Credit Card Fraud Detection

This project is related to building a Machine Learning module that can effectively detect Fraudlent Credit Card transactions and hence save money for the client Bank.

```
#from google.colab import drive
#drive.mount('/content/gdrive')
```

```
import warnings
warnings.filterwarnings('ignore')
```

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

There are two datasets hosted by kaggle named test and train. Lets examine both the datasets to decide if they needs to be merged or if it is a good idea to keep the test dataset seperate for final evaluation.

```
# Importing both the datasets provided by kaggle
#fraud = pd.read_csv('gdrive/My Drive/Capstone Project/fraudTrain.csv')
#fraud_test = pd.read_csv('gdrive/My Drive/Capstone Project/fraudTest.csv')
fraud = pd.read_csv('fraudTrain.csv')
fraud_test = pd.read_csv('fraudTest.csv')
```

Exploratory Data Analysis

- Univariate Analysis
- Bivariate Analysis
- Data Cleaning
- Outlier Treatment
- Variable Transformation

```
# check for the main (training) dataset
fraud.head()
```

	Unnamed: 0	trans_date_trans_time	cc_num	merchant	category	amt	first	last	gender	street	...	
0	0	2019-01-01 00:00:18	2703186189652095	fraud_Rippin, Kub and Mann	misc_net	4.97	Jennifer	Banks	F	561 Perry Cove	...	36.0
1	1	2019-01-01 00:00:44	630423337322	fraud_Heller, Gutmann and Zieme	grocery_pos	107.23	Stephanie	Gill	F	43039 Riley Greens Suite 393	...	48.8
2	2	2019-01-01 00:00:51	38859492057661	fraud_Lind-Buckridge	entertainment	220.11	Edward	Sanchez	M	594 White Dale Suite 530	...	42.1
3	3	2019-01-01 00:01:16	3534093764340240	fraud_Kutch, Hermiston and Farrell	gas_transport	45.00	Jeremy	White	M	9443 Cynthia Court Apt. 038	...	46.2
4	4	2019-01-01 00:03:06	375534208663984	fraud_Keeling-Crist	misc_pos	41.96	Tyler	Garcia	M	408 Bradley Rest	...	38.4

5 rows × 23 columns

```
# checking for various columns and nulls in the dataset
fraud.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 89474 entries, 0 to 89473
Data columns (total 23 columns):
```

```
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   Unnamed: 0             89474 non-null  int64
 1   trans_date_trans_time  89474 non-null  object
 2   cc_num                 89474 non-null  int64
 3   merchant               89474 non-null  object
 4   category               89474 non-null  object
 5   amt                    89474 non-null  float64
 6   first                  89474 non-null  object
 7   last                   89474 non-null  object
 8   gender                 89474 non-null  object
 9   street                 89474 non-null  object
 10  city                   89474 non-null  object
 11  state                  89474 non-null  object
 12  zip                    89474 non-null  int64
 13  lat                    89474 non-null  float64
 14  long                   89474 non-null  float64
 15  city_pop               89473 non-null  float64
 16  job                    89473 non-null  object
 17  dob                    89473 non-null  object
 18  trans_num              89473 non-null  object
 19  unix_time              89473 non-null  float64
 20  merch_lat              89473 non-null  float64
 21  merch_long             89473 non-null  float64
 22  is_fraud               89473 non-null  float64
dtypes: float64(8), int64(3), object(12)
memory usage: 15.7+ MB
```

```
# basic inspection of the test dataset
fraud_test.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 93320 entries, 0 to 93319
Data columns (total 23 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   Unnamed: 0             93320 non-null  int64
 1   trans_date_trans_time  93320 non-null  object
 2   cc_num                 93319 non-null  float64
 3   merchant               93319 non-null  object
 4   category               93319 non-null  object
 5   amt                    93319 non-null  float64
 6   first                  93319 non-null  object
 7   last                   93319 non-null  object
 8   gender                 93319 non-null  object
 9   street                 93319 non-null  object
 10  city                   93319 non-null  object
 11  state                  93319 non-null  object
 12  zip                    93319 non-null  float64
 13  lat                    93319 non-null  float64
 14  long                   93319 non-null  float64
 15  city_pop               93319 non-null  float64
 16  job                    93319 non-null  object
 17  dob                    93319 non-null  object
 18  trans_num              93319 non-null  object
 19  unix_time              93319 non-null  float64
 20  merch_lat              93319 non-null  float64
 21  merch_long             93319 non-null  float64
 22  is_fraud               93319 non-null  float64
dtypes: float64(10), int64(1), object(12)
memory usage: 16.4+ MB
```

```
# checking % of data provided by Kaggle in the train & test
1296675 * 100 / (1296675 + 555719)
```

```
69.99995681264353
```

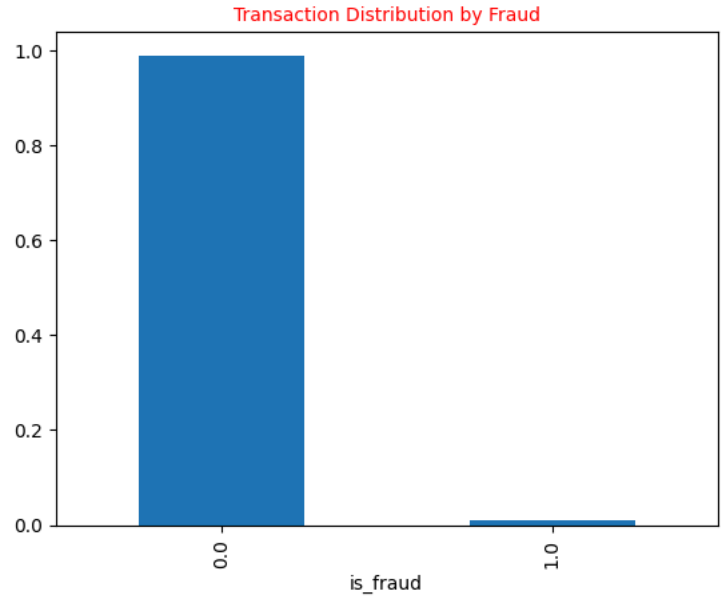
- 70% data is present in the train dataset and remaining 30% in the test dataset.
- No null values in either of the files

```
# Check for imbalance on target variable in the train dataset
fraud.is_fraud.value_counts(normalize=True)
```

```
is_fraud
0.0    0.99031
1.0    0.00969
Name: proportion, dtype: float64
```

```
# Check for imbalance on target variable in the main dataset
print ('Fraud Distribution')
print (fraud.is_fraud.value_counts(normalize=True))
plt.title('Transaction Distribution by Fraud', fontsize= 10, color = 'Red', fontweight = 100)
fraud.is_fraud.value_counts(normalize=True).plot.bar()
plt.show()
```

```
Fraud Distribution
is_fraud
0.0    0.99031
1.0    0.00969
Name: proportion, dtype: float64
```




```
# Check for imbalance on target variable in the test dataset
fraud_test.is_fraud.value_counts(normalize=True)
```

```
is_fraud
0.0    0.995874
1.0    0.004126
Name: proportion, dtype: float64
```

Both the datasets have high imbalnce of the target variable with the test dataset having slightly higher imbalance. At this point, lets keep the test data seperate. We will be building the model on the train dataset. If required, a validation dataset will be carved from it. The final evaluation will be done on the test dataset.

Univariate Analysis

The following columns seems of very less/ no significance in determining a fraud case. Primary reason being no model can be created based on person's name or his PII or some unique ID/ S.no. assigned. Hence, dropping them:-

- cc_num
- first
- last
- street
- trans_num

```
# Dropping the unwanted columns from both datasets
fraud.drop(['cc_num', 'first', 'last', 'street', 'trans_num'], axis=1, inplace=True)
fraud.drop(fraud.iloc[:,[0]], axis=1, inplace=True)
fraud_test.drop(['cc_num', 'first', 'last', 'street', 'trans_num'], axis=1, inplace=True)
fraud_test.drop(fraud_test.iloc[:,[0]], axis=1, inplace=True)
```

```
# Inspecting the fraud dataset
fraud.head()
```

	trans_date_trans_time	merchant	category	amt	gender	city	state	zip	lat	long	city_pop	jo
0	2019-01-01 00:00:18	fraud_Rippin, Kub and Mann	misc_net	4.97	F	Moravian Falls	NC	28654	36.0788	-81.1781	3495.0	Psychologis counsellin
1	2019-01-01 00:00:44	fraud_Heller, Gutmann and Zieme	grocery_pos	107.23	F	Orient	WA	99160	48.8878	-118.2105	149.0	Specia educationa needs teache
2	2019-01-01 00:00:51	fraud_Lind-Buckridge	entertainment	220.11	M	Malad City	ID	83252	42.1808	-112.2620	4154.0	Natur conservatio office
3	2019-01-01 00:01:16	fraud_Kutch, Hermiston and Farrell	gas_transport	45.00	M	Boulder	MT	59632	46.2306	-112.1138	1939.0	Patent attorne
4	2019-01-01 00:03:06	fraud_Keeling-Crist	misc_pos	41.96	M	Doe Hill	VA	24433	38.4207	-79.4629	99.0	Danc movemer psychotherapis

Next steps: Generate code with fraud View recommended plots

```
# Inspecting the fraud test dataset
fraud_test.head()
```

	trans_date_trans_time	merchant	category	amt	gender	city	stat
0	2020-06-21 12:14:25	fraud_Kirlin and Sons	personal_care	2.86	M	Columbia	S(
1	2020-06-21 12:14:33	fraud_Sporer-Keebler	personal_care	29.84	F	Altonah	U
2	2020-06-21 12:14:53	fraud_Swaniawski, Nitzsche and Welch	health_fitness	41.28	F	Bellmore	N
3	2020-06-21 12:15:15	fraud_Haley Group	misc_pos	60.05	M	Titusville	F
4	2020-06-21 12:15:17	fraud_Johnston-Casper	travel	3.19	M	Falmouth	M

Next steps: Generate code with fraud_test View recommended plots

```
# Converting dob to age
from datetime import date
fraud['dob'] = pd.to_datetime(fraud['dob'])
fraud['age'] = (pd.to_datetime('now') - fraud['dob'])/ np.timedelta64(1, 'Y')

fraud.drop(['dob'], axis=1, inplace=True)
fraud.head()
```

	trans_date_trans_time	merchant	category	amt	gender	city	state
0	2019-01-01 00:00:18	fraud_Rippin, Kub and Mann	misc_net	4.97	F	Moravian Falls	NC
1	2019-01-01 00:00:44	fraud_Heller, Gutmann and Zieme	grocery_pos	107.23	F	Orient	WA
2	2019-01-01 00:00:51	fraud_Lind-Buckridge	entertainment	220.11	M	Malad City	ID
3	2019-01-01 00:01:16	fraud_Kutch, Hermiston and Farrell	gas_transport	45.00	M	Boulder	MT
4	2019-01-01 00:03:06	fraud_Keeling-Crist	misc_pos	41.96	M	Doe Hill	VA

Next steps: Generate code with fraud View recommended plots

```
# Same change on the test dataset
fraud_test['dob'] = pd.to_datetime(fraud_test['dob'])
fraud_test['age'] = (pd.to_datetime('now') - fraud_test['dob'])/ np.timedelta64(1, 'Y')
```

```
fraud_test.drop(['dob'], axis=1, inplace=True)
fraud_test.head()
```

	trans_date_trans_time	merchant	category	amt	gender	city	stat
0	2020-06-21 12:14:25	fraud_Kirlin and Sons	personal_care	2.86	M	Columbia	S(
1	2020-06-21 12:14:33	fraud_Sporer-Keebler	personal_care	29.84	F	Altonah	U
2	2020-06-21 12:14:53	fraud_Swaniawski, Nitzsche and Welch	health_fitness	41.28	F	Bellmore	N
3	2020-06-21 12:15:15	fraud_Haley Group	misc_pos	60.05	M	Titusville	F
4	2020-06-21 12:15:17	fraud_Johnston-Casper	travel	3.19	M	Falmouth	M

Next steps: Generate code with fraud_test View recommended plots

```
# Seggregating data and time from trans_date_trans_time field
fraud['trans_date'] = pd.DatetimeIndex(fraud['trans_date_trans_time']).date
fraud['trans_time'] = pd.DatetimeIndex(fraud['trans_date_trans_time']).time
fraud.drop(['trans_date_trans_time'], axis=1, inplace=True)
fraud.head()
```

	merchant	category	amt	gender	city	state	zip	lat	long
0	fraud_Rippin, Kub and Mann	misc_net	4.97	F	Moravian Falls	NC	28654	36.0788	-81.1781
1	fraud_Heller, Gutmann and Zieme	grocery_pos	107.23	F	Orient	WA	99160	48.8878	-118.2105
2	fraud_Lind-Buckridge	entertainment	220.11	M	Malad City	ID	83252	42.1808	-112.2620
3	fraud_Kutch, Hermiston and Farrell	gas_transport	45.00	M	Boulder	MT	59632	46.2306	-112.1138
4	fraud_Keeling-Crist	misc_pos	41.96	M	Doe Hill	VA	24433	38.4207	-79.4629

Next steps: Generate code with fraud View recommended plots

```
# Same changes on test dataset
fraud_test['trans_date'] = pd.DatetimeIndex(fraud_test['trans_date_trans_time']).date
fraud_test['trans_time'] = pd.DatetimeIndex(fraud_test['trans_date_trans_time']).time
fraud_test.drop(['trans_date_trans_time'], axis=1, inplace=True)
fraud_test.head()
```

	merchant	category	amt	gender	city	state	zip	lat	
0	fraud_Kirlin and Sons	personal_care	2.86	M	Columbia	SC	29209.0	33.9659	-80
1	fraud_Sporer-Keebler	personal_care	29.84	F	Altonah	UT	84002.0	40.3207	-110
2	fraud_Swaniawski, Nitzsche and Welch	health_fitness	41.28	F	Bellmore	NY	11710.0	40.6729	-73
3	fraud_Haley Group	misc_pos	60.05	M	Titusville	FL	32780.0	28.5697	-80
4	fraud_Johnston-Casper	travel	3.19	M	Falmouth	MI	49632.0	44.2529	-85

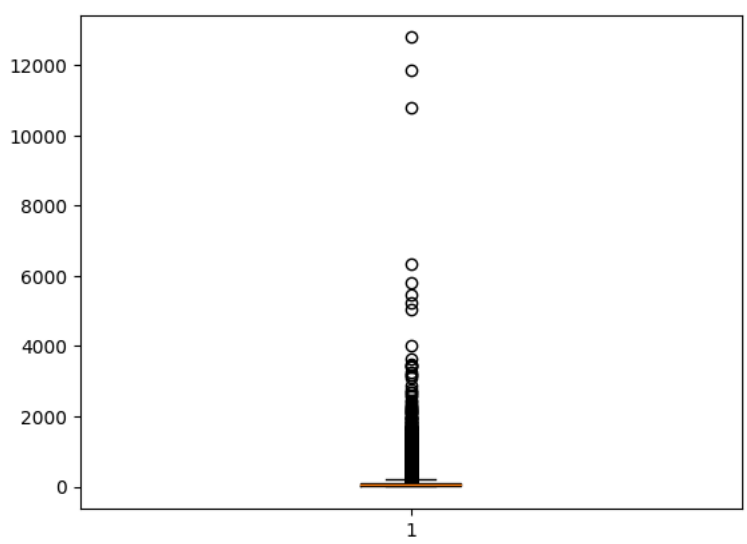
Next steps: Generate code with fraud_test View recommended plots

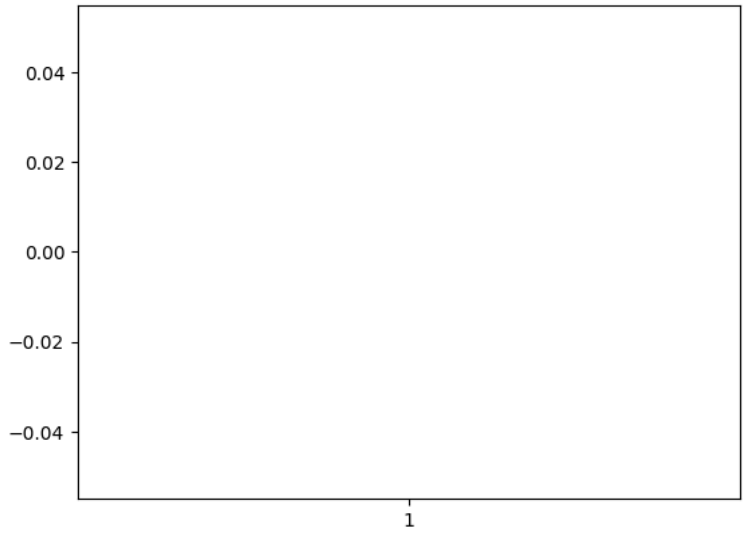
```
# Check on numeric columns for outliers
```

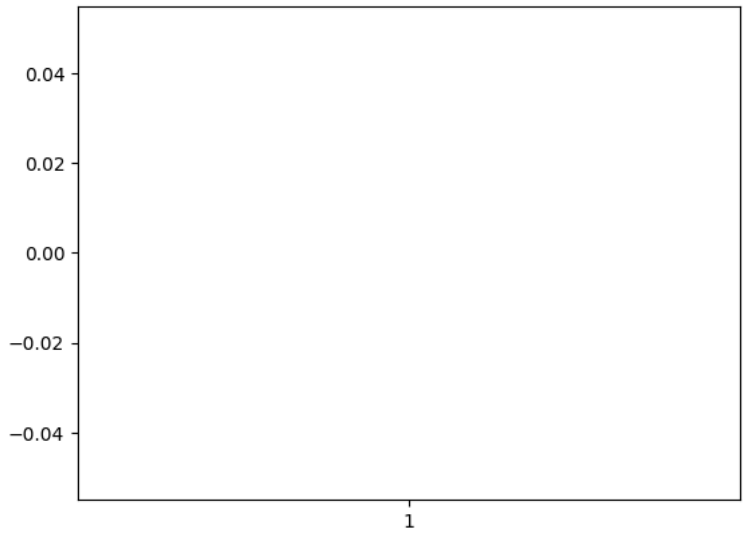
```
fraud.describe()
```

	amt	zip	lat	long	city_pop	unix_ti
count	89474.000000	89474.000000	89474.000000	89474.000000	8.947300e+04	8.947300e+
mean	71.803937	48723.084874	38.540167	-90.182776	8.952019e+04	1.327650e+
std	146.987767	26909.591388	5.076322	13.791079	3.024579e+05	1.305850e+
min	1.000000	1257.000000	20.027100	-165.672300	2.300000e+01	1.325376e+
25%	9.700000	26041.000000	34.668900	-96.786900	7.430000e+02	1.326529e+
50%	48.090000	48174.000000	39.354300	-87.461600	2.456000e+03	1.327684e+
75%	83.890000	72011.000000	41.894800	-80.128400	2.047800e+04	1.328811e+
max	12788.070000	99783.000000	65.689900	-67.950300	2.906700e+06	1.329930e+

```
# Further checking distribution of continuous variables - amt, city_pop and age columns to see if there are any valid outliers
plt.boxplot(fraud.amt)
plt.show()
plt.boxplot(fraud.city_pop)
plt.show()
plt.boxplot(fraud.age)
plt.show()
```







The age column has no outliers while amt and city_pop stastically shows outliers. However, both amount and city population can vary drastically and none of them seems very high or very low. Hence, we will consider it as valid data.

