# Table of Contents



# Part 1: Introduction


## Background

Fraudulent transactions in financial payments represent a significant challenge for the global financial industry. As digital payments become increasingly prevalent, the sophistication and frequency of fraud attempts have surged, posing a substantial threat to consumers, businesses, and financial institutions. According to a report by the Association of Certified Fraud Examiners (ACFE), businesses worldwide lose an estimated 5% of their annual revenues to fraud, amounting to a staggering $4.5 trillion globally [[1]](https://legacy.acfe.com/report-to-the-nations/2020/). In addition, a study by Juniper Research highlighted that online payment fraud losses were expected to grow from \$22 billion in 2020 to over $48 billion by 2023, driven by the increasing adoption of online and mobile payment methods.

Fraudulent transactions erode consumer trust in financial systems and impact the financial health of business. Ensuring robust fraud prevention mechanisms is essential to maintain confidence in digital payments and protect consumers from financial loss and identity theft. It will also help in safeguarding the financial stability of small and medium size businesses.

## Data

#### Context

BankSim is an agent-based simulator of bank payments based on a sample of aggregated transactional data provided by a bank in Spain. The main purpose of BankSim is the generation of synthetic data that can be used for fraud detection research. Statistical and a Social Network Analysis (SNA) of relations between merchants and customers were used to develop and calibrate the model. Our ultimate goal is for BankSim to be usable to model relevant scenarios that combine normal payments and injected known fraud signatures. The data sets generated by BankSim contain no personal information or disclosure of legal and private customer transactions. Therefore, it can be shared by academia, and others, to develop and reason about fraud detection methods. Synthetic data has the added benefit of being easier to acquire, faster and at less cost, for experimentation even for those that have access to their own data. We argue that BankSim generates data that usefully approximates the relevant aspects of the real data.

#### Content

We ran BankSim for 180 steps (approx. six months), several times and calibrated the parameters in order to obtain a distribution that get close enough to be reliable for testing. We collected several log files and selected the most accurate. We injected thieves that aim to steal an average of three cards per step and perform about two fraudulent transactions per day. We produced 594643 records in total. Where 587443 are normal payments and 7200 fraudulent transactions. Since this is a randomised simulation the values are of course not identical to original data.

# Part 2: Exploratory Data Analysis

First thing first, data is everything. If I don't know the data, then I won't be able to produce good analysis result. So, before I do anything fancy, I would like examine each column and understand the characteristics of them. Later, I will perform some bivariate analysis and take a closer look to see how these variables interplay with one another.

In [ ]:
```
# This Python 3 environment comes with many helpful analytics libraries installed
# It is defined by the kaggle/python Docker image: https://github.com/kaggle/docker-py
# For example, here's several helpful packages to load
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

# Input data files are available in the read-only "../input/" directory
# For example, running this (by clicking run or pressing Shift+Enter) will list all fi

import os
```

```
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))

# You can write up to 20GB to the current directory (/kaggle/working/) that gets prese
# You can also write temporary files to /kaggle/temp/, but they won't be saved outside
```

In [ ]:

```
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats import chi2_contingency
import sklearn
```

In [ ]:

```
banksim = pd.read_csv('/kaggle/input/banksim1/bs140513_032310.csv')
```

In [5]:

```
banksim.head()
```

Out[5]:

| | step | customer | age | gender | zipcodeOri | merchant | zipMerchant | category | amou |
|---|---|---|---|---|---|---|---|---|---|
| **0** | 0 | 'C1093826151' | '4' | 'M' | '28007' | 'M348934600' | '28007' | 'es_transportation' | 4 |
| **1** | 0 | 'C352968107' | '2' | 'M' | '28007' | 'M348934600' | '28007' | 'es_transportation' | 39 |
| **2** | 0 | 'C2054744914' | '4' | 'F' | '28007' | 'M1823072687' | '28007' | 'es_transportation' | 26 |
| **3** | 0 | 'C1760612790' | '3' | 'M' | '28007' | 'M348934600' | '28007' | 'es_transportation' | 17 |
| **4** | 0 | 'C757503768' | '5' | 'M' | '28007' | 'M348934600' | '28007' | 'es_transportation' | 35 |

In [6]:

```
banksim.shape
```

Out[6]:

```
(594643, 10)
```

In [7]:

```
banksim.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 594643 entries, 0 to 594642
Data columns (total 10 columns):
 #   Column       Non-Null Count   Dtype  
---  ------       --------------   -----  
 0   step         594643 non-null  int64  
 1   customer     594643 non-null  object 
 2   age          594643 non-null  object 
 3   gender       594643 non-null  object 
 4   zipcodeOri   594643 non-null  object 
 5   merchant     594643 non-null  object 
 6   zipMerchant  594643 non-null  object 
 7   category     594643 non-null  object 
 8   amount       594643 non-null  float64
 9   fraud        594643 non-null  int64  
dtypes: float64(1), int64(2), object(7)
memory usage: 45.4+ MB
```

There are no missing values in this dataset.

## 2.1 Univariate Analysis

In [8]:
```python
# unique values in age
banksim['age'].unique()
```

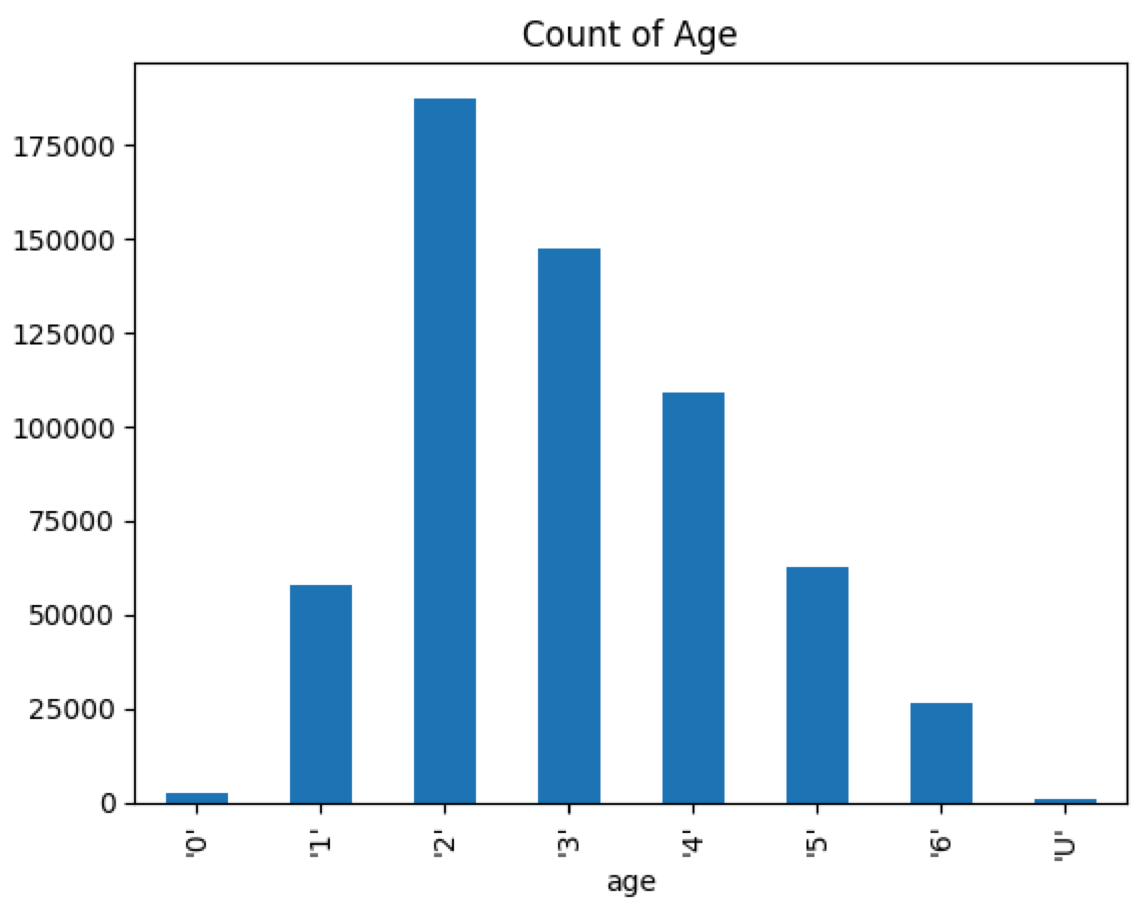
Out[8]:
```
array(["'4'", "'2'", "'3'", "'5'", "'1'", "'6'", "'U'", "'0'"],
      dtype=object)
```

