits'], axis=1)
categorical_features.remove("accountOpenDate")
categorical_features.remove("cardLast4Digits")
```

`isFraud` is our target variable, hence removed from `categorical_features` list.

In [29]:
```
print(f"Numerical Features: {numerical_features}\n")
print(f"Categorical Features: {categorical_features}")
```

```
Numerical Features: ['creditLimit', 'availableMoney', 'transactionAmount',
'currentBalance']

Categorical Features: ['accountNumber', 'acqCountry', 'merchantCountryCode',
'posEntryMode', 'posConditionCode', 'merchantCategoryCode', 'currentExpDat
e', 'dateOfLastAddressChange', 'transactionType', 'cardPresent', 'expiration
DateKeyInMatch', 'CVVmatched']
```

7.1 Preprocessing Categorical Features

In [30]:
```
# Using imported function from src/preprocessing.py
count_unique_numbers(df, categorical_features)
```

Out[30]:

	feature	unique_entry_counts
0	accountNumber	5000
7	dateOfLastAddressChange	2184
6	currentExpDate	165
5	merchantCategoryCode	19
3	posEntryMode	6
1	acqCountry	5
2	merchantCountryCode	5
4	posConditionCode	4
8	transactionType	4
9	cardPresent	2
10	expirationDateKeyInMatch	2
11	CVVmatched	2

We have seen way too many unique entries for a few categorical features, including `accountNumber` and `customerId`. Thus we will apply a function to narrow down the number of unique entries; once the number of unique entries in a column is **greater than 10**, `generalize_categories` function will classify entries that appear less than 10% in our dataset as "Others".

In [31]:
```
# Using imported function from src/preprocessing.py
preprocess_df = df.copy()
for category_column in categorical_features:
    preprocess_df[category_column] = preprocess_df[category_column].replace(
    if len(preprocess_df[category_column].unique().tolist()) > 10:
        preprocess_df = generalize_categories(category_column, preprocess_df
```

In [32]:
```
preprocess_df
```

Out[32]:

	accountNumber	creditLimit	availableMoney	transactionDateTime	transactio
0	Other	5000.0	5000.00	2016-08-13 14:27:32	
1	Other	5000.0	5000.00	2016-10-11 05:05:54	
2	Other	5000.0	5000.00	2016-11-08 09:18:39	
3	Other	5000.0	5000.00	2016-12-10 02:14:50	
4	Other	5000.0	5000.00	2016-03-24 21:04:46	
...	...	...	...	...	
786358	Other	50000.0	48904.96	2016-12-22 18:44:12	
786359	Other	50000.0	48785.04	2016-12-25 16:20:34	
786360	Other	50000.0	48766.15	2016-12-27 15:46:24	
786361	Other	50000.0	48716.72	2016-12-29 00:30:55	
786362	Other	50000.0	48666.83	2016-12-30 20:10:29	

786363 rows × 19 columns

In [33]:
```
count_unique_numbers(preprocess_df, categorical_features)
```

Out[33]:

	feature	unique_entry_counts
5	merchantCategoryCode	16
3	posEntryMode	6
0	accountNumber	5
1	acqCountry	5
2	merchantCountryCode	5
4	posConditionCode	4
8	transactionType	4
9	cardPresent	2
10	expirationDateKeyInMatch	2
11	CVVmatched	2
6	currentExpDate	1
7	dateOfLastAddressChange	1

From above data frame, we observed that none of the entires in `currentExpDate` and `dateOfLastAddressChange` has frequency of 10% or higher. We decided to drop the two columns as they seem to carry little to no predictive power.

In [34]:
```
categorical_features.remove("currentExpDate")
categorical_features.remove("dateOfLastAddressChange")
drop_features.append("currentExpDate")
drop_features.append("dateOfLastAddressChange")
```

7.2 Preprocessing Numerical Columns

As seen from the distribution of numerical columns above (refer to the boxplots), we can see that `creditLimit` column has a single outlier, whereas the other columns have multiple outliers. To intrapolate the missing values, we decided to use `strategy="median"` for the other columns, and `strategy="mean"` for `creditLimit`. This decision stems from the fact that median is a measure of central tendency that is more robust to outliers. Thus, we will be building two separate `SimpleImputer()` for each of mean and median strategies.

In [35]:
```
numerical_features.remove("creditLimit")
credit_feature = ["creditLimit"]
```

In [36]:
```
train_df, test_df = train_test_split(preprocess_df, test_size=0.2, random_st
X_train, y_train = train_df.drop(["isFraud"], axis=1), train_df["isFraud"]
X_test, y_test = test_df.drop(["isFraud"], axis=1), test_df["isFraud"]
```

In [37]:
```
ct = make_column_transformer(
    (
        make_pipeline(SimpleImputer(strategy="median"), StandardScaler()),
        numerical_features,
    ),
    (
        make_pipeline(SimpleImputer(strategy="mean"), StandardScaler()),
        credit_feature,
    ),
    (OneHotEncoder(drop="if_binary", handle_unknown="ignore"), categorical_f
    ("drop", drop_features)
)