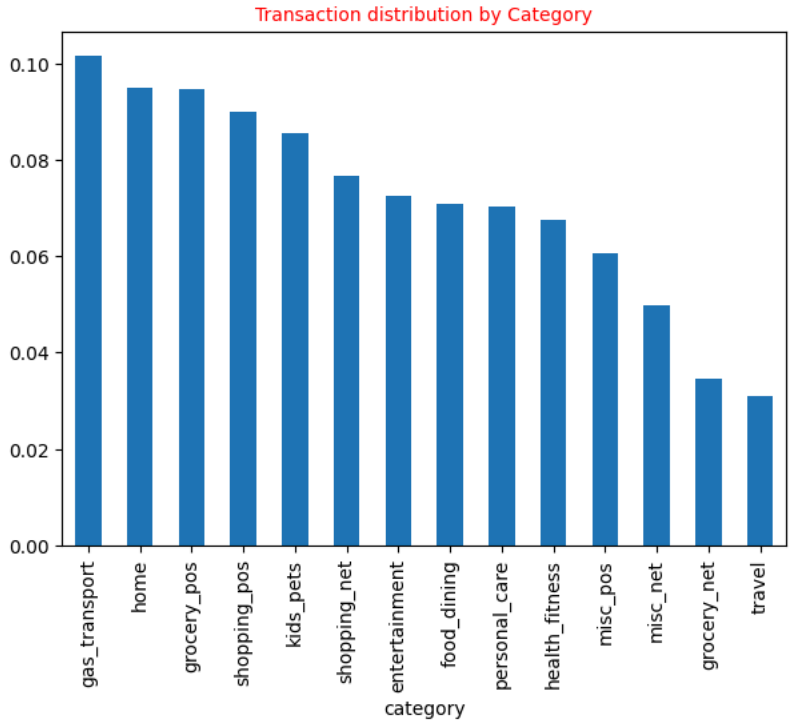
```
# Identifying all the Numeric and non numeric columns
num = []
obj = []
for i in range (0,13):
    if fraud.iloc[:,i].dtype != 'O':
        num.append(i)
    else:
        obj.append(i)
print(num)
print(obj)
col_names = fraud.columns
print(col_names)
```

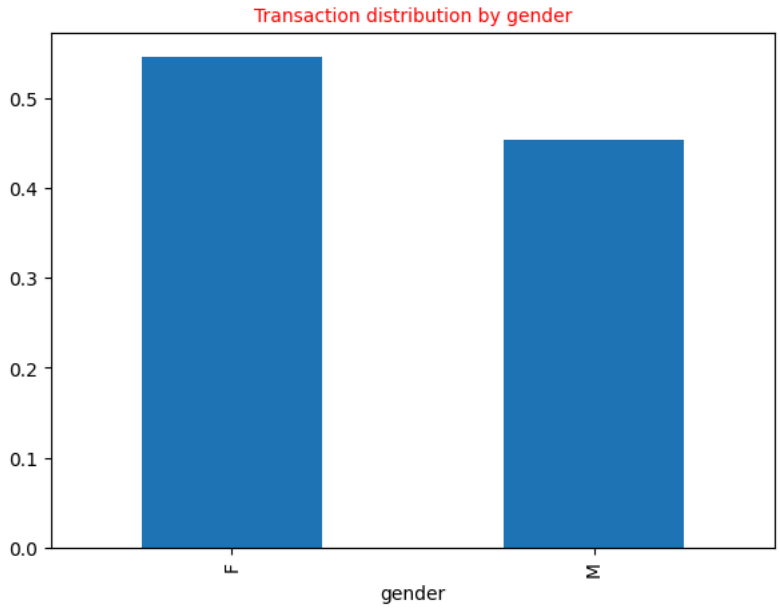
```
[2, 6, 7, 8, 9, 11, 12]
[0, 1, 3, 4, 5, 10]
Index(['merchant', 'category', 'amt', 'gender', 'city', 'state', 'zip', 'lat',
       'long', 'city_pop', 'job', 'unix_time', 'merch_lat', 'merch_long',
       'is_fraud', 'age', 'trans_date', 'trans_time'],
      dtype='object')
```

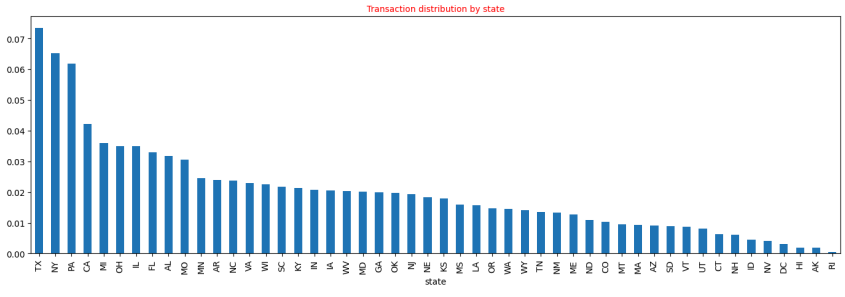
```
# Checking the distribution of object variables
for i in obj:
    print (col_names[i])
    print (fraud.iloc[:,i].value_counts(normalize=True))
    print ('*' * 50)
```

```
MO    0.030646
MN    0.024499
AR    0.024018
NC    0.023783
VA    0.022923
WI    0.022465
SC    0.021805
KY    0.021381
IN    0.020833
IA    0.020553
WV    0.020363
MD    0.020207
GA    0.019916
OK    0.019749
NJ    0.019425
NE    0.018374
KS    0.017893
MS    0.016005
LA    0.015781
OR    0.014697
WA    0.014529
WY    0.014060
TN    0.013512
NM    0.013300
ME    0.012819
ND    0.010852
CO    0.010316
MT    0.009533
MA    0.009332
AZ    0.009031
SD    0.008829
VT    0.008684
UT    0.008159
CT    0.006214
NH    0.006102
ID    0.004504
NV    0.004079
DC    0.003174
HI    0.001989
AK    0.001799
RI    0.000548
Name: proportion, dtype: float64
**************************************************
job
job
Exhibition designer               0.007388
Film/video editor                 0.007254
Naval architect                   0.006706
Surveyor, land/geomatics          0.006672
Designer, ceramics/pottery        0.006427
                                      ...
Environmental manager             0.000279
Engineer, site                    0.000134
Armed forces technical officer    0.000089
Contracting civil engineer        0.000078
Veterinary surgeon                0.000011
Name: proportion, Length: 479, dtype: float64
**************************************************
```

```
# Lets check the transaction distribution by Category, Gender and State variables
plt.figure(figsize = (7,5))
plt.title('Transaction distribution by Category', fontsize= 10, color = 'Red', fontweight = 100)
fraud.category.value_counts(normalize=True).plot.bar()
plt.show()

plt.figure(figsize = (7,5))
plt.title('Transaction distribution by gender', fontsize= 10, color = 'Red', fontweight = 100)
fraud.gender.value_counts(normalize=True).plot.bar()
plt.show()

plt.figure(figsize = (17,5))
plt.title('Transaction distribution by state', fontsize= 10, color = 'Red', fontweight = 100)
fraud.state.value_counts(normalize=True).plot.bar()
plt.show()
```







Bi-Variate Analysis

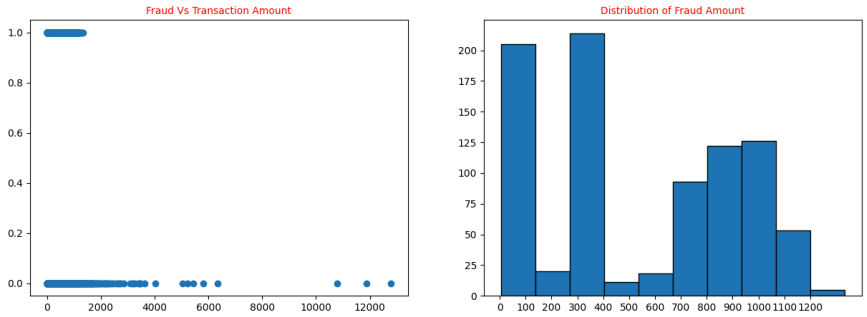
Check for the behaviour of various columns against the is_fraud column

```
fraud.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 89474 entries, 0 to 89473
Data columns (total 18 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   merchant    89474 non-null  object
 1   category    89474 non-null  object
 2   amt         89474 non-null  float64
 3   gender      89474 non-null  object
 4   city        89474 non-null  object
 5   state       89474 non-null  object
 6   zip         89474 non-null  int64
 7   lat         89474 non-null  float64
 8   long        89474 non-null  float64
 9   city_pop    89473 non-null  float64
 10  job         89473 non-null  object
 11  unix_time   89473 non-null  float64
 12  merch_lat   89473 non-null  float64
 13  merch_long  89473 non-null  float64
 14  is_fraud    89473 non-null  float64
 15  age         89473 non-null  float64
 16  trans_date  89474 non-null  object
 17  trans_time  89474 non-null  object
dtypes: float64(9), int64(1), object(8)
memory usage: 12.3+ MB
```

Start coding or generate with AI.

```
# Fraud Vs Amount
plt.figure(figsize=[15,5])
plt.subplot(1,2,1)
plt.title('Fraud Vs Transaction Amount', fontsize= 10, color = 'Red', fontweight = 100)
plt.scatter(fraud.amt, fraud.is_fraud)
plt.subplot(1,2,2)
#fraud.groupby('is_fraud')['amt'].mean().plot.bar()
#plt.xticks((0,1),['Not Fraud', 'Fraud'])
#plt.xticks(rotation=0)
temp = fraud[fraud.is_fraud == 1]
plt.title('Distribution of Fraud Amount', fontsize= 10, color = 'Red', fontweight = 100)
plt.hist(temp.amt, edgecolor='Black')
plt.xticks(np.arange(0, 1300, step=100))

plt.show()
```



As can be seen from above, frauds are happening in transactions with lower amount hence indicating there is a relation in them.

```
# Fraud transactions Vs merchant
# Total number of transactions per merchant
merch_tran_total = fraud.sort_values('merchant').groupby('merchant').count()['is_fraud']
merch_tran_total.head()
```

```
merchant
fraud_Abbott-Rogahn                151
fraud_Abbott-Steuber               121
```

```
fraud_Abernathy and Sons            107
fraud_Abshire PLC                   119
fraud_Adams, Kovacek and Kuhlman     59
Name: is_fraud, dtype: int64
```

```
# Total fraud transactions per merchant
merch_tran_fraud = fraud[fraud.is_fraud == 1]['merchant'].value_counts()
merch_tran_fraud.head()
```

```
merchant
fraud_Gleason-Macejkovic           9
fraud_Koepp-Parker                 9
fraud_Kilback LLC                  9
fraud_Labadie, Treutel and Bode    9
fraud_Bauch-Raynor                 8
Name: count, dtype: int64
```

```
# Percent of fraud transactions per merchant
fraud_perc = merch_tran_fraud/ merch_tran_total * 100
fraud_perc.sort_values(ascending=False)
```

```
merchant
fraud_Gleason-Macejkovic            6.164384
fraud_Towne, Greenholt and Koepp    6.140351
fraud_Labadie, Treutel and Bode     6.081081
fraud_Baumbach, Feeney and Morar    5.833333
fraud_Koepp-Parker                  5.294118
                                        ...
fraud_Zboncak LLC                       NaN
fraud_Zboncak Ltd                       NaN
fraud_Zemlak, Tillman and Cremin        NaN
fraud_Ziemann-Waters                    NaN
fraud_Zulauf LLC                        NaN
Length: 693, dtype: float64
```

Baring a few merchants, most of them have equal distribution of transactions and hence this field may play important role in the model. Changing the alphabetic values to numeric as models expects numeric data.

```
# variable transformation - merchant
from sklearn.preprocessing import LabelEncoder
label_encoder = LabelEncoder()
fraud.merchant = label_encoder.fit_transform(fraud.merchant)
fraud_test.merchant = label_encoder.fit_transform(fraud_test.merchant)
```

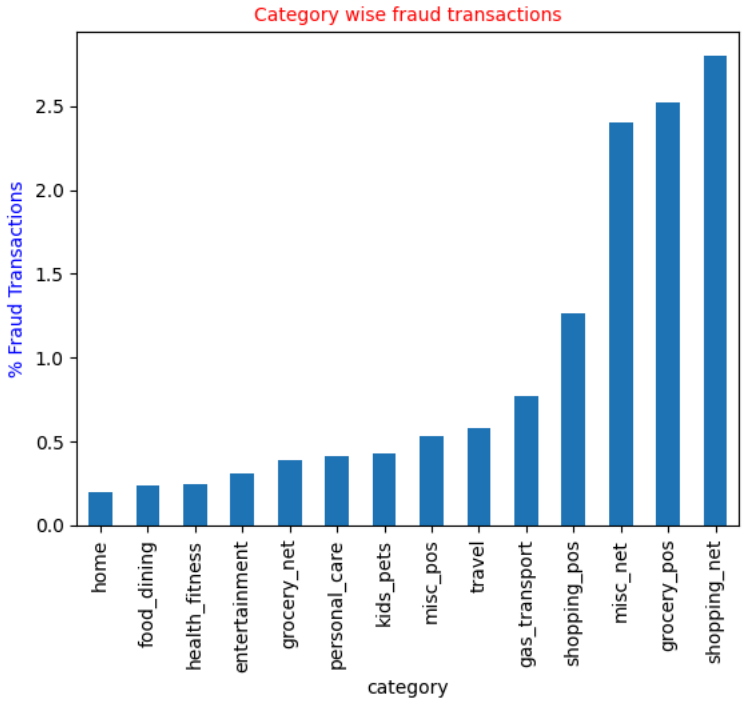
```
# Fraud transactions Vs City
# Percent distribution of fraud based on city
city_tran_total = fraud.sort_values('city').groupby('city').count()['is_fraud']
city_tran_fraud = fraud[fraud.is_fraud == 1]['city'].value_counts()
fraud_perc = city_tran_fraud/ city_tran_total * 100
fraud_perc.sort_values(ascending=False).head()
```

```
city
Crouse            100.0
La Grande         100.0
Greenport         100.0
Madisonville      100.0
West Frankfort    100.0
dtype: float64
```

As can be seen, few cities have all transactions as fraud. All these cities have low transaction rate. There are 58 such cities.

```
# Transforming alphabetic city data into numeric to be processed by the model
fraud.city = label_encoder.fit_transform(fraud.city)
fraud_test.city = label_encoder.fit_transform(fraud_test.city)
```

```
# category Vs fraud
# Percent distribution of fraud based on transaction category
cat_tran_total = fraud.sort_values('category').groupby('category').count()['is_fraud']
cat_tran_fraud = fraud[fraud.is_fraud == 1]['category'].value_counts()
fraud_perc = cat_tran_fraud/ cat_tran_total * 100
plt.title('Category wise fraud transactions', fontsize= 10, color = 'Red', fontweight = 100)
plt.ylabel('% Fraud Transactions', fontdict = {'fontsize': 10, 'color': 'Blue', 'fontweight' : '300'})
fraud_perc.sort_values().plot.bar()
plt.show()
```

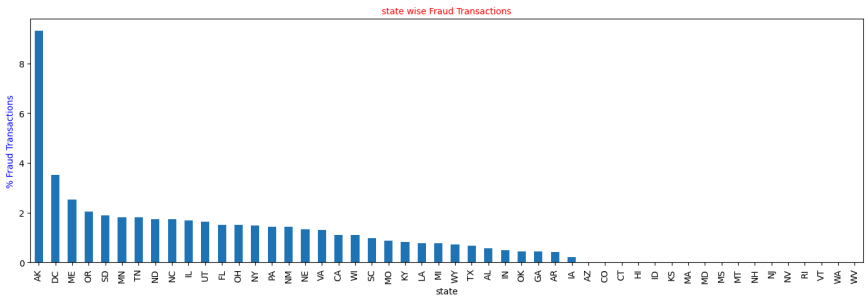



```
# Transforming alphabetic category data into numeric to be processed by the model
fraud.category = label_encoder.fit_transform(fraud.category)
fraud_test.category = label_encoder.fit_transform(fraud_test.category)
```

```
# Gender Vs Fraud
# Percent distribution of fraud based on Gender
gen_tran_total = fraud.sort_values('gender').groupby('gender').count()['is_fraud']
gen_tran_fraud = fraud[fraud.is_fraud == 1]['gender'].value_counts()
fraud_perc = gen_tran_fraud/ gen_tran_total * 100
plt.title('Gender wise Fraud Transactions', fontsize= 10, color = 'Red', fontweight = 100)
fraud_perc.sort_values().plot.bar()
plt.show()
```


```
# Transforming alphabetic gender data into numeric to be processed by the model
fraud.gender = fraud.gender.map({'M': 1, "F": 0})
fraud_test.gender = fraud_test.gender.map({'M': 1, "F": 0})
```

```
# state Vs fraud
# Percent distribution of fraud based on State
plt.figure(figsize = (17,5))
state_tran_total = fraud.sort_values('state').groupby('state').count()['is_fraud']
state_tran_fraud = fraud[fraud.is_fraud == 1]['state'].value_counts()
fraud_perc = state_tran_fraud/ state_tran_total * 100
plt.title('state wise Fraud Transactions', fontsize= 10, color = 'Red', fontweight = 100)
plt.ylabel('% Fraud Transactions', fontdict = {'fontsize': 10, 'color': 'Blue', 'fontweight' : '300'})
fraud_perc.sort_values(ascending=False).plot.bar()
plt.show()
```




```
fraud_perc.sort_values(ascending=False).head()
```

```
state
AK    9.316770
DC    3.521127
ME    2.528335
OR    2.053232
SD    1.898734
dtype: float64
```

This is very significant. While the number of transactions in DE is very less, all of them are fraud transaction. Rest all the states have very low fraud transaction.

```
# Transforming alphabetic state data into numeric to be processed by the model
fraud.state = label_encoder.fit_transform(fraud.state)
fraud_test.state = label_encoder.fit_transform(fraud_test.state)
```

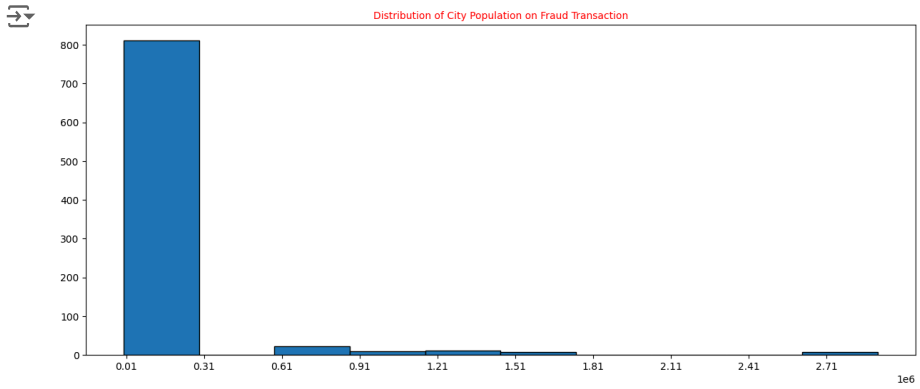
```
# Job Vs Fraud
# Percent distribution of fraud based on Job
job_tran_total = fraud.sort_values('job').groupby('job').count()['is_fraud']
job_tran_fraud = fraud[fraud.is_fraud == 1]['job'].value_counts()
fraud_perc = job_tran_fraud/ job_tran_total * 100
fraud_perc.sort_values(ascending=False).head(20)
```

```
job
Engineer, site                          100.000000
Veterinary surgeon                      100.000000
Armed forces technical officer          100.000000
Contracting civil engineer              100.000000
Solicitor, Scotland                      28.205128
Magazine journalist                      27.450980
Marketing executive                      27.027027
Charity officer                          24.000000
Restaurant manager, fast food            19.148936
Oncologist                               18.181818
English as a foreign language teacher    17.647059
Clinical cytogeneticist                  17.073171
Community development worker             17.073171
Economist                                15.789474
Horticultural consultant                 12.727273
Engineer, building services              11.494253
Medical technical officer                11.458333
Designer, textile                        10.975610
Event organiser                          10.476190
Acupuncturist                            10.416667
dtype: float64
```

There seems certain jobs that have real high % of fraud transactions.

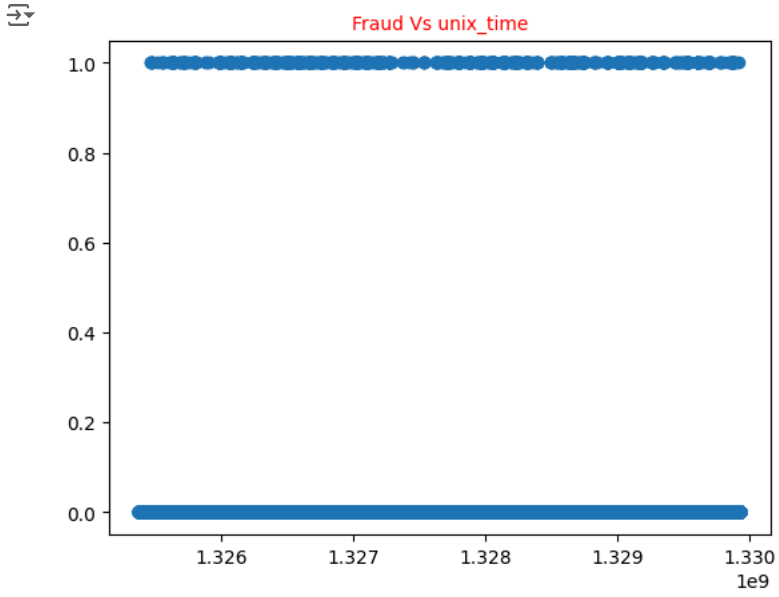
```
# Transforming alphabetic job data into numeric to be processed by the model
fraud.job = label_encoder.fit_transform(fraud.job)
fraud_test.job = label_encoder.fit_transform(fraud_test.job)
```

```
# Fraud Vs City Population
plt.figure(figsize=[15,6])
temp = fraud[fraud.is_fraud == 1]
plt.title('Distribution of City Population on Fraud Transaction', fontsize= 10, color = 'Red', fontweight = 100)
plt.hist(temp.city_pop, edgecolor='Black')
plt.xticks(np.arange(10000, 3000000, step=300000))
plt.show()
```

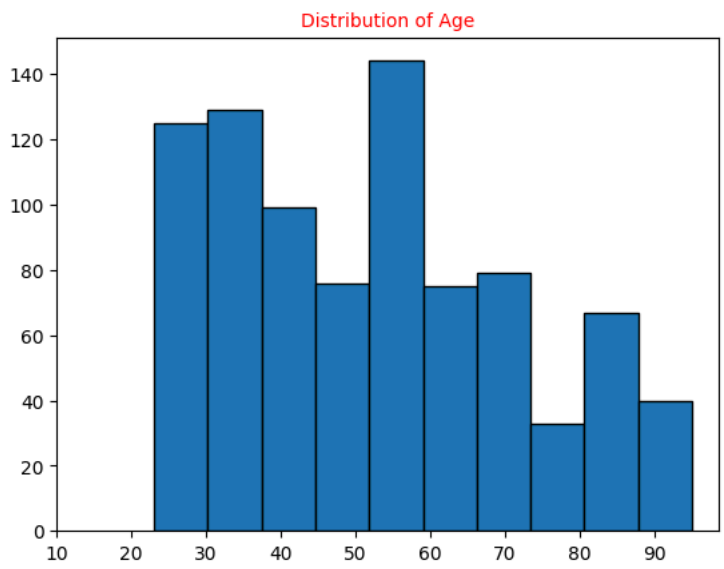


Cities with less population, tends to have more fraud cases.

```
# Fraud Vs Unix Time
plt.title('Fraud Vs unix_time', fontsize= 10, color = 'Red', fontweight = 100)
plt.scatter(fraud.unix_time, fraud.is_fraud)
plt.show()
```



```
# Fraud Vs Age
temp = fraud[fraud.is_fraud == 1]
plt.title('Distribution of Age', fontsize= 10, color = 'Red', fontweight = 100)
plt.hist(temp.age, edgecolor='Black')
plt.xticks(np.arange(10, 100, step=10))
plt.show()
```



So, people in age group 50 to 60 tends to be slightly more victims of fraud.

```
# Fraud Vs Zip
zip_tran_total = fraud.sort_values('zip').groupby('zip').count()['is_fraud']
zip_tran_fraud = fraud[fraud.is_fraud == 1]['zip'].value_counts()
fraud_perc = zip_tran_fraud/ zip_tran_total * 100
fraud_perc.sort_values(ascending=False).head(25)
```

```
zip
70447    100.000000
78208    100.000000
97850    100.000000
11944    100.000000
37411    100.000000
48436    100.000000
95688    100.000000
54980    100.000000
62896    100.000000
28033    100.000000
77027    100.000000
45638     30.232558
56117     29.268293
68869     28.205128
62266     27.450980
92267     27.027027
34120     26.829268
71832     25.714286
92101     24.444444
23666     24.000000
87116     24.000000
23106     23.913043
11955     23.728814
55606     21.839080
47863     21.428571
dtype: float64
```

As is evident from above stats, there are perticular ZIP codes that have 100% frauds.

```
# Fraud Vs lat
lat_tran_total = fraud.sort_values('lat').groupby('lat').count()['is_fraud']
lat_tran_fraud = fraud[fraud.is_fraud == 1]['lat'].value_counts()
fraud_perc = lat_tran_fraud/ lat_tran_total * 100
fraud_perc.sort_values(ascending=False).head()
```

```
lat
45.3304    100.0
29.4400    100.0
29.7396    100.0
37.8979    100.0
30.4287    100.0
dtype: float64
```

