TABLE OF CONTENTS

Click to jump to matching Markdown Header.

INTRODUCTION

Cyrptocurrencies have grown in popularity and market capitalization as corporations, retail investors, and individual adopt them as means of transactions and as a store of value. Bitcoin, introduced to the world in 2008 was the first cryptocurrency and still the largest as a percentage of market share. However, Ethereum and a host of other coins have been making their way into the mainstream.

One of the primary concerns that regulators and investors hold is their association with malicious activity. Because transactions are anonymous and decentralized, they became useful for purchasing goods and services on dark web markets like Silk Road. With anonymous transactions came fraud. Since users can't verify who is on the other end of the transaction, people are regularly defrauded. For widespread adoption, this poses a major concern.

Crypto fraud usually presents itself in a couple of ways. First, when a user makes a transaction on an exchange, they must connect their wallet. If the exchange is hacked, all of the wallets are at risk. This is what happened with Mt. Gox, the largest cyrpto hack of all time. This hack involved 850,000 BTC valued today at ~ 45 billion dollars. Crypto fraud also presents itself in ficticious projects. Founders design projects and raise funds from investors. Many times, these projects end up being fake and the founders route money into their own wallets. It is also critical to note that once a transaction is sent, it cannot be reversed. Transactions are immutable. Unlike when your credit card gets hacked and the bank can provide liability coverage, there is no centralized company backing your ethereum to return your money. This makes security a top priority for crypto users

The model I have designed adds a safeguard for transferring ethereum and ERC20 tokens. If the model believes the transaction may be fraudelent, it will warn the user before they confirm the transaction. This is similar to your bank sending out a warning that your credit card may have been used for unverified transactions. Ideally, this service would be built into an existing wallet provider.

ERC20 tokens are tokens that trade on the Ethereum blockchain and follow the ERC20 standard created by Ethereum developers. They are essentially applications that have a currency tied to them which follows a specified template.

- https://en.wikipedia.org/wiki/Mt._Gox (https://en.wikipedia.org/wiki/Mt._Gox)
- https://www.forbes.com/sites/jeffkauflin/2018/10/29/where-did-the-money-go-inside-the-big-crypto-icos-of-2017/?sh=7a1267cb261b (https://www.forbes.com/sites/jeffkauflin/2018/10/29/where-did-the-money-go-inside-the-big-crypto-icos-of-2017/?sh=7a1267cb261b)
- https://ethereum.org/en/developers/docs/standards/tokens/erc-20/ (https://ethereum.org/en/developers/docs/standards/tokens/erc-20/)

OBTAIN

In [1]:
```
# Load in the data from Kaggle
# https://www.kaggle.com/vagifa/ethereum-frauddetection-dataset

import pandas as pd
pd.set_option('display.max_columns', None)

df = pd.read_csv('transaction_dataset.csv')
```

executed in 1.14s, finished 13:52:21 2021-05-23

In [2]:
```
# Make sure it was loaded correctly

display(df.shape)
df.head()
```

executed in 28ms, finished 13:52:21 2021-05-23

```
(9841, 51)
```

Out[2]:

	Unnamed: 0	Index	Address	FLAG	Avg min between sent tnx	Avg min between received tnx	la
0	0	1	0x00009277775ac7d0d59eaad8fee3d10ac6c805e8	0	844.26	1093.71	7
1	1	2	0x0002b44ddb1476db43c868bd494422ee4c136fed	0	12709.07	2958.44	12
2	2	3	0x0002bda54cb772d040f779e88eb453cac0daa244	0	246194.54	2434.02	5
3	3	4	0x00038e6ba2fd5c09aedb96697c8d7b8fa6632e5e	0	10219.60	15785.09	3
4	4	5	0x00062d1dd1afb6fb02540ddad9cdebfe568e0d89	0	36.61	10707.77	3

Here is a description of the columns of the dataset:

- Index: the index number of a row
- Address: the address of the ethereum account
- FLAG: whether the transaction is fraud or not
- Avg min between sent tnx: Average time between sent transactions for account in minutes
- Avgminbetweenreceivedtnx: Average time between received transactions for account in minutes
- TimeDiffbetweenfirstand_last(Mins): Time difference between the first and last transaction
- Sent_tnx: Total number of sent normal transactions
- Received_tnx: Total number of received normal transactions
- NumberofCreated_Contracts: Total Number of created contract transactions
- UniqueReceivedFrom_Addresses: Total Unique addresses from which account received transactions
- UniqueSentTo_Addresses20: Total Unique addresses from which account sent transactions
- MinValueReceived: Minimum value in Ether ever received
- MaxValueReceived: Maximum value in Ether ever received
- AvgValueReceived5Average value in Ether ever received
- MinValSent: Minimum value of Ether ever sent
- MaxValSent: Maximum value of Ether ever sent
- AvgValSent: Average value of Ether ever sent
- MinValueSentToContract: Minimum value of Ether sent to a contract
- MaxValueSentToContract: Maximum value of Ether sent to a contract
- AvgValueSentToContract: Average value of Ether sent to contracts
- TotalTransactions(IncludingTnxtoCreate_Contract): Total number of transactions
- TotalEtherSent:Total Ether sent for account address
- TotalEtherReceived: Total Ether received for account address
- TotalEtherSent_Contracts: Total Ether sent to Contract addresses

- TotalEtherBalance: Total Ether Balance following enacted transactions
- TotalERC20Tnxs: Total number of ERC20 token transfer transactions
- ERC20TotalEther_Received: Total ERC20 token received transactions in Ether
- ERC20TotalEther_Sent: Total ERC20token sent transactions in Ether
- ERC20TotalEtherSentContract: Total ERC20 token transfer to other contracts in Ether
- ERC20UniqSent_Addr: Number of ERC20 token transactions sent to Unique account addresses
- ERC20UniqRec_Addr: Number of ERC20 token transactions received from Unique addresses
- ERC20UniqRecContractAddr: Number of ERC20token transactions received from Unique contract addresses
- ERC20AvgTimeBetweenSent_Tnx: Average time between ERC20 token sent transactions in minutes
- ERC20AvgTimeBetweenRec_Tnx: Average time between ERC20 token received transactions in minutes
- ERC20AvgTimeBetweenContract_Tnx: Average time ERC20 token between sent token transactions
- ERC20MinVal_Rec: Minimum value in Ether received from ERC20 token transactions for account
- ERC20MaxVal_Rec: Maximum value in Ether received from ERC20 token transactions for account
- ERC20AvgVal_Rec: Average value in Ether received from ERC20 token transactions for account
- ERC20MinVal_Sent: Minimum value in Ether sent from ERC20 token transactions for account
- ERC20MaxVal_Sent: Maximum value in Ether sent from ERC20 token transactions for account
- ERC20AvgVal_Sent: Average value in Ether sent from ERC20 token transactions for account
- ERC20UniqSentTokenName: Number of Unique ERC20 tokens transferred
- ERC20UniqRecTokenName: Number of Unique ERC20 tokens received
- ERC20MostSentTokenType: Most sent token for account via ERC20 transaction
- ERC20MostRecTokenType: Most received token for account via ERC20 transactions

Fraud

- Interpreting fraud as the wallet's owner has reported a fraudulent transaction at least once

In [3]:
```
# Replace spaces with underscores on column names to make it more 'pyth

df.columns = df.columns.str.lstrip()
df.columns = df.columns.str.replace(' ', '_')
```

executed in 3ms, finished 13:52:21 2021-05-23

In [4]:
```
1 df.columns
```
executed in 4ms, finished 13:52:21 2021-05-23

Out[4]:
```
Index(['Unnamed:_0', 'Index', 'Address', 'FLAG', 'Avg_min_between_sent_tn
x',
       'Avg_min_between_received_tnx',
       'Time_Diff_between_first_and_last_(Mins)', 'Sent_tnx', 'Received_T
nx',
       'Number_of_Created_Contracts', 'Unique_Received_From_Addresses',
       'Unique_Sent_To_Addresses', 'min_value_received', 'max_value_recei
ved_',
       'avg_val_received', 'min_val_sent', 'max_val_sent', 'avg_val_sen
t',
       'min_value_sent_to_contract', 'max_val_sent_to_contract',
       'avg_value_sent_to_contract',
       'total_transactions_(including_tnx_to_create_contract',
       'total_Ether_sent', 'total_ether_received',
       'total_ether_sent_contracts', 'total_ether_balance', '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_tn
x',
       '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'],
      dtype='object')
```

In [5]:
```
1 df.head()
```
executed in 23ms, finished 13:52:21 2021-05-23

Out[5]:

	Unnamed:_0	Index	Address	FLAG	Avg_min_between_sent
0	0	1	0x00009277775ac7d0d59eaad8fee3d10ac6c805e8	0	84
1	1	2	0x0002b44ddb1476db43c868bd494422ee4c136fed	0	1270
2	2	3	0x0002bda54cb772d040f779e88eb453cac0daa244	0	24619
3	3	4	0x00038e6ba2fd5c09aedb96697c8d7b8fa6632e5e	0	1021
4	4	5	0x00062d1dd1afb6fb02540ddad9cdebfe568e0d89	0	3

In [6]:
```
# Check for columns that have a low number of unique values

df.nunique()
```

executed in 18ms, finished 13:52:21 2021-05-23

Out[6]:
```
Unnamed:_0                                                  9841
Index                                                       4729
Address                                                     9816
FLAG                                                           2
Avg_min_between_sent_tnx                                    5013
Avg_min_between_received_tnx                                6223
Time_Diff_between_first_and_last_(Mins)                     7810
Sent_tnx                                                     641
Received_Tnx                                                 727
Number_of_Created_Contracts                                   20
Unique_Received_From_Addresses                               256
Unique_Sent_To_Addresses                                     258
min_value_received                                          4589
max_value_received_                                         6302
avg_val_received                                            6767
min_val_sent                                                4719
max_val_sent                                                6647
avg_val_sent                                                5854
min_value_sent_to_contract                                     3
max_val_sent_to_contract                                       4
avg_value_sent_to_contract                                     4
total_transactions_(including_tnx_to_create_contract         897
total_Ether_sent                                            5868
total_ether_received                                        6728
total_ether_sent_contracts                                     4
total_ether_balance                                         5717
Total_ERC20_tnxs                                             300
ERC20_total_Ether_received                                  3460
ERC20_total_ether_sent                                      1415
ERC20_total_Ether_sent_contract                               29
ERC20_uniq_sent_addr                                         107
ERC20_uniq_rec_addr                                          147
ERC20_uniq_sent_addr.1                                         4
ERC20_uniq_rec_contract_addr                                 123
ERC20_avg_time_between_sent_tnx                                1
ERC20_avg_time_between_rec_tnx                                 1
ERC20_avg_time_between_rec_2_tnx                               1
ERC20_avg_time_between_contract_tnx                            1
ERC20_min_val_rec                                           1276
ERC20_max_val_rec                                           2647
ERC20_avg_val_rec                                           3380
ERC20_min_val_sent                                           476
ERC20_max_val_sent                                          1130
ERC20_avg_val_sent                                          1309
ERC20_min_val_sent_contract                                    1
ERC20_max_val_sent_contract                                    1
ERC20_avg_val_sent_contract                                    1
ERC20_uniq_sent_token_name                                    70
ERC20_uniq_rec_token_name                                    121
ERC20_most_sent_token_type                                   305
ERC20_most_rec_token_type                                    467
dtype: int64
```

- Columns with a low number of values may have low variance which makes them less useful for classifcation models

In [7]:
```
# Drop unnecessary columns
# Drop contract related columns because we are focused on detecting fra

cols_to_drop = ['Unnamed:_0', 'Index', 'ERC20_uniq_sent_addr.1', 'ERC20
                'Number_of_Created_Contracts', 'min_value_sent_to_contra
                'avg_value_sent_to_contract', 'total_ether_sent_contract
                'ERC20_uniq_sent_addr.1', 'ERC20_avg_time_between_sent_t
                'ERC20_avg_time_between_rec_2_tnx', 'ERC20_avg_time_betw
                'ERC20_min_val_sent_contract','ERC20_max_val_sent_contra
                ]
df = df.drop(columns=cols_to_drop, axis=1)
```

executed in 4ms, finished 13:52:21 2021-05-23

- Dropping columns that are accidentally repeated
- Dropping columns that only have 1 value but should have more. Attributing this to errors during data collection (continuous numeric data should not only have 1 columns)
- Columns that only contain one variable and are uniform accross the data do not add information for modeling

In [8]:
```
# Clean up column names by removing parenthesis

df = df.rename(columns={'total_transactions_(including_tnx_to_create_co
```

executed in 3ms, finished 13:52:21 2021-05-23

SCRUB

Check for duplicate values

In [9]:
```
# Check for complete duplicates

df.duplicated().sum()
```

executed in 18ms, finished 13:52:21 2021-05-23

Out[9]: 18

In [10]:
```
1 df[df.duplicated(keep=False)]
```

executed in 48ms, finished 13:52:21 2021-05-23

Out[10]:

	Address	FLAG	Avg_min_between_sent_tnx	Avg_min_be
2908	0x4c13f6966dc24c92489344f0fd6f0e61f3489b84	0	5980.35	
2909	0x4c1da8781f6ca312bc11217b3f61e5dfdf428de1	0	7042.64	
2911	0x4c268c7b1d51b369153d6f1f28c61b15f0e17746	0	0.00	
2912	0x4c26a3c12a64f33a3546fbb206c5365ce8e82c20	0	0.00	
2914	0x4c27438a77738153f6cf3ed890b2817d52ebf584	0	9695.78	
2915	0x4c391cc032c9107b596267610a05262c90fc2df7	0	161.24	
2916	0x4c4a03e100b4b104355edc4c50ce12b9a2879547	0	6783.50	
2917	0x4c4da560350e302232a184c8fa16b126a772c326	0	0.00	
2918	0x4c5b0709f66719861e7277c9dcda9175deb3d866	0	2.32	
2922	0x4c7f6d5b287054bf41f9d49ea8d2ca6e6837850b	0	4.72	
2923	0x4c97ccdaa61ca167e8a3b4d425ae6fbf16bcb39c	0	108.00	
2925	0x4cb6f8f060365dd2e4eb949609dd6293bb950ae2	0	22.27	
2927	0x4cbfd5b6b8d69c12fab524a01f5283fcc75d5bc9	0	4432.48	
2928	0x4cc930a91865e30feff38465ace57711e3923881	0	1165.80	
2929	0x4cce03d1fd8aa1d6172cec74d75bd1df9e6a8ac2	0	0.00	
2930	0x4cd3bb2110eda1805dc63abc1959a5ee2d386e9f	0	0.00	
2931	0x4cd526aa2db72eb1fd557b37c6b0394acd35b212	0	3.38	
2932	0x4cdf46740d51a0f6735fe3f1d28fadf00c20cd34	0	3.92	
2933	0x4c13f6966dc24c92489344f0fd6f0e61f3489b84	0	5980.35	
2934	0x4c1da8781f6ca312bc11217b3f61e5dfdf428de1	0	7042.64	
2936	0x4c268c7b1d51b369153d6f1f28c61b15f0e17746	0	0.00	
2937	0x4c26a3c12a64f33a3546fbb206c5365ce8e82c20	0	0.00	
2939	0x4c27438a77738153f6cf3ed890b2817d52ebf584	0	9695.78	
2940	0x4c391cc032c9107b596267610a05262c90fc2df7	0	161.24	
2941	0x4c4a03e100b4b104355edc4c50ce12b9a2879547	0	6783.50	
2942	0x4c4da560350e302232a184c8fa16b126a772c326	0	0.00	
2943	0x4c5b0709f66719861e7277c9dcda9175deb3d866	0	2.32	
2947	0x4c7f6d5b287054bf41f9d49ea8d2ca6e6837850b	0	4.72	
2948	0x4c97ccdaa61ca167e8a3b4d425ae6fbf16bcb39c	0	108.00	
2950	0x4cb6f8f060365dd2e4eb949609dd6293bb950ae2	0	22.27	
2952	0x4cbfd5b6b8d69c12fab524a01f5283fcc75d5bc9	0	4432.48	
2953	0x4cc930a91865e30feff38465ace57711e3923881	0	1165.80	

	Address	FLAG	Avg_min_between_sent_tnx	Avg_min_be
2954	0x4cce03d1fd8aa1d6172cec74d75bd1df9e6a8ac2	0	0.00	
2955	0x4cd3bb2110eda1805dc63abc1959a5ee2d386e9f	0	0.00	
2956	0x4cd526aa2db72eb1fd557b37c6b0394acd35b212	0	3.38	
2957	0x4cdf46740d51a0f6735fe3f1d28fadf00c20cd34	0	3.92	

There are 18 full duplicates. Going to assume this was an error so will delete the 'last' version of the duplicate. They are all flagged as non-fraud

In [11]:
```
# Drop full duplicate values

df = df.drop_duplicates()
```
executed in 16ms, finished 13:52:21 2021-05-23