transformed_df = ct.fit_transform(X_train)
transformed_df
```

Out[37]:
```
<629090x52 sparse matrix of type '<class 'numpy.float64'>'
        with 7827026 stored elements in Compressed Sparse Row format>
```

In [38]:
```
column_names = (
    numerical_features
    + credit_feature
    + ct.named_transformers_["onehotencoder"].get_feature_names_out().tolist
)
print(len(column_names))
```

```
52
```

In [39]:
```
transformed_X_train = pd.DataFrame(transformed_df.toarray(), columns=column_
transformed_X_train
```

Out[39]:

	availableMoney	transactionAmount	currentBalance	creditLimit	accountNum
0	-0.505378	-0.821477	-0.196817	-0.494840	
1	0.092375	-0.870402	-0.630626	-0.280071	
2	-0.164150	-0.311119	-0.665029	-0.494840	
3	-0.678751	1.888795	-0.577148	-0.838470	
4	-0.610822	2.160418	-0.438522	-0.709609	
...	...	...	...	...	
629085	-0.461043	-0.721733	-0.257650	-0.494840	
629086	0.854799	-0.686410	-0.517607	0.364235	
629087	-0.659082	0.067962	0.014087	-0.494840	
629088	-0.405422	-0.677478	-0.333970	-0.494840	
629089	-0.017018	-0.608794	0.678645	0.364235	

629090 rows × 52 columns

In [40]:
```
transformed_df_test = ct.transform(X_test)
transformed_X_test = pd.DataFrame(transformed_df_test.toarray(), columns=col
```

Modeling

For the modelling phase we will do cross validation on three different model algorithms. Logistic regression, random forest and gradient boosting classifier. We will first try logistic regression as it is the model with the least compexity. As it is a linear model it might fail to capture non-linear relationships between the features and our target variable. We will increase the complexity first with random forest and then gradient boosting classifier and check if using more complex models increase our test scores.

Out feature space is quite large as we have more than 50 features. But with more than 500,000 data points in our training set we think our models can handle 50+ features.

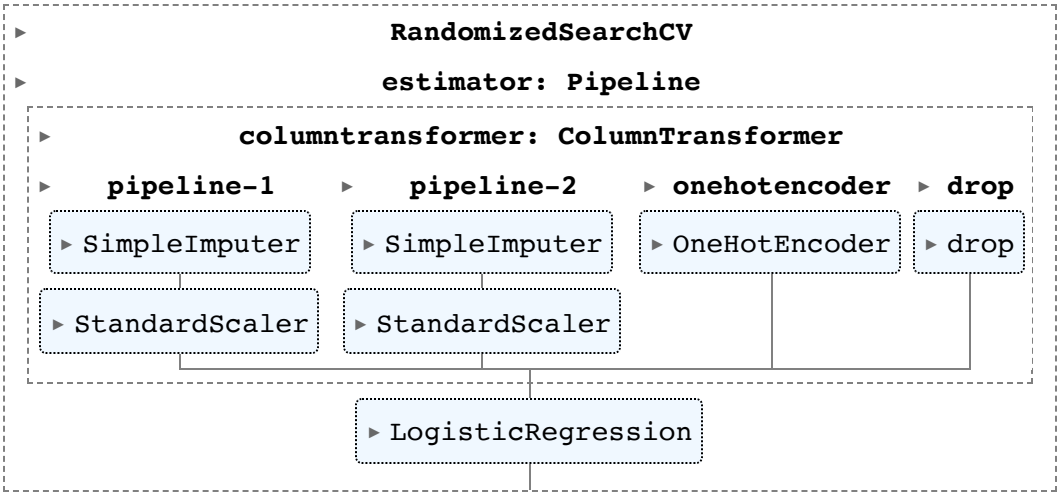
In [46]:
```
# helper function to display roc-auc scores and confusion matrices of train
def create_conf_matrix(clf, X_train, y_train, X_test, y_test):
    X_train_pred = clf.predict(X_train)
    conf_matrix = pd.crosstab(X_train_pred, y_train)
    conf_matrix.index.name = "prediction"
    print("Train confusion matrix:")
    display(conf_matrix)
    print("Train f1 score:",f1_score(X_train_pred, y_train))
    print("----")
    X_test_pred = clf.predict(X_test)
    conf_matrix2 = pd.crosstab(X_test_pred, y_test)
    conf_matrix2.index.name = "prediction"
    print("Test confusion matrix:")
    display(conf_matrix2)
    print("Test f1 score:",f1_score(X_test_pred, y_test))
```

Logistic Regression

In [42]:
```
param_dist_logreg = {"logisticregression__C": 10.0 ** np.arange(-2, 3),
                     "logisticregression__solver": ["newton-cholesky", "lbfg
logreg_search = run_model_with_random_search(
    X_train, y_train, numerical_features, credit_feature, categorical_featur
    drop_features, LogisticRegression, param_dist_logreg)
logreg_search.fit(X_train, y_train)
```