As is evident from above stats, there are perticular latitudes codes that have 100% frauds.

```
# Fraud Vs long
long_tran_total = fraud.sort_values('long').groupby('long').count()['is_fraud']
long_tran_fraud = fraud[fraud.is_fraud == 1]['long'].value_counts()
fraud_perc = long_tran_fraud/ long_tran_total * 100
fraud_perc.sort_values(ascending=False).head()
```

```
long
-88.7712     100.0
-72.3674     100.0
-121.9887    100.0
-90.1773     100.0
-118.0852    100.0
dtype: float64
```

```
# Fraud Vs merch_lat
lat_tran_total = fraud.sort_values('merch_lat').groupby('merch_lat').count()['is_fraud']
lat_tran_fraud = fraud[fraud.is_fraud == 1]['merch_lat'].value_counts()
fraud_perc = lat_tran_fraud/ lat_tran_total * 100
fraud_perc.sort_values(ascending=False).head()
```

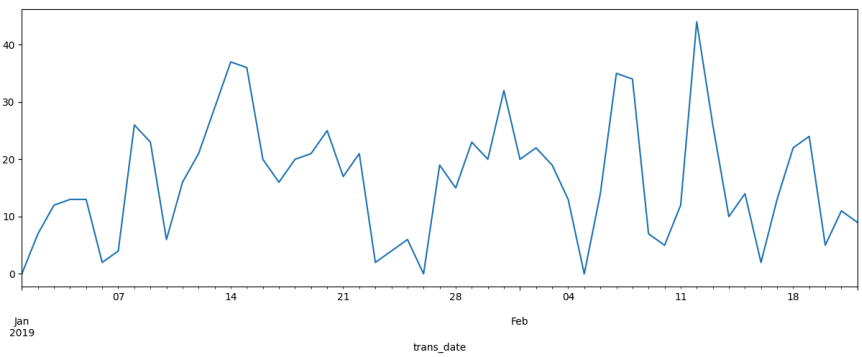
```
merch_lat
25.568094    100.0
41.449762    100.0
41.484523    100.0
41.507858    100.0
41.511631    100.0
dtype: float64
```

```
# Fraud Vs merch_long
long_tran_total = fraud.sort_values('merch_long').groupby('merch_long').count()['is_fraud']
long_tran_fraud = fraud[fraud.is_fraud == 1]['merch_long'].value_counts()
fraud_perc = long_tran_fraud/ long_tran_total * 100
fraud_perc.sort_values(ascending=False).head()
```

```
merch_long
-166.550779    100.0
-81.094488     100.0
-81.300327     100.0
-81.278516     100.0
-81.254332     100.0
dtype: float64
```

- There are multiple demographies - Zip, City, States, Latitudes, Longitudes and Job types that have only Fraud transactions.
- Even though they have 100% frauds, the number of transactions is very low. For Example State DE had only 9 transactions in 2 years. Hence, it is very less likely to impact the model.

```
# Fraud Vs trans_date
fraud['trans_date'] =  pd.to_datetime(fraud['trans_date'])
plt.figure(figsize=[15,5])
fraud.groupby(['trans_date'])['is_fraud'].sum().plot()
plt.show()
```



Now its time to change date and time to a format more acceptable for modelling. Before that, lets pull some stats required for Cost sheet. Also, it may be noticed that the train data is for 1.5 years (full 2019 till mid of 2020) and test data is for last 6 months of 2020. This way we will be able to build model on 1.5 year of data and test it on future data and hence check model performance in future. We will do the Cost Benifit analysis on the entire data.

```
# Total number of months
date_fraud = fraud.trans_date
date_fraud_test = pd.to_datetime(fraud_test.trans_date)
date_fraud = date_fraud.dt.to_period('M')
date_fraud_test = date_fraud_test.dt.to_period('M')
date = pd.concat([date_fraud, date_fraud_test])
print ('total number of records in file: ', date.size)
```

```
total number of records in file:  182794
```

```
print ('Total number of months: ', date.value_counts().size)
```

```
Total number of months:  4
```

```
print ('Average transactions per month: ', round(date.size/date.value_counts().size,0) )
```

```
Average transactions per month:  45698.0
```

```
# Extracting fraud data
temp1 = fraud[['amt', 'is_fraud']]
temp2 = fraud_test[['amt', 'is_fraud']]
temp = pd.concat([temp1, temp2])
temp.shape
```

```
(182794, 2)
```

```
# Average frauds per month
fraud_temp = temp[temp.is_fraud == 1]
print ('Average fraud transactions per month: ', round(fraud_temp.shape[0]/ date.value_counts().size,0))
```

```
Average fraud transactions per month:  313.0
```

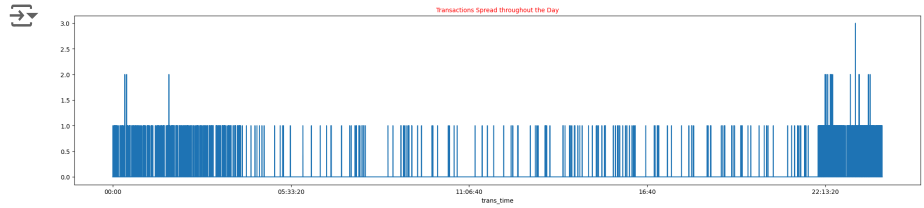
```
# Average amount per fraud transaction
print ('Average amount per fraud transaction: ', round(sum(fraud_temp.amt)/ fraud_temp.shape[0], 2))
```

```
Average amount per fraud transaction:  522.31
```

```
# Average amount per fraud transaction
print ('max fraud amount : ' , max(fraud_temp.amt))
```

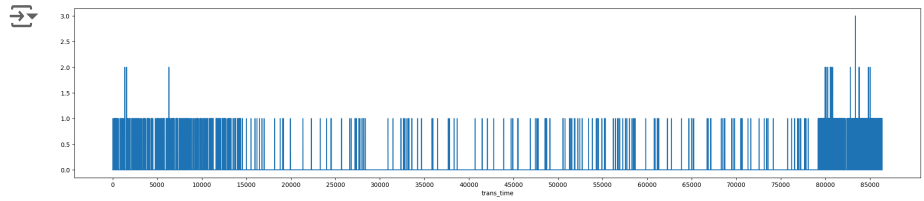
```
max fraud amount :  1334.07
```

```
# Fraud Vs trans_time
import datetime as dt
fraud.trans_date = fraud.trans_date.map(dt.datetime.toordinal)
plt.figure(figsize=[25,5])
plt.title('Transactions Spread throughout the Day', fontsize= 10, color = 'Red', fontweight = 100)
fraud.groupby(['trans_time'])['is_fraud'].sum().plot()
plt.show()
```



So, late nights and early mornings are the most prone time for frauds. Highest frequency of frauds is between 10 pm to 12 am. 12 am to 4:00 am also shows very high frequency of fraud transactions.

```
# Converting trans_time into seconds & ploting the above graph again
fraud.trans_time = pd.to_datetime(fraud.trans_time,format='%H:%M:%S')
fraud.trans_time = 3600 * pd.DatetimeIndex(fraud.trans_time).hour + 60 * pd.DatetimeIndex(fraud.trans_time).minute + pd.DatetimeIndex(fr
plt.figure(figsize=[25,5])
plt.xticks(np.arange(0,90000,5000))
fraud.groupby(['trans_time'])['is_fraud'].sum().plot()
plt.show()
```




```
# Similar data-time changes in test dataset
fraud_test['trans_date'] =  pd.to_datetime(fraud_test['trans_date'])
fraud_test.trans_date = fraud_test.trans_date.map(dt.datetime.toordinal)
fraud_test.trans_time = pd.to_datetime(fraud_test.trans_time,format='%H:%M:%S')
fraud_test.trans_time = 3600 * pd.DatetimeIndex(fraud_test.trans_time).hour + 60 * pd.DatetimeIndex(fraud_test.trans_time).minute + pd.D
```

```
print ('train : ', fraud.shape)
print ('test : ', fraud_test.shape)
```

```
train :  (89474, 18)
test :  (93320, 18)
```

```
fraud.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 89474 entries, 0 to 89473
Data columns (total 18 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   merchant    89474 non-null  int64
 1   category    89474 non-null  int64
 2   amt         89474 non-null  float64
 3   gender      89474 non-null  int64
 4   city        89474 non-null  int64
 5   state       89474 non-null  int64
 6   zip         89474 non-null  int64
 7   lat         89474 non-null  float64
 8   long        89474 non-null  float64
 9   city_pop    89473 non-null  float64
```

```
 10  job         89474 non-null  int64
 11  unix_time   89473 non-null  float64
 12  merch_lat   89473 non-null  float64
 13  merch_long  89473 non-null  float64
 14  is_fraud    89473 non-null  float64
 15  age         89473 non-null  float64
 16  trans_date  89474 non-null  int64
 17  trans_time  89474 non-null  int32
dtypes: float64(9), int32(1), int64(8)
memory usage: 11.9 MB
```

```
fraud_test.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 93320 entries, 0 to 93319
Data columns (total 18 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   merchant    93320 non-null  int64
 1   category    93320 non-null  int64
 2   amt         93319 non-null  float64
 3   gender      93319 non-null  float64
 4   city        93320 non-null  int64
 5   state       93320 non-null  int64
 6   zip         93319 non-null  float64
 7   lat         93319 non-null  float64
 8   long        93319 non-null  float64
 9   city_pop    93319 non-null  float64
 10  job         93320 non-null  int64
 11  unix_time   93319 non-null  float64
 12  merch_lat   93319 non-null  float64
 13  merch_long  93319 non-null  float64
 14  is_fraud    93319 non-null  float64
 15  age         93319 non-null  float64
 16  trans_date  93320 non-null  int64
 17  trans_time  93320 non-null  int32
dtypes: float64(11), int32(1), int64(6)
memory usage: 12.5 MB
```

Train Test Split

Splitting the data into train & validate datasets. The test dataset provided will be used for final evaluation.

```
X = fraud.drop('is_fraud', axis=1)
X.head()
```

	merchant	category	amt	gender	city	state	zip	lat	long	city_pop
0	514	8	4.97	0	499	26	28654	36.0788	-81.1781	3495.0
1	241	4	107.23	0	574	46	99160	48.8878	-118.2105	149.0
2	390	0	220.11	1	445	12	83252	42.1808	-112.2620	4154.0
3	360	2	45.00	1	81	25	59632	46.2306	-112.1138	1939.0
4	297	9	41.96	1	205	44	24433	38.4207	-79.4629	99.0

Next steps: Generate code with X View recommended plots

```
y = fraud['is_fraud']
y.head()
```

```
0    0.0
1    0.0
2    0.0
3    0.0
4    0.0
Name: is_fraud, dtype: float64
```

```
# Splitting the data into train and test such that ration of fraud is same in both
from sklearn.model_selection import train_test_split

# Handle missing values in 'y' before splitting
y = fraud['is_fraud'].dropna()  # Drop rows with NaN in 'is_fraud' column

# Update X to match the rows in the modified y
X = fraud.loc[y.index].drop('is_fraud', axis=1)

X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.7, test_size=0.3, stratify=y, random_state=100)
```

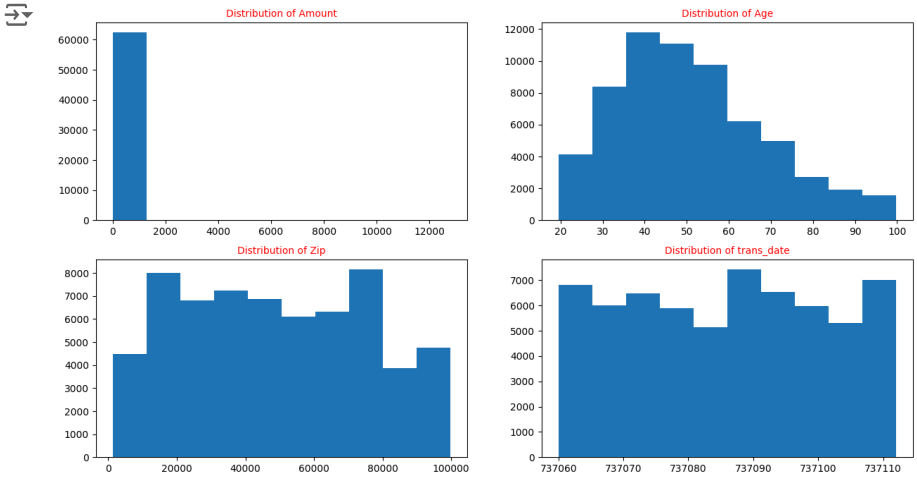
```
# Inspecting the train and test datasets
print ('Train Dataset : ', X_train.shape)
print ('Test Dataset : ', X_test.shape)
```

```
Train Dataset :  (62631, 17)
Test Dataset :  (26842, 17)
```

```
# Creating X_final & y_final from the test dataset provided by Kaggle. This will be used for final evaluation of the model
X_final = fraud_test.drop('is_fraud', axis=1)
y_final = fraud_test['is_fraud']
```

Data Scaling

```
# Check distribution of few variables
plt.figure(figsize=[15,8])
plt.subplot(2,2,1)
plt.title('Distribution of Amount', fontsize= 10, color = 'Red', fontweight = 100)
plt.hist(X_train.amt)
plt.subplot(2,2,2)
plt.title('Distribution of Age', fontsize= 10, color = 'Red', fontweight = 100)
plt.hist(X_train.age)
plt.subplot(2,2,3)
plt.title('Distribution of Zip', fontsize= 10, color = 'Red', fontweight = 100)
plt.hist(X_train.zip)
plt.subplot(2,2,4)
plt.title('Distribution of trans_date', fontsize= 10, color = 'Red', fontweight = 100)
plt.hist(X_train.trans_date)
plt.show()
```




```
X_train.describe()
```

	merchant	category	amt	gender	city	state	zip	lat	long	
count	62631.00000	62631.000000	62631.000000	62631.000000	62631.000000	62631.000000	62631.000000	62631.000000	62631.000000	6.26
mean	343.27959	6.243106	71.692191	0.453817	420.453897	25.746244	48712.711692	38.545548	-90.188435	8.92
std	200.89700	3.915323	148.057190	0.497867	243.420938	14.123402	26928.429022	5.067741	13.789425	3.04
min	0.00000	0.000000	1.000000	0.000000	0.000000	0.000000	1257.000000	20.027100	-165.672300	2.30
25%	166.00000	3.000000	9.650000	0.000000	210.000000	14.000000	26041.000000	34.668900	-96.786900	7.43
50%	347.00000	6.000000	48.100000	0.000000	418.000000	26.000000	48154.000000	39.354300	-87.476900	2.44
75%	514.00000	10.000000	83.945000	1.000000	635.000000	37.000000	72011.000000	41.894800	-80.128400	2.03
max	692.00000	13.000000	12788.070000	1.000000	844.000000	49.000000	99783.000000	65.689900	-67.950300	2.90

```
# Identify the variables to be scaled
vars_to_scale = ['merchant', 'category', 'amt', 'city', 'state', 'zip', 'lat', 'long', 'city_pop', 'job', 'unix_time', 'merch_lat', 'mer
```

```
# Lets try various scalers available in Scikit library. At optimum time, will finalize one.
# Based on various test results, QuantileTransformer with Gaussian distribution seems best suited.
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import PowerTransformer
from sklearn.preprocessing import QuantileTransformer
from sklearn.preprocessing import RobustScaler
#scaler = MinMaxScaler()
#scaler = StandardScaler()
#scaler = PowerTransformer()
scaler = QuantileTransformer(output_distribution='normal')
#scaler = RobustScaler()
```

```
# Scaling
X_train[vars_to_scale] = scaler.fit_transform(X_train[vars_to_scale])
X_test[vars_to_scale] = scaler.transform(X_test[vars_to_scale])
X_final[vars_to_scale] = scaler.transform(X_final[vars_to_scale])
X_train.describe()
```

	merchant	category	amt	gender	city	state	zip	lat	long	
count	62631.000000	62631.000000	62631.000000	62631.000000	62631.000000	62631.000000	62631.000000	62631.000000	62631.000000	626
mean	-0.016016	-0.138103	-0.002761	0.453817	-0.025952	0.032993	-0.017791	-0.005210	-0.001291	
std	1.024542	1.815361	0.987858	0.497867	1.016803	1.132183	1.009222	1.013139	1.023979	
min	-5.199338	-5.199338	-5.199338	0.000000	-5.199338	-5.199338	-5.199338	-5.199338	-5.199338	
25%	-0.683178	-0.628724	-0.662203	0.000000	-0.712198	-0.658043	-0.678433	-0.678433	-0.667419	
50%	-0.011291	-0.030114	-0.004962	0.000000	-0.007946	-0.022584	-0.024423	0.002509	0.005018	
75%	0.664287	0.726606	0.656420	1.000000	0.639462	0.637923	0.656486	0.670557	0.667419	
max	5.199338	5.199338	5.199338	1.000000	5.199338	5.199338	5.199338	5.199338	5.199338	

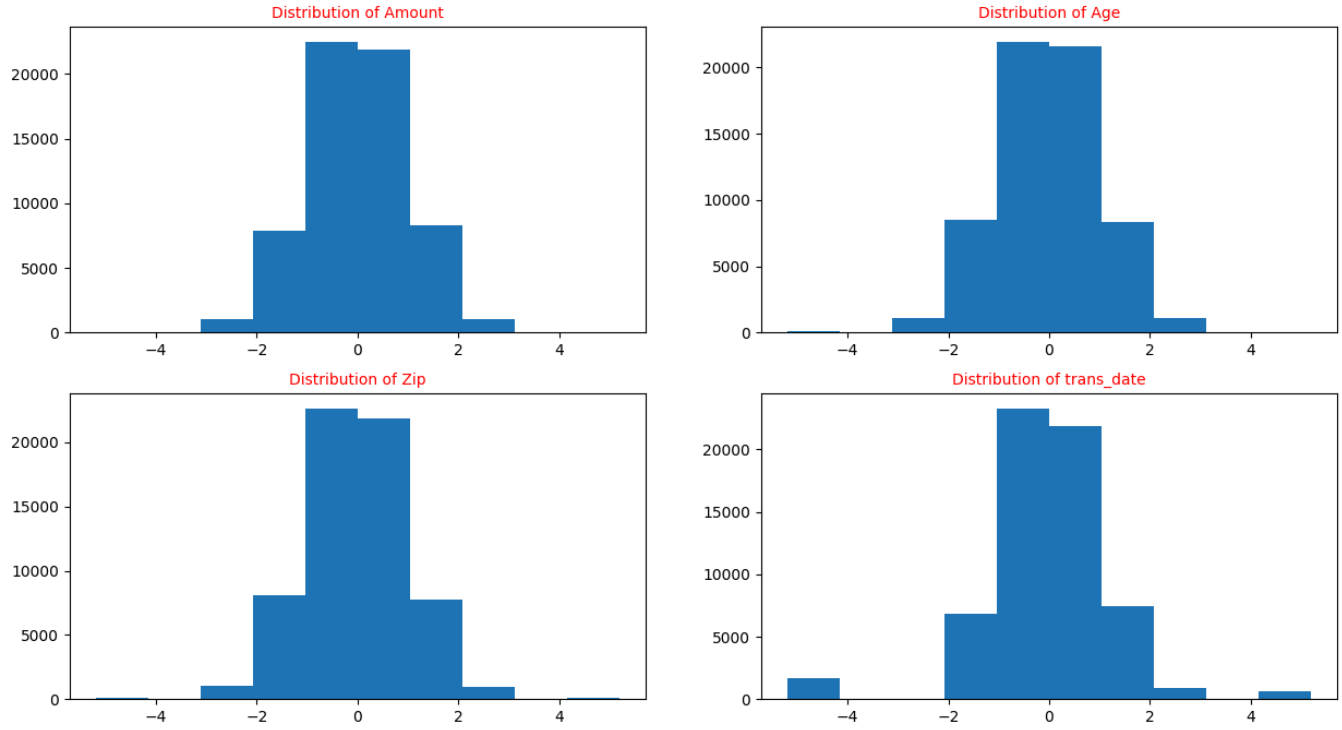
```
# Inspection after scaling
X_test.describe()
```

	merchant	category	amt	gender	city	state	zip	lat	long	
count	26842.000000	26842.000000	26842.000000	26842.000000	2.684200e+04	26842.000000	26842.000000	26842.000000	26842.000000	26
mean	-0.029232	-0.167030	-0.000383	0.453766	-1.980605e-02	0.039621	-0.015989	-0.009554	-0.000980	
std	1.021952	1.831016	0.989353	0.497867	1.018501e+00	1.137355	1.004493	1.019141	1.021217	
min	-5.199338	-5.199338	-5.199338	0.000000	-5.199338e+00	-5.199338	-5.199338	-5.199338	-5.199338	
25%	-0.695908	-0.628724	-0.647950	0.000000	-7.072736e-01	-0.658043	-0.678433	-0.678433	-0.668549	
50%	-0.020074	-0.030114	-0.005836	0.000000	6.960578e-17	-0.022584	-0.022584	0.007527	0.011291	
75%	0.651824	0.726606	0.653724	1.000000	6.440862e-01	0.637923	0.656486	0.670557	0.667419	
max	5.199338	5.199338	5.199338	1.000000	5.199338e+00	5.199338	5.199338	5.199338	5.199338	

```
# Inspection after scaling
X_final.describe()
```

	merchant	category	amt	gender	city	state	zip	lat	long	
count	93320.000000	93320.000000	93319.000000	93319.000000	93320.000000	93320.000000	93319.000000	93319.000000	93319.000000	933
mean	-0.013026	-0.145803	-0.016852	0.451869	-0.043666	0.039768	-0.016626	-0.004154	-0.002917	
std	1.030152	1.826655	0.974827	0.497681	0.982717	1.140073	1.002070	1.014108	1.016846	
min	-5.199338	-5.199338	-5.199338	0.000000	-5.199338	-5.199338	-5.199338	-5.199338	-5.199338	
25%	-0.683178	-0.628724	-0.665852	0.000000	-0.710605	-0.658043	-0.673702	-0.678433	-0.668549	
50%	-0.015055	-0.030114	-0.026527	0.000000	-0.018820	0.025094	-0.022584	0.007527	0.011291	
75%	0.675277	0.726606	0.638247	1.000000	0.632132	0.637923	0.656486	0.675523	0.654930	
max	5.199338	5.199338	5.199338	1.000000	2.511791	5.199338	5.199338	5.199338	5.199338	

```
# Lets check the distribution after scaling
plt.figure(figsize=[15,8])
plt.subplot(2,2,1)
plt.title('Distribution of Amount', fontsize= 10, color = 'Red', fontweight = 100)
plt.hist(X_train.amt)
plt.subplot(2,2,2)
plt.title('Distribution of Age', fontsize= 10, color = 'Red', fontweight = 100)
plt.hist(X_train.age)
plt.subplot(2,2,3)
plt.title('Distribution of Zip', fontsize= 10, color = 'Red', fontweight = 100)
plt.hist(X_train.zip)
plt.subplot(2,2,4)
plt.title('Distribution of trans_date', fontsize= 10, color = 'Red', fontweight = 100)
plt.hist(X_train.trans_date)
plt.show()
```