According to the paper, each age number is defined as follows:

| age | Rank |
|---|---|
| 0 | <=18 |
| 1 | 19-25 |
| 2 | 26-35 |
| 3 | 36-45 |
| 4 | 46-55 |
| 5 | 56-65 |
| 6 | >65 |
| U | Unknown |

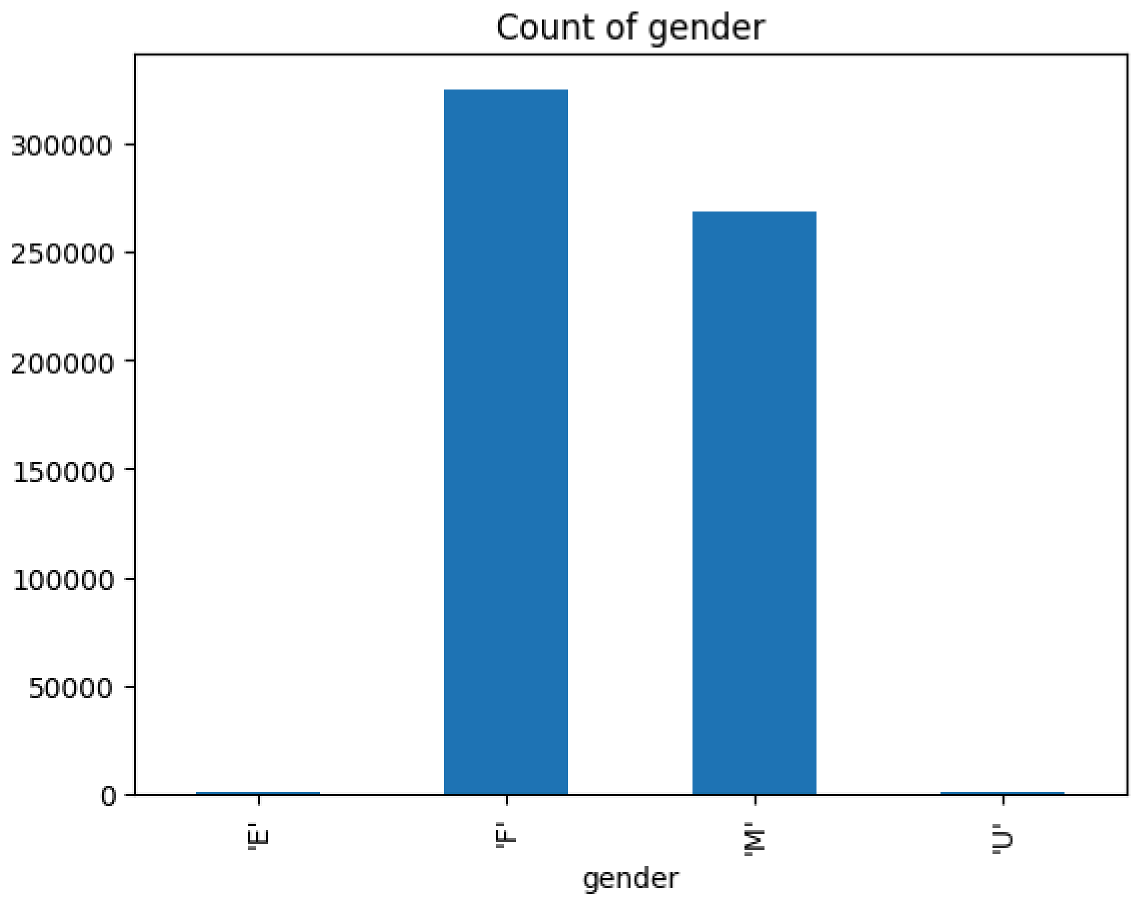
In [9]:
```python
# distribution of age variable
banksim.groupby(['age']).size().plot(kind = 'bar', title='Count of Age')
```

Out[9]:
```
<Axes: title={'center': 'Count of Age'}, xlabel='age'>
```



In [10]:
```
# distribution of gender variable
banksim.groupby(['gender']).size().plot(kind = 'bar', title='Count of gender')
```

Out[10]: <Axes: title={'center': 'Count of gender'}, xlabel='gender'>




According to the paper, gender could be classified as follows:

| idGender | Description |
|---|---|
| E | ENTERPRISE |
| F | FEMALE |
| M | MALE |
| U | UNKNOWN |

In [11]:
```
banksim['gender'].value_counts(normalize = True)
```

Out[11]:
```
gender
'F'    0.545815
'M'    0.451338
'E'    0.001981
'U'    0.000866
Name: proportion, dtype: float64
```

In [12]:
```
# unique values of the zip location where the transactions happened
banksim['zipcodeOri'].unique()
```

Out[12]:
```
array(["'28007'"], dtype=object)
```

In [13]:
```python
# unique values of the zip location of the merchants
banksim['zipMerchant'].unique()
```

