Ethereum Fraud Detection

Sections:

- Housekeeping and Imports
- Data Loading
- Data Exploration
- Data Transformation and Preparation
- Model Exploration and Performance Analysis
- Final Model Building(Hyperparamter Tuning)
- Feature Importance and Selection

Housekeeping and Imports

For importing libraries necessary for the project, and for basic preprocessing functions.

We're going to import commonly used Data Science libraries

In [ ]:
```
## IMPORT REQUIRED PYTHON LIBRARIES
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.preprocessing import StandardScaler
from imblearn.over_sampling import SMOTE, SMOTENC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.pipeline import make_pipeline
from sklearn import tree
from scipy import stats
from xgboost import XGBClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score,confusion_matrix,classification_report,roc_auc_score, roc_curve
from sklearn.model_selection import StratifiedKFold, cross_val_score, GridSearchCV, RandomizedSearchCV
import warnings
warnings.filterwarnings('ignore')
from numpy import sort
from sklearn.feature_selection import SelectFromModel
```

Data Loading

For loading data files into appropriate variables.

In [ ]:
```
## DOWNLOAD DATA AND READ IT WITH PANDAS
url1 = 'https://raw.githubusercontent.com/RushilPatel0703/DAE_Project/main/eth_illicit_features.csv'
data_1 = pd.read_csv(url1)

url2 = 'https://raw.githubusercontent.com/RushilPatel0703/DAE_Project/main/transaction_dataset.csv'
data_2 = pd.read_csv(url2)
pd.reset_option('max_rows')
```

Data Exploration

Section for **exploratory analysis** on the available data.

The exploration techniques vary for numerical, categorical, or time-series variables. Currently,

Here we typically:

- look at example records in the dataset
- Merging datasets, keeping features that are present in both datasets
- investigate the datatypes of variables in the dataset
- calculate and investigate descriptive statistics (ex: central tendencies, variability etc.)
- investigate distribution of feature vectors (ex: to check for skewness and outliers)
- investigate distribution of prediction vector
- check out the relationship (ex: correlation) between different features
- check out the relationship between feature vectors and prediction vector

Common steps to check the health of the data:

- Check for missing data
- Check the skewness of the data, outlier detection
- etc...

Look at Example Records

In [ ]:
```
data_1.head()
```

Out[ ]:

	address	flag	minTimeBetweenSentTnx	maxTimeBetweenSentTnx	avgTimeBetweenSentTnx	minTimeBet
0	0xd0cc2b24980cbcca47ef755da88b220a82291407	1	0.0	2387389.0	5.807655e+04	
1	0x4cdc1cba0aeb5539f2e0ba158281e67e0e54a9b1	1	0.0	0.0	0.000000e+00	
2	0x00e01a648ff41346cdeb873182383333d2184dd1	1	37.0	25112882.0	1.710279e+06	
3	0x858457daa7e087ad74cdeeceab8419079bc2ca03	1	0.0	642460.0	1.576106e+04	
4	0x240e125c20a4cc84bd6e7f8d1fd07aff4c06d43d	1	0.0	0.0	0.000000e+00	

5 rows × 34 columns

In [ ]:
```
data_2.head()
```

Out[ ]:

	Unnamed: 0	Index	Address	FLAG	Avg min between sent tnx	Avg min between received tnx	Time Diff between first and last (Mins)	Sent tnx	Received Tnx	Number of Created Contracts	...	ERC
0	0	1	0x00009277775ac7d0d59eaad8fee3d10ac6c805e8	0	844.26	1093.71	704785.63	721	89	0	...	0.
1	1	2	0x0002b44ddb1476db43c868bd494422ee4c136fed	0	12709.07	2958.44	1218216.73	94	8	0	...	2.
2	2	3	0x0002bda54cb772d040f779e88eb453cac0daa244	0	246194.54	2434.02	516729.30	2	10	0	...	0.
3	3	4	0x00038e6ba2fd5c09aedb96697c8d7b8fa6632e5e	0	10219.60	15785.09	397555.90	25	9	0	...	100.
4	4	5	0x00062d1dd1afb6fb02540ddad9cdebfe568e0d89	0	36.61	10707.77	382472.42	4598	20	1	...	0.

5 rows × 51 columns

Merging datasets, keeping features that are present in both datasets

In [ ]:
```
# DROPPING COLUMNS NOT IN COMMON WITH data_2
data_1.drop(['minTimeBetweenSentTnx', 'maxTimeBetweenSentTnx', 'minTimeBetweenRecTnx', 'maxTimeBetweenRecTnx',
       'dailyMax', 'ratioRecSent', 'ratioSentTotal', 'ratioRecTotal',
       'giniSent', 'giniRec', 'txFreq', 'stdBalanceEth'], inplace=True, axis=1)
```

In [ ]:
```
data_1.columns
```

Out[ ]:
```
Index(['address', 'flag', 'avgTimeBetweenSentTnx', 'avgTimeBetweenRecTnx',
       'sentTransactions', 'receivedTransactions', 'createdContracts',
       'numUniqSentAddress', 'numUniqRecAddress', 'minValSent', 'maxValSent',
       'avgValSent', 'minValReceived', 'maxValReceived', 'avgValReceived',
       'totalTransactions', 'totalEtherSent', 'totalEtherReceived',
       'totalEtherSentContracts', 'totalEtherBalance'],
      dtype='object')
```

In [ ]:
```
# DROPPING COLUMNS NOT IN COMMON WITH  data_1
data_2.drop(['Unnamed: 0', 'Index', 'Time Diff between first and last (Mins)',
       'min value sent to contract', 'max val sent to contract', 'avg value sent to contract',
       ' Total ERC20 tnxs', ' ERC20 total Ether received',
       ' ERC20 total ether sent', ' ERC20 total Ether sent contract',
       ' ERC20 uniq sent addr', ' ERC20 uniq rec addr',
       ' ERC20 uniq sent addr.1', ' ERC20 uniq rec contract addr',
       ' ERC20 avg time between sent tnx', ' ERC20 avg time between rec tnx',
       ' ERC20 avg time between rec 2 tnx',
       ' ERC20 avg time between contract tnx', ' ERC20 min val rec',
       ' ERC20 max val rec', ' ERC20 avg val rec', ' ERC20 min val sent',
       ' ERC20 max val sent', ' ERC20 avg val sent',
       ' ERC20 min val sent contract', ' ERC20 max val sent contract',
       ' ERC20 avg val sent contract', ' ERC20 uniq sent token name',
       ' ERC20 uniq rec token name', ' ERC20 most sent token type',
       ' ERC20_most_rec_token_type'], inplace=True, axis=1)
```

In [ ]:
```
data_2.columns
```

Out[ ]:
```
Index(['Address', 'FLAG', 'Avg min between sent tnx',
       'Avg min between received tnx', 'Sent tnx', 'Received Tnx',
       'Number of Created Contracts', 'Unique Received From Addresses',
       'Unique Sent To Addresses', 'min value received', 'max value received ',
       'avg val received', 'min val sent', 'max val sent', 'avg val sent',
       'total transactions (including tnx to create contract',
       'total Ether sent', 'total ether received',
       'total ether sent contracts', 'total ether balance'],
      dtype='object')
```

In [ ]:
```
# RENAMING COLUMNS TO COLUMN NAMES IN data_1 FOR EASE OF
data_2.rename(columns= {'Address':'address', 'FLAG':'flag', 'Avg min between sent tnx':'avgTimeBetweenSentTnx',
                        'Avg min between received tnx':'avgTimeBetweenRecTnx', 'Sent tnx':'sentTransactions',
                        'Received Tnx':'receivedTransactions', 'Number of Created Contracts':'createdContracts
                        'Unique Received From Addresses':'numUniqRecAddress', 'Unique Sent To Addresses':'numUn
                        'min value received':'minValReceived','max value received ':'maxValReceived',
                        'avg val received':'avgValReceived', 'min val sent':'minValSent', 'max val sent':'maxVa
                        'avg val sent': 'avgValSent', 'total transactions (including tnx to create contract':'t
                        'total Ether sent':'totalEtherSent', 'total ether received':'totalEtherReceived',
                        'total ether sent contracts':'totalEtherSentContracts', 'total ether balance':'totalEth
```

In [ ]:
```
df = pd.concat([data_1, data_2]) # COMBINE data_1 AND data_2
```

Data-types, completeness Information

Using the Pandas "info" function, in addition to the data-type information for the dataset, we can look at counts of available records/missing records too.