In [12]:
```
# Make sure there are no duplicate values left

df.duplicated().sum()
```
executed in 17ms, finished 13:52:21 2021-05-23

Out[12]: 0

Since address must be unique, ensure that there are no duplicate address values

In [13]:
```
df[df.duplicated(subset=['Address'], keep=False) ]
```
executed in 23ms, finished 13:52:21 2021-05-23

Out[13]:

	Address	FLAG	Avg_min_between_sent_tnx	Avg_min_be
2910	0x4c24af967901ec87a6644eb1ef42b680f58e67f5	0	3098.05	
2913	0x4c271764eadcf0d07e5a937b2de290294c9d11c2	0	0.00	
2919	0x4c7520df888aa4569a37ac7d132f89c65821f0af	0	0.00	
2920	0x4c77f6b01da78d053d5885e43bce5239b623dd3e	0	88542.07	
2921	0x4c7accc2689708892be29256fbe9d45a92f0aa97	0	1698.92	
2924	0x4cad652b71519a7a68d05dada31122c4c9a5ed95	0	71349.06	
2926	0x4cb981a7a2956cdfa8afbf454cb4e1b13c69aeb9	0	0.00	
2935	0x4c24af967901ec87a6644eb1ef42b680f58e67f5	0	3098.05	
2938	0x4c271764eadcf0d07e5a937b2de290294c9d11c2	0	0.00	
2944	0x4c7520df888aa4569a37ac7d132f89c65821f0af	0	0.00	
2945	0x4c77f6b01da78d053d5885e43bce5239b623dd3e	0	88542.07	
2946	0x4c7accc2689708892be29256fbe9d45a92f0aa97	0	1698.92	
2949	0x4cad652b71519a7a68d05dada31122c4c9a5ed95	0	71349.06	
2951	0x4cb981a7a2956cdfa8afbf454cb4e1b13c69aeb9	0	0.00	

Only difference is the ERC20 most sent token type and ERC2 most rec token type. They are all non-fraud accounts. Every address is unique and there is no way of certifying which address has the correct information. Rather than randomly choosing to drop one set (7 duplicates), going to drop all of them.

In [14]:
```
1 df = df.drop_duplicates(subset='Address')
```
executed in 5ms, finished 13:52:21 2021-05-23

In [15]:
```
1 # Make sure there are no duplicate addresses
2
3 df.duplicated(subset='Address').sum()
```
executed in 5ms, finished 13:52:21 2021-05-23

Out[15]: 0

Check for Null Values

In [16]:
```
1 df.info()
```

executed in 8ms, finished 13:52:21 2021-05-23

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 9816 entries, 0 to 9840
Data columns (total 35 columns):
 #   Column                            Non-Null Count  Dtype  
---  ------                            --------------  -----  
 0   Address                           9816 non-null   object 
 1   FLAG                              9816 non-null   int64  
 2   Avg_min_between_sent_tnx          9816 non-null   float64
 3   Avg_min_between_received_tnx      9816 non-null   float64
 4   Time_diff_first_last              9816 non-null   float64
 5   Sent_tnx                          9816 non-null   int64  
 6   Received_Tnx                      9816 non-null   int64  
 7   Unique_Received_From_Addresses    9816 non-null   int64  
 8   Unique_Sent_To_Addresses          9816 non-null   int64  
 9   min_value_received                9816 non-null   float64
 10  max_value_received_               9816 non-null   float64
 11  avg_val_received                  9816 non-null   float64
 12  min_val_sent                      9816 non-null   float64
 13  max_val_sent                      9816 non-null   float64
 14  avg_val_sent                      9816 non-null   float64
 15  Total_tnx                         9816 non-null   int64  
 16  total_Ether_sent                  9816 non-null   float64
 17  total_ether_received              9816 non-null   float64
 18  total_ether_balance               9816 non-null   float64
 19  Total_ERC20_tnxs                  8987 non-null   float64
 20  ERC20_total_Ether_received        8987 non-null   float64
 21  ERC20_total_ether_sent            8987 non-null   float64
 22  ERC20_uniq_sent_addr              8987 non-null   float64
 23  ERC20_uniq_rec_addr               8987 non-null   float64
 24  ERC20_uniq_rec_contract_addr      8987 non-null   float64
 25  ERC20_min_val_rec                 8987 non-null   float64
 26  ERC20_max_val_rec                 8987 non-null   float64
 27  ERC20_avg_val_rec                 8987 non-null   float64
 28  ERC20_min_val_sent                8987 non-null   float64
 29  ERC20_max_val_sent                8987 non-null   float64
 30  ERC20_avg_val_sent                8987 non-null   float64
 31  ERC20_uniq_sent_token_name        8987 non-null   float64
 32  ERC20_uniq_rec_token_name         8987 non-null   float64
 33  ERC20_most_sent_token_type        8975 non-null   object 
 34  ERC20_most_rec_token_type         8965 non-null   object 
dtypes: float64(26), int64(6), object(3)
memory usage: 2.7+ MB
```

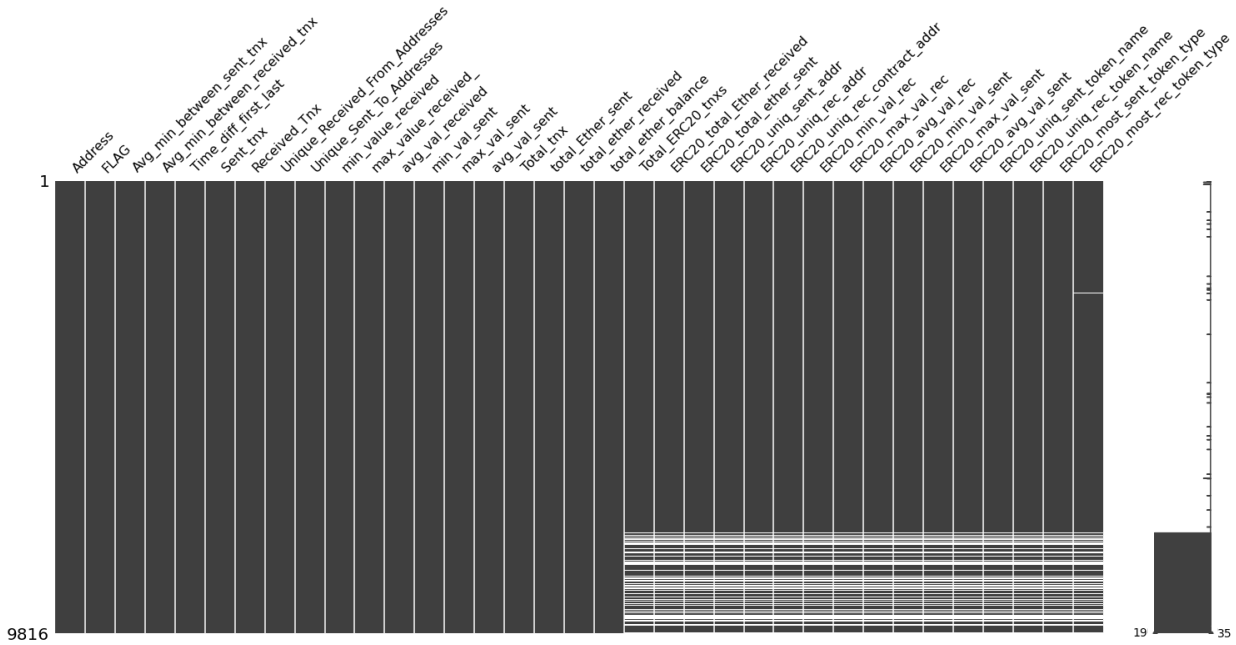
All of th ERC20 columns are missing aprox. 829 values. The first 24 columns which are only focused on ethereum do not contain any null values.

Will also have to inspect if there are placeholder values for null values

In [17]:
```
1 import missingno as msno
2 msno.matrix(df)
```

executed in 1.07s, finished 13:52:22 2021-05-23

Out[17]: <AxesSubplot:>

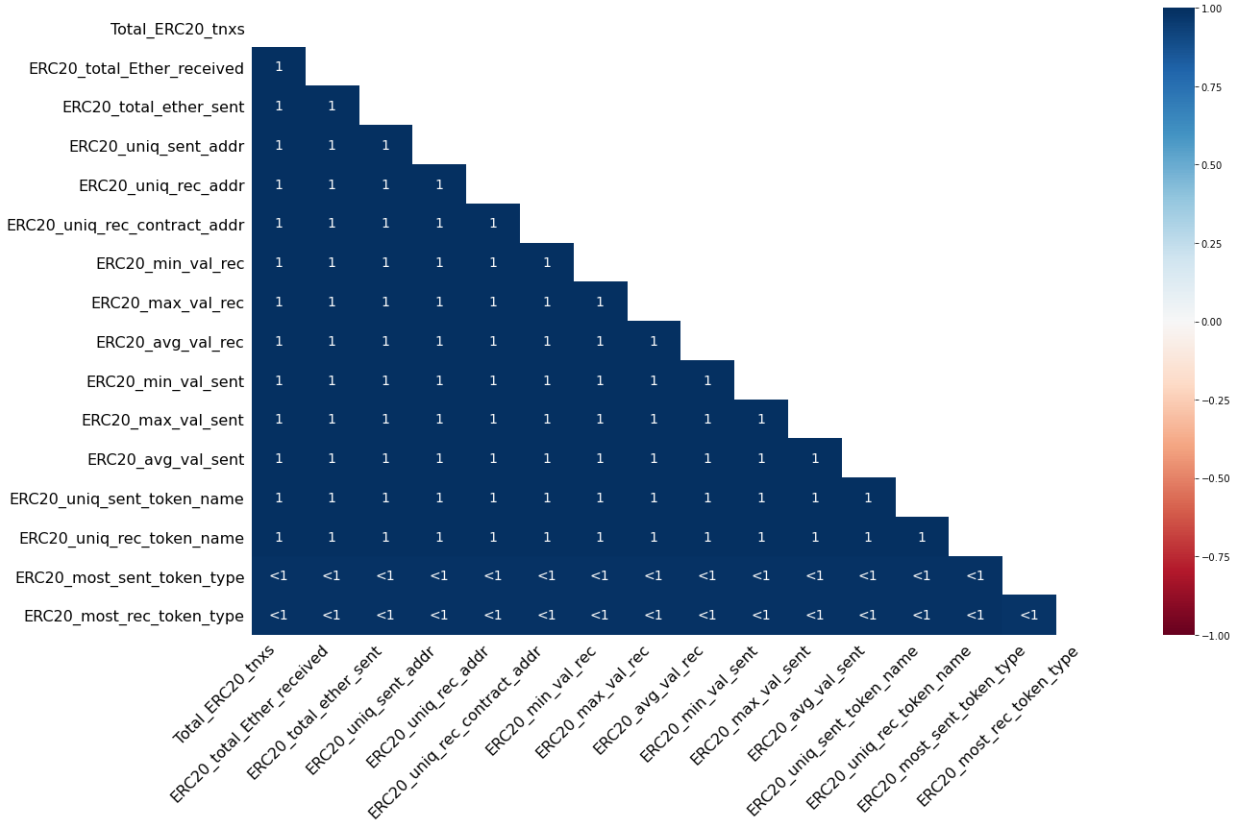


Most of the columns are completely filled in but the ERC tokens all contain null values in the same area. This makes me believe that fraudalent values may systematically be missing. Since they are not missing completely at random, will fill them in with a 'missing' value to indicate that they are missing and possibly uncover a pattern

In [18]:
```
1 msno.heatmap(df)
```

executed in 704ms, finished 13:52:23 2021-05-23

Out[18]: <AxesSubplot:>



The perfect correlation confirms that if one ERC value is missing, all of them will be classified as missing

In [19]:
```
# Check where in the dataset missing values appear