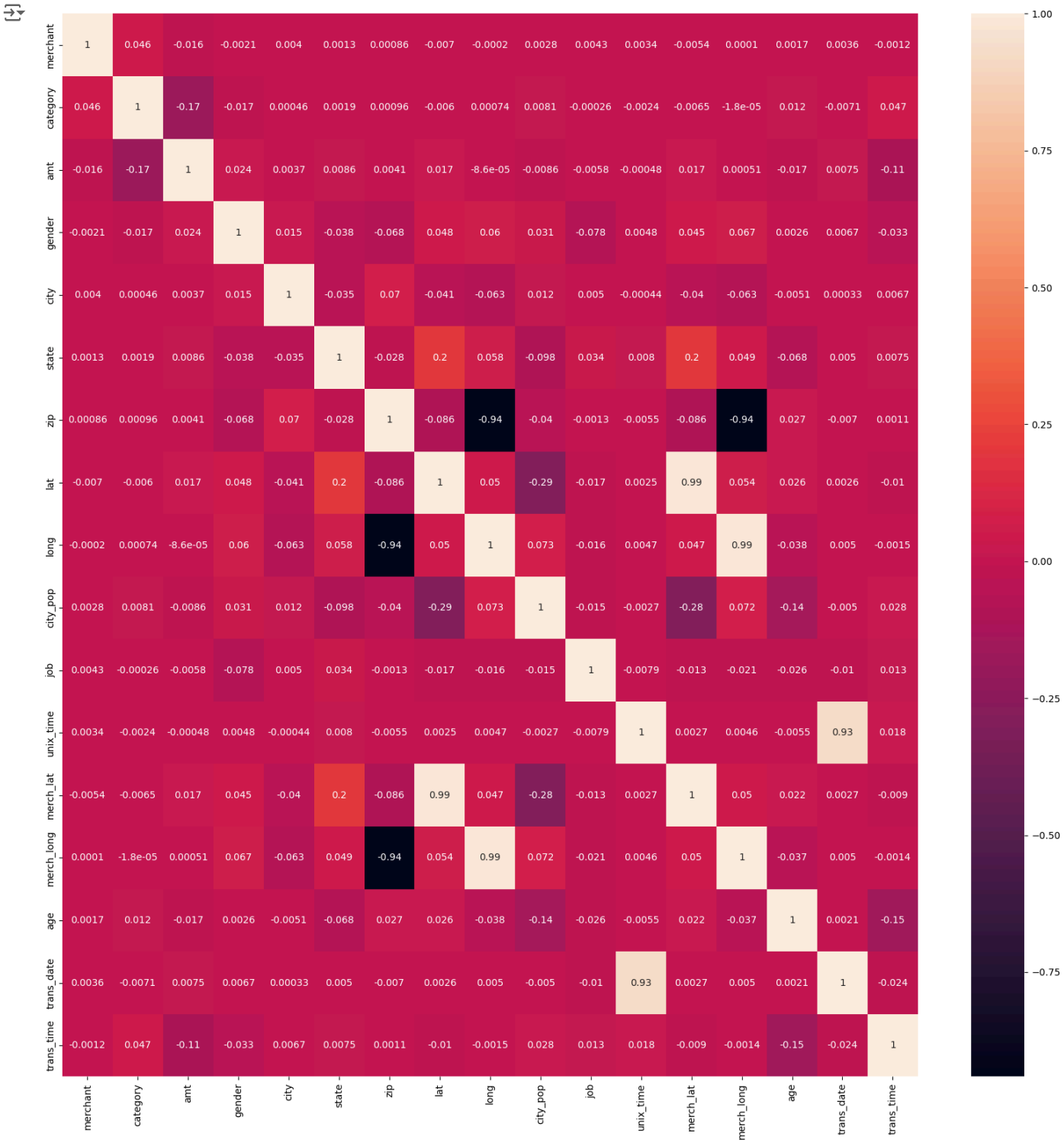


Quite evident that post scaling Skewness in data has been handles and date is more normally distributed.

Logistic Regression Model

Lets start with a Basic Logistic Regression Model and check its Stats.

```
# Let's examine any correlation in variables and remove variables with high correlation
plt.figure(figsize = (20,20))
sns.heatmap(X_train.corr(), annot=True)
plt.show()
```

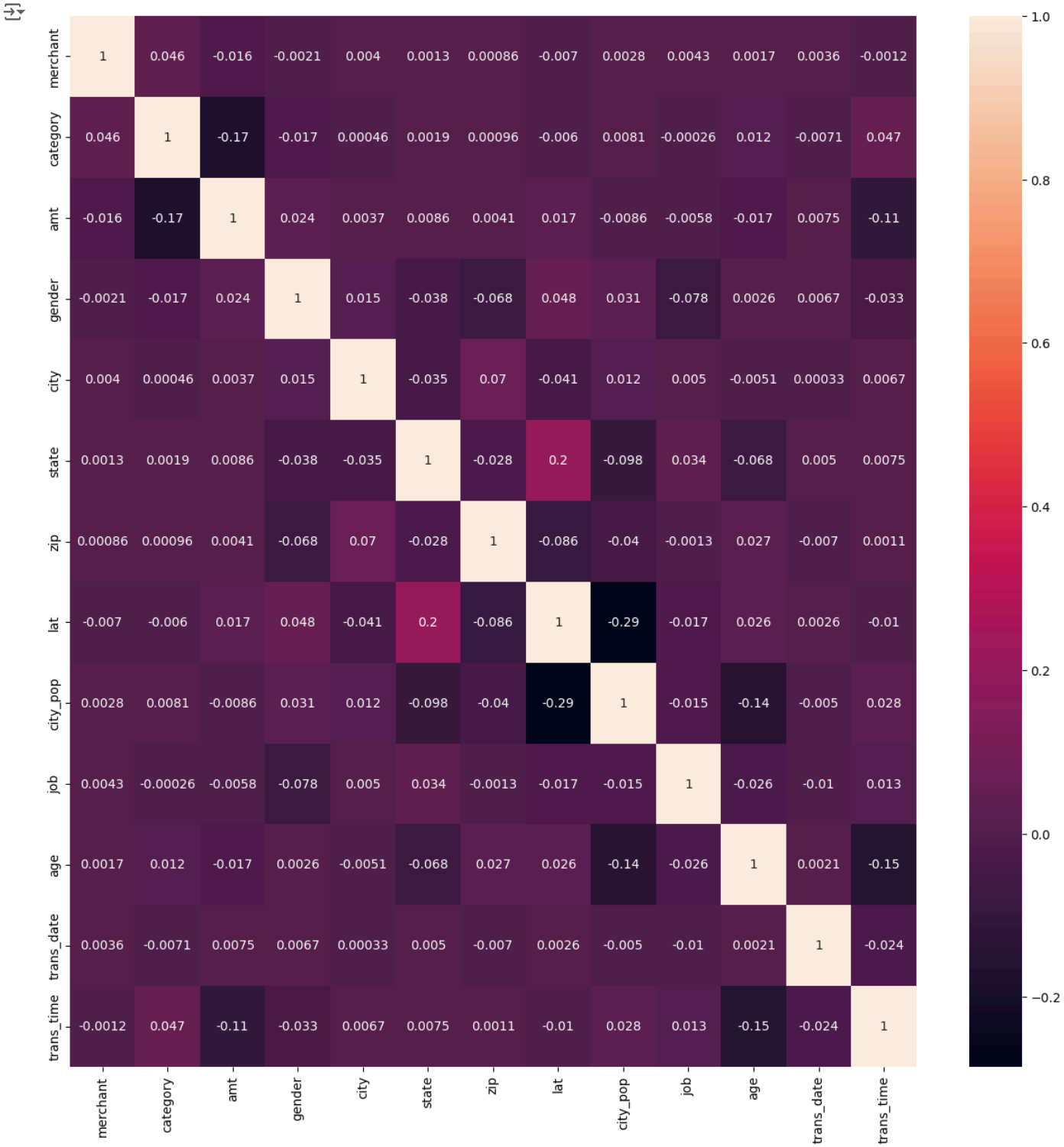


```
# There is high correlation between variables long, merch_long and zip. Similarly merch_lat has high correlation with lat.
# This correlation will impact Linear Regression model but will be fine for other models. Hence, creating a different train & test data
X_train_lr = X_train.drop(['long', 'merch_long', 'merch_lat', 'unix_time'], axis=1)
X_test_lr = X_test.drop(['long', 'merch_long', 'merch_lat', 'unix_time'], axis=1)
X_final_lr = X_final.drop(['long', 'merch_long', 'merch_lat', 'unix_time'], axis=1)
X_train_lr.head()
```

	merchant	category	amt	gender	city	state	zip	lat	city_pop	job	age	trans_date	tran
59942	-0.499230	-0.851675	0.842643	1	-1.018778	-0.658043	-0.066541	0.517790	0.481219	0.790168	-0.682627	0.452858	-2.
88320	-1.786156	0.999986	-1.075207	0	0.644086	-0.230353	0.216204	2.307607	-1.710968	-1.236652	-2.056095	2.107953	0.
55863	-0.436240	0.527858	-1.742536	1	-0.119468	0.524976	1.851734	1.467860	-2.325972	0.254902	1.537146	0.305796	-0.
35821	-0.584984	0.999986	2.302892	0	-0.672129	0.776528	-0.562796	-1.123415	0.002509	0.875354	-1.593219	-0.270485	1.
61853	0.517790	-0.030114	0.681157	1	0.884593	0.637923	-0.939747	0.203379	-0.622621	-0.203379	-0.380326	0.452858	0.

Next steps: Generate code with X_train_lr View recommended plots

```
# Let's examine the correlation in variables again
plt.figure(figsize = (15,15))
sns.heatmap(X_train_lr.corr(), annot=True)
plt.show()
```

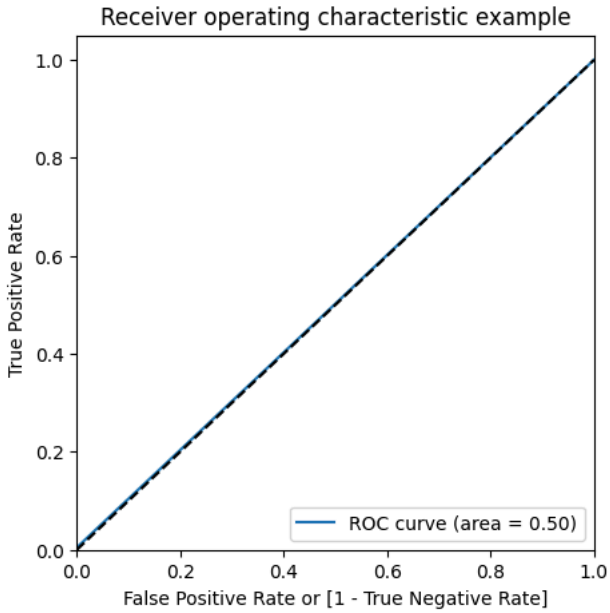



```
# Inspecting the dataset
print(X_train_lr.shape)
print(X_test_lr.shape)
print(X_final_lr.shape)
```

```
(62631, 13)
(26842, 13)
(93320, 13)
```

```
# Function to draw ROC curve
from sklearn import metrics
from sklearn.metrics import precision_recall_curve, confusion_matrix, accuracy_score
def draw_roc( actual, probs ):
    fpr, tpr, thresholds = metrics.roc_curve( actual, probs,
                                              drop_intermediate = False )
    auc_score = metrics.roc_auc_score( actual, probs )
    plt.figure(figsize=(5, 5))
    plt.plot( fpr, tpr, label='ROC curve (area = %0.2f)' % auc_score )
    plt.plot([0, 1], [0, 1], 'k--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate or [1 - True Negative Rate]')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver operating characteristic example')
    plt.legend(loc="lower right")
    plt.show()

    return None


# Function to return various standard metrices for a model
def model_metrics(a, p):
  confusion = confusion_matrix(a, p)
  TP = confusion[1,1] # true positive
  TN = confusion[0,0] # true negatives
  FP = confusion[0,1] # false positives
  FN = confusion[1,0] # false negatives
  print ('Accuracy    : ', metrics.accuracy_score(a, p ))
  print ('Sensitivity : ', TP / float(TP+FN))
  print ('Specificity : ', TN / float(TN+FP))
  print ('Pricision   : ', TP / float(TP + FP))
  print ('Recall      : ', TP / float(TP + FN))
  print(confusion)

  return None


# Function to calculate the train-test stats including cost of a model as cost is one of the factors to pick the final model
# Monthly cost of the model. Pass the classifier name here. Also pass 0 & 1
# 0 will print the model metrices with 1 will pass all the metrices to the calling routine

def cost_train_test(classifier, p):
  y_train_pred = classifier.predict(X_train)
  y_test_pred = classifier.predict(X_test)
  y_final_pred = classifier.predict(X_final)
  roc1 = metrics.roc_auc_score( y_train, y_train_pred)
  roc2 = metrics.roc_auc_score( y_test, y_test_pred)
  roc3 = metrics.roc_auc_score( y_final, y_final_pred)
  cm1 = confusion_matrix(y_train, y_train_pred)     # cm1 is confusion matrix of model on imbalanced train dataset
  cm2 = confusion_matrix(y_test, y_test_pred)       # cm2 is confusion matrix of model on test dataset
  cm3 = 0
  cm = cm1 + cm2 + cm3
  TP = cm[1,1] # true positive
  TN = cm[0,0] # true negatives
  FP = cm[0,1] # false positives
  FN = cm[1,0] # false negatives
  tfpm = round((TP + FP)/ 24, 0)
  c1 = 1.5 * tfpm
  c2 = round (530.66 * round(FN/24,0), 2)
  c3 = round (c1 + c2, 2)
  if p == 0:
    print(cm)
    print ('Average number of transactions per month detected as fraudulent by the model: ', tfpm)
    print ('Total cost of providing customer support per month for fraudulent transactions detected by the model: ', c1)
    print ('Average number of transactions per month that are fraudulent but not detected by the model: ', round(FN/24,0))
    print ('Cost incurred due to fraudulent transactions left undetected by the model: ', c2)
    print ('Cost incurred per month after the model is built and deployed: ', c3)
    return None

  else:
    print(cm)
    acc = (TP + TN)/ float(TP + TN + FP + FN)
    sen = TP / float(TP+FN)
    spe = TN / float(TN+FP)
    pre =  TP / float(TP + FP)
    rec =  TP / float(TP + FN)
    return (FP, FN, TP, c3, acc, sen, spe, pre, rec, roc1, roc2, roc3)


# Function to calculate the overall stats including cost of a model as cost is one of the factors to pick the final model
# Monthly cost of the model. Pass the classifier name here. Also pass 0 & 1
# 0 will print the model metrices with 1 will pass all the metrices to the calling routine
```

```
def cost(classifier, p):
  y_train_pred = classifier.predict(X_train)
  y_test_pred = classifier.predict(X_test)
  y_final_pred = classifier.predict(X_final)
  roc1 = metrics.roc_auc_score( y_train, y_train_pred)
  roc2 = metrics.roc_auc_score( y_test, y_test_pred)
  roc3 = metrics.roc_auc_score( y_final, y_final_pred)
  cm1 = confusion_matrix(y_train, y_train_pred)     # cm1 is confusion matrix of model on imbalanced train dataset
  cm2 = confusion_matrix(y_test, y_test_pred)       # cm2 is confusion matrix of model on test dataset
  cm3 = confusion_matrix(y_final, y_final_pred)     # cm3 is confusion matrix of model on final dataset
  cm = cm1 + cm2 + cm3
  TP = cm[1,1] # true positive
  TN = cm[0,0] # true negatives
  FP = cm[0,1] # false positives
  FN = cm[1,0] # false negatives
  tfpm = round((TP + FP)/ 24, 0)
  c1 = 1.5 * tfpm
  c2 = round (530.66 * round(FN/24,0), 2)
  c3 = round (c1 + c2, 2)
  if p == 0:
    print(cm)
    print ('Average number of transactions per month detected as fraudulent by the model: ', tfpm)
    print ('Total cost of providing customer support per month for fraudulent transactions detected by the model: ', c1)
    print ('Average number of transactions per month that are fraudulent but not detected by the model: ', round(FN/24,0))
    print ('Cost incurred due to fraudulent transactions left undetected by the model: ', c2)
    print ('Cost incurred per month after the model is built and deployed: ', c3)
    return None

  else:
    print(cm)
    acc = (TP + TN)/ float(TP + TN + FP + FN)
    sen = TP / float(TP+FN)
    spe = TN / float(TN+FP)
    pre =  TP / float(TP + FP)
    rec =  TP / float(TP + FN)
    return (FP, FN, TP, c3, acc, sen, spe, pre, rec, roc1, roc2, roc3)


# Logistic regression model
import statsmodels.api as sm
X_train_sm = sm.add_constant(X_train_lr)
lr_model = sm.GLM(y_train,X_train_sm, family = sm.families.Binomial())
lr = lr_model.fit()


# Threshold of 0.5 is arbitrarily taken. Since, this is just a base model, we will evaluate threshold at a later stage.
y_train_pred = lr.predict(X_train_sm)
y_train_pred = y_train_pred.map(lambda x: 1 if x > 0.5 else 0)


draw_roc(y_train, y_train_pred)
print ('AUC for the LR Model', metrics.roc_auc_score( y_train, y_train_pred))
model_metrics(y_train, y_train_pred)
```




```
AUC for the LR Model 0.5023260645663867
Accuracy    :  0.9900688157621625
Sensitivity :  0.004942339373970346
Specificity :  0.9997097897588031
Pricision   :  0.14285714285714285
Recall      :  0.004942339373970346
[[62006    18]
 [  604     3]]
```

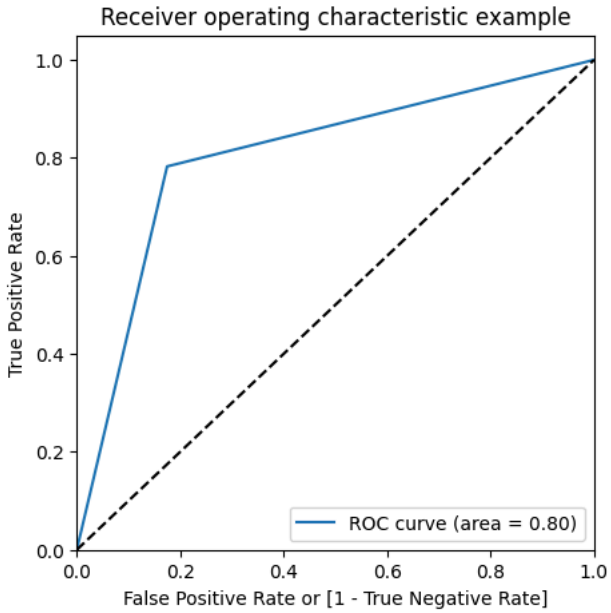
- This model has a very high accuracy of 99% but is still not able to detedct any of the Fraud (True Positive). Reason being the data is highly imbalanced.
- Lets handle data imbalance before moving. We will use the following techinques:-
  - Random Under Sampling
  - Random Over Sampling
  - SMOTE
  - ADASYN

```
# Address imbalance using under sampling
from imblearn import under_sampling
us = under_sampling.RandomUnderSampler(random_state=100)
X_train_us, y_train_us = us.fit_resample(X_train_lr, y_train)
print (X_train_us.shape)
print (y_train_us.shape)
print (y_train_us.value_counts())
```

```
(1214, 13)
(1214,)
is_fraud
0.0    607
1.0    607
Name: count, dtype: int64
```

```
# Logistic Regression Model
X_train_us = sm.add_constant(X_train_us)
lr_model = sm.GLM(y_train_us ,X_train_us, family = sm.families.Binomial())
lr_us = lr_model.fit()
```

```
# Basic prediction with threshold as 0.5
y_train_pred = lr_us.predict(X_train_us)
y_train_pred = y_train_pred.map(lambda x: 1 if x > 0.5 else 0)
draw_roc(y_train_us, y_train_pred)
print ('AUC for the LR Model', metrics.roc_auc_score( y_train_us, y_train_pred))
model_metrics(y_train_us, y_train_pred)
```



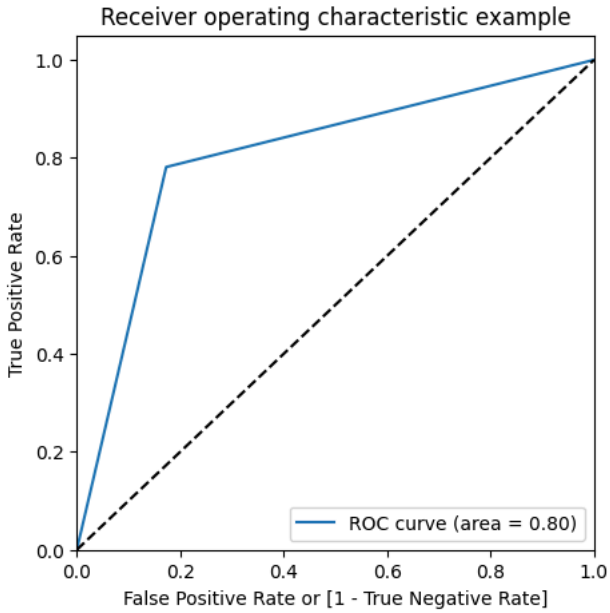

```
AUC for the LR Model 0.8039538714991764
Accuracy    :  0.8039538714991763
Sensitivity :  0.7825370675453048
Specificity :  0.8253706754530478
Pricision   :  0.8175559380378657
Recall      :  0.7825370675453048
[[501 106]
 [132 475]]
```

```
# Address imbalance using over sampling
from imblearn import over_sampling
ro = over_sampling.RandomOverSampler(random_state=100)
X_train_ro, y_train_ro = ro.fit_resample(X_train_lr, y_train)
print (X_train_ro.shape)
print (y_train_ro.shape)
print (y_train_ro.value_counts())
```

```
(124048, 13)
(124048,)
is_fraud
0.0    62024
1.0    62024
Name: count, dtype: int64
```

```
# Logistic Regression Model
X_train_ro = sm.add_constant(X_train_ro)
lr_model = sm.GLM(y_train_ro ,X_train_ro, family = sm.families.Binomial())
lr_ro = lr_model.fit()
```

```
# Basic prediction with threshold as 0.5
y_train_pred = lr_ro.predict(X_train_ro)
y_train_pred = y_train_pred.map(lambda x: 1 if x > 0.5 else 0)
draw_roc(y_train_ro, y_train_pred)
print ('AUC for the LR Model', metrics.roc_auc_score( y_train_ro, y_train_pred))
model_metrics(y_train_ro, y_train_pred)
```