```
/Users/yiluo/miniconda3/envs/571/lib/python3.10/site-packages/sklearn/linear
_model/_logistic.py:460: ConvergenceWarning: lbfgs failed to converge (statu
s=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regre
ssion
  n_iter_i = _check_optimize_result(
/Users/yiluo/miniconda3/envs/571/lib/python3.10/site-packages/sklearn/linear
_model/_logistic.py:460: ConvergenceWarning: lbfgs failed to converge (statu
s=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regre
ssion
  n_iter_i = _check_optimize_result(
/Users/yiluo/miniconda3/envs/571/lib/python3.10/site-packages/sklearn/linear
_model/_logistic.py:460: ConvergenceWarning: lbfgs failed to converge (statu
s=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regre
ssion
  n_iter_i = _check_optimize_result(
/Users/yiluo/miniconda3/envs/571/lib/python3.10/site-packages/sklearn/linear
_model/_logistic.py:460: ConvergenceWarning: lbfgs failed to converge (statu
s=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regre
ssion
  n_iter_i = _check_optimize_result(
```

Out[42]:



In [43]:
```
pd.DataFrame(logreg_search.cv_results_)[["rank_test_score", 'mean_test_score
                                         "param_logisticregression__C", "par
                                         "mean_fit_time"]].sort_values("rank
```

Out[43]:

rank_test_score	mean_test_score	mean_train_score	param_logisticregression__C	pa
1	0.062224	0.062489	0.1	
2	0.062179	0.062460	0.1	
3	0.062164	0.062463	100.0	
4	0.062136	0.062454	1.0	
5	0.062121	0.062457	10.0	

In [44]:
```
create_conf_matrix(logreg_search, X_train, y_train, X_test, y_test)
```

```
Train confusion matrix:
```

prediction	isFraud False	isFraud True
False	407909	2825
True	211240	7116