missing=list(df[df['FLAG']==1].index)
```
executed in 4ms, finished 13:52:23 2021-05-23

In [20]:
```
import numpy as np
np.array(missing)
print(np.min(missing))
np.max(missing)
```
executed in 3ms, finished 13:52:23 2021-05-23

```
7662
```

Out[20]:
```
9840
```

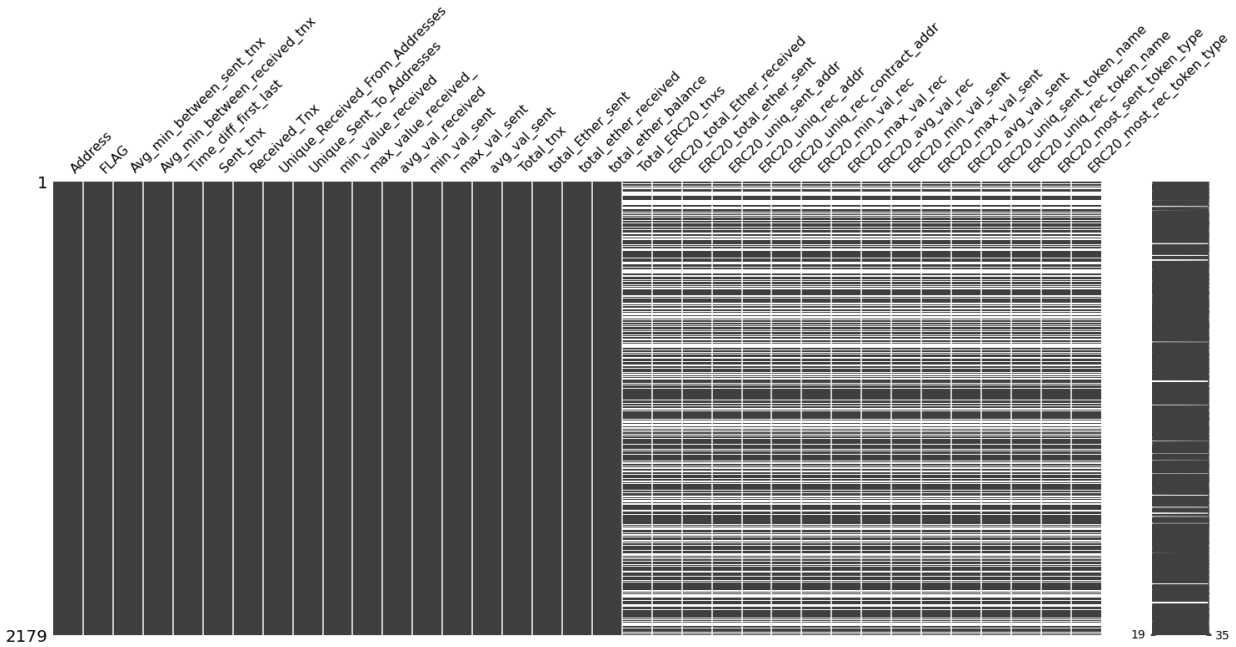
It is clear based on the matrix that the fraud instances are consecutive and correspond with the missing ERC values.

It makes sense that many of the fraud cases are missing values because a user may have had their account hacked and not been aware of transaction being made. For example, if a user connected their wallet to an exchange that was hacked they may not know where the transactions were sent

In [21]:
```
df_missing = df[df['FLAG']==1]
```
executed in 3ms, finished 13:52:23 2021-05-23

In [22]:
```
msno.matrix(df_missing)
```
executed in 346ms, finished 13:52:23 2021-05-23

Out[22]:
```
<AxesSubplot:>
```



In [23]:
```
1 df_missing.isna().sum()
```
executed in 5ms, finished 13:52:23 2021-05-23

Out[23]:
```
Address                             0
FLAG                                0
Avg_min_between_sent_tnx            0
Avg_min_between_received_tnx        0
Time_diff_first_last                0
Sent_tnx                            0
Received_Tnx                        0
Unique_Received_From_Addresses      0
Unique_Sent_To_Addresses            0
min_value_received                  0
max_value_received_                 0
avg_val_received                    0
min_val_sent                        0
max_val_sent                        0
avg_val_sent                        0
Total_tnx                           0
total_Ether_sent                    0
total_ether_received                0
total_ether_balance                 0
Total_ERC20_tnxs                  829
ERC20_total_Ether_received        829
ERC20_total_ether_sent            829
ERC20_uniq_sent_addr              829
ERC20_uniq_rec_addr               829
ERC20_uniq_rec_contract_addr      829
ERC20_min_val_rec                 829
ERC20_max_val_rec                 829
ERC20_avg_val_rec                 829
ERC20_min_val_sent                829
ERC20_max_val_sent                829
ERC20_avg_val_sent                829
ERC20_uniq_sent_token_name        829
ERC20_uniq_rec_token_name         829
ERC20_most_sent_token_type        829
ERC20_most_rec_token_type         829
dtype: int64
```

Confirm that the missing ERC20 values are in the fraud portion of the dataset

In [24]:
```
1 # How many ERC values are missing
2
3 829/len(df_missing)
```
executed in 2ms, finished 13:52:23 2021-05-23

Out[24]:
```
0.38044974759063793
```

Going to classify missing ERC values as 'missing' to demonstrate that they have meaning. Missing ERC value means the user does not know the specifics of their transactions.

About 40% of the fraudulent cases contain missing ERC20 information

In [25]:
```
df['erc_missing'] = df.loc[:,'Total_ERC20_tnxs':'ERC20_avg_val_sent'].i
```
executed in 5ms, finished 13:52:23 2021-05-23

In [26]:
```
df['erc_missing'].value_counts()
```
executed in 4ms, finished 13:52:23 2021-05-23

Out[26]:
```
False    8987
True      829
Name: erc_missing, dtype: int64
```

There is a now a column that denotes 'missing' if ERC values are null. It appears that in many of the fraud cases, ERC20 token information is missing. If the wallent was hacked, the user may be unaware of the transaction being made which is why the data is missing.

In [27]:
```
# Turn into numeric variable so it can be processed by sklearn

