Financial Crime Technology

Let's start by importing the important libraries

In [1]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import datetime
import seaborn as sns
from datetime import date
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
```

In [2]:

```
#fraud_final.columns
```

Data Loading and Overview

In [3]:

```
users = pd.read_csv('users.csv')
countries = pd.read_csv('countries.csv')
fraudsters = pd.read_csv('fraudsters.csv')
transactions = pd.read_csv('transactions.csv')
curr = pd.read_csv('currency_details.csv')
rates = pd.read_csv('fx_rates.csv')
```

Data Preparation

Let's check the number of the rows and columns in each of the files supplied - We will use a small function to return the name of a dataframe and help us checking the data frames in a better way.

In [4]:

```
def get_df_name(df):
    name =[x for x in globals() if globals()[x] is df][0]
    return name
```

In [5]:

```
data = [countries,transactions,curr,rates, users,fraudsters]
for file in data:
     print(f' {get_df_name(file)} dataset below has {file.shape[0]} rows and {file.shape[
1]} columns')
     print(file.head(3))
     print('/n')
```

```
 countries dataset below has 239 rows and 5 columns
  code         name code3  numcode  phonecode
0   AF  Afghanistan   AFG      4.0         93
1   AL      Albania   ALB      8.0        355
2   DZ      Algeria   DZA     12.0        213
/n
 transactions dataset below has 688651 rows and 12 columns
   Unnamed: 0 CURRENCY  AMOUNT      STATE               CREATED_DATE  \
0           0      GBP    3738  COMPLETED  2015-10-11 09:05:43.016000
1           1      GBP     588  COMPLETED  2015-10-11 20:08:39.150000
2           2      GBP    1264  COMPLETED  2015-10-11 11:37:40.908000

  MERCHANT_CATEGORY MERCHANT_COUNTRY ENTRY_METHOD  \
```

```
0               bar              AUS         misc
1               NaN               CA         misc
2               NaN              UKR         misc

                                 USER_ID          TYPE SOURCE  \
0  7285c1ec-31d0-4022-b311-0ad9227ef7f4  CARD_PAYMENT   GAIA
1  20100a1d-12bc-41ed-a5e1-bc46216e9696  CARD_PAYMENT   GAIA
2  0fe472c9-cf3e-4e43-90f3-a0cfb6a4f1f0  CARD_PAYMENT   GAIA

                                     ID
0  5a9ee109-e9b3-4598-8dd7-587591e6a470
1  28d68bf4-460b-4c8e-9b95-bcda9ab596b5
2  1f1e8817-d40b-4c09-b718-cfc4a6f211df
/n
 curr dataset below has 208 rows and 4 columns
  currency  iso_code  exponent  is_crypto
0      AED     784.0       2.0      False
1      AFN     971.0       2.0      False
2      ALL       8.0       2.0      False
/n
 rates dataset below has 84 rows and 3 columns
  base_ccy  ccy         rate
0      EUR  AED     0.239336
1      EUR  AUD     0.639595
2      EUR  BTC  6617.495728
/n
 users dataset below has 10300 rows and 11 columns
   Unnamed: 0  FAILED_SIGN_IN_ATTEMPTS     KYC  BIRTH_YEAR COUNTRY   STATE  \
0           0                        0  PASSED        1971      GB  ACTIVE
1           1                        0  PASSED        1982      GB  ACTIVE
2           2                        0  PASSED        1973      ES  ACTIVE

                 CREATED_DATE TERMS_VERSION   PHONE_COUNTRY  HAS_EMAIL  \
0  2017-08-06 07:33:33.341000    2018-05-25  GB||JE||IM||GG          1
1  2017-03-07 10:18:59.427000    2018-01-01  GB||JE||IM||GG          1
2  2018-05-31 04:41:24.672000    2018-09-20              ES          1

                                     ID
0  1872820f-e3ac-4c02-bdc7-727897b60043
1  545ff94d-66f8-4bea-b398-84425fb2301e
2  10376f1a-a28a-4885-8daa-c8ca496026bb
/n
 fraudsters dataset below has 300 rows and 2 columns
   Unnamed: 0                               user_id
0           0  5270b0f4-2e4a-4ec9-8648-2135312ac1c4
1           1  848fc1b1-096c-40f7-b04a-1399c469e421
2           2  27c76eda-e159-4df3-845a-e13f4e28a8b5
/n
```

A better look at the columns present in each data frame:

In [6]:

```
data = [countries,transactions,curr,rates, users,fraudsters]
for file in data:
    print(f'The columns of {get_df_name(file)} dataset are {file.columns.tolist()}')
```

```
The columns of countries dataset are ['code', 'name', 'code3', 'numcode', 'phonecode']
The columns of transactions dataset are ['Unnamed: 0', 'CURRENCY', 'AMOUNT', 'STATE', 'CR
EATED_DATE', 'MERCHANT_CATEGORY', 'MERCHANT_COUNTRY', 'ENTRY_METHOD', 'USER_ID', 'TYPE',
'SOURCE', 'ID']
The columns of curr dataset are ['currency', 'iso_code', 'exponent', 'is_crypto']
The columns of rates dataset are ['base_ccy', 'ccy', 'rate']
The columns of users dataset are ['Unnamed: 0', 'FAILED_SIGN_IN_ATTEMPTS', 'KYC', 'BIRTH_
YEAR', 'COUNTRY', 'STATE', 'CREATED_DATE', 'TERMS_VERSION', 'PHONE_COUNTRY', 'HAS_EMAIL',
'ID']
The columns of fraudsters dataset are ['Unnamed: 0', 'user_id']
```

So we notice from the above data that the number of fraudsters is less (300) than the number of users! Let's compare the list of fraudsters to the list of all users to check if we have a match. To do this, we will merge the

fraudesters dataset with the user datasets bases on the user_id.

But first we need to unify the naming of columns in different data frames to make sure the merging is done correctly!

In [7]:

```
# We have to rename columns belonging to different data frames
users.rename(columns={"ID": "user_id"}, inplace=True)
users.rename(columns={"CREATED_DATE": "user_date","STATE":'user_state'}, inplace=True)
transactions.rename(columns={"USER_ID": "user_id", "CREATED_DATE": "trans_date", "ID": "
txn_id", "CURRENCY": "currency", "AMOUNT":'amount', "STATE":'txn_state'}, inplace=True)
curr.rename(columns={"CURRENCY": "currency"}, inplace=True)
rates.rename(columns={"ccy": "currency"}, inplace=True)
```

300 is the number of fraudsters and all these 300 users exist in the bigger 'user' dataframe - we can merge different columns from different data sets to the fraudsters list and this will allow us to analyse the data of these fraudsters users and come up with a criteria to predict future fraudsters!

Let's merge the 'fraud' dataset that has the fraudulent IDs with the transactions dataset.

Also, in order to have a standard money value accross all transactions we are going to only consider the EUR as a base currency

In [8]:

```
eur_rates = rates[rates['base_ccy']=='EUR']
print(eur_rates.head(3))
```

```
  base_ccy currency         rate
0      EUR      AED     0.239336
1      EUR      AUD     0.639595
2      EUR      BTC  6617.495728
```

We can see that conversion rates from EUR ONLY are to be conidered.

To create a data frame that contains a list of fraudsters along with details about transactions they made, we should merge fraudtsers and transactions like below:

In [9]:

```
fraud = pd.merge(pd.merge(fraudsters,transactions,on='user_id', how='left'),curr,on='cur
rency', how='left')
fraud_users = pd.merge(fraud, users, on='user_id', how='left')
fraud_final = pd.merge(fraud_users, eur_rates[['currency','rate']], on='currency', how='
left')
```

In [10]:

```
fraud_final[fraud_final['currency']=='EUR'].head(2)
```

Out[10]:

	Unnamed: 0_x	user_id	Unnamed: 0_y	currency	amount	txn_state	trans_date	MERCHANT_CATEGORY	MERCHA
29	7	e7876b06-bcd8-4193-9eaa-b477313f6f1a	557704.0	EUR	500.0	DECLINED	2018-06-15 13:28:33.243000	point_of_interest	
82	15	2707dd70-86d3-4823-ad0f-91a340bccb88	593364.0	EUR	2000.0	COMPLETED	2018-06-05 10:23:25.908000	NaN	

2 rows × 27 columns

Now and after merging the datasets, we have a dataset with fraudulent activities with the following columns:

In [11]:

```
fraud_final.columns
```

Out[11]:

```
Index(['Unnamed: 0_x', 'user_id', 'Unnamed: 0_y', 'currency', 'amount',
       'txn_state', 'trans_date', 'MERCHANT_CATEGORY', 'MERCHANT_COUNTRY',
       'ENTRY_METHOD', 'TYPE', 'SOURCE', 'txn_id', 'iso_code', 'exponent',
       'is_crypto', 'Unnamed: 0', 'FAILED_SIGN_IN_ATTEMPTS', 'KYC',
       'BIRTH_YEAR', 'COUNTRY', 'user_state', 'user_date', 'TERMS_VERSION',
       'PHONE_COUNTRY', 'HAS_EMAIL', 'rate'],
      dtype='object')
```

The description says: exponent column can be used to convert the integer amounts in the transactions table into cash amounts. (e.g for 5000 GBP, exponent = 2, so we apply: 5000/(10^2) = 50 GBP).

Based on the above info above and to have euros as standard currency we should divide our dataframe by the exponent and multiply by the rate (from any currency to euro).

In [12]:

```
# Changing amounts to euro while taking into consideration the exponents:
fraud_final['amount'] = fraud_final['amount'] / (10**fraud_final['exponent'])
fraud_final.loc[fraud_final['currency']!='EUR', 'amount'] = fraud_final['amount'] *fraud
_final['rate']
```

To double-check if our algorithm is working correctly, we can compare one value before and after. We can see from the two results below that the original amount was 59700 GBP and after using the exponent and the exchange rate to EUR, the new value is 673.58 euros which is the correct value!! Also, our amounts in euro are correct which was proved out by the same reasoning!

In [13]:

```
# To check if changes to the right currency are correct we will check two particular tran
sactions (GB and EUR)
# GB values:
fraud_final[fraud_final['trans_date']=='2018-06-29 12:34:41.413000']
```

Out[13]:

	Unnamed: 0_x	user_id	Unnamed: 0_y	currency	amount	txn_state	trans_date	MERCHANT_CATEGORY	MERC
1	1	848fc1b1-096c-40f7-b04a-1399c469e421	599236.0	GBP	673.583033	COMPLETED	2018-06-29 12:34:41.413000		NaN

1 rows × 27 columns

In [14]:

```
# EUR values:
fraud_final[fraud_final['trans_date']=='2018-06-15 13:28:33.243000']
```

Out[14]:

	Unnamed: 0_x	user_id	Unnamed: 0_y	currency	amount	txn_state	trans_date	MERCHANT_CATEGORY	MERCHANT
29	7	e7876b06-bcd8-4193-9eaa-	557704.0	EUR	5.0	DECLINED	2018-06-15 13:28:33.243000	point_of_interest	

Unnamed: 0_x	b477313f6f1a user_id	Unnamed: 0_y	currency	amount	txn_state	trans_date	MERCHANT_CATEGORY	MERCHANT

1 rows × 27 columns

In [15]:

```
# Now our final fraud data frame has the following columns:
fraud_final.columns
```

Out[15]:

```
Index(['Unnamed: 0_x', 'user_id', 'Unnamed: 0_y', 'currency', 'amount',
       'txn_state', 'trans_date', 'MERCHANT_CATEGORY', 'MERCHANT_COUNTRY',
       'ENTRY_METHOD', 'TYPE', 'SOURCE', 'txn_id', 'iso_code', 'exponent',
       'is_crypto', 'Unnamed: 0', 'FAILED_SIGN_IN_ATTEMPTS', 'KYC',
       'BIRTH_YEAR', 'COUNTRY', 'user_state', 'user_date', 'TERMS_VERSION',
       'PHONE_COUNTRY', 'HAS_EMAIL', 'rate'],
      dtype='object')
```

In [16]:

```
# We need to drop the original index columns that appear here as unnamed
fraud_final.drop(['Unnamed: 0_x', 'Unnamed: 0_y', 'Unnamed: 0','TERMS_VERSION'], axis=1,i
nplace=True)
```

In [17]:

```
fraud_final.columns
```

Out[17]:

```
Index(['user_id', 'currency', 'amount', 'txn_state', 'trans_date',
       'MERCHANT_CATEGORY', 'MERCHANT_COUNTRY', 'ENTRY_METHOD', 'TYPE',
       'SOURCE', 'txn_id', 'iso_code', 'exponent', 'is_crypto',
       'FAILED_SIGN_IN_ATTEMPTS', 'KYC', 'BIRTH_YEAR', 'COUNTRY', 'user_state',
       'user_date', 'PHONE_COUNTRY', 'HAS_EMAIL', 'rate'],
      dtype='object')
```

We will create four separate columns, day, hour, month, year and week. These two columns could help us finding the anomalies in our transactions. As maybe fraudsters users make more transactions in a day (or a hour/week/month) than a normal user.

In [18]:

```
fraud_final['trans_date'] = pd.to_datetime(fraud_final['trans_date'],errors='coerce')
fraud_final['day'] = fraud_final['trans_date'].dt.date
fraud_final['hour'] = fraud_final['trans_date'].dt.hour
fraud_final['month'] = fraud_final['trans_date'].dt.month
fraud_final['year'] = fraud_final['trans_date'].dt.year
fraud_final['week'] = fraud_final['trans_date'].dt.week
```

In order to create a data frame that does not have any user_ids from the fraud list, we will use the ~users['user_id'] to create a data frame called regular_final that has no previously listed fraudsters!

In [19]:

```
regular = users[~users['user_id'].isin(fraudsters['user_id'])]
regular_trans = pd.merge(pd.merge(regular,transactions,on='user_id', how='left'),curr,on
='currency', how='left')
regular_final = pd.merge(regular_trans, eur_rates[['currency','rate']], on='currency', h
ow='left')
regular_final['amount'] = regular_final['amount'] / (10**regular_final['exponent'])
regular_final.loc[regular_final['currency']!='EUR', 'amount'] = regular_final['amount'] *
regular_final['rate']
regular_final['trans_date'] = pd.to_datetime(regular_final['trans_date'],errors = 'coerc
e')
regular_final['day'] = regular_final['trans_date'].dt.date
regular_final['hour'] = regular_final['trans_date'].dt.hour
regular_final['month'] = regular_final['trans_date'].dt.month
```

```
regular_final['year'] = regular_final['trans_date'].dt.year
regular_final['week'] = regular_final['trans_date'].dt.week
regular_final.drop(['Unnamed: 0_x', 'Unnamed: 0_y'], axis=1, inplace=True)
regular_final.shape[0]
```

Out[19]:

```
676386
```

Our data is not labelled (except for the list of fraudsters) which means we need to come up with rules that could be followed to detect frausters. We are going to check the following indicators that might help us understand anomalies in the data:

- **Outliers in transaction values (euros). We will use ROBUST z_score to detect outliers and transactions with values higher than z_score = 3.5. Robust z_score as the name suggets is a robust metric against outliers and gives better results than the normal z_score. 3.5 means any values that is higher than 3.5 times the standard deviation will be considered as an outlier.**
- **The empirical rule, also referred to as the three-sigma rule or 68-95-99.7 rule, is a statistical rule which states that for a normal distribution, almost all data falls within three standard deviations (denoted by σ) of the mean (denoted by μ). Broken down, the empirical rule shows that 68% falls within the first standard deviation (μ ± σ), 95% within the first two standard deviations (μ ± 2σ), and 99.7% within the first three standard deviations (μ ± 3σ)**
- **Number of transactions made in a hour/day/week/month as fraudsters might do more transactions in a cerain time frame.**
- **Country where the transaction was made from and compare it to the country where the user account was created**
- **Time of the day of the transaction**
- **#### Clustering: Clustering could be followed here on one dataframe that joins the list of fraudsters and all other users together in one dataset. Any data point that is close to the fraudsters cluster will be flagged as fraud and should be investigated more.**

Data Exploration - EDA

Let's start by checking if we have NAs in our frausters data frame.

In [20]:

```
fraud_final.isna().sum()
```

Out[20]:

```
user_id                      0
currency                     1
amount                       1
txn_state                    1
trans_date                   1
MERCHANT_CATEGORY        10418
MERCHANT_COUNTRY          5465
ENTRY_METHOD                 1
TYPE                         1
SOURCE                       1
txn_id                       1
iso_code                     4
exponent                     1
is_crypto                    1
FAILED_SIGN_IN_ATTEMPTS      0
KYC                          0
BIRTH_YEAR                   0
COUNTRY                      0
user_state                   0
user_date                    0
PHONE_COUNTRY                0
HAS_EMAIL                    0
rate                       714
day                          1
```