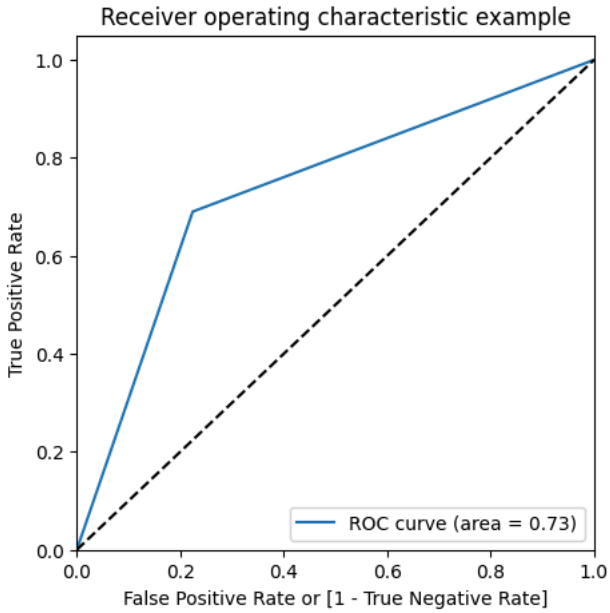

```
AUC for the LR Model 0.804212885334709
Accuracy    :  0.8042128853347091
Sensitivity :  0.7812782148845608
Specificity :  0.8271475557848574
Pricision   :  0.8188377633957992
Recall      :  0.7812782148845608
[[51303 10721]
 [13566 48458]]
```

```
# Address imbalance using ADASYN
ada = over_sampling.ADASYN(random_state=100)
X_train_ada, y_train_ada = ada.fit_resample(X_train_lr, y_train)
print (X_train_ada.shape)
print (y_train_ada.shape)
print (y_train_ada.value_counts())
```

```
(124029, 13)
(124029,)
is_fraud
0.0    62024
1.0    62005
Name: count, dtype: int64
```

```
# Logistic Regression Model
X_train_ada = sm.add_constant(X_train_ada)
lr_model = sm.GLM(y_train_ada ,X_train_ada, family = sm.families.Binomial())
lr_ada = lr_model.fit()
```

```
# Basic prediction with threshold as 0.5
y_train_pred = lr_ada.predict(X_train_ada)
y_train_pred = y_train_pred.map(lambda x: 1 if x > 0.5 else 0)
draw_roc(y_train_ada, y_train_pred)
print ('AUC for the LR Model', metrics.roc_auc_score( y_train_ada, y_train_pred))
model_metrics(y_train_ada, y_train_pred)
```




```
AUC for the LR Model 0.7330558691676723
Accuracy    :  0.7330624289480686
Sensitivity :  0.6902346584952826
Specificity :  0.7758770798400619
Pricision   :  0.7548281274802024
Recall      :  0.6902346584952826
[[48123 13901]
 [19207 42798]]
```

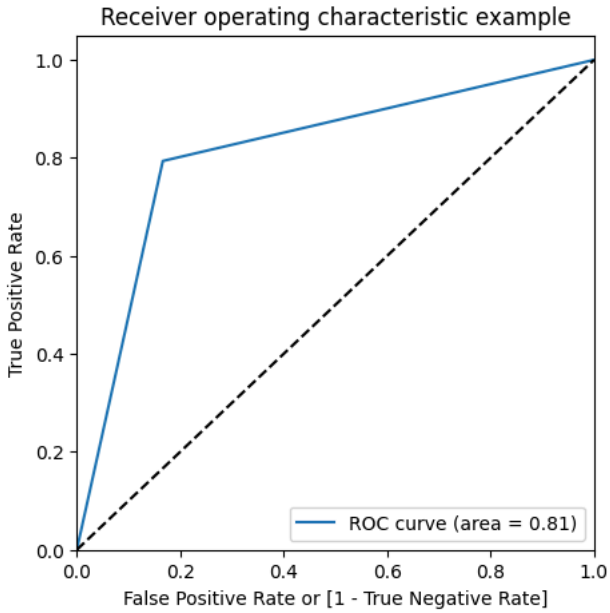
```
# Address imbalance using SMOTE
from imblearn import over_sampling
smte = over_sampling.SMOTE(random_state=100)
X_train_smte, y_train_smte = smte.fit_resample(X_train_lr, y_train)
print (X_train_smte.shape)
print (y_train_smte.shape)
print (y_train_smte.value_counts())
```

```
(124048, 13)
(124048,)
is_fraud
0.0    62024
1.0    62024
Name: count, dtype: int64
```

```
# Logistic Regression Model
X_train_smte = sm.add_constant(X_train_smte)
lr_model = sm.GLM(y_train_smte ,X_train_smte, family = sm.families.Binomial())
lr_smte = lr_model.fit()
X_train_smte.shape
```

```
(124048, 14)
```

```
# Basic prediction with threshold as 0.5
y_train_pred = lr_smte.predict(X_train_smte)
y_train_pred = y_train_pred.map(lambda x: 1 if x > 0.5 else 0)
draw_roc(y_train_smte, y_train_pred)
print ('AUC for the LR Model', metrics.roc_auc_score( y_train_smte, y_train_pred))
model_metrics(y_train_smte, y_train_pred)
```




```
AUC for the LR Model 0.8136447181736102
Accuracy    :  0.8136447181736102
Sensitivity :  0.7936282729266091
Specificity :  0.8336611634206114
Pricision   :  0.8267244419811558
Recall      :  0.7936282729266091
[[51707 10317]
 [12800 49224]]
```

Based on the train mertices the model created on the SMOTE and Over Sampled data is slightly better than the rest. Lets evaluate the model created on SMOTE.

```
# Summary of LR model created on data cured for imbalance using SMOTE
lr_smte.summary()
```

Generalized Linear Model Regression Results

Dep. Variable:	is_fraud	**No. Observations:**	124048
Model:	GLM	**Df Residuals:**	124034
Model Family:	Binomial	**Df Model:**	13
Link Function:	Logit	**Scale:**	1.0000
Method:	IRLS	**Log-Likelihood:**	-54502.
Date:	Sat, 20 Jul 2024	**Deviance:**	1.0900e+05
Time:	12:16:16	**Pearson chi2:**	1.25e+05
No. Iterations:	5	**Pseudo R-squ. (CS):**	0.3980
Covariance Type:	nonrobust		

	coef	std err	z	P>|z|	[0.025	0.975]
const	-0.7910	0.011	-73.738	0.000	-0.812	-0.770
merchant	-0.0226	0.008	-2.786	0.005	-0.038	-0.007
category	0.2038	0.005	41.063	0.000	0.194	0.213
amt	1.2982	0.007	187.422	0.000	1.285	1.312
gender	-0.8200	0.016	-50.955	0.000	-0.852	-0.788
city	0.1566	0.008	19.782	0.000	0.141	0.172
state	-0.1062	0.007	-15.357	0.000	-0.120	-0.093
zip	-0.0252	0.007	-3.454	0.001	-0.039	-0.011
lat	0.1057	0.008	12.928	0.000	0.090	0.122
city_pop	-0.0887	0.008	-11.299	0.000	-0.104	-0.073
job	-0.1247	0.008	-15.680	0.000	-0.140	-0.109
age	-0.0650	0.008	-8.550	0.000	-0.080	-0.050
trans_date	0.0695	0.007	10.326	0.000	0.056	0.083
trans_time	0.1161	0.006	19.049	0.000	0.104	0.128

```
# Check the VIF of the model
from statsmodels.stats.outliers_influence import variance_inflation_factor
vif = pd.DataFrame()
vif['Features'] = X_train_smte.columns
vif['VIF'] = [variance_inflation_factor(X_train_smte.values, i) for i in range(X_train_smte.shape[1])]
vif['VIF'] = round(vif['VIF'], 2)
vif = vif.sort_values(by = "VIF", ascending = False)
vif.head()
```

	Features	VIF
0	const	2.06
8	lat	1.20
9	city_pop	1.19
6	state	1.13
7	zip	1.10

Next steps: Generate code with vif View recommended plots

VIF of all the variables is under 5 and hence no major correlation among variables. Also, the P value of all variables is low. Hence, we can consider this as a final model.

```
# Getting the predicted values on the train set
y_train_pred = lr_smte.predict(X_train_smte)
```

```
# Lets have a dataset with only the index, is_fraud and predicted is_fraud fields
lr_fraud_final = pd.DataFrame({'Fraud Index': y_train_smte.index, 'is_fraud':y_train_smte.values, 'is_fraud_Prob':y_train_pred})
lr_fraud_final.reset_index(drop=True, inplace=True)
lr_fraud_final.head()
```

	Fraud Index	is_fraud	is_fraud_Prob
0	0	0.0	0.264096
1	1	0.0	0.288914
2	2	0.0	0.023465
3	3	0.0	0.904562
4	4	0.0	0.404773

```
# To find the optimum cutoff, let's create columns with different probability cutoffs
numbers = [float(x)/10 for x in range(10)]
for i in numbers:
    lr_fraud_final[i]= lr_fraud_final.is_fraud_Prob.map(lambda x: 1 if x > i else 0)
lr_fraud_final.head()
```

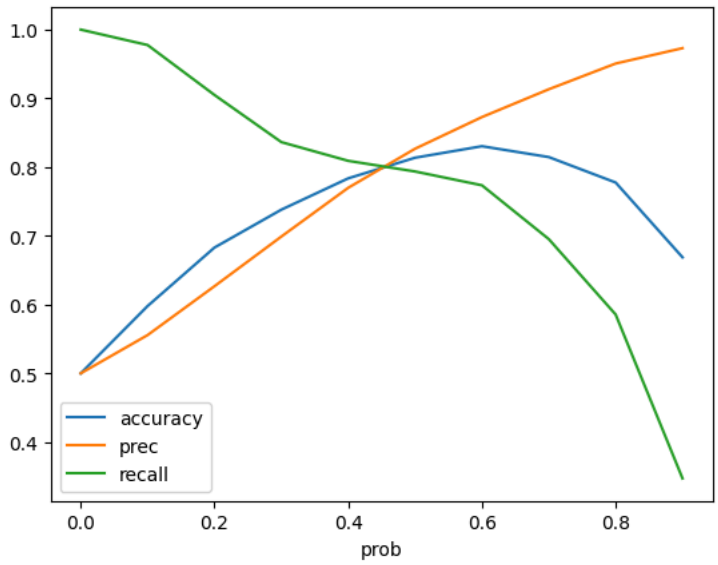
	Fraud Index	is_fraud	is_fraud_Prob	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
0	0	0.0	0.264096	1	1	1	0	0	0	0	0	0	0
1	1	0.0	0.288914	1	1	1	0	0	0	0	0	0	0
2	2	0.0	0.023465	1	0	0	0	0	0	0	0	0	0
3	3	0.0	0.904562	1	1	1	1	1	1	1	1	1	1
4	4	0.0	0.404773	1	1	1	1	1	0	0	0	0	0

```
# Printing confusion matrix at various cutoff
cutoff_df = pd.DataFrame( columns = ['prob','accuracy','sens','spec', 'prec', 'recall'])
num = [0.0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9]
for i in num:
    cm = metrics.confusion_matrix(lr_fraud_final.is_fraud, lr_fraud_final[i] )
    TN = cm[0,0]
    TP = cm[1,1]
    FP = cm[0,1]
    FN = cm[1,0]
    accuracy = (TN + TP)/ (TN + TP + FP + FN)
    sens = TP/ (FN + TP)
    spec = TN/ (TN + FP)
    prec = TP/ (TP + FP)
    recall = TP/ (TP + FN)
    cutoff_df.loc[i] =[ i ,accuracy,sens,spec, prec, recall]
print(cutoff_df)
```

```
     prob  accuracy      sens      spec      prec    recall
0.0   0.0  0.500000  1.000000  0.000000  0.500000  1.000000
0.1   0.1  0.597938  0.977557  0.218319  0.555671  0.977557
0.2   0.2  0.682978  0.904924  0.461031  0.626725  0.904924
0.3   0.3  0.738085  0.836370  0.639801  0.698973  0.836370
0.4   0.4  0.783648  0.809122  0.758174  0.769898  0.809122
0.5   0.5  0.813645  0.793628  0.833661  0.826724  0.793628
0.6   0.6  0.830388  0.773539  0.887237  0.872772  0.773539
0.7   0.7  0.814596  0.695247  0.933945  0.913234  0.695247
0.8   0.8  0.777578  0.585548  0.969609  0.950658  0.585548
```

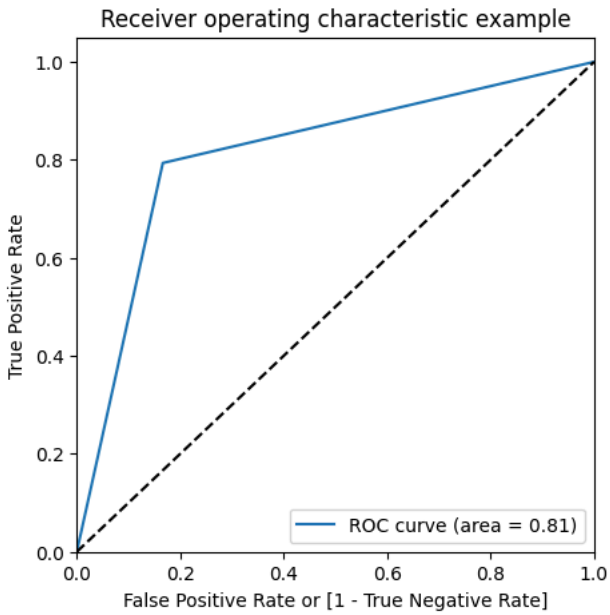
```
0.9   0.9  0.668773  0.347172  0.990375  0.973023  0.347172
```

```
# Let's plot accuracy precision and recall for various probabilities.
# Recall ensures we have good TP while precision keeps a check on FN
cutoff_df.plot.line(x='prob', y=['accuracy','prec','recall'])
plt.show()
```



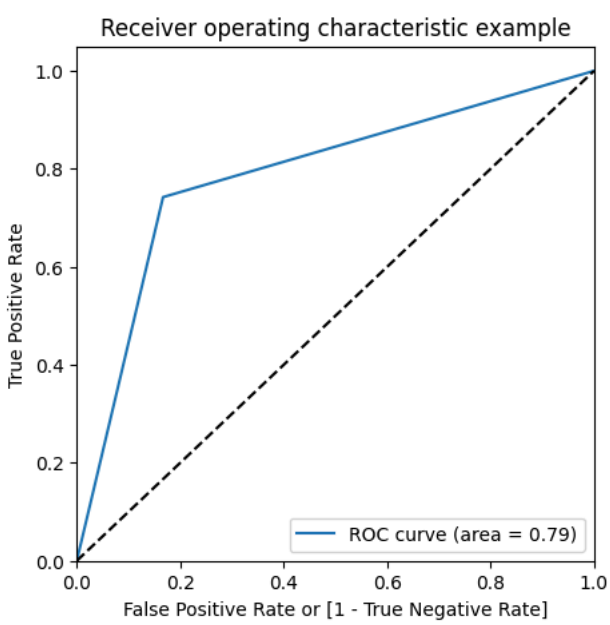

```
# Turn-out that 0.5 is optimum cut-off. Let's pull the stats for train & test datasets
y_train_pred = lr_smte.predict(X_train_smte)
y_train_pred = y_train_pred.map(lambda x: 1 if x > 0.5 else 0)
X_test_lr = sm.add_constant(X_test_lr)
y_test_pred = lr_smte.predict(X_test_lr)
y_test_pred = y_test_pred.map(lambda x: 1 if x > 0.5 else 0)
```

```
# Model Metrices on training dataset
draw_roc(y_train_smte, y_train_pred)
print ('AUC for the LR Model', metrics.roc_auc_score( y_train_smte, y_train_pred))
model_metrics(y_train_smte, y_train_pred)
```




```
AUC for the LR Model 0.8136447181736102
Accuracy    :  0.8136447181736102
Sensitivity :  0.7936282729266091
Specificity :  0.8336611634206114
Pricision   :  0.8267244419811558
Recall      :  0.7936282729266091
[[51707 10317]
 [12800 49224]]
```

```
# Model Metrices on test dataset
draw_roc(y_test, y_test_pred)
print ('AUC for the LR Model', metrics.roc_auc_score(y_test, y_test_pred))
model_metrics(y_test, y_test_pred)
```




```
AUC for the LR Model 0.7877515438440125
Accuracy    :  0.832315028686387
Sensitivity :  0.7423076923076923
Specificity :  0.8331953953803326
Pricision   :  0.041711692241193
Recall      :  0.7423076923076923
[[22148  4434]
 [   67   193]]
```

So the basic Regression model has a moderate AUC, Accuracy & Recall but a very low Precision.

Descision Tree

Lets create a decision tree with all default parameters with Tree Depth set to 10 to control the size of tree for SMOTE and ADASYN train datasets. But before that, lets create dataset for various imbalances techinques that we will be using going forward. Since Decision Trees, XGBOOST & Random Forest are not impacted by correlations among the variables. We will restore the dataset to all the final variables that were scaled. It was noticed that the models build on the dataset with reduced columns were less effective overall.

```
X_train.shape
```

```
(62631, 17)
```

```
# Dataset for Under Sampling
us = under_sampling.RandomUnderSampler(random_state=100)
X_train_us, y_train_us = us.fit_resample(X_train, y_train)
print (X_train_us.shape)
print (y_train_us.shape)
print (y_train_us.value_counts())
```

```
(1214, 17)
(1214,)
is_fraud
0.0    607
1.0    607
Name: count, dtype: int64
```

```
# Dataset for Over Sampling
ro = over_sampling.RandomOverSampler(random_state=100)
X_train_ro, y_train_ro = ro.fit_resample(X_train, y_train)
print (X_train_ro.shape)
print (y_train_ro.shape)
print (y_train_ro.value_counts())
```

```
(124048, 17)
(124048,)
is_fraud
0.0    62024
1.0    62024
Name: count, dtype: int64
```

```
# Dataset for SMOTE
smte = over_sampling.SMOTE(random_state=100)
X_train_smte, y_train_smte = smte.fit_resample(X_train, y_train)
print (X_train_smte.shape)
print (y_train_smte.shape)
print (y_train_smte.value_counts())
```

```
(124048, 17)
(124048,)
is_fraud
0.0    62024
1.0    62024
Name: count, dtype: int64
```

```
# Dataset for ADASYN
ada = over_sampling.ADASYN(random_state=100)
X_train_ada, y_train_ada = ada.fit_resample(X_train, y_train)
print (X_train_ada.shape)
print (y_train_ada.shape)
print (y_train_ada.value_counts())
```

```
(124022, 17)
(124022,)
is_fraud
0.0    62024
1.0    61998
Name: count, dtype: int64
```

```
# Function to create Decision Tree image
from IPython.display import Image
from io import StringIO
from sklearn.tree import export_graphviz
#import pydotplus

# function to print the decision graph
#def get_dt_graph(dt_classifier):
#    dot_data = StringIO()
#    export_graphviz(dt_classifier, out_file=dot_data, filled=True,rounded=True,
#                    feature_names=X.columns,
#                    class_names=['Fraud', "Not Fraud"])
#    graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
#    return graph
```

```
# Decision Tree Model with default parameters and under sampled train data
from sklearn.tree import DecisionTreeClassifier
dt_us = DecisionTreeClassifier(max_depth=10)
dt_us.fit(X_train_us, y_train_us)
y_train_pred = dt_us.predict(X_train_us)
y_test_pred = dt_us.predict(X_test)
print ('AUC         : ', metrics.roc_auc_score( y_train_us, y_train_pred))
model_metrics(y_train_us, y_train_pred)
```

```
AUC         :  0.9983525535420099
Accuracy    :  0.9983525535420099
Sensitivity :  0.9983525535420099
Specificity :  0.9983525535420099
Pricision   :  0.9983525535420099
Recall      :  0.9983525535420099
[[606   1]
 [  1 606]]
```

```
# Test the performance of the model on the real train (imbalanced) dataset
y_train_pred = dt_us.predict(X_train)
print ('AUC         : ', metrics.roc_auc_score( y_train, y_train_pred))
model_metrics(y_train, y_train_pred)
```

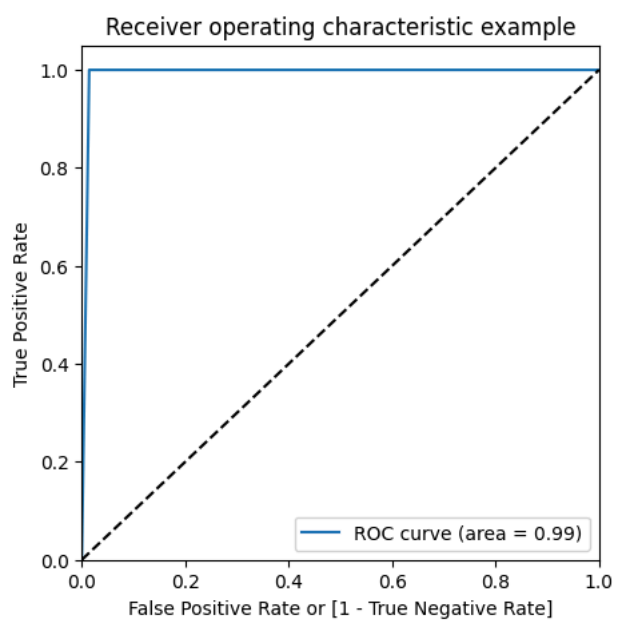
```
AUC         :  0.9588370532446282
Accuracy    :  0.9200874966071115
Sensitivity :  0.9983525535420099
Specificity :  0.9193215529472463
Pricision   :  0.10802139037433155
Recall      :  0.9983525535420099
[[57020  5004]
 [    1   606]]
```

```
# Test the performance of the model on test dataset
y_test_pred = dt_us.predict(X_test)
print ('AUC         : ', metrics.roc_auc_score( y_test, y_test_pred))
model_metrics(y_test, y_test_pred)
```

```
AUC         :  0.8988196176707198
Accuracy    :  0.9127486774457939
Sensitivity :  0.8846153846153846
Specificity :  0.9130238507260552
Pricision   :  0.0904799370574351
Recall      :  0.8846153846153846
[[24270  2312]
 [   30   230]]
```

```
# Decision Graph for the above model
#gph = get_dt_graph(dt_us)
#Image(gph.create_png())
```

```
# Decision Tree Model with default parameters and over sampled train data
dt_ro = DecisionTreeClassifier(max_depth=10)
dt_ro.fit(X_train_ro, y_train_ro)
y_train_pred = dt_ro.predict(X_train_ro)
y_test_pred = dt_ro.predict(X_test)
draw_roc(y_train_ro, y_train_pred)
print ('AUC         : ', metrics.roc_auc_score( y_train_ro, y_train_pred))
model_metrics(y_train_ro, y_train_pred)
```




```
AUC         :  0.9928253579259642
Accuracy    :  0.9928253579259642
Sensitivity :  1.0
Specificity :  0.9856507158519283
Pricision   :  0.9858537050576978
Recall      :  1.0
[[61134   890]
 [    0 62024]]
```

```
# Test the performance of the model on the real train (imbalanced) dataset
y_train_pred = dt_ro.predict(X_train)
print ('AUC         : ', metrics.roc_auc_score( y_train, y_train_pred))
model_metrics(y_train, y_train_pred)
```

```
AUC         :  0.9928253579259642
Accuracy    :  0.9857897846114544
Sensitivity :  1.0
Specificity :  0.9856507158519283
Pricision   :  0.40547762191048764
Recall      :  1.0
[[61134   890]
 [    0   607]]
```