```
Train f1 score: 0.06233984677853848
----
Test confusion matrix:
```

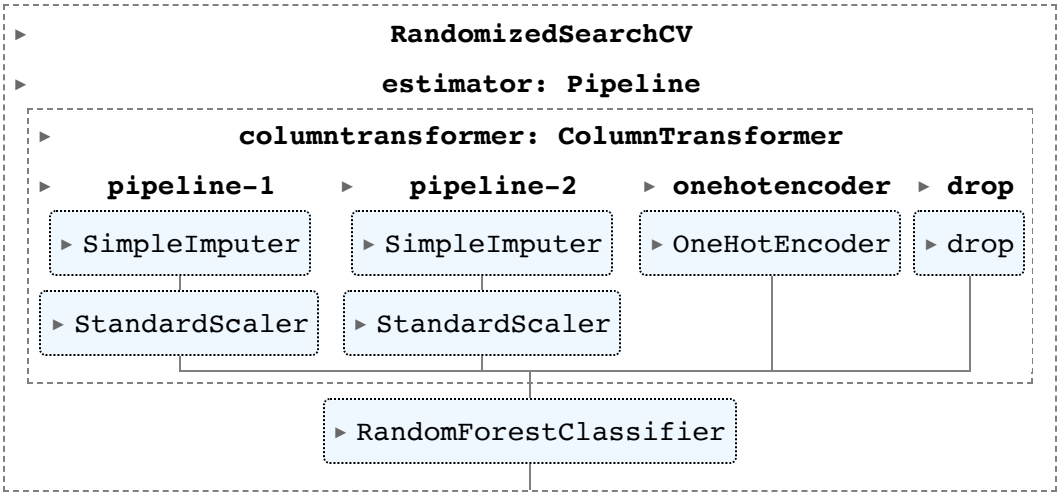
prediction	isFraud False	isFraud True
False	101601	722
True	53196	1754

```
Test f1 score: 0.06108731236722041
```

Random Forest Classifier

In [45]:
```
param_dist_rfclf = {"randomforestclassifier__n_estimators": 50 * np.array([1
                    "randomforestclassifier__max_depth": [5, 10, 20, None],
                    "randomforestclassifier__max_features": ['sqrt', 'log2']}
rfclf_search = run_model_with_random_search(
    X_train, y_train, numerical_features, credit_feature, categorical_featur
    drop_features, RandomForestClassifier, param_dist_rfclf)
rfclf_search.fit(X_train, y_train)
```

Out[45]:



In [46]:
```
pd.DataFrame(rfclf_search.cv_results_)[["rank_test_score", 'mean_test_score'
                                        "param_randomforestclassifier__n_es
                                        "param_randomforestclassifier__max_
                                        "mean_fit_time"]].sort_values("rank
```

Out[46]:

rank_test_score	mean_test_score	mean_train_score	param_randomforestclassifier__r
1	0.075337	0.082307	
2	0.075088	0.081377	
3	0.010174	0.998187	
4	0.009225	0.999799	
5	0.008297	0.998086	

In [47]:
```
create_conf_matrix(rfclf_search, X_train, y_train, X_test, y_test)
```

Train confusion matrix:

prediction	isFraud False	True
False	455564	3021
True	163585	6920