```
hour                              1
month                             1
year                              1
week                              1
dtype: int64
```

We have quiet few NAs in the MERCHANT_CATEGORY and MERCHANT_COUNTRY column we will keep them for now! The rest of the columns don't have NAs so nothing to be done with any of them.

Let's check the statistics of the all features present in the fraud dataframe.

Looking at the two tables below, there is an interesting pattern here. The average amount of money spent during one transaction is around e316.9. This piece of info could help us flag the transactions that might be fraud.

In [21]:

```
# Let's check the statistics of the all features present in the fraud dataframe
fraud_final.describe()
```

Out[21]:

	amount	iso_code	exponent	FAILED_SIGN_IN_ATTEMPTS	BIRTH_YEAR	HAS_EMAIL	rate	
count	14543.000000	14540.000000	14543.000000	14544.000000	14544.000000	14544.000000	13830.000000	145
mean	316.589318	834.928198	2.001238	0.008938	1987.944513	0.998144	2.531843	
std	1846.320115	45.733986	0.086170	0.123228	9.442633	0.043048	97.440706	
min	0.000000	203.000000	2.000000	0.000000	1936.000000	0.000000	0.045900	
25%	5.641399	826.000000	2.000000	0.000000	1984.000000	1.000000	1.128280	
50%	22.520465	826.000000	2.000000	0.000000	1990.000000	1.000000	1.128280	
75%	191.807564	826.000000	2.000000	0.000000	1994.000000	1.000000	1.128280	
max	80963.450794	985.000000	8.000000	2.000000	2000.000000	1.000000	6617.495728	

As part as our analysis we should have a look at any correlated features. The closer to 1, the higher correlation exists between the two features. The result below proves no correlation exists between our features (except rate/exponent)

In [22]:

```
corr = fraud_final.corr()
corr.style.background_gradient(cmap='coolwarm')
```

```
/Users/mac/anaconda3/lib/python3.6/site-packages/matplotlib/colors.py:527: RuntimeWarning
: invalid value encountered in less
  xa[xa < 0] = -1
```

Out[22]:

	amount	iso_code	exponent	FAILED_SIGN_IN_ATTEMPTS	BIRTH_YEAR	HAS_EMAIL	
amount	1	-0.00174712	-0.00245563	0.0138744	-0.00758669	0.00228099	-(
iso_code	-0.00174712	1	nan	-0.00674403	0.0211344	0.0084206	
exponent	-0.00245563	nan	1	-0.00104198	0.00921159	0.000619494	
FAILED_SIGN_IN_ATTEMPTS	0.0138744	-0.00674403	-0.00104198	1	0.0433285	0.0031283	-(
BIRTH_YEAR	-0.00758669	0.0211344	0.00921159	0.0433285	1	0.068257	(
HAS_EMAIL	0.00228099	0.0084206	0.000619494	0.0031283	0.068257	1	0.

	amount	iso_code	exponent	FAILED_SIGN_IN_ATTEMPTS	BIRTH_YEAR	HAS_EMAIL
rate	0.00245634	-0.177236	0.999999	-0.00105786	0.00956528	0.000637092
hour	0.0149879	-0.00587688	-0.0114094	0.0120035	-0.00148246	-0.00937771
month	-0.0238306	-0.0836946	-0.0118722	-0.030608	-0.0537517	-0.033574
year	0.0410297	0.0191447	0.0100015	0.0505089	0.139915	0.057806
week	-0.0231412	-0.080543	-0.0133099	-0.033725	-0.0566137	-0.0326224

Let's start looking at boxplots of fraud users. The values are too high to be graphed by a normal boxplot therefore we will use log transformation before using the boxplot.

Outliers:

In order to find fraudsters (different than the ones we were already given in the fraudsters dataset) we will write a fucntion that uses a robust Z_score method to find outliers. We will set the threshold to 3 which means any amount of transaction that is higher than 3 std devitions from the mean will be considered as an outlier and should be looked at more closely. The robust z score is better than the normal z score as it uses median absolute deviation.

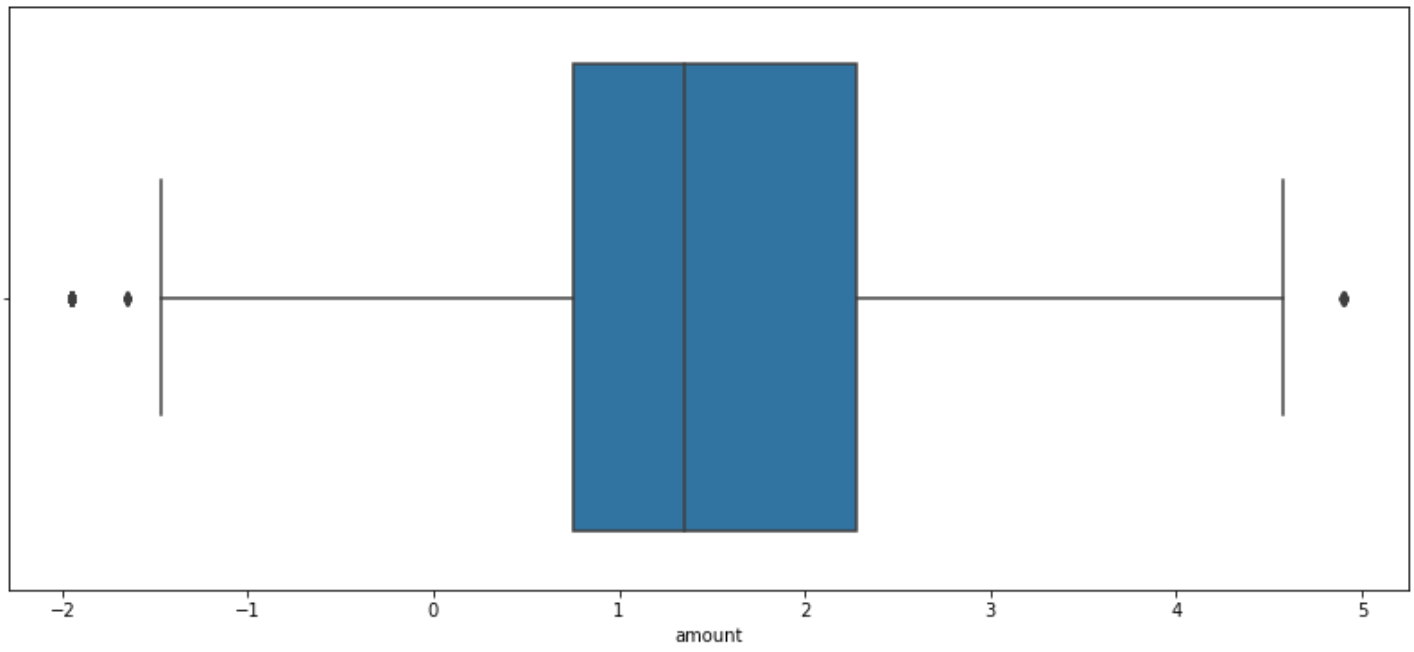
In [23]:

```
plt.figure(figsize=(14,6))
sns.boxplot(np.log10(fraud_final['amount']))
```

```
/Users/mac/anaconda3/lib/python3.6/site-packages/pandas/core/series.py:853: RuntimeWarnin
g: divide by zero encountered in log10
  result = getattr(ufunc, method)(*inputs, **kwargs)
```

Out[23]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x117a61da0>
```



The boxplot for fraudsters dataset shows the presence of outliers. Let's take a closer look at the outliers using the z_scores. The fraudsters dataframe had some NAs in it so we will delete them as our data size permits. In our case this method is better than imputation.

In [24]:

```
fraud_final.dropna(inplace=True)
```

```
outliers_fraud = outliers_modified_z_score(fraud_final['amount'])
outlier_index = pd.DataFrame(np.column_stack(outliers_fraud),columns=['index_outlier'])
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-24-5d87ebf9e5bd> in <module>()
      1 fraud_final.dropna(inplace=True)
----> 2 outliers_fraud = outliers_modified_z_score(fraud_final['amount'])
      3 outlier_index = pd.DataFrame(np.column_stack(outliers_fraud),columns=['index_outl
ier'])

NameError: name 'outliers_modified_z_score' is not defined
```

In [ ]:

```
outlier_index.info()
```

The result above shows even our fraud list has some outliers in it (1250 exactly) when it comes to the amount of money taken.

What is the average amount of money taken during a fraudulent transaction? median?

In [ ]:

```
fraud_final['amount'].mean()
```

In [ ]:

```
fraud_final['amount'].median()
```

What is the average total amount of money a fraudulant user might take?

In [ ]:

```
fraud_final.groupby('user_id')['amount'].sum().mean()
```

Per user, what is the mean amount of money taken?