df['erc_missing'] = df['erc_missing'].map({False:0,
                          True:1})
```
executed in 4ms, finished 13:52:23 2021-05-23

In [28]:
```
df['erc_missing'].value_counts()
```
executed in 3ms, finished 13:52:23 2021-05-23

Out[28]:
```
0    8987
1     829
Name: erc_missing, dtype: int64
```

Feature Engineering

An example of fraud could be a bad actor gaining access to your wallet and sending the balance or certain number of tokens to their own wallet. Characteristics of this may look like:

- Significantly larger max_val_sent than avg_val_sent because the hacker was flushing out their entire account in one swoop
- Lower market cap ERC20 tokens are more likely to result in crypto scams (ie ICO bubble in 2017)

Organizing Tokens by MarketCap

- Name of the token does not provide additional values and it will be impossible to OHE names of tokens based on cardinality of most recieved and most sent ERC coins (527 unique coins)
- Cross reference coingecko to retrieve market cap for respective coins
  - Marketcap: Current Price * # of Coins in circulation
- 1 denotes coin that does not have a listed market cap
  - Using 1 as opposed to 0 because 0 means that the wallet does not trade any ERC20 coins, 1 represents that the user does trade ERC20 tokens they are just very low market cap coins

Source: https://www.coingecko.com/en (https://www.coingecko.com/en)

In [29]:
```
# List of unique received tokens

df['ERC20_most_rec_token_type'].unique()
```

executed in 4ms, finished 13:52:23 2021-05-23

Out[29]:
```
array(['Numeraire', 'Livepeer Token', 'XENON', 'EOS', '0', 'AICRYPTO',
       'DATAcoin', 'PoSToken', 'KyberNetwork', 'None', 'Bancor',
       'OmiseGO', 'ONOT', 'Tronix', 'StatusNetwork',
       'SAFE.AD - 20% DISCOUNT UNTIL 1 MAY', nan, 'Storj', 'bitqy',
       'Beauty Coin', 'SONM', 'NEVERDIE', 'INS Promo', 'TenXPay',
       'Cybereits Token', 'FunFair', 'DGD', 'iEx.ec Network Token',
       'AION', 'Aragon', 'Cofoundit', 'Golem', 'CRYPTOPUNKS', 'Nitro',
       'Ether Token', 'VeChain', 'Reputation', 'Intelion', 'Dochain',
       'SwarmCity', 'BAT', 'LockTrip', 'Humaniq', 'KickCoin', 'BOX Toke
n',
       'MobileGo', 'Monaco', 'Azbit', 'Nexium', 'ZGC', 'www.pnztrust.co
m',
       'Send your ETH to this contract and earn 2.55% every day for Live-
long. https://255eth.club', (https://255eth.club',)
       'empowr', 'An Etheal Promo', 'BitClave', 'Bytom',
       'Identity Hub Token', 'Celsius', 'Raiden', 'Trustcoin', 'ARP',
       'Qtum', 'Poker Chips', 'Loopring', 'Penta Network Token', 'Salt',
       'ICONOMI', 'Edgeless', 'BitCAD', 'SAN', 'iDAG SPACE',
       'Guaranteed Entrance Token', 'Telcoin', 'Poker IO', 'Data',
       'HuobiToken', 'DICE', 'Polybius', 'Kin', 'Tokenomy',
       'Hut34 Entropy', 'DMTS', 'MT Token', 'Testamint',
       'Bulleon Promo Token', 'blockwell.ai KYC Casper Token',
       'Time New Bank', 'TokenCard', 'BPTN', 'Super Wallet Token',
       'Cash Poker Pro', 'SNGLS', 'BlockchainPoland',
       'https://findtherabbit.me', 'Oyster Pearl', 'WisePlat Token',
       'ethBo', ' ', 'ChainLink Token', 'Delphy Token', 'BlitzPredict',
       'Pundi X Token', 'PROVER.IO additional 5% discount', 'Stox',
       'DragonGameCoin', 'OWN', 'Populous', 'Veritaseum', 'UG Token',
       'Soarcoin', 'LGO', 'FinShi Capital Tokens', 'Melon', 'Delta',
       'More Gold Coin', 'Aeternity', 'TAAS', 'TzLibre Token',
       'StatusGenesis', 'BitClave-ConsumerActivityToken', 'AnyCoinVer10',
       'CANDY', 'OpenANX', 'MKRWrapper', 'minereum', 'UnityIngot',
       'CYRUS', 'Aurora', 'Amber', 'WAX Token', 'Pro', 'Upfiring',
       'district0x', 'Global ICO Token', 'MEX', 'Centra', 'DALECOIN',
       'IGNITE', 'Carrots', 'Dentacoin', 'Dao.Casino', 'BCShareS',
       'Coineal Token', 'IOT Chain', 'Brickblock', 'TOKOK', 'Fair Token',
       'AirCoin', 'BMB', 'Guppy', 'DMarket', 'TrueUSD', 'PILLAR',
       'WinETHFree', 'Yooba token', 'Musiconomi', 'WFee', 'ZEON', 'TIME',
       'Live Stars Token', 'vSlice', 'Cryptonex (CNX)', 'Ink Protocol',
       'iXledger', 'ICON', 'OPEN Chain', 'Civic', 'TheDAO', 'Walton',
       'Request', 'Individual Content &amp;amp; Skill Token', 'NKN',
       'Flyp.me', 'RHOC', 'MediShares', 'bitqy10', 'Trip.io Ad',
       'NimiqNetwork', 'Divi Exchange Token', 'EnjinCoin', 'VIU',
       'Patientory', 'Invox Finance Token', 'ZRX', 'yocoinclassic',
       'Gnosis', 'Crypto.com', 'Skraps', 'Dai Stablecoin v1.0', 'DEBITU
M',
       'Bilian', 'REP', 'Frikandel', 'DCORP', 'HackerGold', 'Bloom',
       'Ether', 'Pundi X', 'Kuende Token', 'Mysterium', 'FIFA.win',
       'Nebulas', 'Merculet', 'Asobicoin promo', 'WINGS', 'freetoken.pr
o',
       'QRL', 'ChangeNOW', 'elixor', 'Authorship', 'VectoraicToken',
       'FloodToken', 'XCELTOKEN', 'FundRequest', 'Six Domain Asset',
```