```
Train f1 score: 0.07669884619221264
----
Test confusion matrix:
```

isFraud	False	True
prediction		
False	113500	840
True	41297	1636

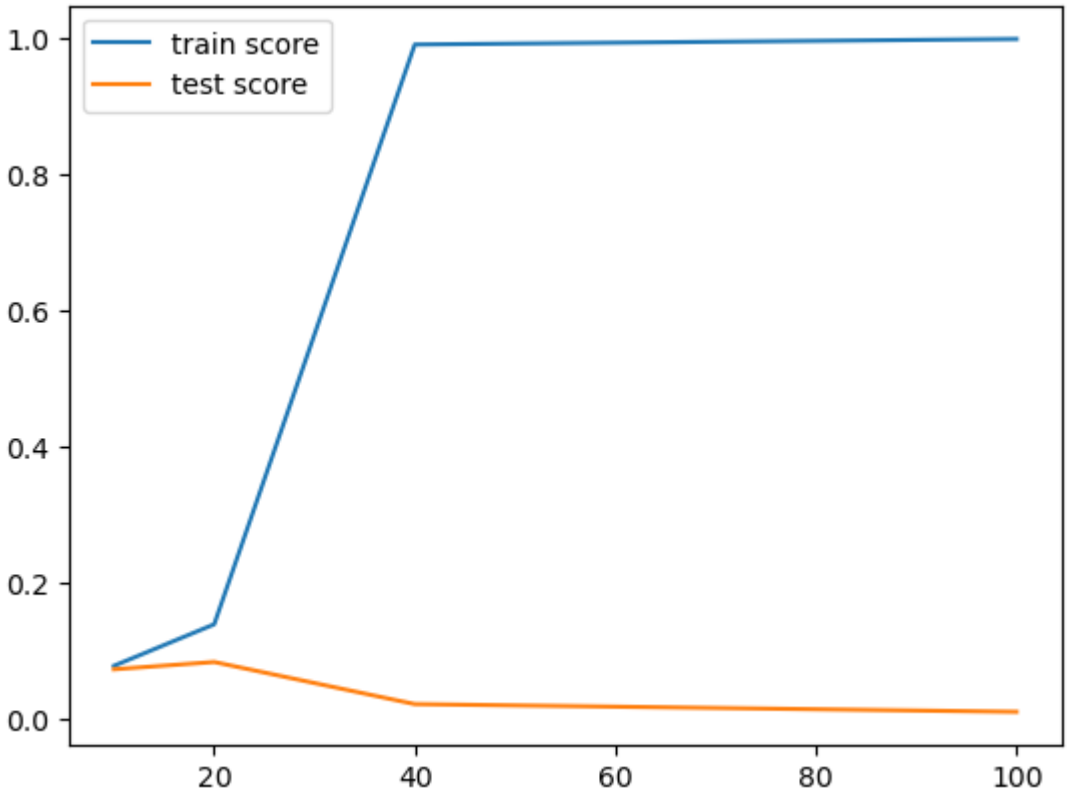
```
Test f1 score: 0.07205620031271334
```

In [48]:
```
train_scores = []
test_scores = []
max_depths = [10, 20, 40, 100, None]
for max_depth in max_depths:
    pipe = make_pipeline(ct, RandomForestClassifier(class_weight = "balanced
    pipe.fit(X_train, y_train)
    X_train_pred = pipe.predict(X_train)
    train_scores.append(f1_score(X_train_pred, y_train))
    X_test_pred = pipe.predict(X_test)
    test_scores.append(f1_score(X_test_pred, y_test))
    print(f"{max_depth} done")
```

```
10 done
20 done
40 done
100 done
None done
```

The plot below shows train and test scores as we increase model complexity by changing max_depth argument

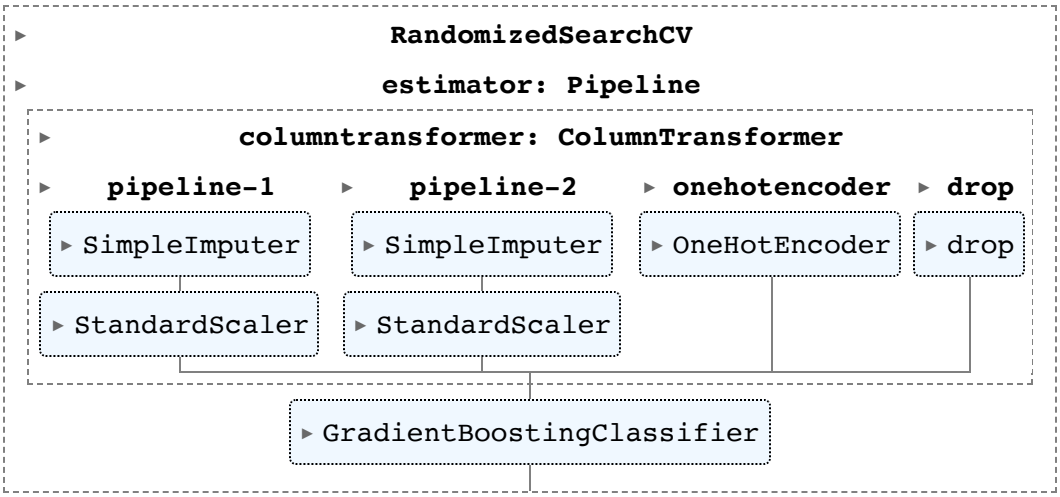
In [49]:
```
plt.plot(max_depths, train_scores, label='train score')
plt.plot(max_depths, test_scores, label='test score')
# Adding a legend
plt.legend()