In [ ]:

```
fraud_final.groupby('user_id')['amount'].mean().mean()
```

Let's check the number of transactions a fraudster would make in one hour? Also, the average/median amount of money withdrawn

In [ ]:

```
fraud_final.groupby(['user_id','hour'])['amount'].count().mean()
```

What is the top 10 merchant countries of the fraudsters?

In [ ]:

```
fraud_final.groupby(['MERCHANT_COUNTRY'])['amount'].count().nlargest(10)
```

In [ ]:

```
fraud_final.groupby(['COUNTRY'])['amount'].count().nlargest(10)
```

Let's check the number of fraudsters with an account created in one country while fraud is happening in a different country - According to the numbers below, some accounts created in GB are trying to take out frauduent money from machines located in 25 other countries! Followed by Lithuania

In [ ]:

```
fraud_final.groupby('COUNTRY')['MERCHANT_COUNTRY'].nunique().nlargest(10)
```

Let's check the age of fraudsters now! From the histogram below, we could clearly see that fraudulent users's birth year is between 1980 - 2000.

In [ ]:

```
fraud_final.hist('BIRTH_YEAR')
```

To look closer at the age of fraudsters, we group the fraudsters dataframe by age and we could see from the results below that most fraudsters were born in 1998

In [ ]:

```
fraud_final.groupby('BIRTH_YEAR')['user_id'].count().nlargest(10)
```

In [ ]:

```
fraud_final['hour'].value_counts()
```

In [ ]:

```
fraud_final['hour'].hist(bins=24)
```

The time of fraudulent activities peaks at 11 in the morning and it becomes quiet in the early morning hours

In [ ]:

```
fraud_final['month'].hist(bins=12)
```

In [ ]:

```
fraud_final['month'].value_counts()
```

It is intersting to see that the month of December sees the lowest amount of fraud cases while the month of June sees the highest number of fraud transactions.

What kind of merchants are the most popular to perform fraud? Looking at the results below, we conclude ATM is the most popular and it is followed closely by point of interest.

In [ ]:

```
fraud_final['MERCHANT_CATEGORY'].value_counts().nlargest(10)
```

In [ ]:

```
fraud_final['MERCHANT_COUNTRY'].value_counts().nlargest()
```

In [ ]:

```
plt.scatter(fraud_final['amount'], fraud_final['FAILED_SIGN_IN_ATTEMPTS'], c='blue', alp
ha=0.5)
```

Number of transactions per day is:

In [ ]:

```
fraud_final.groupby(['user_id','day'])['amount'].count().mean()
```

At this stage, we have multiple markers that we can use to detect fraudsters:

- **The mean amount of money taken in a fraudulent transaction is around 210 euros while the median is 25 euros**
- **The mean amount of money taken per user is around 208 euros**
- **The median amount of money taken durind a fraud transaction is 25**
- **The average sum of transactions that a fraudulant user might take is 3918 euros.**
- **The average number of transactions a fraudster might make in one day is around 3.2**
- **The average number of transactions a fraudster might make in one hour is around 3**
- **Top merchant countries of fraud are GBR and IRL followed by Poland**
- **The birth year of a fraudster is mostly 1998 and 1991**
- **The fraudulent activities peak at 11 in the morning**
- **The month of June has the highest number of fraud transactions**
- **ATM is the most preferred type or merchants for fraudsters and followed directly by points of interests. ## Using these criteria above we will pinpoint 5 fraudsters and 5 high risk transactions as any transaction that presents all the above criteria can be considered as fraudulent.**

Find 5 likely fraudsters (not already found in fraudsters.csv!), provide their user_ids, and explain how you found them and why they are likely fraudsters. (15 points) Find an additional 5 high risk users, explain the financial crime typology that is likely here, and how you will conduct enhanced due diligence on these specific users. (10 points)

To find 5 fraudsters and 5 high risk users, we will use the above criteria which act now as a rules based model:

In [ ]:

```
# filter the users data frame by Transactions higher than 210
txn210 = regular_final[regular_final['amount'] > 210]
# filter by transactions made by someone born in 1998
birth = txn210[txn210['BIRTH_YEAR']==1998]
# filter by transactions made in GBR
country = birth[birth['MERCHANT_COUNTRY']=='GBR']
# filter by transactions done in ATM
frauds = country[country['TYPE']=='ATM']
```

Users likely to be fraudsters are:

In [ ]:

```
frauds['user_id'].tolist()
```

For the high risk users, their accounts should be flagged and monitored closely to ensure they re not doing any fraudulent transactions.