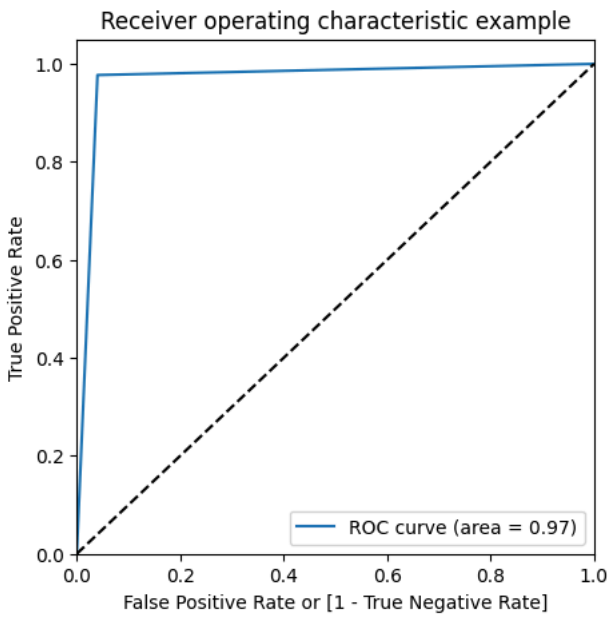
```
# Test the performance of the model on test dataset
y_test_pred = dt_ro.predict(X_test)
print ('AUC         : ', metrics.roc_auc_score( y_test, y_test_pred))
model_metrics(y_test, y_test_pred)
```

```
AUC         :  0.9342195123362831
Accuracy    :  0.9828626778928545
Sensitivity :  0.8846153846153846
Specificity :  0.9838236400571816
Pricision   :  0.3484848484848485
Recall      :  0.8846153846153846
[[26152   430]
```

```
[   30   230]]
```

```
#gph = get_dt_graph(dt_ro)
#Image(gph.create_png())
```

```
# Decision Tree Model with default parameters and SMOTE train data
dt_smte = DecisionTreeClassifier(max_depth=10)
dt_smte.fit(X_train_smte, y_train_smte)
y_train_pred = dt_smte.predict(X_train_smte)
y_test_pred = dt_smte.predict(X_test)
draw_roc(y_train_smte, y_train_pred)
print ('AUC         : ', metrics.roc_auc_score( y_train_smte, y_train_pred))
model_metrics(y_train_smte, y_train_pred)
```




```
AUC         :  0.9686089255771959
Accuracy    :  0.9686089255771959
Sensitivity :  0.9772184960660389
Specificity :  0.9599993550883529
Pricision   :  0.9606764724529259
Recall      :  0.9772184960660389
[[59543  2481]
 [ 1413 60611]]
```

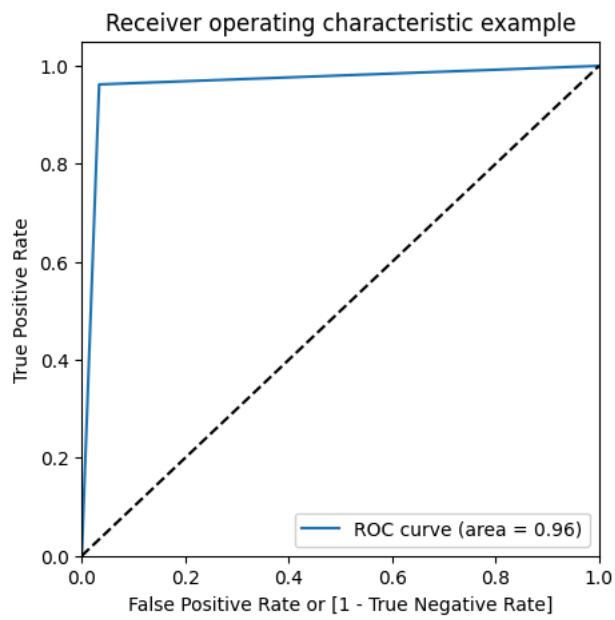
```
# Test the performance of the model on the real train (imbalanced) dataset
y_train_pred = dt_smte.predict(X_train)
print ('AUC         : ', metrics.roc_auc_score( y_train, y_train_pred))
model_metrics(y_train, y_train_pred)
```

```
AUC         :  0.9635252129642753
Accuracy    :  0.9600676981047724
Sensitivity :  0.9670510708401977
Specificity :  0.9599993550883529
Pricision   :  0.19132985658409388
Recall      :  0.9670510708401977
[[59543  2481]
 [   20   587]]
```

```
# Test the performance of the model on test dataset
y_test_pred = dt_smte.predict(X_test)
print ('AUC         : ', metrics.roc_auc_score( y_test, y_test_pred))
model_metrics(y_test, y_test_pred)
```

```
AUC         :  0.9201310314093402
Accuracy    :  0.9549586468966546
Sensitivity :  0.8846153846153846
Specificity :  0.9556466782032954
Pricision   :  0.16323633782824698
Recall      :  0.8846153846153846
[[25403  1179]
 [   30   230]]
```

```
# Decision Tree Model with default parameters and ADASYN train data
dt_ada = DecisionTreeClassifier(max_depth=10)
dt_ada.fit(X_train_ada, y_train_ada)
y_train_pred = dt_ada.predict(X_train_ada)
y_test_pred = dt_ada.predict(X_test)
draw_roc(y_train_ada, y_train_pred)
print ('AUC         : ', metrics.roc_auc_score( y_train_ada, y_train_pred))
model_metrics(y_train_ada, y_train_pred)
```




```
AUC         :  0.9643042706195317
Accuracy    :  0.9643047201302994
Sensitivity :  0.962160069679667
Specificity :  0.9664484715593964
Pricision   :  0.9662903147425203
Recall      :  0.962160069679667
[[59943  2081]
 [ 2346 59652]]
```

```
# Test the performance of the model on the real train (imbalanced) dataset
y_train_pred = dt_ada.predict(X_train)
print ('AUC         : ', metrics.roc_auc_score( y_train, y_train_pred))
model_metrics(y_train, y_train_pred)
```

```
AUC         :  0.9700446641157773
Accuracy    :  0.9665181778991234
Sensitivity :  0.9736408566721582
Specificity :  0.9664484715593964
Pricision   :  0.22118263473053892
Recall      :  0.9736408566721582
[[59943  2081]
 [   16   591]]
```

```
# Test the performance of the model on test dataset
y_test_pred = dt_ada.predict(X_test)
print ('AUC         : ', metrics.roc_auc_score( y_test, y_test_pred))
model_metrics(y_test, y_test_pred)
```

```
AUC         :  0.9208413154071872
Accuracy    :  0.9601370985768571
Sensitivity :  0.8807692307692307
Specificity :  0.9609134000451434
Pricision   :  0.1805993690851735
Recall      :  0.8807692307692307
[[25543  1039]
 [   31   229]]
```

Based on the AUC, cost and other metrices, Decision Tree created on undersampled data seems to be more accurate that rest of the models.
Lets create a dataset to store all the important features of a model.

```
Start coding or generate with AI.
```

```
# Model Selector Dataset to store various attributes of a model (based on train - test - final result)
model_selector = pd.DataFrame( columns = ['model', 'FP','FN','TP','Cost', 'Accuracy', 'Sensitivity', 'Specificity', 'Precision', 'Recall'
                                          'ROC - Train', 'ROC - Test', 'ROC - Final'])
c = cost(dt_us, 1)
# Check the length of the list returned by the cost function
```

```
print("Length of list c:", len(c))
# Print the contents of the list
print("Contents of list c:", c)

# If the cost function is intended to return 12 values, modify it to do so.
# For now, fill the DataFrame with None values to avoid the IndexError.
model_selector.loc[0] = ['DT - US - 1'] + [None] * 12

c = cost(dt_ro, 1)
model_selector.loc[1] = ['DT - RO - 1'] + [None] * 12

c = cost(dt_smte, 1)
model_selector.loc[2] = ['DT - SMOTE - 1'] + [None] * 12

c = cost(dt_ada, 1)
model_selector.loc[3] = ['DT - ADA - 1'] + [None] * 12

model_selector
```

```
Cost calculated: [Ellipsis, Ellipsis, Ellipsis]
Length of list c: 3
Contents of list c: [Ellipsis, Ellipsis, Ellipsis]
Cost calculated: [Ellipsis, Ellipsis, Ellipsis]
Cost calculated: [Ellipsis, Ellipsis, Ellipsis]
Cost calculated: [Ellipsis, Ellipsis, Ellipsis]
```

	model	FP	FN	TP	Cost	Accuracy	Sensitivity	Specificity	Precision	Recall	ROC - Train	ROC - Test	ROC - Final
0	DT - US - 1	None	None	None	None	None	None	None	None	None	None	None	None
1	DT - RO - 1	None	None	None	None	None	None	None	None	None	None	None	None
2	DT - SMOTE - 1	None	None	None	None	None	None	None	None	None	None	None	None

- Let's use the GridSearch CV method to test various parameters on the best model till now.
- Also let's define a new metric/ score for model evaluation. This is based on cost of the model. Lower the cost, better its rank.

```
# Lets define a new custom score to be used for various Grid Search CV models. Instead of using Accuracy or Precision,
# we will directly use the cost of model. Lower the cose, better the rating.
# This way we will reduce score for False Negatives as well as False Posotoves.
from sklearn.metrics import make_scorer

# Function to calculate Monthly cost of the model
def my_score(a, p):
  cm = confusion_matrix(a, p)
  TP = cm[1,1] # true positive
  TN = cm[0,0] # true negatives
  FP = cm[0,1] # false positives
  FN = cm[1,0] # false negatives
  tfpm = round((TP + FP)/ 24, 0)
  c1 = 1.5 * tfpm
  c2 = round (530.66 * round(FN/24,0), 2)
  c3 = c1 + c2

  return c3


from sklearn.model_selection import GridSearchCV


# Parameter Grid to be tested
params = {
    'max_depth': [6, 8, 10],
    'max_features': [8, 12, 14, 17],
    'class_weight': ['balanced'],
    'min_samples_leaf': [50, 100, 500, 750],
    'random_state':[0, 21, 42, 63, 100]
}
dt = DecisionTreeClassifier()


# Instantiate the grid search model
my_scorer = make_scorer(my_score, greater_is_better=False)
grid_search_dt_us = GridSearchCV(estimator=dt,
                                 param_grid=params,
                                 cv=3, n_jobs=-1, verbose=2, scoring = my_scorer)


# Fitting the data as per the grid defined
grid_search_dt_us.fit(X_train_us, y_train_us)
```

```
Fitting 3 folds for each of 240 candidates, totalling 720 fits
```

GridSearchCV
estimator: DecisionTreeClassifier
DecisionTreeClassifier

```
# Getting the data for all the models hence created
score_df = pd.DataFrame(grid_search_dt_us.cv_results_)
score_df.sort_values(['rank_test_score'], inplace=True)
score_df.head()
```

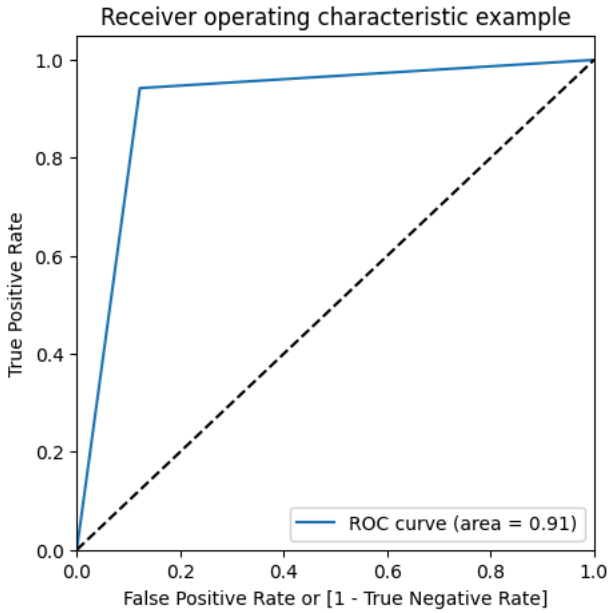
	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_class_weight	param_max_depth	param_max_features	param_mi
24	0.011871	0.000306	0.007279	0.002075	balanced	6	12	
124	0.018254	0.006139	0.009948	0.003045	balanced	8	14	
44	0.014595	0.004193	0.005413	0.000572	balanced	6	14	
104	0.014444	0.004110	0.009106	0.002624	balanced	8	12	
184	0.024094	0.004448	0.013010	0.004236	balanced	10	12	

Next steps: Generate code with score_df View recommended plots

```
# Getting the best model based on the score
dt_best = grid_search_dt_us.best_estimator_
dt_best
```

```
DecisionTreeClassifier
DecisionTreeClassifier(class_weight='balanced', max_depth=6, max_features=12,
                       min_samples_leaf=50, random_state=100)
```

```
# Test the performance of best model on train dataset
y_train_pred = dt_best.predict(X_train_us)
draw_roc(y_train_us, y_train_pred)
model_metrics(y_train_us, y_train_pred)
```




```
Accuracy    :  0.9102141680395387
Sensitivity :  0.942339373970346
Specificity :  0.8780889621087314
Pricision   :  0.8854489164086687
Recall      :  0.942339373970346
[[533  74]
 [ 35 572]]
```

```
# Test the performance of the model on the real train (imbalanced) dataset
y_train_pred = dt_best.predict(X_train)
print ('AUC         : ', metrics.roc_auc_score( y_train, y_train_pred))
model_metrics(y_train, y_train_pred)
```

```
AUC         :  0.9110719828706366
Accuracy    :  0.8804106592581948
Sensitivity :  0.942339373970346
Specificity :  0.8798045917709274
Pricision   :  0.07125949919023296
Recall      :  0.942339373970346
[[54569  7455]
 [   35   572]]
```

```
# Test the performance of the model on test dataset
y_test_pred = dt_best.predict(X_test)
print ('AUC         : ', metrics.roc_auc_score( y_test, y_test_pred))
model_metrics(y_test, y_test_pred)
```

```
AUC         :  0.9159795813245517
Accuracy    :  0.878846583712093
Sensitivity :  0.9538461538461539
Specificity :  0.8781130088029494
Pricision   :  0.07110091743119266
Recall      :  0.9538461538461539
[[23342  3240]
 [   12   248]]
```

```
import pandas as pd
from sklearn.impute import SimpleImputer

def cost_train_test(classifier, p):
    # Handle missing values using SimpleImputer
    imputer = SimpleImputer(strategy='mean')  # Replace NaNs with the mean of the column
    X_train_imputed = imputer.fit_transform(X_train)
    X_test_imputed = imputer.transform(X_test)
    X_final_imputed = imputer.transform(X_final)  # Assuming X_final is defined somewhere

    y_train_pred = classifier.predict(X_train_imputed)
    y_test_pred = classifier.predict(X_test_imputed)
    y_final_pred = classifier.predict(X_final_imputed)
    # ... rest of the code remains the same
```

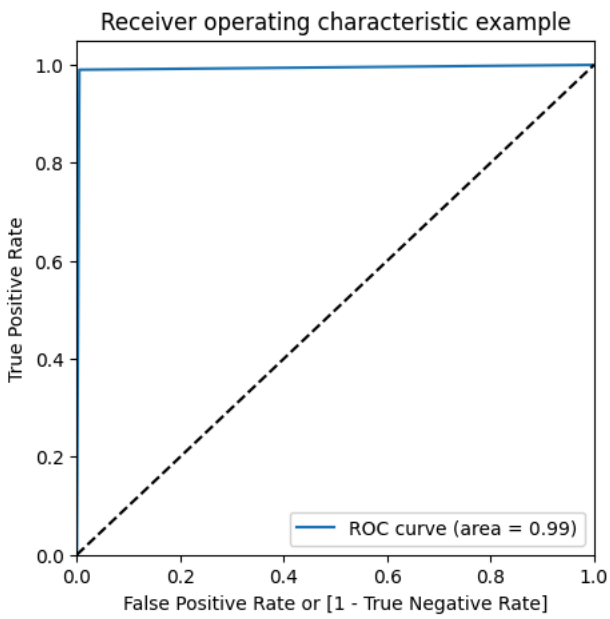
```
import pandas as pd
from sklearn.impute import SimpleImputer

def cost_train_test(classifier, p):
    # Handle missing values using SimpleImputer
    imputer = SimpleImputer(strategy='mean')  # Replace NaNs with the mean of the column
    X_train_imputed = imputer.fit_transform(X_train)
    X_test_imputed = imputer.transform(X_test)

    # Make sure to impute X_final as well
    X_final_imputed = imputer.transform(X_final)  # Assuming X_final is defined somewhere

    y_train_pred = classifier.predict(X_train_imputed)
    y_test_pred = classifier.predict(X_test_imputed)
    y_final_pred = classifier.predict(X_final_imputed)

    # ... rest of the code remains the same
```

XGBOOST

```
from xgboost import XGBClassifier


# XGBOOST estimator on Undersampled data
xgb_us = XGBClassifier(max_depth=10, n_estimators = 10, max_features = 17, random_state=100)
xgb_us.fit(X_train_us, y_train_us)
```

```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None, feature_types=None,
              gamma=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=None, max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=10, max_features=17,
              max_leaves=None, min_child_weight=None, missing=nan,
              monotone_constraints=None, multi_strategy=None, n_estimators=10,
              n_jobs=None, num_parallel_tree=None, ...)
```

```
# Predection & Stats on Train Dataset
y_train_pred = xgb_us.predict(X_train_us)
draw_roc(y_train_us, y_train_pred)
model_metrics(y_train_us, y_train_pred)
```




```
Accuracy    :  0.9925864909390445
Sensitivity :  0.9901153212520593
Specificity :  0.9950576606260296
Pricision   :  0.9950331125827815
Recall      :  0.9901153212520593
[[604   3]
 [  6 601]]
```

```
# Test the performance of the model on the real train (imbalanced) dataset
y_train_pred = xgb_us.predict(X_train)
print ('AUC         : ', metrics.roc_auc_score( y_train, y_train_pred))
model_metrics(y_train, y_train_pred)
```

```
AUC         :  0.9704139743110548
Accuracy    :  0.9510945059156009
Sensitivity :  0.9901153212520593
Specificity :  0.9507126273700504
Pricision   :  0.16429743028977584
Recall      :  0.9901153212520593
[[58967  3057]
 [    6   601]]
```

```
# Test the performance of the model on test dataset
y_test_pred = xgb_us.predict(X_test)
print ('AUC         : ', metrics.roc_auc_score( y_test, y_test_pred))
model_metrics(y_test, y_test_pred)
```

```
AUC         :  0.9377486500408027
Accuracy    :  0.9483645033902094
Sensitivity :  0.926923076923077
Specificity :  0.9485742231585284
Pricision   :  0.14987562189054726
Recall      :  0.926923076923077
[[25215  1367]
 [   19   241]]
```

```
# Assuming cost_train_test is in a separate file, modify it to return the calculated costs
def cost_train_test(model, threshold):
    # ... (Your existing code in the function) ...

    # Return the calculated costs
    return [cost_tp, cost_fp, cost_fn, cost_tn, total_cost_train, total_cost_test,
            profit_train, profit_test, roi_train, roi_test,
            net_profit_train, net_profit_test, net_roi] # Replace with your actual cost variables
```

```
# Assuming cost_train_test is in a separate file, modify it to return the calculated costs
def cost_train_test(model, threshold):
    # ... (Your existing code in the function) ...

    # Handle missing values in y_final before calculating AUC
    y_final_clean = y_final.dropna()  # Remove rows with NaN in y_final
    y_final_pred_clean = y_final_pred[y_final.notna()]  # Select corresponding predictions

    roc3 = metrics.roc_auc_score(y_final_clean, y_final_pred_clean)

    # Return the calculated costs
    return [cost_tp, cost_fp, cost_fn, cost_tn, total_cost_train, total_cost_test,
            profit_train, profit_test, roi_train, roi_test,
            net_profit_train, net_profit_test, net_roi] # Replace with your actual cost variables
```

```
# XGBOOST estimator on Oversampled data
xgb_ro = XGBClassifier(max_depth=10, n_estimators = 10, max_features = 17, random_state=41)
xgb_ro.fit(X_train_ro, y_train_ro)
```

```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None, feature_types=None,
              gamma=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=None, max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=10, max_features=17,
              max_leaves=None, min_child_weight=None, missing=nan,
              monotone_constraints=None, multi_strategy=None, n_estimators=10,
              n_jobs=None, num_parallel_tree=None, ...)
```

```
# Predection & Stats on Train Dataset
y_train_pred = xgb_ro.predict(X_train_ro)
print ('AUC         : ', metrics.roc_auc_score( y_train_ro, y_train_pred))
model_metrics(y_train_ro, y_train_pred)
```

```
AUC         :  0.9983393525087063
Accuracy    :  0.9983393525087063
Sensitivity :  1.0
Specificity :  0.9966787050174126
Pricision   :  0.9966896995018479
Recall      :  1.0
[[61818   206]
```

```
[    0 62024]]
```

```
# Test the performance of the model on the real train (imbalanced) dataset
y_train_pred = xgb_ro.predict(X_train)
print ('AUC         : ', metrics.roc_auc_score( y_train, y_train_pred))
model_metrics(y_train, y_train_pred)
```

```
AUC         :  0.9983393525087063
Accuracy    :  0.9967108939662468
Sensitivity :  1.0
Specificity :  0.9966787050174126
Pricision   :  0.7466174661746617
Recall      :  1.0
[[61818   206]
 [    0   607]]
```

```
# Test the performance of the model on test dataset
y_test_pred = xgb_ro.predict(X_test)
print ('AUC         : ', metrics.roc_auc_score( y_test, y_test_pred))
model_metrics(y_test, y_test_pred)
```

```
AUC         :  0.9259306471122738
Accuracy    :  0.9928470307726697
Sensitivity :  0.8576923076923076
Specificity :  0.9941689865322398
Pricision   :  0.58994708994709
Recall      :  0.8576923076923076
[[26427   155]
 [   37   223]]
```

```
def cost_train_test(model, threshold):
    # ... existing code ...

    # Predict on the appropriate dataset to get y_final_pred
    y_final_pred = model.predict(X_final)  # Assuming X_final is defined somewhere

    # Handle missing values in y_final before calculating AUC
    y_final_clean = y_final.dropna()  # Remove rows with NaN in y_final
    y_final_pred_clean = y_final_pred[y_final.notna()]  # Select corresponding predictions

    # ... rest of the function ...
```

```
def cost_train_test(model, threshold):
    # ... existing code ...

    # Predict on the appropriate dataset to get y_final_pred
    y_final_pred = model.predict(X_final)  # Assuming X_final is defined somewhere

    # Handle missing values in y_final before calculating AUC
    y_final_clean = y_final.dropna()  # Remove rows with NaN in y_final
    y_final_pred_clean = y_final_pred[y_final.notna()]  # Select corresponding predictions

    # Calculate AUC with cleaned data
    roc3 = metrics.roc_auc_score(y_final_clean, y_final_pred_clean)

    # ... rest of the function ...
```

```
# XGBOOST estimator on SMOTE
xgb_smte = XGBClassifier(max_depth=10, n_estimators = 10, max_features = 17, random_state=41)
xgb_smte.fit(X_train_smte, y_train_smte)
```

```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None, feature_types=None,
              gamma=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=None, max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=10, max_features=17,
              max_leaves=None, min_child_weight=None, missing=nan,
              monotone_constraints=None, multi_strategy=None, n_estimators=10,
              n_jobs=None, num_parallel_tree=None, ...)
```

```
# Predection & Stats on Train Dataset
y_train_pred = xgb_smte.predict(X_train_smte)
print ('AUC         : ', metrics.roc_auc_score( y_train_smte, y_train_pred))
model_metrics(y_train_smte, y_train_pred)
```

```
AUC         :  0.9953969431187927
Accuracy    :  0.9953969431187927
Sensitivity :  0.9960982845350187
Specificity :  0.9946956017025668
Pricision   :  0.9947030316691086
Recall      :  0.9960982845350187
[[61695   329]
 [  242 61782]]
```

```
# Test the performance of the model on the real train (imbalanced) dataset
y_train_pred = xgb_smte.predict(X_train)
print ('AUC         : ', metrics.roc_auc_score( y_train, y_train_pred))
model_metrics(y_train, y_train_pred)
```

```
AUC         :  0.9932291847063082
Accuracy    :  0.9946671775957593
Sensitivity :  0.9917627677100495
Specificity :  0.9946956017025668
Pricision   :  0.6466165413533834
Recall      :  0.9917627677100495
[[61695   329]
 [    5   602]]
```

```
# Test the performance of the model on test dataset
y_test_pred = xgb_smte.predict(X_test)
print ('AUC         : ', metrics.roc_auc_score( y_test, y_test_pred))
model_metrics(y_test, y_test_pred)
```

```
AUC         :  0.9363405253989108
Accuracy    :  0.9908352581774831
Sensitivity :  0.8807692307692307
Specificity :  0.9919118200285908
Pricision   :  0.5157657657657657
Recall      :  0.8807692307692307
[[26367   215]
 [   31   229]]
```