# Display the plot
plt.show()
```



Gradient Boosting Classifier

In [43]:
```
param_dist_gbclf = {"gradientboostingclassifier__n_estimators": 50 * np.arra
                    "gradientboostingclassifier__max_depth": [3, 5, 10],
                    "gradientboostingclassifier__learning_rate": [0.05, 0.1,
gbclf_search = run_model_with_random_search(
    X_train, y_train, numerical_features, credit_feature, categorical_featur
    drop_features, GradientBoostingClassifier, param_dist_gbclf, need_class_
gbclf_search.fit(X_train, y_train)
```

Out[43]:



In [44]:
```
pd.DataFrame(gbclf_search.cv_results_)[["rank_test_score", 'mean_test_score'
                                        "param_gradientboostingclassifier__
```

```
                                                "param_gradientboostingclassifier__
                                                "mean_fit_time"]].sort_values("rank
```

Out[44]:

rank_test_score	mean_test_score	mean_train_score	param_gradientboostingclassifier
1	0.029037	0.854230	
2	0.005036	0.084095	
3	0.001205	0.003011	
4	0.001203	0.005612	
5	0.000603	0.001206	

In [47]:
```
create_conf_matrix(gbclf_search, X_train, y_train, X_test, y_test)
```

```
Train confusion matrix:
```

prediction	isFraud	False	True
False		619036	5607
True		113	4334

```
Train f1 score: 0.6024464831804281
----
Test confusion matrix:
```

prediction	isFraud	False	True
False		153956	2418
True		841	58