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

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 21987 entries, 0 to 9840
Data columns (total 20 columns):
 #   Column                   Non-Null Count  Dtype
---  ------                   --------------  -----
 0   address                  21987 non-null  object
 1   flag                     21987 non-null  int64
 2   avgTimeBetweenSentTnx    21987 non-null  float64
 3   avgTimeBetweenRecTnx     21987 non-null  float64
 4   sentTransactions         21987 non-null  int64
 5   receivedTransactions     21987 non-null  int64
 6   createdContracts         21987 non-null  int64
 7   numUniqSentAddress       21987 non-null  int64
 8   numUniqRecAddress        21987 non-null  int64
 9   minValSent               21987 non-null  float64
 10  maxValSent               21987 non-null  float64
 11  avgValSent               21987 non-null  float64
 12  minValReceived           21987 non-null  float64
 13  maxValReceived           21987 non-null  float64
 14  avgValReceived           21987 non-null  float64
 15  totalTransactions        21987 non-null  int64
 16  totalEtherSent           21987 non-null  float64
 17  totalEtherReceived       21987 non-null  float64
 18  totalEtherSentContracts  21987 non-null  float64
 19  totalEtherBalance        21987 non-null  float64
dtypes: float64(12), int64(7), object(1)
memory usage: 3.5+ MB
```

In [ ]:
```
df.isnull().sum() # LOOK IF THERE IS ANY MISSING VALUES
```

Out[ ]:
```
address                    0
flag                       0
avgTimeBetweenSentTnx      0
avgTimeBetweenRecTnx       0
sentTransactions           0
receivedTransactions       0
createdContracts           0
numUniqSentAddress         0
numUniqRecAddress          0
minValSent                 0
maxValSent                 0
avgValSent                 0
minValReceived             0
maxValReceived             0
avgValReceived             0
totalTransactions          0
totalEtherSent             0
totalEtherReceived         0
totalEtherSentContracts    0
totalEtherBalance          0
dtype: int64
```

Descriptive Statistics

In [ ]:
```
df.describe()
# Additonal:
#     We can also make a guess at the skewness of the data at this stage by looking at the difference between
#     the means and medians of numerical features
```

Out[ ]:

	flag	avgTimeBetweenSentTnx	avgTimeBetweenRecTnx	sentTransactions	receivedTransactions	createdContracts	numUniqSentAd
count	21987.000000	2.198700e+04	2.198700e+04	21987.000000	21987.000000	21987.000000	21987.0
mean	0.333333	2.017807e+06	8.849104e+07	337.691409	448.022786	2.641788	66.6
std	0.471415	3.068441e+07	1.905694e+08	1496.171520	1807.664973	122.922835	496.8
min	0.000000	0.000000e+00	0.000000e+00	0.000000	0.000000	0.000000	0.0
25%	0.000000	0.000000e+00	1.020000e+00	1.000000	2.000000	0.000000	1.0
50%	0.000000	2.171600e+02	1.033698e+04	3.000000	8.000000	0.000000	2.0
75%	1.000000	3.269873e+04	6.604927e+07	32.000000	42.000000	0.000000	8.0
max	1.000000	8.291299e+08	8.309932e+08	10000.000000	10000.000000	9995.000000	9950.0

In [ ]:
```
df.duplicated().sum() # LOOK FOR ANY DUPLICATES IN DATA
```

Out[ ]:
```
125
```

In [ ]:
```
df.drop_duplicates(inplace=True) # REMOVING DUPLICATES
df.duplicated().sum()
```

Out[ ]:
```
0
```

Visualizaton: Distribution of features

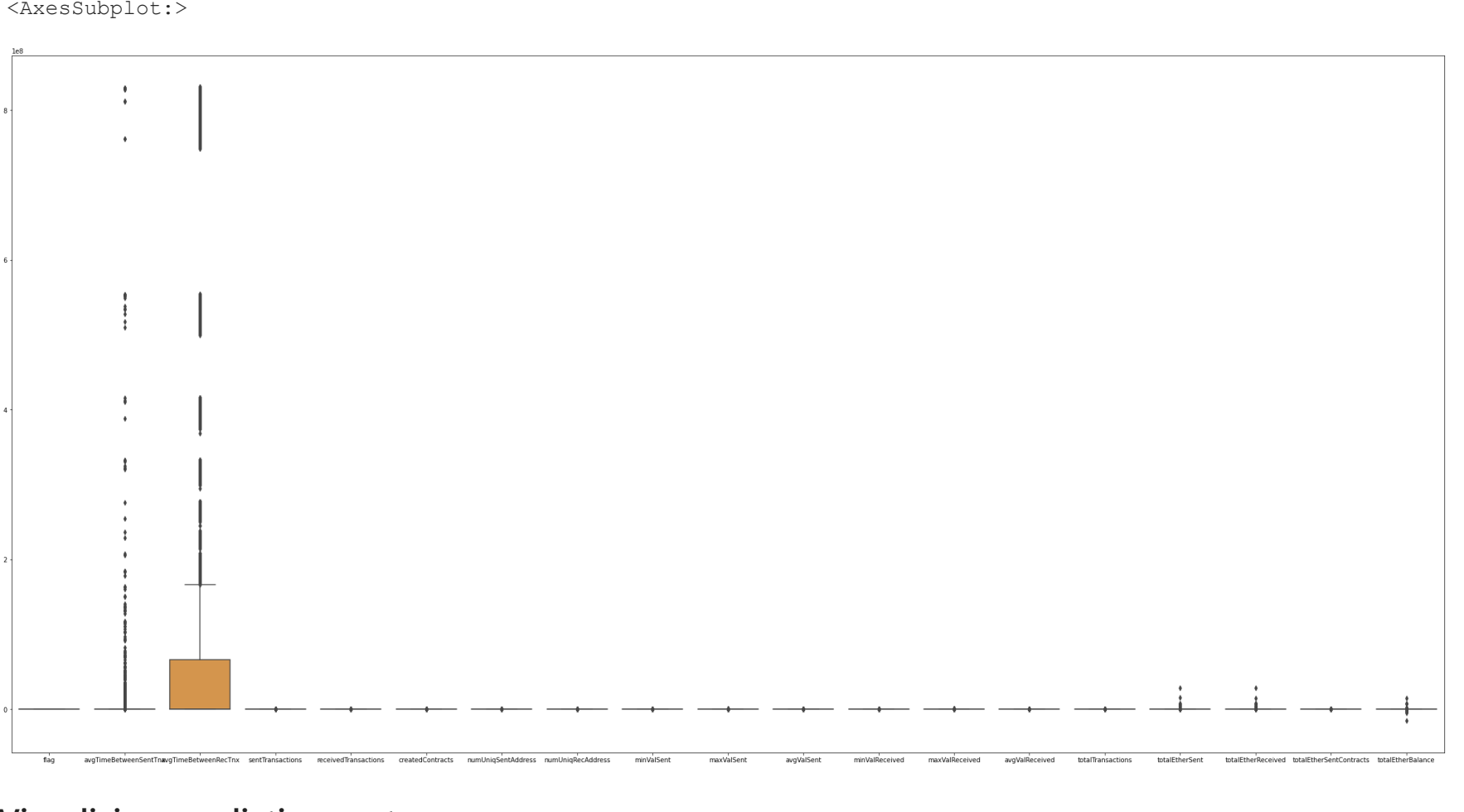
Visualization techniques differ depending on the type of the feature vector (i.e. numerical: continuous or discrete, categorical: ordinal etc). Techniques will also depend on the type of data being dealt with, and the insight that we want to extract from it.

Common visualization techniques include:

- Bar Plots: Visualize the frequency distribution of categorical features.
- Histograms: Visualize the frequency distribution of numerical features.
- Box Plots: Visualize a numerical feature, while providing more information like the median, lower/upper quantiles etc..
- Scatter Plots: Visualize the relationship (usually the correlation) between two features. Can include a goodness of fit line, to serve as a regression plot.

In [ ]:
```
# PLOT FEATURES TO SEE THE OUTLIERS
plt.figure(figsize=(40, 20))
sns.boxplot(data=df)
```

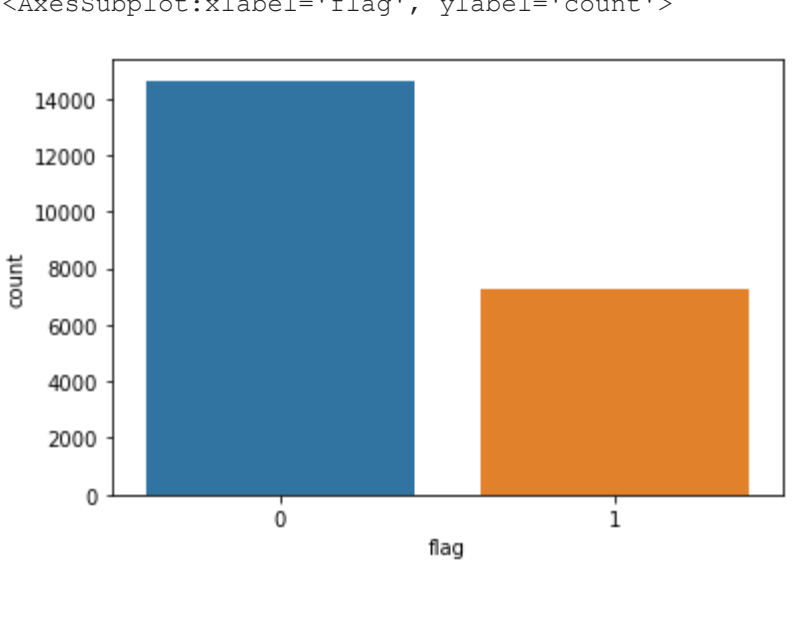
Out[ ]:
```
<AxesSubplot:>
```



Visualizing prediction vector

In [ ]:
```
sns.countplot(x="flag", data=df) # We notice a bias in our dataset where there are more samples with no fraud t
```

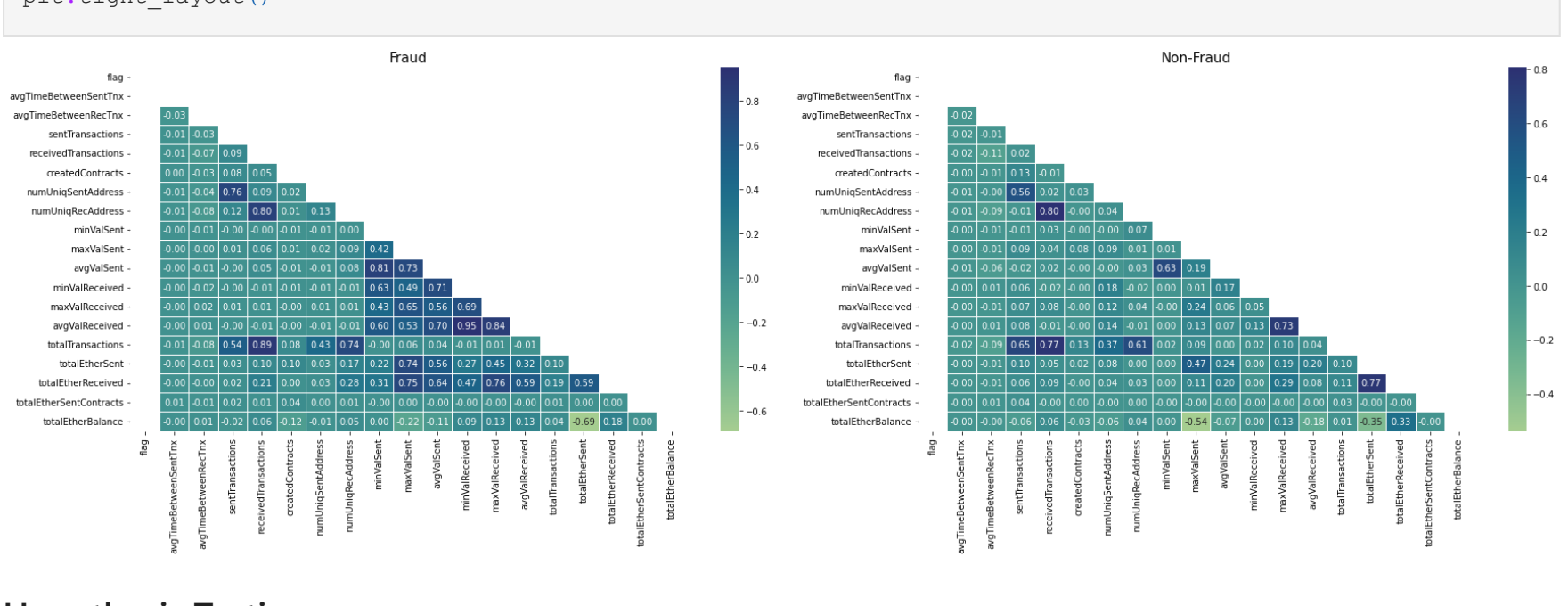
Out[ ]:
```
<AxesSubplot:xlabel='flag', ylabel='count'>
```



Visualizing: Correlation between features under fraud and non-fraud transactions

In [ ]:
```
fraud_df = np.where(df['flag'] == 1)
fraud_df = df.iloc[fraud_df]