Out[13]:
```
array(["'28007'"], dtype=object)
```

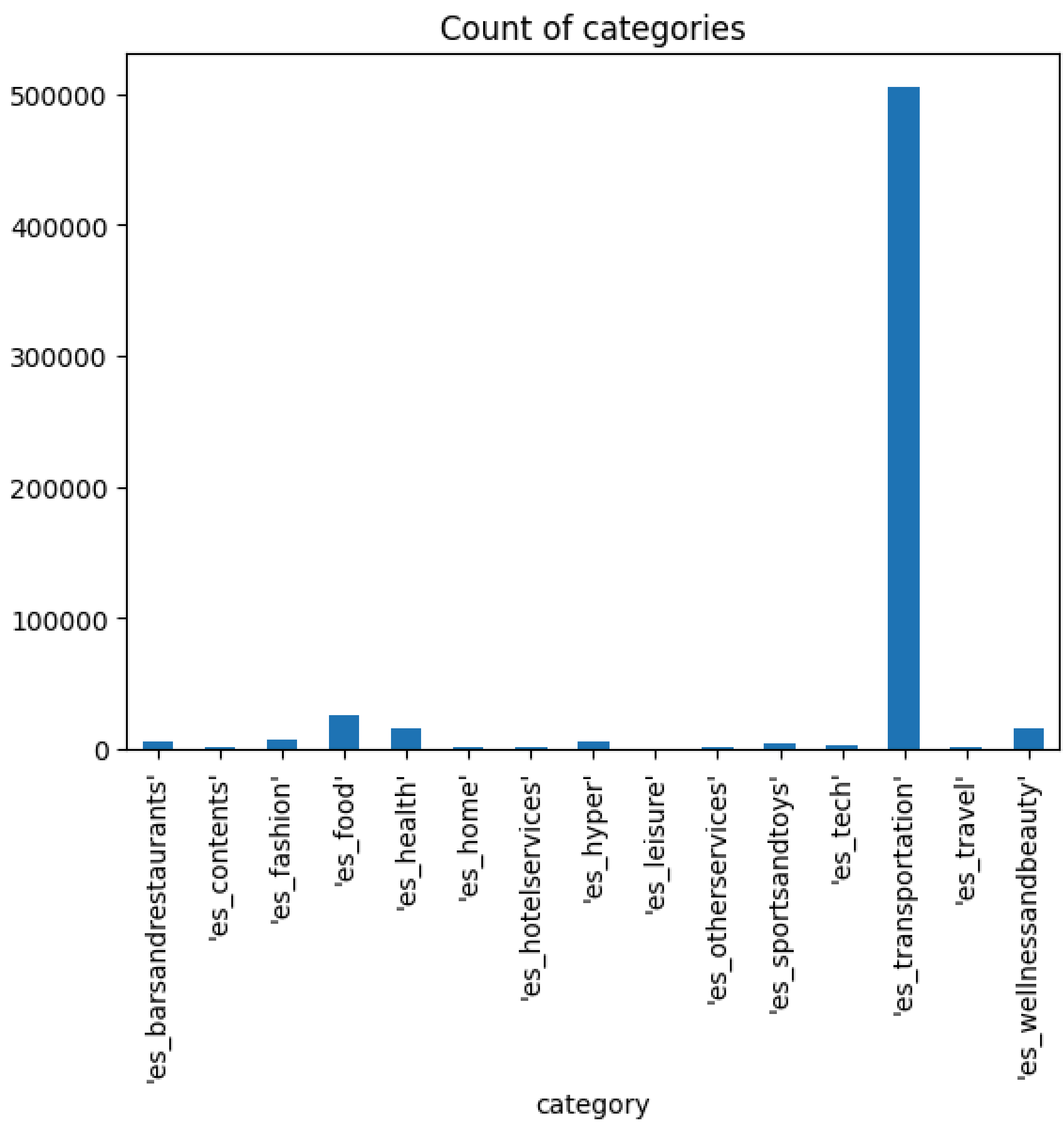
So, all the transactions happened in one location.

In [14]:
```python
# how many categories of transactions are there
banksim['category'].unique()
```

Out[14]:
```
array(["'es_transportation'", "'es_health'", "'es_otherservices'",
       "'es_food'", "'es_hotelservices'", "'es_barsandrestaurants'",
       "'es_tech'", "'es_sportsandtoys'", "'es_wellnessandbeauty'",
       "'es_hyper'", "'es_fashion'", "'es_home'", "'es_contents'",
       "'es_travel'", "'es_leisure'"], dtype=object)
```

In [15]:
```python
# distribution of transaction categories
banksim.groupby(['category']).size().plot(kind = 'bar', title='Count of categories')
```

Out[15]:
```
<Axes: title={'center': 'Count of categories'}, xlabel='category'>
```



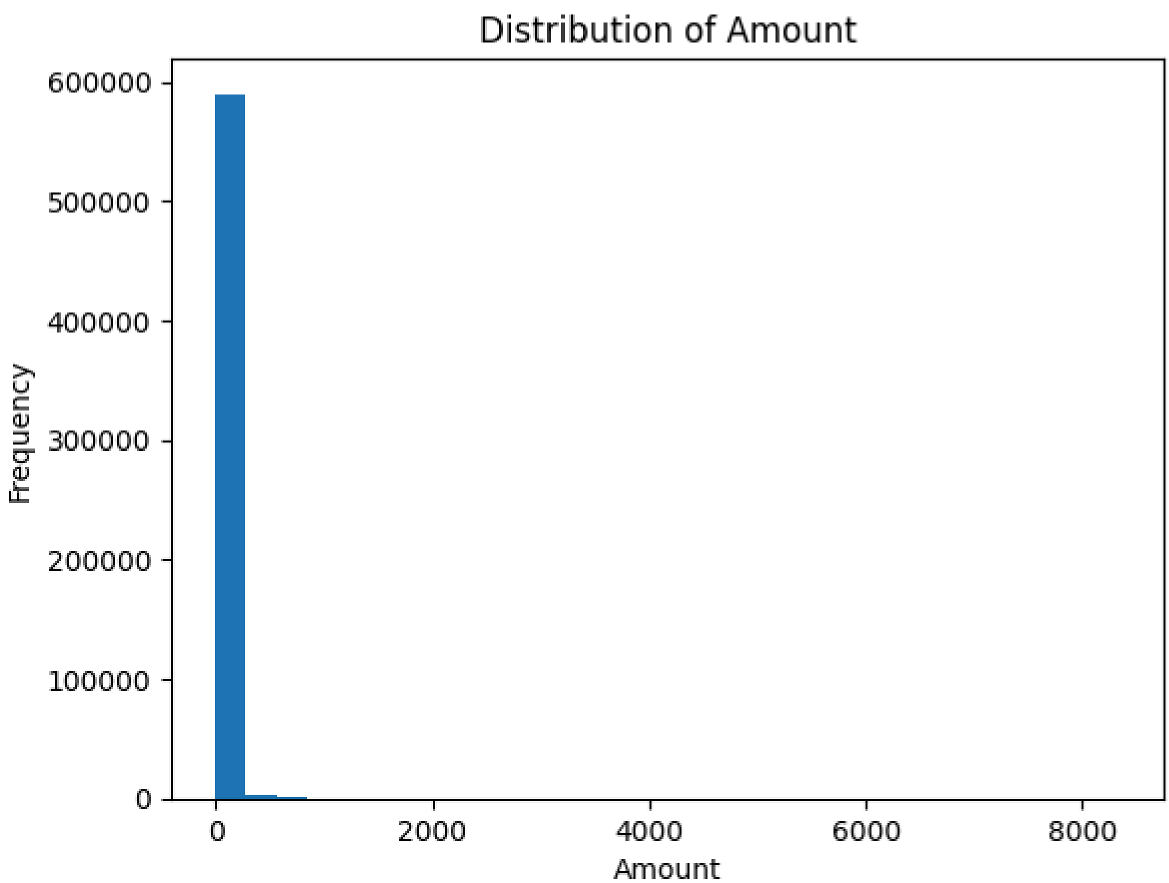
In [16]:
```
print(banksim['amount'].max())
print(banksim['amount'].min())
```

```
8329.96
0.0
```