```
        'Lunyr', 'DIW Token', 'RLC', 'Promodl', 'Silent Notary Token',
        'MKR', 'Ethos', 'Freyr Coin', 'IBCCoin', 'Love Chain', '\x01',
        'RAZOOM', 'NucleusVision', 'Insolar', 'SanDianZhong', 'WaykiCoin',
        'Credo Token', 'Beth', 'Fysical',
        'Blockchain Certified Data Token', 'GSG coin', 'WELL Token',
        'ALFA NTOK', 'Indorse', 'CryptoLah', 'The Force Token',
        'ENDO.network Promo Token', 'BigBang Game Coin Token', 'VIN',
        'COPYTRACK', 'MATRIX AI Network', 'BeautyChain', 'Cappasity',
        'Yun Planet', 'Free BOB Tokens - BobsRepair.com', 'UG Coin',
        'Egretia', 'BCDN', 'Nectar', 'Quantum',
        'Arcona Distribution Contract', 'CoinDash', 'JewCoin', 'Cryptone
x',
        'Zombie X Chain', 'Cindicator', 'ICO',
        'Huobi Airdrop HuobiAirdrop.com', 'EMO tokens', 'CVNToken',
        'ESSENTIA', 'CanYaCoin', '21Million', 'shellchains.com',
        '0xBitcoin Token', 'Unicorns', 'CGCOINS', 'RCoinVer70', 'BNB',
        'CGW', 'AdEx', 'Dragon', 'Covalent Token', 'Proof Test',
        'PowerLedger', 'YouDeal Token', 'Bitcoineum', 'DJANGO UNCHAIN',
        'Genesis Vision', 'BRAT', 'HeroCoin', 'ArcBlock', 'DEW', 'SGCC',
        'SIPC', 'MedToken', 'MCAP', 'Decentraland', 'Ethbits', 'Etherbal
l',
        'Pluton', 'Zilliqa', 'TRUE Token', 'Jury.Online', 'FluzFluz',
        'Genaro X', 'CandyHCoin', 'YESTERDAY', 'InsurePal', 'FirstBlood',
        'SingularityNET', 'FXPay', 'DRC Token', 'ATLANT', 'OPEN',
        'ECHARGE', 'Bigbom', 'E4ROW', 'IOSToken', 'Mothership',
        'Deprecated', 'Atonomi', 'VTChain', 'Xaurum', 'Biograffi',
        'Primas', 'TurnGreenToken', 'SIGMA', 'Dignity', 'PayPie',
        'SkinCoin', 'Bi ecology Token', 'Substratum', 'OCoin', 'Mavrodi',
        'BCG.to', 'timereum', 'Hiveterminal Token', 'ABYSS', 'DAY',
        'Trade', 'TaTaTu', 'QunQunCommunities', 'LEADCOIN', 'PIX', 'BSB',
        'Gifto', 'Wrapped Ether', 'Ethmon', 'AIT', 'Katalyse', 'Penis',
        'Republic', 'BANKEX', 'ElectrifyAsia', 'CreditBIT', 'Hero Origen',
        'CyberVeinToken', 'Welcome Coin', 'KredX Token', 'Relex', 'GECoi
n',
        '1irst', '$P4C3', 'GRID', 'BTOCoin', 'BCT Token', 'MINDOL',
        'SPECTRE SUBSCRIBER2 TOKEN', 'UnikoinGold', 'LuckCash', 'Network',
        'Covesting', 'Herocoin', 'BAI', 'Tierion Network Token',
        'MoneyToken IMT Token (promo)', 'Worldcore', 'KingOfCandy',
        'DOG: The Anti-Scam Reward Token', 'ZMINE Token', 'LikeCoin',
        'Maximine Coin', 'BitDegree', 'Mithril Token', 'Bitcoin EOS',
        'KEY', 'ViteToken', 'Enumivo', 'LocalCoinSwap Cryptoshare',
        'Avocado', 'Decentralized Application Coin', 'EBCoin', 'Storiqa',
        'Cevac Token', 'Olive', 'Blockwell say NOTSAFU', 'ELF',
        'Fortecoin', 'ROOMDAO COIN (RDC)', 'Celer Network',
        'AI Gaming Coin', 'ThoreCash', 'Cashaa',
        'FinallyUsableCryptoKarma', 'ICTA', 'GSENetwork', 'Lino', 'ERC20',
        'DIGIBYTE', 'Helbiz', 'Bounty', 'Hash Power Token',
        'WhalesburgToken', 'CargoX', 'Signals Network Token', 'Ethereum',
        'BitAir', 'Ponder Airdrop Token', 'Hms Token', 'BAX', 'GOT', 'Rv
T',
        'Hydro', 'BBN', 'Jolly Boots', 'OPTin Token', 'Everest',
        'SpherePay', 'Polymath', 'NGOT', 'Monetha', 'BinaryCoin',
        'AppCoins', 'FUCKtoken', 'EtherEcash',
        'A2A(B) STeX Exchange Token', 'BUZCOIN', 'ABCC invite',
        'Litecoin One', 'Energem', 'NOAHCOIN', 'Electronic Energy Coin',
        'ArtisTurba', 'Authoreon', 'ICE ROCK MINING', 'savedroid', 'Meta
l',
```

```
       'USDDex Stablecoin', 'Dropil', 'Amplify', 'CosmoCoin', 'Petroleu
m',
       'Titanium BAR Token', 'LendConnect', 'BizCoin', 'OOOBTCTOKEN',
       'Rebellious', 'Lendroid Support Token', 'USD Coin', 'QKC',
       'TemboCoin', 'Crypterium', 'Snovio', 'Galbi', 'SinghCoin',
       'Matic Network', 'Havven', 'CyberMiles', 'WORLD of BATTLES',
       'SCAM Stamp Token', 'EasyEosToken', 'INS Promo1'], dtype=object)
```

In [30]:

```
# Marketcap sourced from CoinGecko

token_rec_dict = {'Numeraire':330000000, 'Livepeer Token':809000000, '
                  'AICRYPTO': 1, 'DATAcoin': 19000, 'PoSToken':1, 'KyberNe
            'OmiseGO':1, 'ONOT':1, 'Tronix':1, 'StatusNetwork':624000000, '
            'Beauty Coin':1, 'SONM':201000000, 'NEVERDIE':1, 'INS Promo':1,
            'Cybereits Token':1, 'FunFair':1, 'DGD':72000000, 'iEx.ec Netwo
            'AION':211000000, 'Aragon':288000000, 'Cofoundit':1, 'Golem':41
            'VeChain':11920000000, 'Reputation':1, 'Intelion':1, 'Dochain':
            'SwarmCity':720000, 'BAT':1780000000, 'LockTrip':152000000, 'Hu
             'BOX Token':1687000,'MobileGo':1630000, 'Monaco':205000000, 'A
             'ZGC':1, 'empowr':1, 'BitClave':1, 'Bytom':321000000,
             'Celsius': 2760000000, 'Raiden': 53000000, 'Trustcoin':1, 'ARP
            'Qtum':2205000000, 'Poker Chips':1, 'Loopring':751000000, 'Pent
             'Salt':55000000,'ICONOMI':1, 'Edgeless':890000, 'BitCAD':1, 'S
             'Telcoin':2200000000, 'Poker IO':1, 'HuobiToken':6012000000, '
             'Kin':209400000, 'Tokenomy':7600000,'Hut34 Entropy':1, 'DMTS':
             'Testamint':1, 'Bulleon Promo Token':11600,
            'Time New Bank':18000000, 'TokenCard':1, 'BPTN':1, 'Super Walle
            'SNGLS':16000000, 'BlockchainPoland':1,'Oyster Pearl':1, 'WiseP
            'ethBo':1,'Delphy Token':1132164, 'BlitzPredict':781000,
            'Pundi X Token':628000000, 'Stox':825000,'DragonGameCoin':1, 'O
             'Veritaseum':46000000, 'UG Token':1, 'Soarcoin':4600000, 'LGO'
             'FinShi Capital Tokens':1, 'Melon':1, 'Delta':15000000,
            'More Gold Coin':1, 'Aeternity':118000000, 'TAAS':1, 'TzLibre T
            'StatusGenesis':668000000, 'AnyCoinVer10':1,
            'CANDY':195000, 'OpenANX':1, 'MKRWrapper':1, 'minereum':1, 'Uni
            'CYRUS':1, 'Aurora':16000000, 'Amber':1, 'WAX Token':1, 'Upfiri
            'district0x':2100000000, 'Global ICO Token':1, 'MEX':1, 'Centra
            'IGNITE':1, 'Carrots':1, 'Dentacoin':51000000, 'Dao.Casino':1,
            'Coineal Token':600000, 'IOT Chain':13000000, 'Brickblock':1, '
            'AirCoin':1, 'BMB':1, 'Guppy':260000000, 'DMarket':50000000, 'T
            'Yooba token':1, 'Musiconomi':1, 'WFee':1, 'ZEON':1, 'TIME':110
            'Live Stars Token':1, 'vSlice':165000, 'Cryptonex (CNX)':165000
            'iXledger':1, 'ICON':1308000000, 'OPEN Chain':1, 'Civic':319000
            'Request':97000000, 'NKN':350000000,'Flyp.me':1, 'RHOC':1, 'Med
             'bitqy10':1,'NimiqNetwork':74000000, 'Divi Exchange Token':203
            'Patientory':2000000, 'Invox Finance Token':1, 'ZRX':1400000000
            'Gnosis':385000000, 'Skraps':1, 'DEBITUM':514000,'Bilian':1, 'R
             'Frikandel':1, 'DCORP':1, 'HackerGold':1, 'Bloom':8000000,
            'Pundi X':630000000, 'Kuende Token':1, 'Mysterium':23000000, 'F
            'Nebulas':45000000, 'Merculet':7500000, 'WINGS':8700000,
            'QRL':30000000, 'ChangeNOW':1, 'elixor':1, 'Authorship':1, 'Vec
            'FloodToken':1, 'XCELTOKEN':1, 'FundRequest':1, 'Six Domain Ass
            'Lunyr':1, 'DIW Token':1, 'RLC':703000000, 'Promodl':1, 'Silent
            'MKR':4484000000, 'Ethos':1, 'Freyr Coin':1, 'IBCCoin':1, 'Love
            'RAZOOM':1, 'NucleusVision':17000000, 'Insolar':1, 'SanDianZhon
            'Credo Token':1, 'Beth':980000, 'Fysical':1, 'GSG coin':1, 'WEL
            'ALFA NTOK':1, 'Indorse':1400000, 'CryptoLah':1, 'The Force Tok
            'BigBang Game Coin Token':1, 'VIN':1,
            'COPYTRACK':1, 'MATRIX AI Network':13000000, 'Cappasity':1,
            'Yun Planet':1, 'UG Coin':1,'Egretia':16000000, 'BCDN':840000,
            'Arcona Distribution Contract':1, 'CoinDash':1, 'Cryptonex':1,
            'Zombie X Chain':1, 'Cindicator':58000000, 'ICO':1,
            'EMO tokens':1, 'CVNToken':20000000, 'ESSENTIA':6000000, 'CanYa
             '21Million':1, 'shellchains.com':1,
```