non_fraud_df = np.where(df['flag'] == 0)
non_fraud_df = df.iloc[non_fraud_df]
```

In [ ]:
```
fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(25, 8))

mask = np.triu(np.ones_like(fraud_df.corr(), dtype=bool))
ax1.set_title('Fraud', size =15)
sns.heatmap(fraud_df.corr(), annot=True,  linewidth=0.5, fmt=".2f", cmap="crest", mask=mask, ax=ax1)

mask2 = np.triu(np.ones_like(non_fraud_df.corr(), dtype=bool))
ax2.set_title('Non-Fraud', size =15)
sns.heatmap(non_fraud_df.corr(), annot=True,  linewidth=0.5, fmt=".2f", cmap="crest", mask=mask2, ax=ax2)

plt.tight_layout()
```



Hypothesis Testing

In [ ]:
```
def hyp_test(col1, col2, df, alpha):
    fraud_sample = np.array([np.mean(df[df[col1]==1].sample(20)[col2].values) for i in range(500)])
    no_fraud_sample = np.array([np.mean(df[df[col1]==0].sample(20)[col2].values) for i in range(500)])

    effect = np.mean(fraud_sample) - np.mean(no_fraud_sample)
    sigma_first = np.std(fraud_sample)
    sigma_second = np.std(no_fraud_sample)
    sigma_difference = np.sqrt((sigma_first**2)/len(fraud_sample)  +  (sigma_second**2)/len(no_fraud_sample))
    z_score = effect / sigma_difference # getting z-score

    p_value = stats.norm.sf(abs(z_score))*2 # getting p-value

    if p_value > alpha:
        print("We fail to reject Null hypothesis")
    elif p_value <= alpha:
        print("We reject null hypothesis")
```

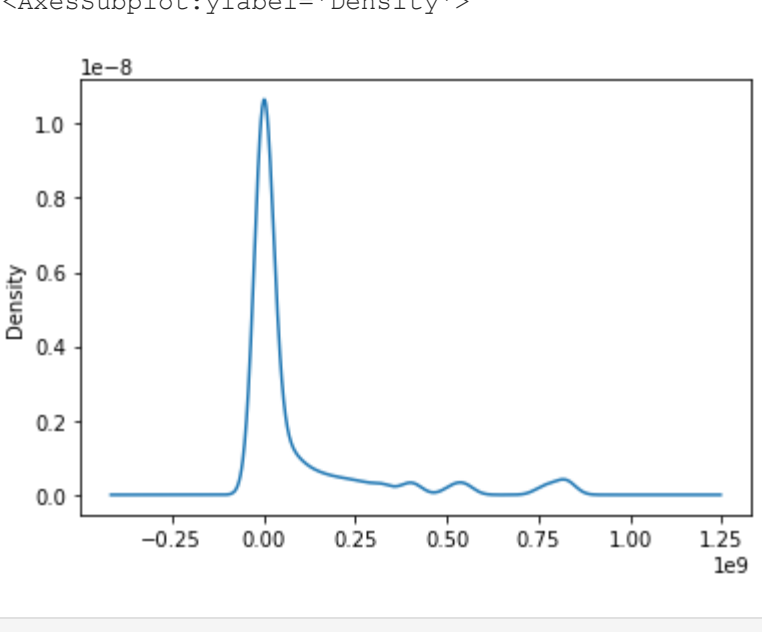
H0: Fradulent transactions are independent of average times between transactions recieved

H1: Fradulent transactions are dependent of average times between transactions recieved

cutoff = 0.05

In [ ]:
```
df['avgTimeBetweenRecTnx'].plot(kind='kde') #Check distribution of average times
```

Out[ ]:
```
<AxesSubplot:ylabel='Density'>
```



In [ ]:
```
alpha = 0.05
hyp_test('flag', 'avgTimeBetweenRecTnx', df, alpha =0.05)
```

```
We reject null hypothesis
```

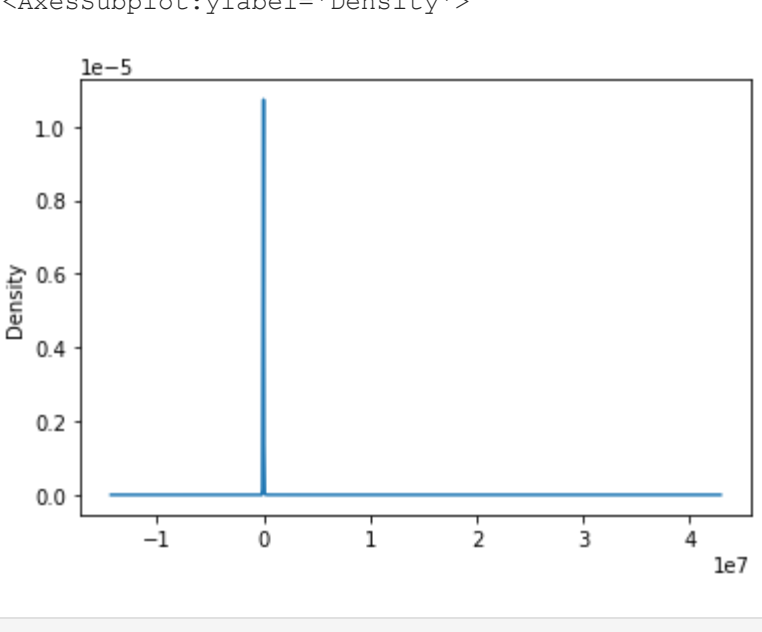
H0: Fradulent transactions are independent of amount of Ether sent

H1: Fradulent transactions are dependent on amount of Ether sent

cutoff = 0.05

In [ ]:
```
df['totalEtherSent'].plot(kind='kde') #Check distribution of total ether sent
```

Out[ ]:
```
<AxesSubplot:ylabel='Density'>
```



In [ ]:
```
alpha = 0.05
hyp_test('flag', 'totalEtherSent', df, alpha =0.05)
```

```
We reject null hypothesis
```

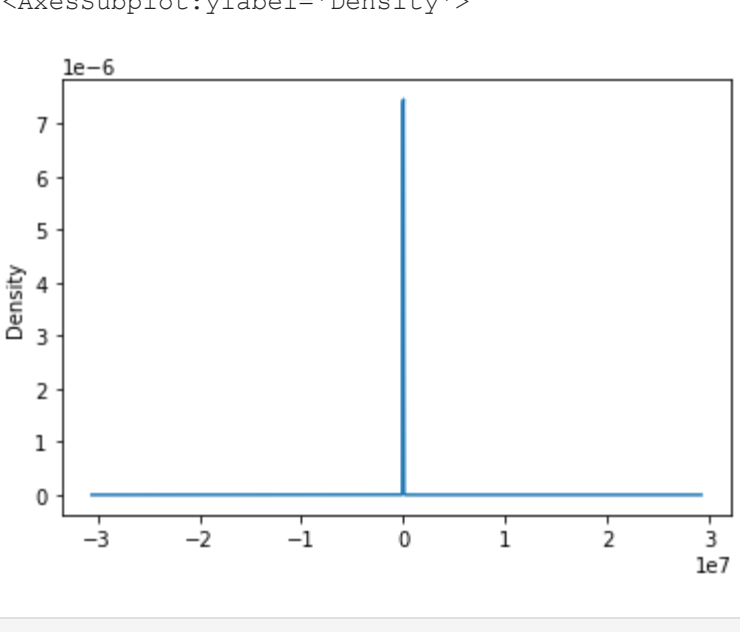
H0: Fradulent transactions are independent of Ether balance

H1: Fradulent transactions are dependent on Ether balance

cutoff = 0.05

In [ ]:
```
df['totalEtherBalance'].plot(kind='kde') #Check distribution of total ether balance
```

Out[ ]:
```
<AxesSubplot:ylabel='Density'>
```



In [ ]:
```
alpha = 0.05
hyp_test('flag', 'totalEtherBalance', df, alpha =0.05)
```