In [17]:
```
banksim['amount'].max() - banksim['amount'].min()
```

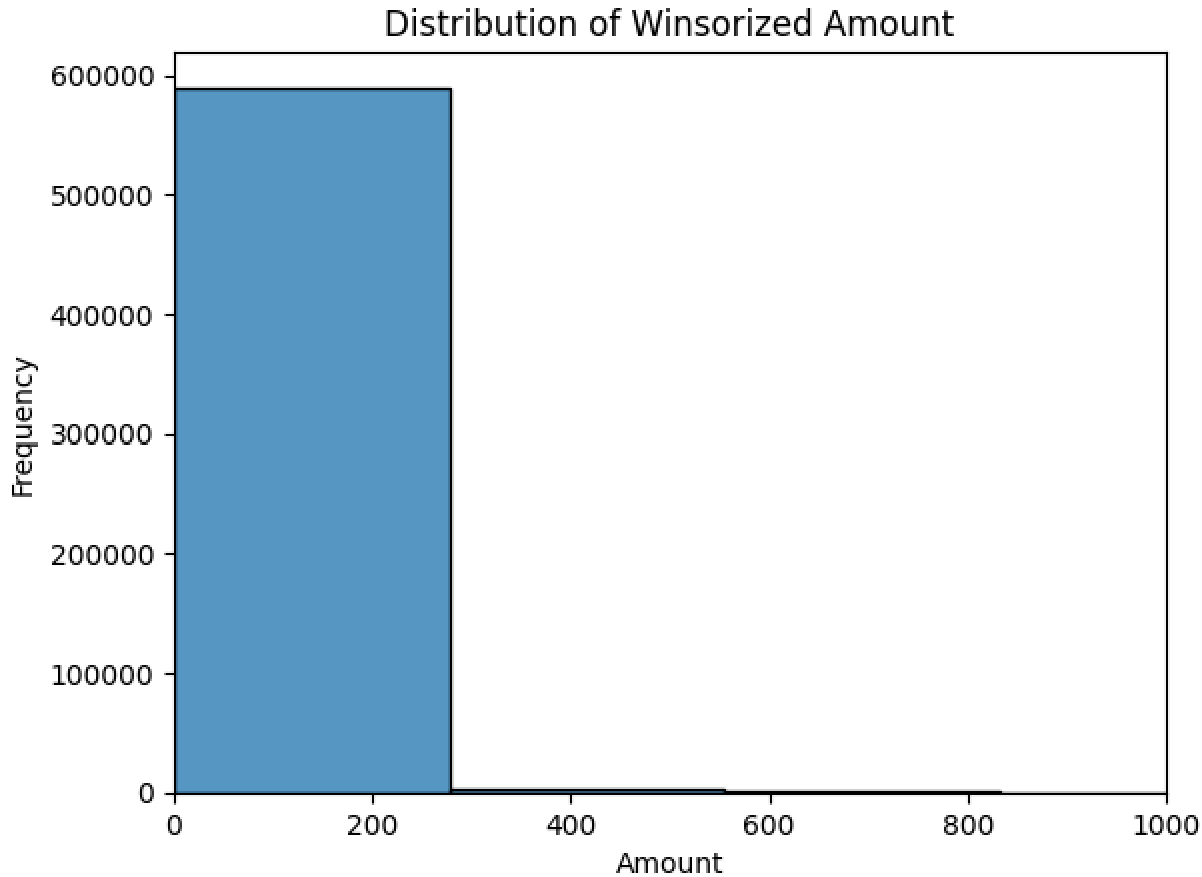
Out[17]:
```
8329.96
```

In [18]:
```
# Distribution of amount
banksim['amount'].plot(kind='hist', bins=30, title='Distribution of Amount')
plt.xlabel('Amount')
plt.ylabel('Frequency')
plt.show()
```



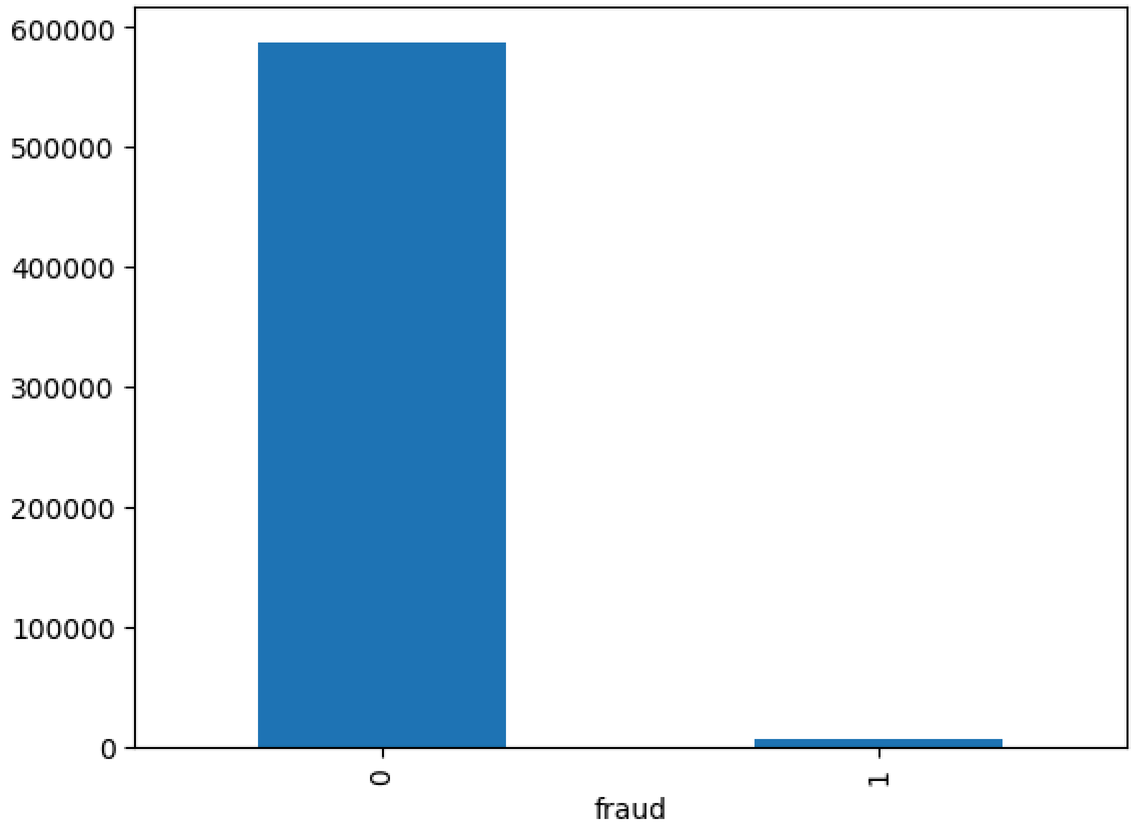

In [19]:
```
# Distribution of amount (truncated version)

ax = sns.histplot(banksim['amount'], bins=30)
ax.set_xlim(0, 1000)  # Set x-axis limits
plt.title('Distribution of Winsorized Amount')
plt.xlabel('Amount')
plt.ylabel('Frequency')
plt.show()
```

```
/opt/conda/lib/python3.10/site-packages/seaborn/_oldcore.py:1119: FutureWarning: use_
inf_as_na option is deprecated and will be removed in a future version. Convert inf v
alues to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
```




In [20]:
```
# percentage of transaction: genuine vs. fraud
banksim['fraud'].value_counts().plot(kind = 'bar')
```