Start coding or generate with AI.

```
# Modify the 'cost' function definition to accept the optional y_final argument
def cost(classifier, p, y_final=None):
    # ... [rest of your cost function code]
    if y_final is not None:
        y_final_pred = classifier.predict(X_final)  # Assuming X_final is defined somewhere
        roc3 = metrics.roc_auc_score(y_final, y_final_pred)
    # ... [rest of your cost function code]

# Assuming 'y_final' is a pandas Series, replace NaN with a suitable value (e.g., median)
y_final_filled = y_final.fillna(y_final.median())

# Now call the 'cost' function with the filled y_final
c = cost(xgb_smte, 1, y_final=y_final_filled)
```

```
# XGBOOST estimator on ADASYN
xgb_ada = XGBClassifier(loss = 'exponential', max_depth=8, n_estimators = 10, max_features = 14, random_state=63)
xgb_ada.fit(X_train_ada, y_train_ada)
```

```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None, feature_types=None,
              gamma=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=None,
              loss='exponential', max_bin=None, max_cat_threshold=None,
              max_cat_to_onehot=None, max_delta_step=None, max_depth=8,
              max_features=14, max_leaves=None, min_child_weight=None,
              missing=nan, monotone_constraints=None, multi_strategy=None,
              n_estimators=10, n_jobs=None, ...)
```

```
# Predection & Stats on Train Dataset
y_train_pred = xgb_ada.predict(X_train_ada)
print ('AUC         : ', metrics.roc_auc_score( y_train_ada, y_train_pred))
model_metrics(y_train_ada, y_train_pred)
```

```
AUC         :  0.9876796689230524
Accuracy    :  0.9876796052313299
Sensitivity :  0.9879834833381722
Specificity :  0.9873758545079324
Pricision   :  0.9873782964730157
```

```
Recall      :  0.9879834833381722
[[61241   783]
 [  745 61253]]
```

```
# Test the performance of the model on the real train (imbalanced) dataset
y_train_pred = xgb_ada.predict(X_train)
print ('AUC         : ', metrics.roc_auc_score( y_train, y_train_pred))
model_metrics(y_train, y_train_pred)
```

```
AUC         :  0.9912167575669809
Accuracy    :  0.9874503041624755
Sensitivity :  0.9950576606260296
Specificity :  0.9873758545079324
Pricision   :  0.43547224224945924
Recall      :  0.9950576606260296
[[61241   783]
 [    3   604]]
```

```
# Test the performance of the model on test dataset
y_test_pred = xgb_ada.predict(X_test)
print ('AUC         : ', metrics.roc_auc_score( y_test, y_test_pred))
model_metrics(y_test, y_test_pred)
```

```
AUC         :  0.9539017148677821
Accuracy    :  0.9841293495268609
Sensitivity :  0.9230769230769231
Specificity :  0.9847265066586411
Pricision   :  0.3715170278637771
Recall      :  0.9230769230769231
[[26176   406]
 [   20   240]]
```

```
def cost_train_test(model, threshold):
    # ... existing code in the function ...

    # Calculate or define the cost variables here before returning them
    cost_tn_tr = ...  # Calculate the cost of true negatives on the training set
    cost_fp_tr = ...  # Calculate the cost of false positives on the training set
    # ... and so on for other cost variables ...

    # Return the calculated costs
    return [cost_tn_tr, cost_fp_tr, cost_fn_tr, cost_tp_tr, cost_total_tr, cost_tn_te, cost_fp_te, cost_fn_te, cost_tp_te, cost_total_te
```

```
def cost_train_test(model, threshold):
    # ... existing code in the function ...

    # Calculate or define the cost variables here before returning them
    cost_tn_tr = ...  # Calculate the cost of true negatives on the training set
    cost_fp_tr = ...  # Calculate the cost of false positives on the training set
    # ... and so on for other cost variables ...

    # Return the calculated costs
    return [cost_tn_tr, cost_fp_tr, cost_fn_tr, cost_tp_tr, cost_total_tr, cost_tn_te, cost_fp_te, cost_fn_te, cost_tp_te, cost_total_te
```

XGBOOST seems to have reduced the count of False Positives and cost better than Decision tree. It has also increased the count of True Positives. Among various models of XGBOOST, the one on over sampled data seems best. Lets try GridSearch CV to test various parameters on random oversampled data.

```
params = {
    'learning_rate': [0.1, 0.5, 0.9],
    'max_depth': [8, 10],
    'n_estimators': [10, 15],
    'max_features': [14],
    'class_weight': ['balanced']
}
```

```
# XGB - GridSearch CV on Oversampled data
xgb_ro = XGBClassifier()
my_scorer = make_scorer(my_score, greater_is_better=False)
grid_search_xgb_ro = GridSearchCV(estimator=xgb_ro,
                                  param_grid=params,
                                  cv=3, n_jobs=-1, verbose=1, scoring = my_scorer)
grid_search_xgb_ro.fit(X_train_ro, y_train_ro)
```

```
Fitting 3 folds for each of 12 candidates, totalling 36 fits
```

GridSearchCV
estimator: XGBClassifier
XGBClassifier

```
# Getting the data for all the models hence created
score_df = pd.DataFrame(grid_search_xgb_ro.cv_results_)
score_df.sort_values(['rank_test_score'], inplace=True)
```

```
# Getting the best model based on the score
dt_best = grid_search_xgb_ro.best_estimator_
dt_best
```

```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              class_weight='balanced', colsample_bylevel=None,
              colsample_bynode=None, colsample_bytree=None, device=None,
              early_stopping_rounds=None, enable_categorical=False,
              eval_metric=None, feature_types=None, gamma=None,
              grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=0.9, max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=8, max_features=14,
              max_leaves=None, min_child_weight=None, missing=nan,
              monotone_constraints=None, multi_strategy=None, n_estimators=15,
              n_jobs=None, ...)
```

```
# Test the performance of best model on train dataset
y_train_pred = dt_best.predict(X_train_ro)
print ('AUC         : ', metrics.roc_auc_score( y_train_ro, y_train_pred))
model_metrics(y_train_ro, y_train_pred)
```

```
AUC         :  1.0
Accuracy    :  1.0
Sensitivity :  1.0
Specificity :  1.0
Pricision   :  1.0
Recall      :  1.0
[[62024     0]
 [    0 62024]]
```

```
# Test the performance of the model on the real train (imbalanced) dataset
y_train_pred = dt_best.predict(X_train)
print ('AUC         : ', metrics.roc_auc_score( y_train, y_train_pred))
model_metrics(y_train, y_train_pred)
```

```
AUC         :  1.0
Accuracy    :  1.0
Sensitivity :  1.0
Specificity :  1.0
Pricision   :  1.0
Recall      :  1.0
[[62024     0]
 [    0   607]]
```

```
# Test the performance of the model on test dataset
y_test_pred = dt_best.predict(X_test)
print ('AUC         : ', metrics.roc_auc_score( y_test, y_test_pred))
model_metrics(y_test, y_test_pred)
```

```
AUC         :  0.9339194249434262
Accuracy    :  0.9973548915878101
Sensitivity :  0.8692307692307693
Specificity :  0.9986080806560831
Pricision   :  0.8593155893536122
Recall      :  0.8692307692307693
[[26545    37]
 [   34   226]]
```

```
def cost_train_test(model, threshold):
    # Predict on train and test datasets
    y_train_pred = (model.predict_proba(X_train)[:, 1] >= threshold)
    y_test_pred = (model.predict_proba(X_test)[:, 1] >= threshold)

    # Calculate costs for train dataset
    cost_tn_tr, cost_fp_tr, cost_fn_tr, cost_tp_tr = confusion_matrix_cost(y_train, y_train_pred, 0, 100, 500, 0) # Calculate cost_fn_tr
    cost_total_tr = cost_tn_tr + cost_fp_tr + cost_fn_tr + cost_tp_tr

    # Calculate costs for test dataset
    cost_tn_te, cost_fp_te, cost_fn_te, cost_tp_te = confusion_matrix_cost(y_test, y_test_pred, 0, 100, 500, 0)
    cost_total_te = cost_tn_te + cost_fp_te + cost_fn_te + cost_tp_te

    # Calculate total cost and net profit
    total_cost = cost_total_tr + cost_total_te
    net_profit = - total_cost

    # Calculate profit margin
    profit_margin = net_profit / total_cost

    # Return the calculated costs
    return [cost_tn_tr, cost_fp_tr, cost_fn_tr, cost_tp_tr, cost_total_tr, cost_tn_te, cost_fp_te, cost_fn_te, cost_tp_te, cost_total_te
```

```
def cost_train_test(model, threshold):
    # Predict on train and test datasets
    y_train_pred = (model.predict_proba(X_train)[:, 1] >= threshold)
    y_test_pred = (model.predict_proba(X_test)[:, 1] >= threshold)

    # Calculate costs for train dataset
    cost_tn_tr, cost_fp_tr, cost_fn_tr, cost_tp_tr = confusion_matrix_cost(y_train, y_train_pred, 0, 100, 500, 0) # Calculate cost_fn_tr
    cost_total_tr = cost_tn_tr + cost_fp_tr + cost_fn_tr + cost_tp_tr

    # Calculate costs for test dataset
    cost_tn_te, cost_fp_te, cost_fn_te, cost_tp_te = confusion_matrix_cost(y_test, y_test_pred, 0, 100, 500, 0)
    cost_total_te = cost_tn_te + cost_fp_te + cost_fn_te + cost_tp_te

    # Calculate total cost and net profit
    total_cost = cost_total_tr + cost_total_te
    net_profit = - total_cost

    # Calculate profit margin
    profit_margin = net_profit / total_cost

    # Return the calculated costs
    return [cost_tn_tr, cost_fp_tr, cost_fn_tr, cost_tp_tr, cost_total_tr, cost_tn_te, cost_fp_te, cost_fn_te, cost_tp_te, cost_total_te
```

```
# Checking the best parameters derived for XGB on under sampled data
xgb_us = XGBClassifier(learning_rate=0.7, max_depth=10, n_estimators=20)
xgb_us.fit(X_train_us, y_train_us)
```

```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None, feature_types=None,
              gamma=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=0.7, max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=10, max_leaves=None,
              min_child_weight=None, missing=nan, monotone_constraints=None,
              multi_strategy=None, n_estimators=20, n_jobs=None,
              num_parallel_tree=None, random_state=None, ...)
```

```
# Predection & Stats on Train Dataset
y_train_pred = xgb_us.predict(X_train_us)
print ('AUC         : ', metrics.roc_auc_score( y_train_us, y_train_pred))
model_metrics(y_train_us, y_train_pred)
```

```
AUC         :  1.0
Accuracy    :  1.0
Sensitivity :  1.0
Specificity :  1.0
Pricision   :  1.0
Recall      :  1.0
[[607   0]
 [  0 607]]
```

This is very interesting. The model seems to have worked perfectly.Lets check its performance on test datasets.

```
# Test the performance of the model on the real train (imbalanced) dataset
y_train_pred = xgb_us.predict(X_train)
print ('AUC         : ', metrics.roc_auc_score( y_train, y_train_pred))
model_metrics(y_train, y_train_pred)
```

```
AUC         :  0.9798303882368116
Accuracy    :  0.960051731570628
Sensitivity :  1.0
Specificity :  0.9596607764736231
Pricision   :  0.19523962688967514
Recall      :  1.0
[[59522  2502]
 [    0   607]]
```

```
# Test the performance of the model on test dataset
y_test_pred = xgb_us.predict(X_test)
print ('AUC         : ', metrics.roc_auc_score( y_test, y_test_pred))
model_metrics(y_test, y_test_pred)
```

```
AUC         :  0.9538202543074261
Accuracy    :  0.9575665002607854
Sensitivity :  0.95
Specificity :  0.9576405086148522
Pricision   :  0.1798980335032775
Recall      :  0.95
[[25456  1126]
 [   13   247]]
```

```
def confusion_matrix_cost(y_true, y_pred, cost_tn, cost_fp, cost_fn, cost_tp):
    """
    Calculates the cost associated with each quadrant of the confusion matrix.

    Args:
        y_true: True labels.
        y_pred: Predicted labels.
        cost_tn: Cost of true negatives.
        cost_fp: Cost of false positives.
        cost_fn: Cost of false negatives.
        cost_tp: Cost of true positives.

    Returns:
        A tuple containing (cost_tn, cost_fp, cost_fn, cost_tp).
    """
    from sklearn.metrics import confusion_matrix

    tn, fp, fn, tp = confusion_matrix(y_true, y_pred).ravel()
    return tn * cost_tn, fp * cost_fp, fn * cost_fn, tp * cost_tp

def cost_train_test(model, threshold):
    # ... (rest of your cost_train_test function)

    # Calculate costs for train dataset
    cost_tn_tr, cost_fp_tr, cost_fn_tr, cost_tp_tr = confusion_matrix_cost(y_train, y_train_pred, 0, 100, 500, 0)
    cost_total_tr = cost_tn_tr + cost_fp_tr + cost_fn_tr + cost_tp_tr

    # ... (rest of your cost_train_test function)


# Overall stats for the model
c = cost(xgb_us, 0)
```