```
We fail to reject Null hypothesis
```

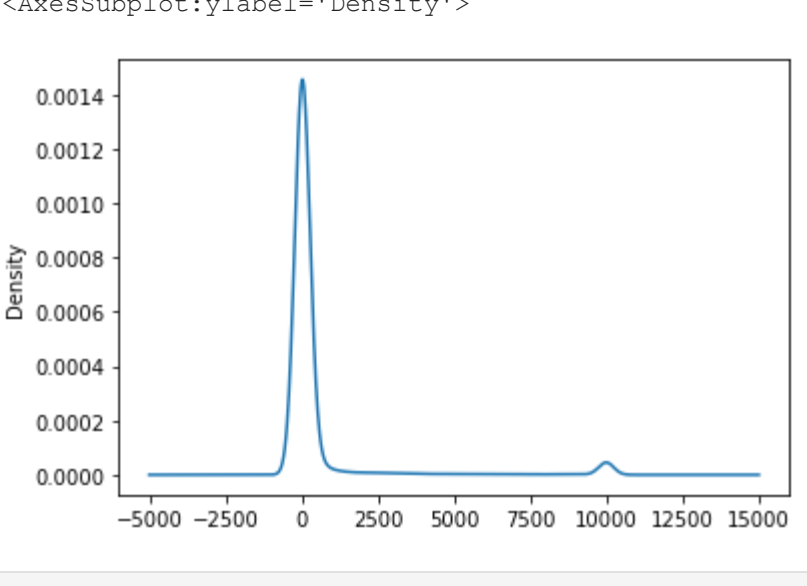
H0: Fradulent transactions are independent of total number of transactions made

H1: Fradulent transactions are dependent on total number of transactions made

cutoff = 0.05

In [ ]:
```
df['receivedTransactions'].plot(kind='kde') #Check distribution of total transactions received
```

Out[ ]:
```
<AxesSubplot:ylabel='Density'>
```



In [ ]:
```
alpha = 0.05
hyp_test('flag', 'receivedTransactions', df, alpha =0.05)
```

```
We reject null hypothesis
```

Outlier Detection

The presence of outliers can often skew results which take into consideration these data points.

In [ ]:
```
df.skew(axis = 0, skipna = True).sort_values() # CHECK THE SKEWNESS OF THE DATA
```

```
C:\Users\rieke\AppData\Local\Temp\ipykernel_3872\3312197560.py:1: FutureWarning: Dropping of nuisance columns i
n DataFrame reductions (with 'numeric_only=None') is deprecated; in a future version this will raise TypeError.
  Select only valid columns before calling the reduction.
  df.skew(axis = 0, skipna = True).sort_values() # CHECK THE SKEWNESS OF THE DATA
```

Out[ ]:
```
totalEtherBalance           -3.218564
flag                         0.719930
avgTimeBetweenRecTnx         2.620069
totalTransactions            3.425968
receivedTransactions         4.673886
sentTransactions             5.568895
numUniqRecAddress            6.753592
numUniqSentAddress          12.528606
avgTimeBetweenSentTnx       21.763417
minValReceived              43.451106
avgValSent                  53.459817
maxValReceived              58.327570
createdContracts            63.004272
maxValSent                  73.548959
totalEtherReceived          83.881310
totalEtherSent              84.067256
avgValReceived              97.952199
minValSent                  98.741249
totalEtherSentContracts    147.452227
dtype: float64
```

Data Transformation and Preparation

Transforming Skewed Continous Features

It is common practice to apply a logarthmic transformation to highly skewed continuous feature distributions. A typical flow for this is in a commented code block below.

In [ ]:
```
# SPLIT DATA TO PREPARE IT FOR THE DATA TRANSFORMATION
X = df.drop('flag',axis=1)
y = df['flag']

df2 = X.copy()
from scipy import stats

# BOX COX DATA TRANSFORMATION TO HANDLE SKEWED DATA(IT TRANSFORMS DATA TO HAVE BETTER DISTRIBUTION)
# makes a better distribution of the data so that outliers have minimal impact on our ml model
# we only transform features that are numerical , not categorical

for col in df2:
  if df2[col].dtypes == 'int64' or df2[col].dtypes == 'float64':
    posdata = df2[col][df2[col] > 0]
    col_data, fitted_lambda = stats.boxcox(posdata)
    df2.loc[df2[col] > 0, col] = col_data
df_final = df2.copy()
```

In [ ]:
```
df_final.skew(axis = 0, skipna = True).sort_values()
```

```
C:\Users\rieke\AppData\Local\Temp\ipykernel_3872\3022671192.py:1: FutureWarning: Dropping of nuisance columns i
n DataFrame reductions (with 'numeric_only=None') is deprecated; in a future version this will raise TypeError.
  Select only valid columns before calling the reduction.
  df_final.skew(axis = 0, skipna = True).sort_values()
```

Out[ ]:
```
totalEtherBalance         -113.841353
totalEtherSentContracts    -32.452553
minValSent                  -0.714737
minValReceived              -0.492611
avgValReceived               0.023142
avgValSent                   0.029651
maxValReceived               0.074183
totalTransactions            0.094538
totalEtherReceived           0.159352
avgTimeBetweenRecTnx         0.171635
receivedTransactions         0.191791
numUniqRecAddress            0.261355
maxValSent                   0.318413
avgTimeBetweenSentTnx        0.337046
sentTransactions             0.466997
totalEtherSent               0.499708
numUniqSentAddress           0.560156
createdContracts             9.983197
dtype: float64
```

We can see that we get less skewed data after applying boxcox, with only 3 features that are still skew compared to 18 skewed features before transforming

Normalizing Numerical Features

Another common practice is to perform some type of scaling on numerical features. Applying scaling doesn't change the shape of each feature's distribution; but ensures that each feature is treated equally when applying supervised learners.

In [ ]:
```
# we standardize the data with StandardScaler we basically just does z = (x-mean)/standard deviation
df_final = df_final.drop('address',axis=1)
standardizer = StandardScaler()
standardized_data = standardizer.fit_transform(df_final)

df_standard = pd.DataFrame(standardized_data, columns=df_final.columns)
```

In [ ]:
```
df_standard.head()
```

Out[ ]:

	avgTimeBetweenSentTnx	avgTimeBetweenRecTnx	sentTransactions	receivedTransactions	createdContracts	numUniqSentAddress	numUniqRe
0	0.990254	1.062077	1.102766	0.481231	-0.09919	1.568607	
1	-1.094791	1.529180	-1.071426	-0.836902	-0.09919	-0.980744	
2	1.708255	1.053293	0.548911	0.501457	-0.09919	1.079560	
3	0.723344	0.762165	1.253017	1.097946	-0.09919	1.608723	
4	-1.094791	1.385448	-1.071426	-0.380741	-0.09919	-0.980744	

Adding Smote data and Splitting data into Train,Val and Test sets

In [ ]:
```
# we will use SMOTE-NC to artificially generate data points to fix the imbalance noticed earlier
# this will only be applied to only the training data
# we will also stratify our y labels because we do have a imbalance

train_ratio = 0.75
validation_ratio = 0.15
test_ratio = 0.10

# train is now 75% of the entire data set
# the _junk suffix means that we drop that variable completely
X_train, X_test, y_train, y_test = train_test_split(df_standard, y, test_size=1 - train_ratio, stratify=y)

# test is now 10% of the initial data set
# validation is now 15% of the initial data set
X_val, X_test, y_val, y_test = train_test_split(X_test, y_test,stratify=y_test, test_size=test_ratio/(test_rati
```

In [ ]:
```
smote_nc = SMOTENC(random_state=42, k_neighbors=2, categorical_features=[0])
X_train_new, y_train_new = smote_nc.fit_resample(X_train,y_train)
```

Model Exploration

We tested our data on multiple models. For each model we print the train data score, and the confusion matrix scores for the test data.

For the case of fraud detection we will focus on the recall, since we want a high recall on true fraudulent transactions and non fraudelent transactions

1.Logistic Regression

In [ ]:
```
clf = LogisticRegression(random_state=0).fit(X_train_new, y_train_new)
print("Accuracy on Train data: ",clf.score(X_train_new, y_train_new))
```

```
Accuracy on Train data:  0.8032166940040095
```

In [ ]:
```
pred = clf.predict(X_val)
print(classification_report(y_val, pred))
```

```
              precision    recall  f1-score   support

           0       0.89      0.81      0.84      2195
           1       0.67      0.79      0.72      1084

    accuracy                           0.80      3279
   macro avg       0.78      0.80      0.78      3279
weighted avg       0.81      0.80      0.80      3279
```

In [ ]:
```
pred = clf.predict(X_test)
print(classification_report(y_test, pred))
```