Out[20]:
```
<Axes: xlabel='fraud'>
```




In [21]:
```
banksim['fraud'].value_counts(normalize = True)
```

Out[21]:
```
fraud
0    0.987892
1    0.012108
Name: proportion, dtype: float64
```

In [22]:
```
# percentage of fraud transaction in banksim
banksim.loc[banksim['fraud'] == 1, 'amount'].sum()/banksim['amount'].sum()
```

Out[22]:
```
0.16966195778993912
```

In [23]:
```
# average amount of fraud transaction
banksim.loc[banksim['fraud'] == 1, 'amount'].sum()/banksim.loc[banksim['fraud'] == 1].
```

Out[23]:
```
530.9265513888889
```

## Univariate Analysis Conclusion:

Observation:

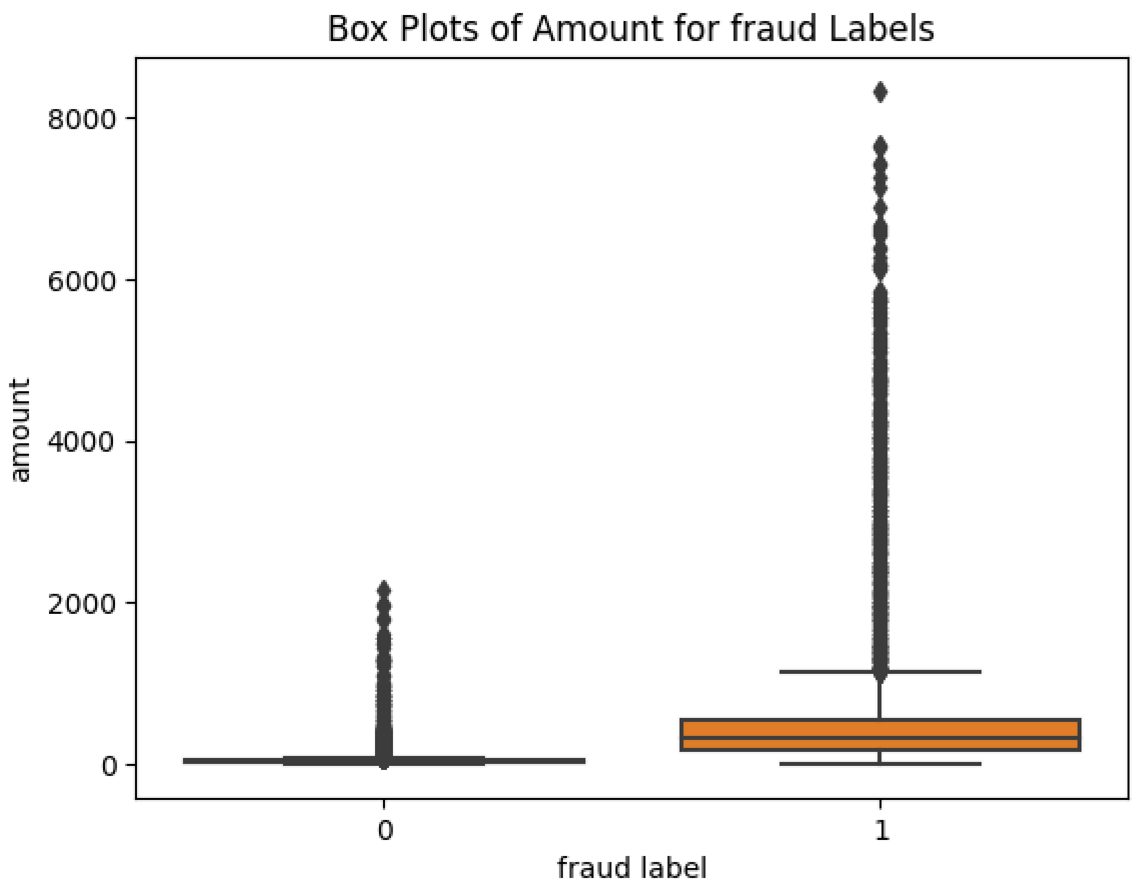
1. `age` : Age groups are encoded into this variable and represented by a single number.
2. `gender` : The proportion of male and female are close ('M': 54%, 'F': 45%), but there are some other values like 'E' for 'Enterprise' and 'U' for unknown.
3. `zip` : All the transactions happened in the same location.
4. `category` : most of the transactions are related to transportation in this dataset.
5. `Amount` : There are some outliers in this variable, and most of the values are less than $500.

6. `Fraud` : Over 98% of the transactions are genuine and less than 2% of transactions are fraud.
7. Amount in fraud transactions takes up to 17% of the total transaction amount in this banksim dataset. Also, the average amount of fraud transaction is around $530.

## 2.2 Bivariate Analysis

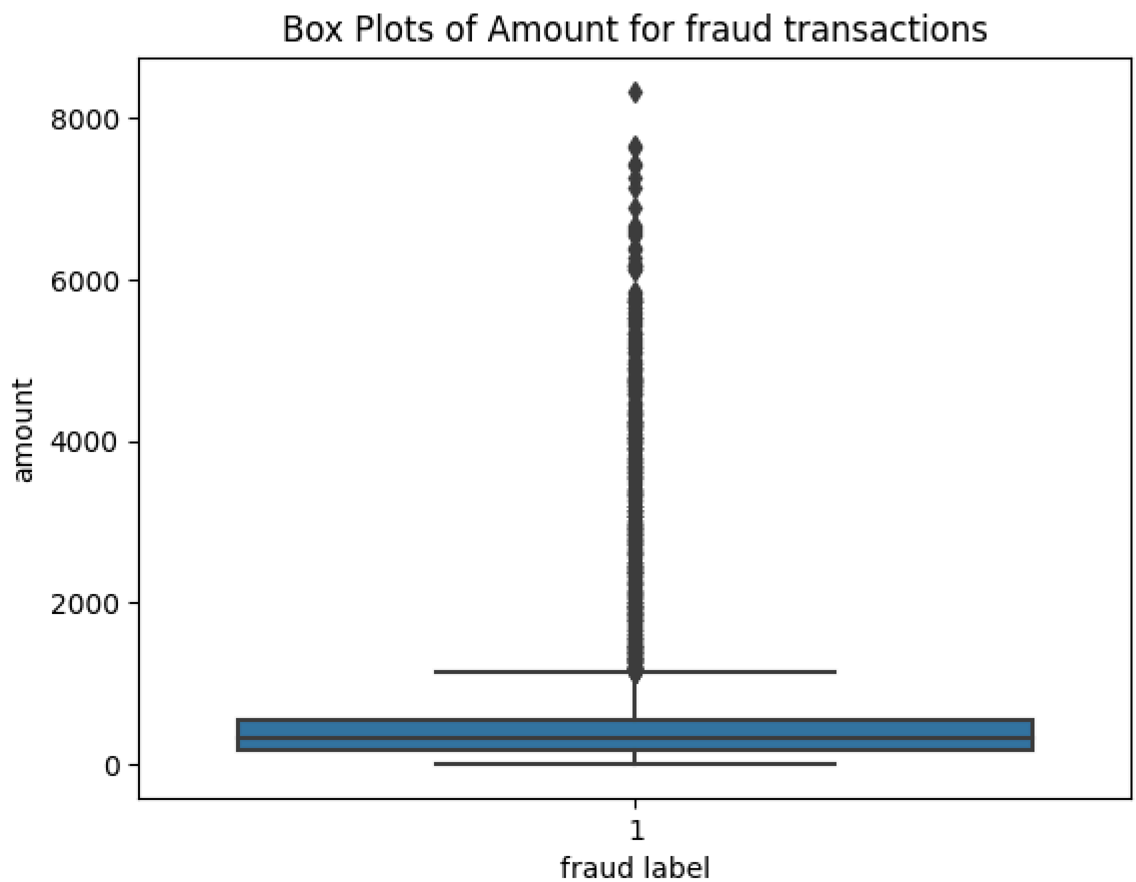
In [24]:

```
# Check distribution of `amount` for genuine and fraud transactions
sns.boxplot(x='fraud', y='amount', data=banksim)
plt.title('Box Plots of Amount for fraud Labels')
plt.xlabel('fraud label')
plt.ylabel('amount')
plt.show()
```