```
        '0xBitcoin Token':1, 'Unicorns':1, 'CGCOINS':1, 'RCoinVer70':1,
        'CGW':1, 'AdEx':135000000, 'Dragon':85000000, 'Covalent Token':
        'PowerLedger':1, 'YouDeal Token':1, 'Bitcoineum':1, 'DJANGO UNC
        'Genesis Vision':38000000, 'BRAT':1, 'HeroCoin':7000000, 'ArcBl
        'SIPC':1, 'MedToken':1, 'MCAP':1, 'Decentraland':1600000000, 'E
        'Pluton':26000000, 'Zilliqa':2323000000, 'TRUE Token':1, 'FluzF
        'Genaro X':14000000, 'CandyHCoin':1, 'YESTERDAY':1, 'InsurePal'
        'SingularityNET':346000000, 'FXPay':3700000, 'DRC Token':1, 'AT
        'ECHARGE':1, 'Bigbom':265000, 'E4ROW':1, 'IOSToken':1, 'Mothers
        'Deprecated':1, 'Atonomi':1, 'VTChain':1, 'Xaurum':1800000, 'Bi
        'Primas':2000000, 'TurnGreenToken':1, 'SIGMA':9000000, 'Dignity
        'SkinCoin':245000, 'Bi ecology Token':1, 'Substratum':1700000,
        'BCG.to':1, 'timereum':1, 'Hiveterminal Token':18000000, 'ABYSS
        'Trade':1, 'TaTaTu':1, 'QunQunCommunities':7000000, 'LEADCOIN':
        'Gifto':1, 'Wrapped Ether':1, 'Ethmon':1, 'AIT':1, 'Katalyse':1
        'Republic':1, 'BANKEX':12000, 'ElectrifyAsia':2148000, 'CreditB
        'CyberVeinToken':1, 'Welcome Coin':1, 'KredX Token':1, 'Relex':
        '1irst':1, '$P4C3':1, 'GRID':10844000, 'BTOCoin':1, 'BCT Token'
        'UnikoinGold':384000, 'LuckCash':1, 'Network':1,
        'Covesting':16000000, 'Herocoin':8500000, 'BAI':1, 'Tierion Netw
        'Worldcore':88000, 'KingOfCandy':1,
        'DOG: The Anti-Scam Reward Token':1, 'ZMINE Token':1300000, 'Li
        'Maximine Coin':1, 'BitDegree':1270000, 'Mithril Token':5300000
        'KEY':38000000, 'ViteToken':1, 'Enumivo':1, 'LocalCoinSwap Cryp
        'Avocado':1, 'EBCoin':1, 'Storiqa':776000,
        'Cevac Token':1, 'Olive':1, 'ELF':174000000,
        'Fortecoin':1, 'ROOMDAO COIN (RDC)':1, 'Celer Network':29500000
        'AI Gaming Coin':1, 'ThoreCash':1, 'Cashaa':330000000,
        'FinallyUsableCryptoKarma':1, 'ICTA':1, 'GSENetwork':1500000, '
        'DIGIBYTE':1600000000, 'Helbiz':1, 'Bounty':1, 'Hash Power Toke
        'WhalesburgToken':30000, 'CargoX':51000000, 'Signals Network To
        'BitAir':1, 'Ponder Airdrop Token':1, 'Hms Token':1, 'BAX':5200
        'Hydro':25000, 'BBN':1, 'Jolly Boots':1, 'OPTin Token':1, 'Evere
        'SpherePay':1, 'Polymath':294000000, 'NGOT':223000, 'Monetha':1
        'AppCoins':190000000, 'EtherEcash':1,
        'BUZCOIN':1, 'ABCC invite':1,
        'Energem':1, 'NOAHCOIN':1, 'Electronic Energy Coin':1,
        'ArtisTurba':1, 'Authoreon':1, 'ICE ROCK MINING':1, 'savedroid'
        'USDDex Stablecoin':1, 'Dropil':1, 'Amplify':1, 'CosmoCoin':1,
        'Titanium BAR Token':1, 'LendConnect':1, 'BizCoin':1, 'OOOBTCTO
        'Rebellious':1, 'Lendroid Support Token':1, 'USD Coin':16000000
        'TemboCoin':1, 'Crypterium':28000000, 'Snovio':1, 'Galbi':1, 'S
        'Matic Network':1, 'Havven':1, 'CyberMiles':18000000, 'WORLD of
        'SCAM Stamp Token':1, 'EasyEosToken':1, 'INS Promo1':1}
```

executed in 18ms, finished 13:52:23 2021-05-23

In [31]:

```
# Create a set of the most received tokens and find the difference for

rec_set = set(token_rec_dict.keys())
sent_set = set(df['ERC20_most_sent_token_type'].unique())
sent_only = list(sent_set.difference(rec_set))
```

executed in 3ms, finished 13:52:23 2021-05-23

In [32]:
```
# Marketcap sourced from CoinGecko
# Added 0

sent_only_dict = {
 'UTRUST':258000000,'Decentralized Application Coin':1,'Happy Coin':1,'
 'Po.et':698000,'Lucky Token':1,'EduCoin':2700000,'1World':2250000,'Cry
 'AdBank':5400000,'CultureVirtue':1,'QUBE':1,
 'WePower':21000000,'Everex':19000000,'realchain':1,'DADI':1,'Aeron':41
 'WTT':165000,'WIKI Token':290000,'IDICE':1,'Storm':356000000,'Sether':
 'The Token Fund':1,'Trace':876000,'Countinghouse Fund':1,'0xcert Proto
 'NapoleonX':9800000,'Blackmoon Crypto Token':1,'Fantom Token':16560000
 'POWERBANK':1,'Aditus':219000,'ChangeBank':1,'Adshares':7760000,'Monol
 'CCRB':1,'Digix Gold Token':4300000,'Maker':4368000000,'Cobinhood':201
 'Measurable Data Token':35000000,'Decent.Bet Token':68000,
 'HOQU Token':1,'Feed':1,'SIRIN':9800000,'QASH':35000000,'AirSwap':5800
 'MOT':1,'Vezt':1,'CoinBene Coin':1840000,'dmb.top':1,'WanCoin':1,'Nebu
 'Identity Hub Token':1,'Aigang':1,'UnlimitedIP Token':13000000,'PRG':1
 'Blocktix':445000,'FOAM Token':23000000,'Banker Token':1,'SunContract'
 'Friendz Coin':1700000,'AVT':6500000, 'Opus':617000,'DRP Utility':1,
 'ARBITRAGE':99000,'CarTaxi':1,'SCAM Seal Token':1,'IDEX Membership':1,
 'BMChain Token':1,'BOMB':3000000,'Piggies':1,'Litecoin One':1,'TezosTK
 'BitDice':1,'Loom':99000000,'ethereumAI Token':1,'eBTC':1,'VIB':160000
 'Propy':81000000,'SeratioCoin':1,'IUNGO':1,'TYT':1,'PangeaCoinICO':1,'
 'ETHWrapper':1,'Bounty0x':732000,'CharterCoin':1, 'CoinBoin':1,'ChainL
 'BANCA':1484000,'BIX Token':1,'Theta Token':1,'DNA':1, '0':0, 0:0}
```

executed in 5ms, finished 13:52:23 2021-05-23

In [33]:
```
# https://www.geeksforgeeks.org/python-merging-two-dictionaries/
# Create one dictionary that contains prices for all coins

def Merge(dict1, dict2):
    res = {**dict1, **dict2}
    return res
```

executed in 2ms, finished 13:52:23 2021-05-23

In [34]:
```
all_coins = Merge(token_rec_dict, sent_only_dict)
```

executed in 1ms, finished 13:52:23 2021-05-23

In [35]:
```
# Turn dictionary into DataFrame to make it easier to evaluate values