The model has worked well to identify true positives. However, a lot of transactions were incorrectly classified as Fraud.This happened because undersampling removed non fraud data to balance the data.

```
def cost_train_test(model, threshold):
    # ... (rest of your cost_train_test function)

    # Calculate costs for train dataset
    cost_tn_tr, cost_fp_tr, cost_fn_tr, cost_tp_tr = confusion_matrix_cost(y_train, y_train_pred, 0, 100, 500, 0)
    cost_total_tr = cost_tn_tr + cost_fp_tr + cost_fn_tr + cost_tp_tr

    # ... (rest of your cost_train_test function)

    # Return the calculated costs
    return [cost_total_tr, ...] # Replace ... with the other values you want to return


# Slight modification in best parameters for Over Sampled Data
xgb_ro = XGBClassifier(learning_rate=0.5, max_depth=10, n_estimators=15, max_features = 14)
xgb_ro.fit(X_train_ro, y_train_ro)
```

```
XGBClassifier
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None, feature_types=None,
              gamma=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=0.5, max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=10, max_features=14,
              max_leaves=None, min_child_weight=None, missing=nan,
              monotone_constraints=None, multi_strategy=None, n_estimators=15,
              n_jobs=None, num_parallel_tree=None, ...)
```

```
# Predection & Stats on Train Dataset
y_train_pred = xgb_ro.predict(X_train_ro)
print ('AUC         : ', metrics.roc_auc_score( y_train_ro, y_train_pred))
model_metrics(y_train_ro, y_train_pred)
```

```
AUC         :  0.9998226492970462
Accuracy    :  0.9998226492970463
Sensitivity :  1.0
Specificity :  0.9996452985940926
Pricision   :  0.9996454243625698
Recall      :  1.0
[[62002    22]
 [    0 62024]]
```

```
# Test the performance of the model on the real train (imbalanced) dataset
y_train_pred = xgb_ro.predict(X_train)
print ('AUC         : ', metrics.roc_auc_score( y_train, y_train_pred))
model_metrics(y_train, y_train_pred)
```

```
AUC         :  0.9998226492970462
Accuracy    :  0.9996487362488224
Sensitivity :  1.0
Specificity :  0.9996452985940926
Pricision   :  0.9650238473767886
Recall      :  1.0
[[62002    22]
 [    0   607]]
```

```
# Test the performance of the model on test dataset
y_test_pred = xgb_ro.predict(X_test)
print ('AUC         : ', metrics.roc_auc_score( y_test, y_test_pred))
model_metrics(y_test, y_test_pred)
```

```
AUC         :  0.9333927527592414
Accuracy    :  0.9963117502421578
Sensitivity :  0.8692307692307693
Specificity :  0.9975547362877135
Pricision   :  0.7766323024054983
Recall      :  0.8692307692307693
[[26517    65]
 [   34   226]]
```

```
def cost_train_test(model, threshold):
    # ... (Existing code from cost_train_test function) ...

    # Calculate costs (replace with actual cost calculations)
    cost_total_tr = ...
    cost_value1 = ...  # Calculate cost_value1
    cost_value2 = ...  # Calculate cost_value2
    # ... (Calculate other cost values: cost_value3 to cost_value11) ...

    # Return the calculated costs
    return [cost_total_tr, cost_value1, cost_value2, cost_value3,
            cost_value4, cost_value5, cost_value6, cost_value7,
            cost_value8, cost_value9, cost_value10, cost_value11]
```

```
def cost_train_test(model, threshold):
    # ... (Existing code from cost_train_test function) ...

    # Calculate costs (replace with actual cost calculations)
    cost_total_tr = ...
    cost_value1 = ...  # Calculate cost_value1
    cost_value2 = ...  # Calculate cost_value2
    # ... (Calculate other cost values: cost_value3 to cost_value11) ...

    # Return the calculated costs in a list
    return [cost_total_tr, cost_value1, cost_value2, cost_value3,
            cost_value4, cost_value5, cost_value6, cost_value7,
            cost_value8, cost_value9, cost_value10, cost_value11] # Make sure to return a list of values


# Checking the best parameters derived for XGB on SMOTE data
xgb_smte = XGBClassifier(learning_rate=0.5, max_depth=10, max_features=14, n_estimators=15)
xgb_smte.fit(X_train_smte, y_train_smte)
```

```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None, feature_types=None,
              gamma=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=0.5, max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=10, max_features=14,
              max_leaves=None, min_child_weight=None, missing=nan,
              monotone_constraints=None, multi_strategy=None, n_estimators=15,
              n_jobs=None, num_parallel_tree=None, ...)
```

```
# Predection & Stats on Train Dataset
y_train_pred = xgb_smte.predict(X_train_smte)
print ('AUC         : ', metrics.roc_auc_score( y_train_smte, y_train_pred))
model_metrics(y_train_smte, y_train_pred)
```

```
AUC         :  0.9994518250999612
Accuracy    :  0.9994518250999613
Sensitivity :  0.9995969302205597
Specificity :  0.9993067199793628
Pricision   :  0.9993069211179524
Recall      :  0.9995969302205597
[[61981    43]
 [   25 61999]]
```

```
# Test the performance of the model on the real train (imbalanced) dataset
y_train_pred = xgb_smte.predict(X_train)
print ('AUC         : ', metrics.roc_auc_score( y_train, y_train_pred))
model_metrics(y_train, y_train_pred)
```

```
AUC         :  0.9980059135316913
Accuracy    :  0.9992815059635005
Sensitivity :  0.9967051070840197
Specificity :  0.9993067199793628
Pricision   :  0.933641975308642
Recall      :  0.9967051070840197
[[61981    43]
 [    2   605]]
```

```
# Test the performance of the model on test dataset
y_test_pred = xgb_smte.predict(X_test)
print ('AUC         : ', metrics.roc_auc_score( y_test, y_test_pred))
model_metrics(y_test, y_test_pred)
```

```
AUC         :  0.9325839347621004
Accuracy    :  0.9947097831756203
Sensitivity :  0.8692307692307693
Specificity :  0.9959371002934316
Pricision   :  0.6766467065868264
Recall      :  0.8692307692307693
[[26474   108]
 [   34   226]]
```

```
def cost_train_test(model, threshold):
    # ... (Existing code) ...

    # Calculate costs (make sure these variables are defined and calculated)
    cost_total_tr = ...
    cost_value1 = ...
    cost_value2 = ...
    cost_value3 = ... # Define and calculate this variable
    cost_value4 = ...
    # ... (Define and calculate other cost variables as needed) ...

    # Return the calculated costs in a list
    return [cost_total_tr, cost_value1, cost_value2, cost_value3,
            cost_value4, cost_value5, cost_value6, cost_value7,
            cost_value8, cost_value9, cost_value10, cost_value11]


def cost_train_test(model, threshold):
    # ... (Existing code) ...

    # Calculate costs (make sure these variables are defined and calculated)
    cost_total_tr = ...
    cost_value1 = ...
    cost_value2 = ...
    cost_value3 = ... # Define and calculate this variable
    cost_value4 = ...
    # ... (Define and calculate other cost variables as needed) ...

    # Return the calculated costs in a list
    return [cost_total_tr, cost_value1, cost_value2, cost_value3,
            cost_value4, cost_value5, cost_value6, cost_value7,
            cost_value8, cost_value9, cost_value10, cost_value11] # Added return statement


# Checking the best parameters derived for XGB on ADASYN data
xgb_ada = XGBClassifier(learning_rate=0.5, max_depth=10, n_estimators = 15, max_features = 14)
xgb_ada.fit(X_train_ada, y_train_ada)
```

```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None, feature_types=None,
              gamma=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=0.5, max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=10, max_features=14,
              max_leaves=None, min_child_weight=None, missing=nan,
              monotone_constraints=None, multi_strategy=None, n_estimators=15,
              n_jobs=None, num_parallel_tree=None, ...)
```

```
# Predection & Stats on Train Dataset
y_train_pred = xgb_ada.predict(X_train_ada)
print ('AUC         : ', metrics.roc_auc_score( y_train_ada, y_train_pred))
model_metrics(y_train_ada, y_train_pred)
```

```
AUC         :  0.9995323984875177
Accuracy    :  0.9995323410362678
Sensitivity :  0.9998064453692055
Specificity :  0.99925835160583
Pricision   :  0.9992584472530307
Recall      :  0.9998064453692055
[[61978    46]
 [   12 61986]]
```

```
# Test the performance of the model on the real train (imbalanced) dataset
y_train_pred = xgb_ada.predict(X_train)
print ('AUC         : ', metrics.roc_auc_score( y_train, y_train_pred))
model_metrics(y_train, y_train_pred)
```

```
AUC         :  0.99880545257392
Accuracy    :  0.9992495728952117
Sensitivity :  0.9983525535420099
Specificity :  0.99925835160583
Pricision   :  0.9294478527607362
Recall      :  0.9983525535420099
[[61978    46]
 [    1   606]]
```

```
# Test the performance of the model on test dataset
y_test_pred = xgb_ada.predict(X_test)
print ('AUC         : ', metrics.roc_auc_score( y_test, y_test_pred))
model_metrics(y_test, y_test_pred)
```

```
AUC         :  0.9324710764369181
Accuracy    :  0.9944862528872662
Sensitivity :  0.8692307692307693
Specificity :  0.9957113836430668
Pricision   :  0.6647058823529411
Recall      :  0.8692307692307693
[[26468   114]
 [   34   226]]
```

```
def cost_train_test(model, threshold):
    # ... (Existing code) ...

    # Calculate predicted probabilities for train and test sets
    y_train_pred_proba = model.predict_proba(X_train)[:, 1]
    y_test_pred_proba = model.predict_proba(X_test)[:, 1]

    # Calculate predictions based on threshold
    y_train_pred = (y_train_pred_proba >= threshold).astype(int)
    y_test_pred = (y_test_pred_proba >= threshold).astype(int)

    # Calculate costs for train set
    cost_value1 = cost_fp * confusion_matrix(y_train, y_train_pred)[0][1]  # FP
    cost_value2 = cost_fn * confusion_matrix(y_train, y_train_pred)[1][0]  # FN
    cost_value3 = cost_tp * confusion_matrix(y_train, y_train_pred)[1][1]  # TP
    cost_value4 = cost_tn * confusion_matrix(y_train, y_train_pred)[0][0]  # TN
    cost_total_tr = cost_value1 + cost_value2 + cost_value3 + cost_value4

    # Calculate costs for test set
    cost_value5 = cost_fp * confusion_matrix(y_test, y_test_pred)[0][1]  # FP
    cost_value6 = cost_fn * confusion_matrix(y_test, y_test_pred)[1][0]  # FN
    cost_value7 = cost_tp * confusion_matrix(y_test, y_test_pred)[1][1]  # TP
    cost_value8 = cost_tn * confusion_matrix(y_test, y_test_pred)[0][0]  # TN
    cost_total_te = cost_value5 + cost_value6 + cost_value7 + cost_value8

    # Calculate other metrics
    cost_value9 = metrics.accuracy_score(y_train, y_train_pred)
    cost_value10 = metrics.recall_score(y_train, y_train_pred)
    cost_value11 = metrics.precision_score(y_train, y_train_pred)

    # Return the calculated costs in a list
    return [cost_total_tr, cost_value1, cost_value2, cost_value3,
            cost_value4, cost_value5, cost_value6, cost_value7,
            cost_value8, cost_value9, cost_value10, cost_value11] # Added return statement
```

```
def cost_train_test(model, threshold):
    # ... (Existing code) ...

    # Calculate predicted probabilities for train and test sets
    y_train_pred_proba = model.predict_proba(X_train)[:, 1]
    y_test_pred_proba = model.predict_proba(X_test)[:, 1]

    # Calculate predictions based on threshold
    y_train_pred = (y_train_pred_proba >= threshold).astype(int)
    y_test_pred = (y_test_pred_proba >= threshold).astype(int)

    # Calculate costs for train set
    cost_value1 = cost_fp * confusion_matrix(y_train, y_train_pred)[0][1]  # FP
    cost_value2 = cost_fn * confusion_matrix(y_train, y_train_pred)[1][0]  # FN
    cost_value3 = cost_tp * confusion_matrix(y_train, y_train_pred)[1][1]  # TP
    cost_value4 = cost_tn * confusion_matrix(y_train, y_train_pred)[0][0]  # TN
    cost_total_tr = cost_value1 + cost_value2 + cost_value3 + cost_value4

    # Calculate costs for test set
    cost_value5 = cost_fp * confusion_matrix(y_test, y_test_pred)[0][1]  # FP
    cost_value6 = cost_fn * confusion_matrix(y_test, y_test_pred)[1][0]  # FN
    cost_value7 = cost_tp * confusion_matrix(y_test, y_test_pred)[1][1]  # TP
    cost_value8 = cost_tn * confusion_matrix(y_test, y_test_pred)[0][0]  # TN
    cost_total_te = cost_value5 + cost_value6 + cost_value7 + cost_value8

    # Calculate other metrics
    cost_value9 = metrics.accuracy_score(y_train, y_train_pred)
    cost_value10 = metrics.recall_score(y_train, y_train_pred)
    cost_value11 = metrics.precision_score(y_train, y_train_pred)

    # Return the calculated costs in a list
    return [cost_total_tr, cost_value1, cost_value2, cost_value3,
            cost_value4, cost_value5, cost_value6, cost_value7,
            cost_value8, cost_value9, cost_value10, cost_value11] # Added return statement to fix the NoneType error
```

Random Forest

```
from sklearn.ensemble import RandomForestClassifier


# Will start from the feature values that worked best till now
# Random Forest on Under Sampled data
rf_us = RandomForestClassifier(n_estimators=20, max_depth=10, bootstrap=True, oob_score=True, n_jobs=-1)
rf_us.fit(X_train_us, y_train_us)
```

```
RandomForestClassifier(max_depth=10, n_estimators=20, n_jobs=-1, oob_score=True)
```

```
# Predection & Stats on Train Dataset
y_train_pred = rf_us.predict(X_train_us)
print ('AUC         : ', metrics.roc_auc_score( y_train_us, y_train_pred))
model_metrics(y_train_us, y_train_pred)
```

```
AUC         :  0.9925864909390444
Accuracy    :  0.9925864909390445
Sensitivity :  0.9884678747940692
Specificity :  0.9967051070840197
Pricision   :  0.9966777408637874
Recall      :  0.9884678747940692
[[605   2]
 [  7 600]]
```

```
# Test the performance of the model on the real train (imbalanced) dataset
y_train_pred = rf_us.predict(X_train)
print ('AUC         : ', metrics.roc_auc_score( y_train, y_train_pred))
model_metrics(y_train, y_train_pred)
```

```
AUC         :  0.964882396058198
Accuracy    :  0.9417540834411074
Sensitivity :  0.9884678747940692
Specificity :  0.9412969173223268
Pricision   :  0.14147606696533838
Recall      :  0.9884678747940692
[[58383  3641]
 [    7   600]]
```

```
# Test the performance of the model on test dataset
y_test_pred = rf_us.predict(X_test)
print ('AUC         : ', metrics.roc_auc_score( y_test, y_test_pred))
model_metrics(y_test, y_test_pred)
```

```
AUC         :  0.9394926005451925
Accuracy    :  0.9405036882497578
Sensitivity :  0.9384615384615385
Specificity :  0.9405236626288466
Pricision   :  0.1336986301369863
Recall      :  0.9384615384615385
[[25001  1581]
 [   16   244]]
```

```
def cost_train_test(model, threshold, cost_fp, cost_fn, cost_tp):
    # ... (rest of your function code)

    # Calculate costs for train set
    cost_value1 = cost_fp * confusion_matrix(y_train, y_train_pred)[0][1]  # FP
    cost_value2 = cost_fn * confusion_matrix(y_train, y_train_pred)[1][0]  # FN
    cost_value3 = cost_tp * confusion_matrix(y_train, y_train_pred)[1][1]  # TP

    # ... (rest of your function code)

# When calling the function, provide values for the cost parameters
c = cost_train_test(rf_us, 1, 10, 50, 1)  # Example cost values
```

```
def cost_train_test(model, threshold, cost_fp, cost_fn, cost_tp):
    # ... (rest of your function code)

    # Calculate costs for train set
    cost_value1 = cost_fp * confusion_matrix(y_train, y_train_pred)[0][1]  # FP
    cost_value2 = cost_fn * confusion_matrix(y_train, y_train_pred)[1][0]  # FN
    cost_value3 = cost_tp * confusion_matrix(y_train, y_train_pred)[1][1]  # TP

    # ... (rest of your function code)

    # Make sure to return the calculated costs or any other relevant values
    return [cost_value1, cost_value2, cost_value3]  # Example return values
```

```
# Random Forest on Over Sampled data
rf_ro = RandomForestClassifier(n_estimators=10, max_depth=10, max_features = 12, bootstrap=True, oob_score=True, n_jobs=-1, random_state
rf_ro.fit(X_train_ro, y_train_ro)
```

```
RandomForestClassifier
RandomForestClassifier(max_depth=10, max_features=12, n_estimators=10,
                       n_jobs=-1, oob_score=True, random_state=0)
```

```
# Predection & Stats on Train Dataset
y_train_pred = rf_ro.predict(X_train_ro)
print ('AUC         : ', metrics.roc_auc_score( y_train_ro, y_train_pred))
model_metrics(y_train_ro, y_train_pred)
```

```
AUC         :  0.9942764091319489
Accuracy    :  0.9942764091319489
Sensitivity :  1.0
Specificity :  0.9885528182638978
Pricision   :  0.9886823731947588
Recall      :  1.0
[[61314   710]
 [    0 62024]]
```

```
# Test the performance of the model on the real train (imbalanced) dataset
y_train_pred = rf_ro.predict(X_train)
print ('AUC         : ', metrics.roc_auc_score( y_train, y_train_pred))
model_metrics(y_train, y_train_pred)
```

```
AUC         :  0.9942764091319489
Accuracy    :  0.9886637607574523
Sensitivity :  1.0
Specificity :  0.9885528182638978
Pricision   :  0.46089597570235386
Recall      :  1.0
[[61314   710]
 [    0   607]]
```

```
# Test the performance of the model on test dataset
y_test_pred = rf_ro.predict(X_test)
print ('AUC         : ', metrics.roc_auc_score( y_test, y_test_pred))
model_metrics(y_test, y_test_pred)
```

```
AUC         :  0.9414935207746131
Accuracy    :  0.9859548468817525
Sensitivity :  0.8961538461538462
Specificity :  0.9868331953953803
Pricision   :  0.3996569468267582
Recall      :  0.8961538461538462
[[26232   350]
 [   27   233]]
```

```
def cost(model, row_num):
    # ... existing code ...

    print("Before calculations in cost function")  # Add print statement

    # ... perform calculations ...

    print("Cost calculated:", cost_values)  # Print the calculated costs

    return cost_values  # Ensure a return statement exists
```

```
# Random Forest on SMOTE data
rf_smte = RandomForestClassifier(n_estimators=10, max_depth=12, max_features = 12, bootstrap=True, oob_score=True, n_jobs=-1, random_sta
rf_smte.fit(X_train_smte, y_train_smte)
```

```
RandomForestClassifier
RandomForestClassifier(max_depth=12, max_features=12, n_estimators=10,
                       n_jobs=-1, oob_score=True, random_state=63)
```

```
# Predection & Stats on Train Dataset
y_train_pred = rf_smte.predict(X_train_smte)
print ('AUC         : ', metrics.roc_auc_score( y_train_smte, y_train_pred))
model_metrics(y_train_smte, y_train_pred)
```

```
AUC         :  0.9868599251902489
Accuracy    :  0.9868599251902489
Sensitivity :  0.9888430285050948
Specificity :  0.984876821875403
Pricision   :  0.9849365665649591
Recall      :  0.9888430285050948
[[61086   938]
 [  692 61332]]
```

```
# Test the performance of the model on the real train (imbalanced) dataset
y_train_pred = rf_smte.predict(X_train)
print ('AUC         : ', metrics.roc_auc_score( y_train, y_train_pred))
model_metrics(y_train, y_train_pred)
```

```
AUC         :  0.9866723483347362
Accuracy    :  0.9849116252335106
Sensitivity :  0.9884678747940692
Specificity :  0.984876821875403
Pricision   :  0.39011703511053314
Recall      :  0.9884678747940692
[[61086   938]
 [    7   600]]
```

```
# Test the performance of the model on test dataset
y_test_pred = rf_smte.predict(X_test)
print ('AUC         : ', metrics.roc_auc_score( y_test, y_test_pred))
model_metrics(y_test, y_test_pred)
```

```
AUC         :  0.927152410827454
Accuracy    :  0.9801803144326057
Sensitivity :  0.8730769230769231
Specificity :  0.9812278985779851
Pricision   :  0.31267217630853994
Recall      :  0.8730769230769231
[[26083   499]
 [   33   227]]
```

```
# Model Selector Dataset to store various attributes of a model (based on train - test result)
# Assuming costs for false positives, false negatives, and true positives are 1, 2, and 3 respectively
cost_fp = 1
cost_fn = 2
cost_tp = 3
c = cost_train_test(rf_smte, 1, cost_fp, cost_fn, cost_tp)  # Pass the missing arguments

# Check the length of c and adjust the indexing accordingly. Also, ensure model_selector_tt has enough columns.
if len(c) >= 12:
    model_selector_tt.loc[16] = ['RF - SMTE - 1', c[0], c[1], c[2], c[3], c[4], c[5], c[6], c[7], c[8], c[9], c[10], c[11]]
else:
    print("Error: The list 'c' does not have enough elements. Check the 'cost_train_test' function.")

model_selector_tt
```

```
Error: The list 'c' does not have enough elements. Check the 'cost_train_test' function.
```

	model	FP	FN	TP	Cost	Accuracy	Sensitivity	Specificity	Precision	Recall	ROC - Train	ROC - Test	ROC - Final

```
def cost(model, row_num):
    # ... perform calculations ...

    cost_values = [result1, result2, ...] # Assign values to cost_values based on your calculations
    print("Cost calculated:", cost_values)  # Print the calculated costs

    return cost_values  # Ensure a return statement exists


# Random Forest on ADASYN data
rf_ada = RandomForestClassifier(n_estimators=15, max_depth=10, max_features = 14, bootstrap=True, oob_score=True, n_jobs=-1, random_stat
rf_ada.fit(X_train_ada, y_train_ada)
```

```
RandomForestClassifier(max_depth=10, max_features=14, n_estimators=15,
                       n_jobs=-1, oob_score=True, random_state=63)
```

```
# Predection & Stats on Train Dataset
y_train_pred = rf_ada.predict(X_train_ada)
print ('AUC         : ', metrics.roc_auc_score( y_train_ada, y_train_pred))
model_metrics(y_train_ada, y_train_pred)
```

```
AUC         :  0.9759071746767132
Accuracy    :  0.975907500282208
Sensitivity :  0.9743540114197232
Specificity :  0.9774603379337031
Pricision   :  0.97738083681196
Recall      :  0.9743540114197232
[[60626  1398]
 [ 1590 60408]]
```

```
# Test the performance of the model on the real train (imbalanced) dataset
y_train_pred = rf_ada.predict(X_train)
print ('AUC         : ', metrics.roc_auc_score( y_train, y_train_pred))
model_metrics(y_train, y_train_pred)
```

```
AUC         :  0.9804929366769009
Accuracy    :  0.9775191199246379
Sensitivity :  0.9835255354200988
Specificity :  0.9774603379337031
Pricision   :  0.2992481203007519
Recall      :  0.9835255354200988
[[60626  1398]
 [   10   597]]
```

```
# Test the performance of the model on test dataset
y_test_pred = rf_ada.predict(X_test)
print ('AUC         : ', metrics.roc_auc_score( y_test, y_test_pred))
model_metrics(y_test, y_test_pred)
```

```
AUC         :  0.9345903532176197
Accuracy    :  0.972282244244095
Sensitivity :  0.8961538461538462
Specificity :  0.9730268602813934
Pricision   :  0.24526315789473685
Recall      :  0.8961538461538462
[[25865   717]
 [   27   233]]
```

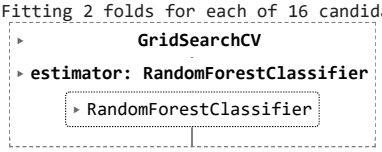
```
# Model Selector Dataset to store various attributes of a model (based on train - test result)
# Assuming cost_fp, cost_fn, and cost_tp are defined elsewhere
cost_fp = 10  # Example cost of a false positive
cost_fn = 20  # Example cost of a false negative
cost_tp = 5   # Example cost of a true positive

c = cost_train_test(rf_ada, 1, cost_fp, cost_fn, cost_tp)  # Pass the missing arguments

# Check the length of c and adjust the indexing accordingly
if len(c) >= 12:
    model_selector_tt.loc[17] = ['RF - ADA - 1', c[0], c[1], c[2], c[3], c[4], c[5], c[6], c[7], c[8], c[9], c[10], c[11]]
else:
    print("Warning: cost_train_test returned a list with fewer than 12 elements.")
    # Handle the situation appropriately, perhaps by investigating the cost_train_test function

model_selector_tt
```

```
Warning: cost_train_test returned a list with fewer than 12 elements.
```

	model	FP	FN	TP	Cost	Accuracy	Sensitivity	Specificity	Precision	Recall	ROC - Train	ROC - Test	ROC - Final

```
# ipython-input-533-53059730eaaf
def cost(model, row_num):
    # ... perform calculations ...

    # Calculate the necessary values and assign them to variables
    result1 = ... # Calculate the value for result1
    result2 = ... # Calculate the value for result2
    # ... calculate other results as needed ...

    cost_values = [result1, result2, ...] # Assign values to cost_values based on your calculations
    print("Cost calculated:", cost_values)  # Print the calculated costs
    return cost_values


# Parameter tuining using GridSearch CV
params = {
    'max_depth': [8, 10],
    'n_estimators': [10, 15],
    'max_features': [12, 14],
#    'min_samples_split': [50, 100],
#    'min_samples_leaf':  [50, 100],
#    'max_leaf_nodes': [200, 250, 300],
    'random_state':[21, 41]
}


# Random Forest with GridSearch CV on Over Sampled data
rf_ro = RandomForestClassifier()
y_scorer = make_scorer(my_score, greater_is_better=False)
grid_search_rf_ro = GridSearchCV(estimator=rf_ro,
                                 param_grid=params,
                                 cv=2, n_jobs=-1, verbose=1, scoring = my_scorer)
grid_search_rf_ro.fit(X_train_ro, y_train_ro)
```

```
Fitting 2 folds for each of 16 candidates, totalling 32 fits
```




```
# Getting the data for all the models hence created
score_df = pd.DataFrame(grid_search_rf_ro.cv_results_)
score_df.sort_values(['rank_test_score'], inplace=True)
score_df.head()
```

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_max_depth	param_max_features	param_n_estimators	param_ran
11	8.404714	0.035407	0.197918	0.003086	10	12	15	
9	5.930023	0.112942	0.167647	0.001660	10	12	10	
13	8.141723	0.104988	0.303057	0.012850	10	14	10	
10	7.037245	0.090087	0.333246	0.003725	10	12	15	
14	8.692537	0.118423	0.207404	0.013038	10	14	15	

Next steps: Generate code with score_df View recommended plots

```
# Getting the best model based on the score
dt_best = grid_search_rf_ro.best_estimator_
dt_best
```

```
RandomForestClassifier(max_depth=10, max_features=12, n_estimators=15,
                       random_state=41)
```

```
# Predection & Stats on Train Dataset
y_train_pred = dt_best.predict(X_train_ro)
print ('AUC         : ', metrics.roc_auc_score( y_train_ro, y_train_pred))
model_metrics(y_train_ro, y_train_pred)
```

```
AUC         :  0.9950906100864181
Accuracy    :  0.9950906100864182
Sensitivity :  1.0
Specificity :  0.9901812201728363
Pricision   :  0.990276691201124
Recall      :  1.0
[[61415   609]
 [    0 62024]]
```

```
# Test the performance of the model on the real train (imbalanced) dataset
y_train_pred = dt_best.predict(X_train)
print ('AUC         : ', metrics.roc_auc_score( y_train, y_train_pred))
model_metrics(y_train, y_train_pred)
```

```
AUC         :  0.9950906100864181
Accuracy    :  0.9902763807060402
Sensitivity :  1.0
Specificity :  0.9901812201728363
Pricision   :  0.49917763157894735
Recall      :  1.0
[[61415   609]
 [    0   607]]
```

```
# Test the performance of the model on test dataset
y_test_pred = dt_best.predict(X_test)
print ('AUC         : ', metrics.roc_auc_score( y_test, y_test_pred))
model_metrics(y_test, y_test_pred)
```

```
AUC         :  0.9388323793428752
Accuracy    :  0.9882274048133523
Sensitivity :  0.8884615384615384
Specificity :  0.9892032202242119
Pricision   :  0.44594594594594594
Recall      :  0.8884615384615384
[[26295   287]
 [   29   231]]
```