In [25]:

```
sns.boxplot(x='fraud', y='amount', data=banksim[banksim['fraud'] == 1])
plt.title('Box Plots of Amount for fraud transactions')
plt.xlabel('fraud label')
plt.ylabel('amount')
plt.show()
```




Observation: Looks like most of the fraud transactions are less than $1500. However, there are some transactions that include large amounts and need more investigation.

In [26]:
```
# fraud transaction across categories
banksim[banksim['fraud'] == 1].groupby(by = ['fraud', 'category']).size()
```

Out[26]:
```
fraud  category
1      'es_barsandrestaurants'     120
       'es_fashion'                116
       'es_health'                1696
       'es_home'                   302
       'es_hotelservices'          548
       'es_hyper'                  280
       'es_leisure'                474
       'es_otherservices'          228
       'es_sportsandtoys'         1982
       'es_tech'                   158
       'es_travel'                 578
       'es_wellnessandbeauty'      718
dtype: int64
```

Fraud transactions are more frequent in health and sportsandtoys.

In [27]:
```
# fraud transactions among gender
banksim.groupby(by = ['fraud', 'gender']).size()
```

Out[27]:
```
fraud  gender
0      'E'         1171
       'F'       319807
       'M'       265950
       'U'          515
1      'E'            7
       'F'         4758
       'M'         2435
dtype: int64
```

Looks like fraud transactions are more frequent among female customers' transactions.

In [28]:
```
set(banksim['customer']).intersection(set(banksim['merchant']))
```

Out[28]:
```
set()
```

Observation: There is no overlap between customer and merchant.

In [29]:
```
# Test the independency between category and fraud
fraud_cat_crosstab = pd.crosstab(banksim['category'], banksim['fraud'])
chi2, p, dof, expected = chi2_contingency(fraud_cat_crosstab)
print(chi2, p, dof, expected )
```

```
193862.64201580346 0.0 14 [[6.29583505e+03 7.71649544e+01]
 [8.74284327e+02 1.07156731e+01]
 [6.37585429e+03 7.81457110e+01]
 [2.59361138e+04 3.17886194e+02]
 [1.59376599e+04 1.95340061e+02]
 [1.96195330e+03 2.40466969e+01]
 [1.72288346e+03 2.11165355e+01]
 [6.02416477e+03 7.38352255e+01]
 [4.92958056e+02 6.04194449e+00]
 [9.00957408e+02 1.10425919e+01]
 [3.95354336e+03 4.84566370e+01]
 [2.34130379e+03 2.86962093e+01]
 [4.99002966e+05 6.11603399e+03]
 [7.19185299e+02 8.81470059e+00]
 [1.49033371e+04 1.82662875e+02]]
```

Looks like there is association between `category` and `fraud`

## Bivariate Analysis Conclusion:

1. Most of the fraud transactions are less than $1500. However, there are some transactions that include large amounts and need more investigation. (To-do)
2. Fraud transactions are more frequent in **health** and **sportsandtoys**.
3. Fraud transactions targets more frequency on female consumers.
4. Most of the transactions are customers payment to merchants, while a small portion of the transactions are initiated by enterprises.
5. There is a association between `category` and `fraud`.

# Part 3： Feature Engineering

To aviod data leakage, I will split the data first and then do feature transformation. Here's the step I will perform:

1. Separate feature set and target variable first;
2. Perform train-test split to obtain two datasets;
3. For each train and test set, perform feature transformation separately

In [30]:
```
from sklearn.preprocessing import StandardScaler, RobustScaler
from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedKFold
from sklearn.preprocessing import OneHotEncoder
from sklearn.model_selection import KFold
from imblearn.over_sampling import SMOTE
```

In [31]:
```
# create feature set and dependent variable
X = banksim.drop(['customer', 'merchant', 'zipcodeOri', 'zipMerchant', 'fraud'], axis
y = banksim['fraud']
```

I drop `zip` in the feature set because there is only one value in both `zip` columns and they are the same.

In [32]:
```
# stratified random sampling to make sure the proportion of imbalanced classes are mai
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25, stratify =

X_train_archive = X_train.copy()
```

In [33]:
```
# check the proportion of different classes
print(banksim['fraud'].value_counts()[0]/banksim['fraud'].value_counts()[1])
print(sum(y_train == 0)/sum(y_train == 1))
print(sum(y_test == 0)/sum(y_test == 1))
```

```
81.58930555555555
81.58925925925926
81.58944444444444
```

In [34]:
```
def OneHotEncoding(df, enc, categories):
    """
    A helper function that applies one-hot encoding for categories variables.
    """
    transformed = pd.DataFrame(enc.transform(df[categories]).toarray(),
                               columns = enc.get_feature_names_out(categories))
    return pd.concat([df.reset_index(drop=True), transformed], axis=1).drop(categories
```

In [35]:
```
# apply one-hot encoding to categorical variables
categories = ['age', 'gender']
enc_ohe = OneHotEncoder()
enc_ohe.fit(X_train[categories])
```

Out[35]:

```
▾ OneHotEncoder
OneHotEncoder()
```

In [36]:
```
# transform both train and test set
X_train = OneHotEncoding(X_train, enc_ohe, categories)
X_test = OneHotEncoding(X_test, enc_ohe, categories)
```

In [37]:
```
std_scaler = StandardScaler()
rob_scaler = RobustScaler()
```

In [38]:
```
# for amount use robust scaler since there are outliers
X_train['scaled_amount'] = rob_scaler.fit_transform(X_train['amount'].values.reshape(-
X_test['scaled_amount'] = rob_scaler.fit_transform(X_test['amount'].values.reshape(-1,

# as the steps are mostly linear here, I use standard scaler
X_train['scaled_step'] = std_scaler.fit_transform(X_train['step'].values.reshape(-1,1)
X_test['scaled_step'] = std_scaler.fit_transform(X_test['step'].values.reshape(-1,1))
X_train.drop(['amount', 'step'], axis = 1, inplace = True)
X_test.drop(['amount', 'step'], axis = 1, inplace = True)
```

In [39]:
```
# Based on our observation of the transaction `category`, frequency information plays
# this column and our target variable. Thus, I will perform WoE transform.

def calculate_smooth_woe(feature_set, feature, target_var, alpha = 1, beta = 2):
    df = feature_set.copy()
    df['target'] = target_var.copy()

    total_pos = target_var.sum()
    total_neg = target_var.count() - total_pos

    # Group by the feature and calculate counts
    grouped = df.groupby(feature).agg({'target': ['sum', 'count']})
    grouped.columns = ['Pos', 'Total']
    grouped['Neg'] = grouped['Total'] - grouped['Pos']

    # Calculate WoE
    grouped['WoE'] = np.log((grouped['Pos'] + alpha / total_pos + beta) / (grouped['Ne

    return grouped[['WoE']].reset_index()
```