```
              precision    recall  f1-score   support

           0       0.88      0.81      0.84      1464
           1       0.67      0.78      0.72       723

    accuracy                           0.80      2187
   macro avg       0.78      0.79      0.78      2187
weighted avg       0.81      0.80      0.80      2187
```

2.KNN

In [ ]:
```
neigh = KNeighborsClassifier(n_neighbors=3)
neigh.fit(X_train_new, y_train_new)
print("Accuracy on Train data: ",neigh.score(X_train_new, y_train_new))
```

```
Accuracy on Train data:  0.9532987060324403
```

In [ ]:
```
pred = neigh.predict(X_val)
print(classification_report(y_val, pred))
```

```
              precision    recall  f1-score   support

           0       0.93      0.88      0.90      2195
           1       0.78      0.86      0.82      1084

    accuracy                           0.87      3279
   macro avg       0.85      0.87      0.86      3279
weighted avg       0.88      0.87      0.88      3279
```

In [ ]:
```
pred = neigh.predict(X_test)
print(classification_report(y_test, pred))
```

```
              precision    recall  f1-score   support

           0       0.93      0.90      0.91      1464
           1       0.81      0.86      0.83       723

    accuracy                           0.89      2187
   macro avg       0.87      0.88      0.87      2187
weighted avg       0.89      0.89      0.89      2187
```

3.Naive Bayes

In [ ]:
```
gnb = GaussianNB()
gnb.fit(X_train_new, y_train_new)
print("Accuracy on Train data: ",gnb.score(X_train_new, y_train_new))
```

```
Accuracy on Train data:  0.5744486969199927
```

In [ ]:
```
pred = gnb.predict(X_val)
print(classification_report(y_val, pred))
```

```
              precision    recall  f1-score   support

           0       0.90      0.22      0.35      2195
           1       0.38      0.95      0.54      1084

    accuracy                           0.46      3279
   macro avg       0.64      0.58      0.44      3279
weighted avg       0.73      0.46      0.41      3279
```

In [ ]:
```
pred = gnb.predict(X_test)
print(classification_report(y_test, pred))
```

```
              precision    recall  f1-score   support

           0       0.90      0.20      0.33      1464
           1       0.37      0.95      0.53       723

    accuracy                           0.45      2187
   macro avg       0.63      0.58      0.43      2187
weighted avg       0.72      0.45      0.40      2187
```

4.SVM

In [ ]:
```
svm = make_pipeline(SVC(gamma='auto'))
svm.fit(X_train_new, y_train_new)
print("Accuracy on Train data: ",svm.score(X_train_new, y_train_new))
```

```
Accuracy on Train data:  0.8863677783852743
```

In [ ]:
```
pred = svm.predict(X_val)
print(classification_report(y_val, pred))
```

```
              precision    recall  f1-score   support

           0       0.94      0.86      0.90      2195
           1       0.76      0.90      0.82      1084

    accuracy                           0.87      3279
   macro avg       0.85      0.88      0.86      3279
weighted avg       0.88      0.87      0.87      3279
```

In [ ]:
```
pred = svm.predict(X_test)
print(classification_report(y_test, pred))
```

```
              precision    recall  f1-score   support

           0       0.93      0.86      0.90      1464
           1       0.76      0.88      0.82       723

    accuracy                           0.87      2187
   macro avg       0.85      0.87      0.86      2187
weighted avg       0.88      0.87      0.87      2187
```

5.Decision Tree

In [ ]:
```
dt = tree.DecisionTreeClassifier(random_state=0)
dt.fit(X_train_new, y_train_new)
print("Accuracy on Train data: ",dt.score(X_train_new, y_train_new))
```

```
Accuracy on Train data:  0.998587570621469
```

In [ ]:
```
pred = dt.predict(X_val)
print(classification_report(y_val, pred))
```

```
              precision    recall  f1-score   support

           0       0.93      0.90      0.91      2195
           1       0.81      0.86      0.83      1084

    accuracy                           0.89      3279
   macro avg       0.87      0.88      0.87      3279
weighted avg       0.89      0.89      0.89      3279
```

```python
pred = dt.predict(X_test)
print(classification_report(y_test, pred))
```

```
              precision    recall  f1-score   support

           0       0.92      0.90      0.91      1464
           1       0.80      0.84      0.82       723

    accuracy                           0.88      2187
   macro avg       0.86      0.87      0.86      2187
weighted avg       0.88      0.88      0.88      2187
```

6.XGBoost

```python
xgb_cl = xgb.XGBClassifier()

xgb_cl.fit(X_train_new, y_train_new)
print("Accuracy on Train data: ",xgb_cl.score(X_train_new, y_train_new))
```

```
Accuracy on Train data:  0.981957353745216
```

```python
pred = xgb_cl.predict(X_val)
print(classification_report(y_val, pred))
```

```
              precision    recall  f1-score   support

           0       0.96      0.94      0.95      2195
           1       0.88      0.91      0.89      1084

    accuracy                           0.93      3279
   macro avg       0.92      0.92      0.92      3279
weighted avg       0.93      0.93      0.93      3279
```

```python
pred = xgb_cl.predict(X_test)
print(classification_report(y_test, pred))
```

```
              precision    recall  f1-score   support

           0       0.96      0.94      0.95      1464
           1       0.89      0.91      0.90       723

    accuracy                           0.93      2187
   macro avg       0.92      0.93      0.93      2187
weighted avg       0.93      0.93      0.93      2187
```

7.Random Forest Tree

```python
rf = RandomForestClassifier(n_estimators = 100, random_state=0)
rf.fit(X_train_new, y_train_new)
print("Accuracy on Train data: ",rf.score(X_train_new, y_train_new))
```

```
Accuracy on Train data:  0.998587570621469
```

```python
pred = rf.predict(X_val)
print(classification_report(y_val, pred))
```

```
              precision    recall  f1-score   support

           0       0.94      0.94      0.94      2195
           1       0.87      0.89      0.88      1084

    accuracy                           0.92      3279
   macro avg       0.91      0.91      0.91      3279
weighted avg       0.92      0.92      0.92      3279
```

```python
pred = rf.predict(X_test)
print(classification_report(y_test, pred))
```

```
              precision    recall  f1-score   support

           0       0.94      0.94      0.94      1464
           1       0.88      0.88      0.88       723

    accuracy                           0.92      2187
   macro avg       0.91      0.91      0.91      2187
weighted avg       0.92      0.92      0.92      2187
```

Final Model Building(Hyperparamter Tuning)

From the results above we saw that the XGBoost model gave us the highest f1 and recall, we will now do some paramter tuning and then make final conclusions which parameters should be used based on the results for future detection.

XGBoost

```python
params={
"learning_rate" : [0.05, 0.10, 0.20, 0.30 ] ,
"max_depth" : [ 3, 5, 8, 10, 15],
"min_child_weight" : [ 1, 3, 5, 7 ],
"gamma" : [ 0.0, 0.1, 0.2 , 0.3, 0.4 ],
"colsample_bytree" : [ 0.3, 0.4, 0.5 , 0.7 ]
}
classifier = XGBClassifier()
grid_search= GridSearchCV(estimator = classifier,param_grid=params,scoring='f1',n_jobs=-1,cv=3, verbose=1)
grid_search.fit(X_train_new,y_train_new.ravel())
grid_search.best_params_
```

```
Fitting 3 folds for each of 3200 candidates, totalling 9600 fits
{'colsample_bytree': 0.5,
 'gamma': 0.2,
 'learning_rate': 0.2,
 'max_depth': 15,
 'min_child_weight': 1,
 'n_estimators': 100}
```