```
Test f1 score: 0.03437037037037037
```

Discussion

In our attempt to distinguish between fraud and non-fraud transactions in customer accounts, we experimented with three classification models: Logistic Regression, Random Forest Classifier, and Gradient Boost Classifier. However, we faced significant challenges in developing an effective model due to the extreme imbalance in our data.

1. Data Preprocessing and EDA Analysis

Data Preprocessing is the critical step before feeding the data into machine learning model. Before processing data, Exploratory Data Analysis allow us to examine our features and choose the right methods to impute missing values, process numerical features and transform categorical features. Many columns have more than half of value

counts as missing values so dropping these columns can enhance performance and prevent overfitting to ml models. Later, we applied one hot encoding to binary categorical features and label encoding to categorical featues with more than 2 values, preventing generating too many features and overfitting the ml models.

2. Imbalanced Data Hanlding Dealing imbalanced data can be a chanllenging problem as the number of non-fraudulent transactions far exceeds the number of fraudulent ones. Here are some strategies to handle imbalanced data in this context. Ml models ususally don't generalize well on imbalanced datasets. Balancing datasets before building ml models generally is consctructive in model learning. We used undersampling method to size down the majority class and created a balanced data for ml model buidling.

3. Model Selection and Evaluation

In the modeling phase, we employed cross-validation on three different classification algorithms: Logistic Regression, Random Forest, and Gradient Boosting Classifier. Each model was strategically chosen to incrementally increase in complexity. This approach allows us to assess whether more complex models improve our ability to detect fraud in customer transactions compared to the initial Logistic Regression model.

- Logistic Regression: We initiated with Logistic Regression due to its simplicity. However, being a linear model, it may struggle to capture non-linear relationships between features and the target variable.

- Random Forest: Next, we elevated the complexity with the Random Forest Classifier. This ensemble method is capable of capturing non-linear patterns through the combination of multiple decision trees.

- Gradient Boosting Classifier: The Gradient Boosting Classifier, chosen for its boosting technique, further increased model complexity. Boosting focuses on improving the weaknesses of previous models, potentially enhancing overall predictive performance.

- Logistic Regression Results:

We utilized Randomized Search to tune hyperparameters for Logistic Regression, including regularization strength (C) and solver options. The balanced class weights were applied to address the imbalanced nature of the data. However, Logistic Regression yielded limited success, as indicated by low f1 scores on both the training and test sets. Logistic Regression exhibits limited ability to capture fraud instances, emphasizing the need for enhanced model complexity.

- Random Forest Results:

Randomized Search was similarly employed to optimize hyperparameters for the Random Forest Classifier, considering the number of estimators, maximum depth, and maximum features. While the Random Forest Classifier exhibited an improvement in f1 scores compared to Logistic Regression, there is room for further enhancement.according to the results, our model is underfitting.

- Gradient Boosting Classifier Results:

We extended our exploration to the Gradient Boosting Classifier, adjusting hyperparameters such as the number of estimators, maximum depth, and learning rate. The Gradient Boost Classifier shows promising results on the training set with a high f1 score of 0.872. However, its performance on the test set drops significantly (f1 score of 0.0386), indicating potential overfitting.

- conclusion: According to the results, we would prefer to choosing random forest model because it achieves the highest F1 score on test dataset. However it is still too low to apply.And for gradient boosting classifier, it achieves a good result on training data while a bad result on testing result, making it not a reliable model.

4. Possible Reasons for Our Model Results:

The extremely imbalanced nature of the data, where fraud instances are rare compared to non-fraud cases, posed a substantial challenge. Imbalance can lead to models favoring the majority class, resulting in low sensitivity and f1 scores for the minority class (fraud).

5. Future Steps and Improvements:

In conclusion, our initial exploration indicates the need for further model refinement to effectively distinguish between fraud and non-fraud transactions. The suggestions outlined above aim to address current limitations and pave the way for improved fraud detection capabilities in our classification mode Other resampling methods can be tried: Resampling Techniques: Oversampling the Minority Class: Increase the number of samples in the minority class (fraudulent transactions) by duplicating them or using techniques like SMOTE (Synthetic Minority Over-sampling Technique) which creates synthetic samples.

- Feature Engineering:

Consider incorporating frequency encoding to address the class imbalance. This technique assigns weights to different classes based on their frequency, potentially improving model performance. Exploring New Features and Encoding Techniques: In addition to the models and strategies discussed, we are planning to explore the implementation of frequency encoding, particularly focusing on the 'merchant_name' feature. This approach holds potential for being a significant predictor of fraud. Frequency encoding will involve assigning weights to merchant names based on their