In [40]:
```
# obtain the woe mapping table for category variable
woe_table_train = calculate_smooth_woe(X_train, 'category', y_train)
woe_table_test = calculate_smooth_woe(X_test, 'category', y_test)
```

In [41]:
```
# replace category with category_woe
X_train = pd.merge(X_train, woe_table_train, left_on = 'category', right_on = 'categor
X_train = X_train.drop(['category'], axis = 1)
X_train = X_train.rename(columns = {'WoE': 'category_woe'})

X_test = pd.merge(X_test, woe_table_test, left_on = 'category', right_on = 'category',
X_test = X_test.drop(['category'], axis = 1)
X_test = X_test.rename(columns = {'WoE': 'category_woe'})
```

In [42]:
```
# Apply SMOTE to the training data
smote = SMOTE(random_state=42)
X_train_resampled, y_train_resampled = smote.fit_resample(X_train, y_train)
```

In [43]:
```
print(X_train.shape)
print(X_train_resampled.shape)
```

```
(445982, 15)
(881164, 15)
```

In [44]:
```
# see the class distribution for
y_train_resampled.value_counts()
```

Out[44]:
```
fraud
0    440582
1    440582
Name: count, dtype: int64
```

# Part 4: Modeling

Steps that I follow:

1. Build a baseline model to test the test metrics on test set;
2. Compare models generated by different algorithms to determine the best model;
3. Perform grid search to fine tune the chosen model;
4. Test the model performance on a test set.

## 4.1 Baseline Models and Model Comparison

In [45]:
```
# model training
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
import xgboost as xgb

# Check the ROC
from sklearn.metrics import roc_curve
from sklearn import metrics
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
```

In [46]:
```
X_train.describe()
```

Out[46]:

| | age_'0' | age_'1' | age_'2' | age_'3' | age_'4' | age_'5' | |
|---|---|---|---|---|---|---|---|
| **count** | 445982.000000 | 445982.000000 | 445982.000000 | 445982.000000 | 445982.000000 | 445982.000000 | 44! |
| **mean** | 0.004115 | 0.097952 | 0.314744 | 0.247429 | 0.183532 | 0.105341 | |
| **std** | 0.064012 | 0.297251 | 0.464414 | 0.431519 | 0.387103 | 0.306992 | |
| **min** | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | |
| **25%** | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | |
| **50%** | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | |
| **75%** | 0.000000 | 0.000000 | 1.000000 | 0.000000 | 0.000000 | 0.000000 | |
| **max** | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | |

In [47]:
```
# Logistic Regression
logistic = LogisticRegression(max_iter=500)
logistic.fit(X_train, y_train)
```

Out[47]:
```
▾ LogisticRegression
LogisticRegression(max_iter=500)
```

In [48]:
```
# Decision Tree
dtree = DecisionTreeClassifier()
dtree.fit(X_train, y_train)
```

Out[48]:
```
▾ DecisionTreeClassifier
DecisionTreeClassifier()
```

In [49]:
```
# Random Forest
rf = RandomForestClassifier()
rf.fit(X_train, y_train)
```

Out[49]:
```
▾ RandomForestClassifier
RandomForestClassifier()
```

In [50]:
```
# XGBoost
# Create a DMatrix for XGBoost
dtrain = xgb.DMatrix(X_train, label=y_train)
dtest = xgb.DMatrix(X_test, label=y_test)

# Set parameters for XGBoost
params = {
    'objective': 'binary:logistic',  # Use 'multi:softprob' for multi-class classifica
    'max_depth': 6,
    'eta': 0.3,
    'eval_metric': 'logloss'
}

# Train the model
num_round = 100
bst = xgb.train(params, dtrain, num_round)
```

In [51]:
```
# get predicted class for XGBoost prediction
bst_pred = [1 if prob > 0.5 else 0 for prob in bst.predict(dtest)]
```

In [52]:
```
######  Evaluate the model
# accuracy comparison
print(np.mean(logistic.predict(X_test) == y_test))
print(np.mean(dtree.predict(X_test) == y_test))
print(np.mean(rf.predict(X_test) == y_test))
print(np.mean(bst_pred == y_test))
```

```
0.9628752665460343
0.9396344703721891
0.9822414755719389
0.9831495819347374
```

In [53]:
```
report_logistic = classification_report(y_test, logistic.predict(X_test))
report_dtree    = classification_report(y_test, dtree.predict(X_test))
report_rf       = classification_report(y_test, rf.predict(X_test))
report_xgb      = classification_report(y_test, bst_pred)
```

In [54]:
```
print("Logistic Regression: \n", report_logistic)
print("Decision Tree: \n", report_dtree)
print("Random Forest: \n", report_rf)
print("XGBoost: \n", report_xgb)
```

```
Logistic Regression:
               precision    recall  f1-score   support

           0       1.00      0.96      0.98    146861
           1       0.23      0.87      0.36      1800

    accuracy                           0.96    148661
   macro avg       0.61      0.92      0.67    148661
weighted avg       0.99      0.96      0.97    148661

Decision Tree:
               precision    recall  f1-score   support

           0       1.00      0.94      0.97    146861
           1       0.13      0.69      0.22      1800

    accuracy                           0.94    148661
   macro avg       0.56      0.82      0.59    148661
weighted avg       0.99      0.94      0.96    148661

Random Forest:
               precision    recall  f1-score   support

           0       1.00      0.99      0.99    146861
           1       0.37      0.68      0.48      1800

    accuracy                           0.98    148661
   macro avg       0.68      0.83      0.74    148661
weighted avg       0.99      0.98      0.98    148661

XGBoost:
               precision    recall  f1-score   support

           0       1.00      0.99      0.99    146861
           1       0.38      0.65      0.48      1800

    accuracy                           0.98    148661
   macro avg       0.69      0.82      0.74    148661
weighted avg       0.99      0.98      0.99    148661
```

In [55]:
```
# using resampled dataset to see if the performance has significant improvement

# Build DMatrix using resampled data
dtrain_2 = xgb.DMatrix(X_train_resampled, label = y_train_resampled)

# train model
bst_2 = xgb.train(params, dtrain_2, num_round)
```

```
# prediction
bst_pred_2 = [1 if prob > 0.5 else 0 for prob in bst.predict(dtest)]

# see performance
print(classification_report(y_test, bst_pred_2))
```

```
              precision    recall  f1-score   support

           0       1.00      0.99      0.99    146861
           1       0.38      0.65      0.48      1800

    accuracy                           0.98    148661
   macro avg       0.69      0.82      0.74    148661
weighted avg       0.99      0.98      0.99    148661
```

Conclusion:

Since we care about the correctly identified fraud activities among the transactions, I think precision is the right metrics to choose for comparing model performance. As in our case, we don't want to flag any legitimate transactions as fraudulent.