```python
accuracy = grid_search.best_score_
accuracy
```

```
0.9475196230056412
```

```python
classifier2 = XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=0.5, gamma=0.2, gpu_id=-1,
              importance_type='gain', interaction_constraints='',
              learning_rate=0.2, max_delta_step=0, max_depth=15,
              min_child_weight=1, missing=1, monotone_constraints='()',
              n_estimators=100, n_jobs=0, num_parallel_tree=1, random_state=0,
              reg_alpha=0, reg_lambda=1, scale_pos_weight=1, subsample=1,
              tree_method='exact', validate_parameters=1, verbosity=None)
classifier2.fit(X_train_new,y_train_new)
pred_xgbf = classifier2.predict(X_val)
```

Validation Set

```python
#Checking different metrics for decision tree model after tuning the hyperparameters
print('Checking different metrics for decision tree model after tuning the hyperparameters:\n')
print("Training accuracy: ",classifier2.score(X_val,y_val))
acc_score = accuracy_score(y_val, pred_xgbf)
print('Testing accuracy: ',acc_score)
conf_mat = confusion_matrix(y_val, pred_xgbf)
print('Confusion Matrix: \n',conf_mat)
roc_auc = roc_auc_score(y_val,pred_xgbf)
print('ROC AUC score: ',roc_auc)
class_rep2 = classification_report(y_val,pred_xgbf)
print('Classification Report: \n',class_rep2)
```

```
Checking different metrics for decision tree model after tuning the hyperparameters:

Training accuracy:  0.9365660262275084
Testing accuracy:  0.9365660262275084
Confusion Matrix:
 [[2079  116]
 [  92  992]]
ROC AUC score:  0.931140885440745
Classification Report:
               precision    recall  f1-score   support

           0       0.96      0.95      0.95      2195
           1       0.90      0.92      0.91      1084

    accuracy                           0.94      3279
   macro avg       0.93      0.93      0.93      3279
weighted avg       0.94      0.94      0.94      3279
```

Test Set

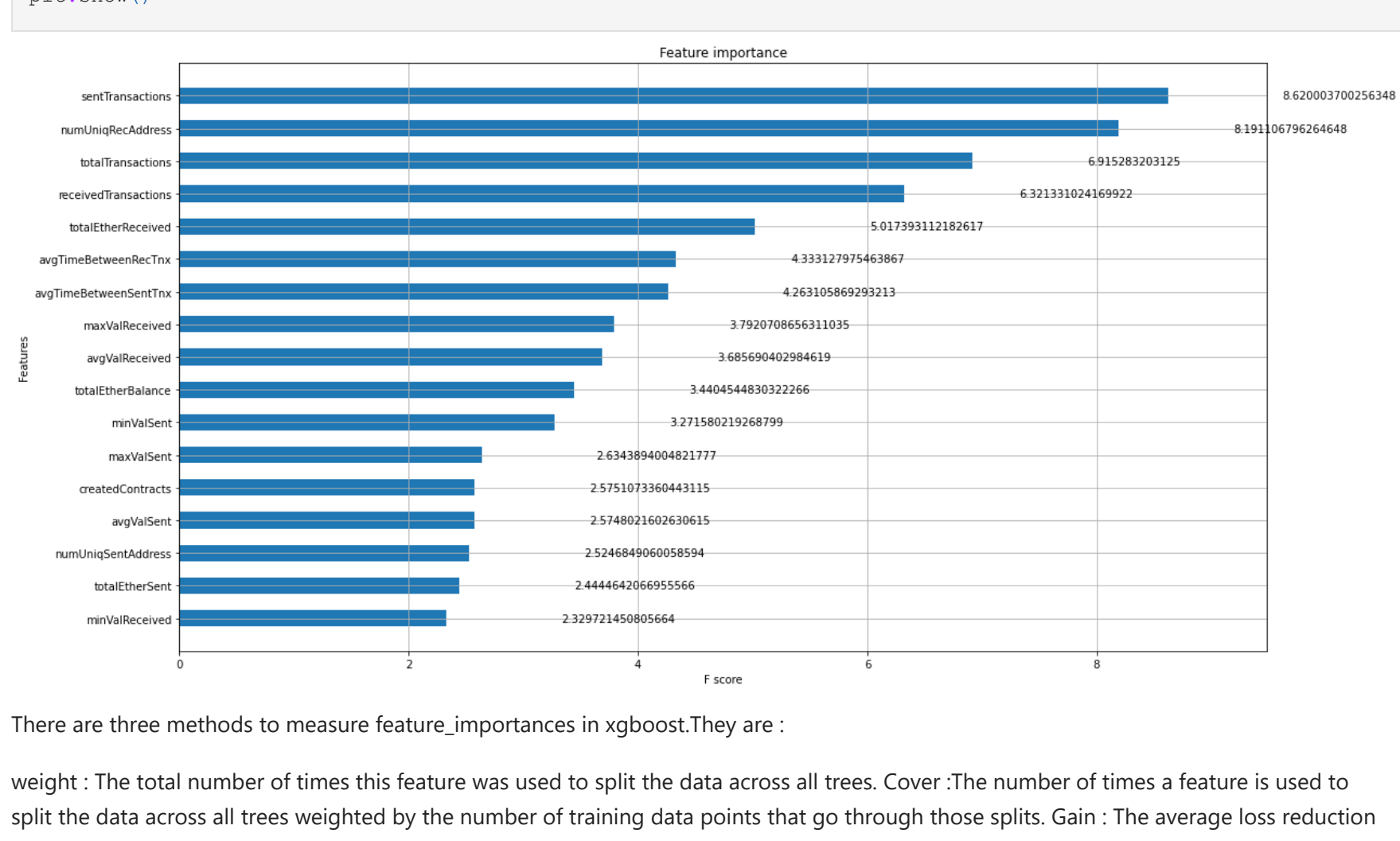
```python
pred_xgbf = classifier2.predict(X_test)

#Checking different metrics for decision tree model after tuning the hyperparameters
print('Checking different metrics for decision tree model after tuning the hyperparameters:\n')
print("Training accuracy: ",classifier2.score(X_test,y_test))
acc_score = accuracy_score(y_test, pred_xgbf)
print('Testing accuracy: ',acc_score)
conf_mat = confusion_matrix(y_test, pred_xgbf)
print('Confusion Matrix: \n',conf_mat)
roc_auc = roc_auc_score(y_test,pred_xgbf)
print('ROC AUC score: ',roc_auc)
class_rep2 = classification_report(y_test,pred_xgbf)
print('Classification Report: \n',class_rep2)
```

```
Checking different metrics for decision tree model after tuning the hyperparameters:

Training accuracy:  0.9350708733424783
Testing accuracy:  0.9350708733424783
Confusion Matrix:
 [[1387   77]
 [  65  658]]
ROC AUC score:  0.9287505951976055
Classification Report:
               precision    recall  f1-score   support