Part of quesions 3 was already ansered in our preious analysis as the features that are indicative to detecting fraudsters were already explained. We will check the regular_final dataframe which is the frame that has no fraudsters. We will try to find patterns in this dataframe that can help us find more fraudsters!

We will start by checking outliers in this dataframe, outliers in the amount withdrawn!

In [ ]:

```
import numpy as np

def outliers_modified_z_score(ys):
    threshold = 3.5

    median_y = np.median(ys)
    median_absolute_deviation_y = np.median([np.abs(y - median_y) for y in ys])
```

```
    modified_z_scores = [0.6745 * (y - median_y) / median_absolute_deviation_y
                         for y in ys]
    return np.where(np.abs(modified_z_scores) > threshold)
```

In [ ]:

```
# Let's drop all rows in the data set where there is NA
regular_final.dropna(inplace=True)
```

We will create a data frame with all indices

In [ ]:

```
outliers = outliers_modified_z_score(regular_final['amount'])
```

In [ ]:

```
outliers_index = pd.DataFrame(np.column_stack(outliers),columns=['index_outlier'])
```

In [ ]:

```
outliers_index.head(5)
```

Now we can use the above index we just created to search for the user_ids of these outliers who took out 95% more money than the rest of the transactions. These user_ids can be seen below:

In [ ]:

```
regular_final['user_id'].loc[outliers_index['index_outlier']].dropna().head(10)
```

My two questions to the data team: 1- Can different user_ids belong to the same account/person? 2- Are online transactions included in the point_of_interest? How could the location of such transaction be accurately found?

In order to detect the fraudsters from the regular dataframe, we are going to create clusters using KMeans clustering method. Hopefully we will get nice clusters that will help us identify the fruadsters in a better way!

In [ ]:

```
df = pd.concat([fraud_final, regular_final],ignore_index=True)
```

In [ ]:

```
# Import the scaler
df_reduced = df[['amount','day','hour','month','COUNTRY', 'MERCHANT_COUNTRY','year','BIR
TH_YEAR']]
# We need to change to numericals in order for us to use the knn clustering methods
df_reduced[['COUNTRY','MERCHANT_COUNTRY']] = df_reduced[['COUNTRY','MERCHANT_COUNTRY']].a
pply(lambda x: x.astype('category') )
cat_columns = df_reduced.select_dtypes(['category']).columns

df_reduced[cat_columns] = df_reduced[cat_columns].apply(lambda x: x.cat.codes)
```

In [ ]:

```
df_reduced.head()
```

In [ ]:

```
regular_final.describe()
```

In order to perform Kmeans clustering, we need to transfrom all the features to numericals

In [ ]:

```
df_reduced.dropna(inplace=True)
df_reduced['day'] = df_reduced['day'].astype(str)
df_reduced['day'] = df_reduced['day'].str.replace("-","").astype(int)
```

In [ ]:

```
df_reduced.head()
```

In [ ]:

```
X = df_reduced.values.astype(np.float)

# Define the scaler and apply to the data
scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)

# Import MiniBatchKmeans
from sklearn.cluster import MiniBatchKMeans

# Define the model
kmeans = MiniBatchKMeans(n_clusters=2, random_state=0)
# Fit the model to the scaled data
kmeans.fit(X_scaled)
# Define the range of clusters to try
clustno = range(1, 10)
# Run MiniBatch Kmeans over the number of clusters
kmeans = [MiniBatchKMeans(n_clusters=i) for i in clustno]
# Obtain the score for each model
score = [kmeans[i].fit(X_scaled).score(X_scaled) for i in range(len(kmeans))]

# Plot the models and their respective score
plt.plot(clustno, score)
plt.xlabel('Number of Clusters')
plt.ylabel('Score')
plt.title('Elbow Curve')
plt.show()
```

From the above elbow Curve we can see clearly that two clusters were identified by kMeans model. and this discovery could help us classify the datapoints into Fraud or Non Fraud cases.

The above result shows we could easily divide our data into two clusters, one that only has fraud transactions and the second cluster will have most of the regular transactions. One way to detect new fraudsters will be finiding the points on the graph that are closer to the fraud cluster. Any point close to the fraud cluster is definitely fraudulent!