Based on comparison, it looks like XGBoost is the best as compared to other three algorithms. It beats the other in both precision and f1-score for the positive class. In addition, I tried using SMOTE to upsample the data and then fitted the XGBoost model on it. However, there is no significant improvement on our metrics. Therefore, I will use XGBoost as my final model and fine tune the hyperparameters on non-resampled training data to get better estimates.

## 4.2 Fine Tuning and Testing

We've decided to use XGBoost as the final model, and now we need to fine tune the hyperparameters to obtain a model that has best performance to detect fraud activities.

In [56]:
```
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import make_scorer, f1_score, precision_score
```

In [57]:
```
param_grid = {
    'max_depth': [3, 4, 5],
    'learning_rate': [0.01, 0.1, 0.2],
    'n_estimators': [100, 200],
    'subsample': [0.8, 0.9, 1.0],
    'colsample_bytree': [0.8, 0.9]
}
```

In [58]:
```
model = xgb.XGBClassifier(use_label_encoder=False, eval_metric='mlogloss')
```

In [59]:
```
# Define the F1 score as the scoring metric
f1_scorer = make_scorer(f1_score, average='weighted')

grid_search = GridSearchCV(estimator=model,
                           param_grid=param_grid,
                           scoring=f1_scorer,
```

```
                            cv=3,
                            verbose=1)
```

In [60]:
```
grid_search.fit(X_train, y_train)
```

```
Fitting 3 folds for each of 108 candidates, totalling 324 fits
```

Out[60]:
```
GridSearchCV
estimator: XGBClassifier
XGBClassifier
```

In [61]:
```
best_model = grid_search.best_estimator_
print("Best parameters found: ", grid_search.best_params_)
```

```
Best parameters found:  {'colsample_bytree': 0.9, 'learning_rate': 0.1, 'max_depth':
4, 'n_estimators': 100, 'subsample': 0.9}
```

In [62]:
```
# Make predictions with the best estimator
y_pred = best_model.predict(X_test)

# Calculate performance metrics
f1 = f1_score(y_test, y_pred, average='weighted')
precision = precision_score(y_test, y_pred, average='weighted')
print("F1 Score: ", f1)
print("Precision Score: ", precision)
```

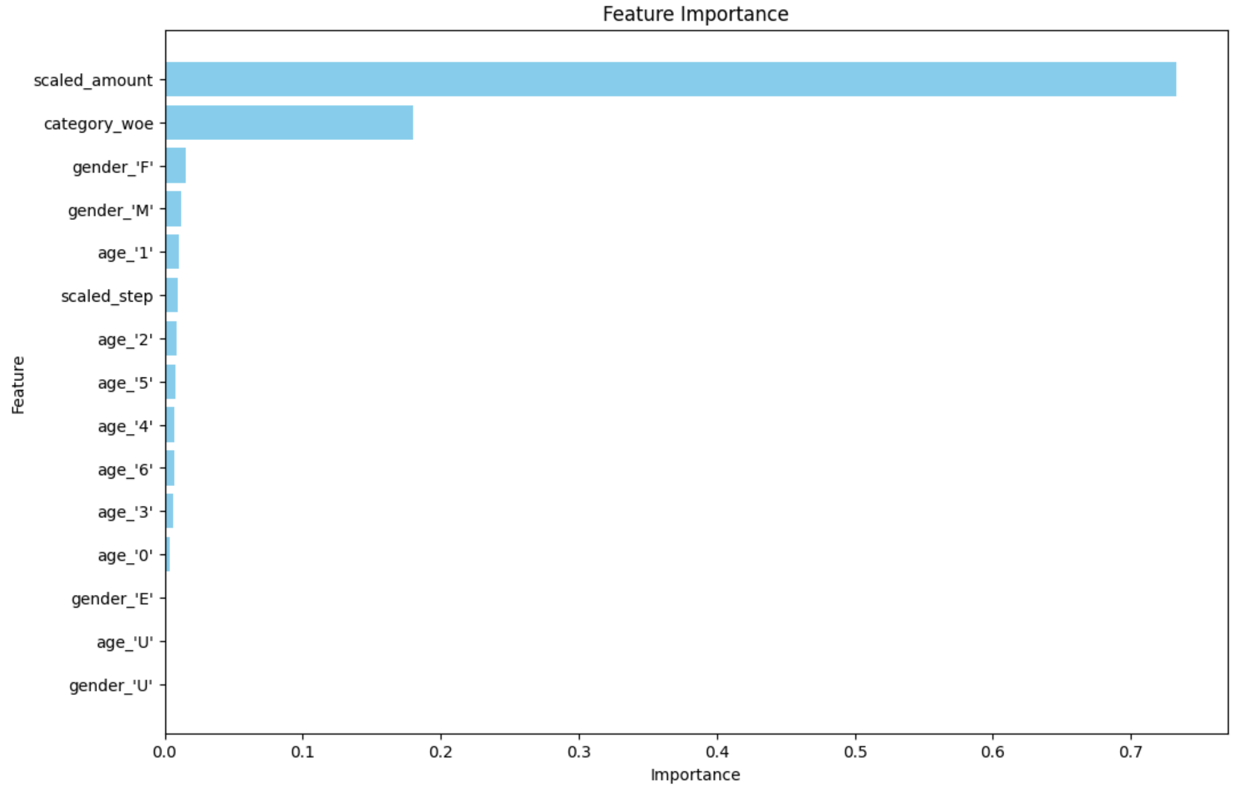
```
F1 Score:  0.987877275105618
Precision Score:  0.9892441637242974
```

In [64]:
```
# Get feature importance
feature_importances = best_model.feature_importances_
print("Feature Importances: ", feature_importances)
```

```
Feature Importances:  [0.00356751 0.01044634 0.00828227 0.00658984 0.0067852  0.00806
248
 0.00675953 0.         0.         0.01526499 0.0116198  0.
 0.7331267  0.00967755 0.17981777]
```

In [72]:
```
# Convert to DataFrame for better readability
feature_names = X_test.columns
importance_df = pd.DataFrame({'Feature': feature_names, 'Importance': feature_importar
importance_df = importance_df.sort_values(by='Importance', ascending=False)
```

In [73]:
```
# Sort the DataFrame by importance
importance_df = importance_df.sort_values(by='Importance', ascending=False)

# Plot the bar plot
plt.figure(figsize=(12, 8))
plt.barh(importance_df['Feature'], importance_df['Importance'], color='skyblue')
plt.xlabel('Importance')
plt.ylabel('Feature')
plt.title('Feature Importance')
plt.gca().invert_yaxis()  # To display the highest importance at the top
plt.show()
```




# Part 5: Conclusion

In this analysis, I examined the fraud activities in a bank simulation dataset. By observations, I found that less than 2% of the transactions are fruad but they take up to 17% of the total amount of all the transactions shown in this data. Among these fraudulent transactions, many of them frequently took place in two categories, health and sportsandtoys (over 50%).

To catch potential fraud in future financial payment, I developed supervised learning model to detect fraud transactions using this dataset. By conducting independency test, I believe that there is a association between the `category` and `fraud` label. So I applied weight of evidence to encode the categories present in this data. Also, I also applied one-hot encoding to take care other low-dimensional categorical variables and scaled the numerical ones for best estimation.

I chose precision and f1-score as I metrics to measure the performance of my fitted model. In the end, XGBoost outperform the three others including logistic regression, decision tree, and random forest. By carefully fine tuning, ultimately, I improved the model precision from 38\% to 98\%.