           0       0.96      0.95      0.95      1464
           1       0.90      0.91      0.90       723

    accuracy                           0.94      2187
   macro avg       0.93      0.93      0.93      2187
weighted avg       0.94      0.94      0.94      2187
```

Feature Importance and Selection

Feature Importance

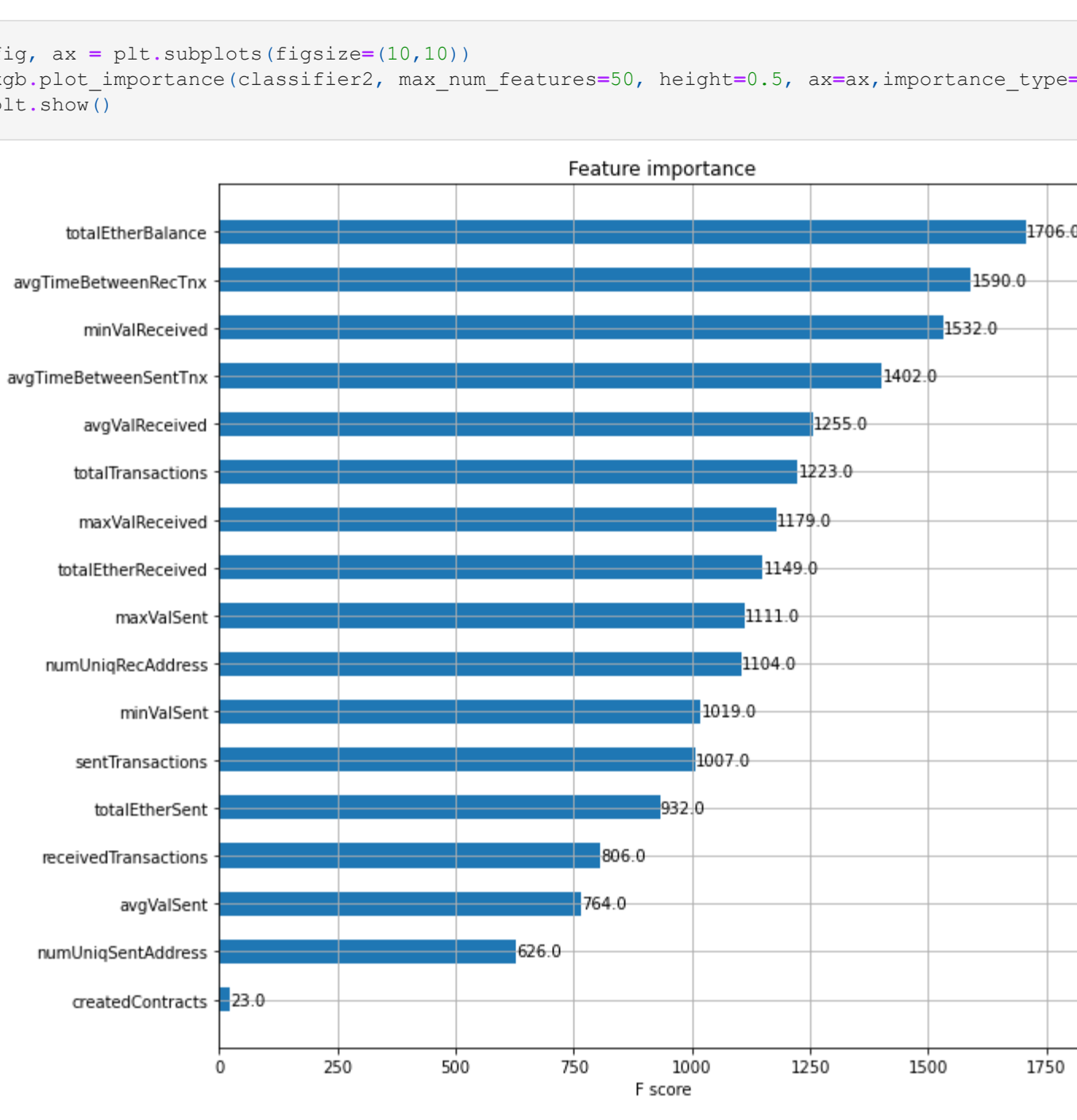
```python
fig, ax = plt.subplots(figsize=(18,10))
xgb.plot_importance(classifier2, max_num_features=50, height=0.5, ax=ax,importance_type='gain')
plt.show()
```



There are three methods to measure feature_importances in xgboost.They are :

weight : The total number of times this feature was used to split the data across all trees. Cover :The number of times a feature is used to split the data across all trees weighted by the number of training data points that go through those splits. Gain : The average loss reduction gained when using this feature for splitting in trees. We used Gain in the above example and the model says when it used sentTransactions, the loss on average was reduced by 8.6%.

```python
fig, ax = plt.subplots(figsize=(10,10))
xgb.plot_importance(classifier2, max_num_features=50, height=0.5, ax=ax,importance_type='weight')
plt.show()
```



When we considered weight,the model says that is used totalEtherBalance 1706 times to split the data across the trees.

Feature Selection

```python
model = XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=0.5, gamma=0.2, gpu_id=-1,
              importance_type='gain', interaction_constraints='',
              learning_rate=0.2, max_delta_step=0, max_depth=15,
              min_child_weight=1, missing=1, monotone_constraints='()',
              n_estimators=100, n_jobs=0, num_parallel_tree=1, random_state=0,
              reg_alpha=0, reg_lambda=1, scale_pos_weight=1, subsample=1,
              tree_method='exact', validate_parameters=1, verbosity=None)
model.fit(X_train_new,y_train_new)
```

```
XGBClassifier

XGBClassifier(base_score=0.5, booster='gbtree', callbacks=None,
              colsample_bylevel=1, colsample_bynode=1, colsample_bytree=0.5,
              early_stopping_rounds=None, enable_categorical=False,
              eval_metric=None, gamma=0.2, gpu_id=-1, grow_policy='depthwise',
              importance_type='gain', interaction_constraints='',
              learning_rate=0.2, max_bin=256, max_cat_to_onehot=4,
              max_delta_step=0, max_depth=15, max_leaves=0, min_child_weight=1,
              missing=1, monotone_constraints='()', n_estimators=100, n_jobs=0,
              num_parallel_tree=1, predictor='auto', random_state=0,
              reg_alpha=0, reg_lambda=1, ...)
```

```python
thresholds = sort(model.feature_importances_)
for thresh in thresholds:
    # select features using threshold
    selection = SelectFromModel(model, threshold=thresh, prefit=True)
    select_X_train = selection.transform(X_train_new)
    # train model
    selection_model = XGBClassifier()
    selection_model.fit(select_X_train, y_train_new)
    # eval model
    select_X_test = selection.transform(X_test)
    y_pred = selection_model.predict(select_X_test)
    predictions = [round(value) for value in y_pred]
    accuracy = accuracy_score(y_test, predictions)
    print("Thresh=%.3f, n=%d, Accuracy: %.2f%%" % (thresh, select_X_train.shape[1], accuracy*100.0))
```

```
Thresh=0.000, n=18, Accuracy: 93.42%
Thresh=0.032, n=17, Accuracy: 93.42%
Thresh=0.034, n=16, Accuracy: 93.46%
Thresh=0.035, n=15, Accuracy: 93.37%
Thresh=0.035, n=14, Accuracy: 93.05%
Thresh=0.035, n=13, Accuracy: 93.37%
Thresh=0.036, n=12, Accuracy: 92.59%
Thresh=0.045, n=11, Accuracy: 93.28%
Thresh=0.047, n=10, Accuracy: 92.55%
Thresh=0.051, n=9, Accuracy: 91.18%
Thresh=0.052, n=8, Accuracy: 90.81%
Thresh=0.058, n=7, Accuracy: 91.40%
Thresh=0.059, n=6, Accuracy: 90.53%
Thresh=0.069, n=5, Accuracy: 86.51%
Thresh=0.087, n=4, Accuracy: 83.31%
Thresh=0.095, n=3, Accuracy: 82.90%
Thresh=0.112, n=2, Accuracy: 77.37%
Thresh=0.118, n=1, Accuracy: 63.05%
```

From the results above, if we want a balanced performance, accuracy tradeoff, we would go with only using n=11 with a treshold of 0.045 on the feature importances.

For future work we suggest at looking to combine multiple models with each other, or even using a model for each subsets of the datasets(one model for demographics, another model for transactions etc.) Other complex models like anomoly detection neural networks can also be a godd recommendation.