occurrence frequency, which could highlight repetitive fraudulent patterns associated with specific merchants. By integrating this technique, we aim to enhance the model's sensitivity towards subtle cues of fraud that might be overlooked by traditional methods. This addition is expected to provide a more nuanced understanding of the transactional data, thereby improving our model's capability to detect fraud more accurately and efficiently. The integration of such feature-specific techniques, combined with our existing models, could pave the way for a more robust and effective fraud detection system.

- Exploring New Features and Encoding Techniques: In addition to the models and strategies discussed, we are planning to explore the implementation of frequency encoding, particularly focusing on the 'merchant_name' feature. This approach holds potential for being a significant predictor of fraud. Frequency encoding will involve assigning weights to merchant names based on their occurrence frequency, which could highlight repetitive fraudulent patterns associated with specific merchants. By integrating this technique, we aim to enhance the model's sensitivity towards subtle cues of fraud that might be overlooked by traditional methods. This addition is expected to provide a more nuanced understanding of the transactional data, thereby improving our model's capability to detect fraud more accurately and efficiently. The integration of such feature-specific techniques, combined with our existing models, could pave the way for a more robust and effective fraud detection system. m. .

- Adjust Threshold: We should vary the classification threshold. By default, many models use a threshold of 0.5 (probability greater than or equal to 0.5 for the positive class). We can experiment with thresholds like 0.4, 0.3, or others to observe how it affects the trade-off between precision and recall.

- Scoring Metrics Adjustment:

In light of our research question, where recall holds significance, it is advisable to prioritize its utilization for result analysis. However, acknowledging the potential escalation of false negatives, we must also account for precision. To strike a balance, we can assign different weights to these metrics in our scoring system. Experimenting with alternative scoring metrics, such as precision-recall curves or the area under the precision-recall curve, proves to be more fitting for imbalanced datasets. i* Exploring Alternative Loss Functions for Imbalanced Fraud Detection: In addressing the challenge of imbalanced datasets in fraud detection, it's crucial to consider the implementation of alternative loss functions tailored to this specific context. Loss functions such as 'Weighted Cross-Entropy' or 'Focal Loss' have shown promise in such scenarios. Weighted Cross-Entropy adjusts the loss based on the class frequencies, giving more weight to the minority class, thereby emphasizing the importance of correctly predicting fraud cases. Focal Loss, on the other hand, focuses on hard-to-classify examples and down-weighs the loss for well-classified cases, which can be particularly useful in

distinguishing between fraud and non-fraud transactions. The choice of loss function significantly impacts the model's ability to learn from the underrepresented class and should be carefully selected based on the specific characteristics and requirements of the fraud detection task. Experimentation with these loss functions can lead to better handling of the imbalanced nature of fraud data, potentially enhancing the model's effectiveness in identifying fraudulent activities

- Adopting AUC-ROC Curve Metrics for Enhanced Model Selection: In our ongoing efforts to refine our model selection process, we are considering the adoption of the AUC-ROC (Area Under the Receiver Operating Characteristics) curve metrics as a more comprehensive measure for evaluating our models. The AUC-ROC curve is particularly advantageous in imbalanced datasets like ours, as it provides a clear picture of how well the model distinguishes between the two classes (fraud and non-fraud) across different thresholds. Unlike the F1 score, which primarily focuses on the balance between precision and recall at a specific threshold, the AUC-ROC curve evaluates model performance over a range of threshold values, offering a more holistic view of its capability. This approach allows us to better understand the trade-offs between true positive rates and false positive rates, making it a crucial tool in the fine-tuning of our models for optimal fraud detection. By transitioning to AUC-ROC curve metrics, we aim to achieve a more nuanced and effective model evaluation, thereb

6. . 4Model Complexity and resource:

Explore more complex models or ensemble methods that can capture intricate patterns in the data. Deep learning approaches, such as neural networks, may be worth investing. However, resources should be prioritized when productionizing ml models.ransactions.

References

Capital One. (2018). Capital One Data Science Challenge. In CapitalOneRecruiting GitHub Repository. https://github.com/CapitalOneRecruiting/DS

Python Software Foundation. Python Language Reference, version 3.11.6. Available at http://www.python.org

Timbers, T., Lee, M. & Ostblom, J. (2023). Breast Cancer Predictor. https://github.com/ttimbers/breast_cancer_predictor_py/tree/0.0.1

Pedregosa, F., Varoquaux, Ga"el, Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... others. (2011). Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12(Oct), 2825–2830

McKinney, W., & others. (2010). Data structures for statistical computing in python. In Proceedings of the 9th Python in Science Conference (Vol. 445, pp. 51–56).

Caporal, J. (2023). Identity Theft and Credit Card Fraud Statistics for 2023. In the Ascent. https://www.fool.com/the-ascent/research/identity-theft-credit-card-fraud-statistics/