series_coins = pd.Series(all_coins)
df_all_coins = pd.DataFrame(series_coins)
df_all_coins = df_all_coins.rename(columns={0:'price'})
```

executed in 3ms, finished 13:52:23 2021-05-23

In [36]:
```
1 df_all_coins.head()
```
executed in 4ms, finished 13:52:23 2021-05-23

Out[36]:

	price
Numeraire	330000000
Livepeer Token	809000000
XENON	1
EOS	10373000000
AICRYPTO	1

In [37]:
```
1 pd.options.display.float_format = '{:.5f}'.format
2
3 df_all_coins.describe()
```
executed in 8ms, finished 13:52:23 2021-05-23

Out[37]:

	price
count	529.00000
mean	246935794.49149
std	1431097854.20461
min	0.00000
25%	1.00000
50%	1.00000
75%	8500000.00000
max	18872000000.00000

We get a sense that the data is extremely skewed because we have quite a few 1 values and then very large values on the other side of the distribution

Create a system to bin the values based on market cap size

In [38]:
```
1 # Filter only coins that have listed market caps
2
3 df_priced=df_all_coins[df_all_coins['price']>1]
```
executed in 2ms, finished 13:52:23 2021-05-23

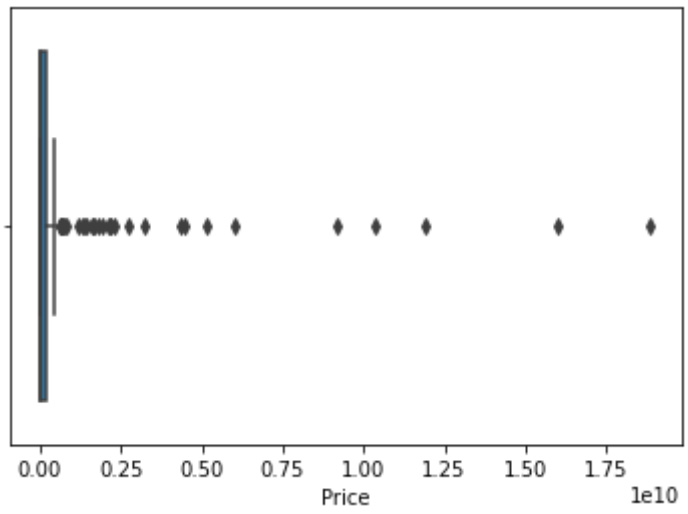
In [39]:
```
1 df_priced.describe()
```
executed in 7ms, finished 13:52:23 2021-05-23

Out[39]:

	price
count	217.00000
mean	601977119.70507
std	2188973819.33492
min	11600.00000
25%	1840000.00000
50%	16000000.00000
75%	190000000.00000
max	18872000000.00000

Displays distribution of coins with listed market caps. More informative than all prices because the 1 values were severly skewing the dataset

In [40]:
```
1 import matplotlib.pyplot as plt
2 import seaborn as sns
3
4 fig, ax = plt.subplots()
5 sns.boxplot(data=df_priced, x='price', ax=ax)
6 ax.set_xlabel('Price');
```
executed in 62ms, finished 13:52:23 2021-05-23



Many outliers on the right side of the distribution

ERC20 Coins Received

In [41]:
```
# Create column that shows marketcap in dollars for the most received E

df['cap_dollars_rec'] = df['ERC20_most_rec_token_type'].map(all_coins)
```
executed in 4ms, finished 13:52:23 2021-05-23

In [42]:
```
# Confirm values appear

df['cap_dollars_rec']
```
executed in 3ms, finished 13:52:23 2021-05-23

Out[42]:
```
0         330000000.00000
1         809000000.00000
2                1.00000
3                1.00000
4       10373000000.00000
               ...
9836       1500000.00000
9837                 nan
9838                 nan
9839                 nan
9840             1.00000
Name: cap_dollars_rec, Length: 9816, dtype: float64
```

In [43]:
```
# Display bounds for each market cap
# Use 20th percentiles to bin market cap values

print(f"Small Cap Upper Bound: ${round(np.quantile(df_priced['price'],
print(f"Sm_Mid Cap Upper Bound: ${round(np.quantile(df_priced['price'],
print(f"Mid Cap Upper Bound: ${round(np.quantile(df_priced['price'], 0.
print(f"Large Cap Upper Bound: ${round(np.quantile(df_priced['price'],
print(f"Ultra Cap Upper Bound: ${round(np.quantile(df_priced['price'],
```
executed in 4ms, finished 13:52:24 2021-05-23

```
Small Cap Upper Bound: $1,416,800.0
Sm_Mid Cap Upper Bound: $9,320,000.0
Mid Cap Upper Bound: $35,000,000.0
Large Cap Upper Bound: $282,400,000.0
Ultra Cap Upper Bound: $18,872,000,000.0
```

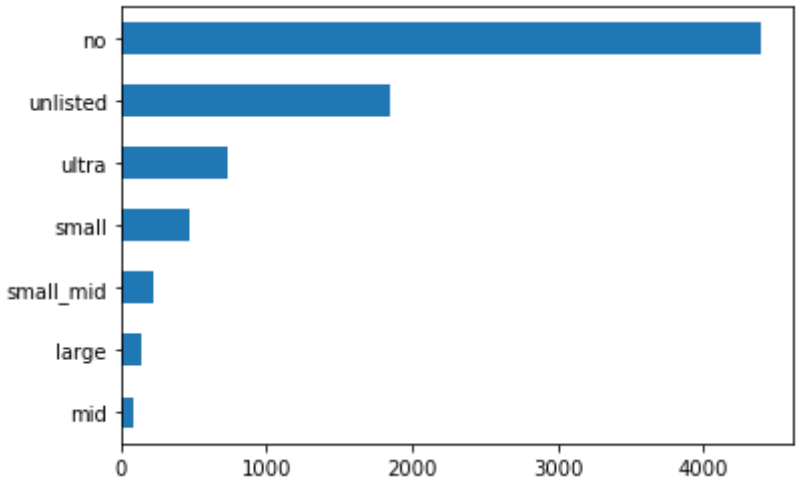
Define market cap based on 20th percent quantiles of the distribution

- Small Cap: $2 - $1,416,800
- Small-Mid Cap: $1,416,800 - $9,320,000
- Mid Cap: $9,320,000 - $35,000,000
- Large Cap: $35,000,000 - $282,400,000
- Ultra Cap: $282,400,000 - $18,872,000,000

In [44]:
```
def market_cap_weight(x):
    """
    Compute if market cap by dollar should be grouped as small, sm-mid,
    large, or ultra
    Parameters: DataFrame values
    Output: Market cap described by an ordinal value
    """
    if x == 0:
        return 'no'
    if x == 1:
        return 'unlisted'
    elif x < 1416800:
        return 'small'
    elif x < 9320000:
        return 'small_mid'
    elif x < 35000000:
        return 'mid'
    elif x < 282400000:
        return 'large'
    elif x < 18872000000:
        return 'ultra'

```
executed in 3ms, finished 13:52:24 2021-05-23

In [45]:
```
# Create column that displays values 0-6 depending on market cap of ave

df['weights_rec'] = df['cap_dollars_rec'].map(market_cap_weight)
```
executed in 4ms, finished 13:52:24 2021-05-23

In [46]:
```
df['weights_rec'].value_counts(ascending=True).plot(kind='barh');
```
executed in 75ms, finished 13:52:24 2021-05-23



About 4000 wallets (identified by address) have never received an ERC20 token. This is not surprising given that Ethereum and Bitcoin have such a dominant market share of the crypto market. Many investors do not have experience with these alternative coins. After 'no' comes unlisted tokens. Thse are tokens that don't have a listed value on CoinGecko. After comes ultra, small, small_mid, large, and then mid.

There's no discenrable linear relationship between market cap of coin and liklihold of user receiving the coin. Given that there are far more unlisted coins (those with very low market cap values) it makes some sense that they are the predonominant holding.

In [47]:
```
# Check for null values

df['weights_rec'].isna().sum()
```
executed in 3ms, finished 13:52:24 2021-05-23

Out[47]: 1930

Null values emerged from certain coins being misclassified during the data collection period. For example some coins were websites, advertisements, or exchanges and not actual coins. Also, some coins were given 'None' value instead of 0. 0 meaning no ERC20 tokens received in the wallet.

Going to fill null values with 'no', having a mislisted coin or None is the equivalant as never having an ERC20 coin in the wallet

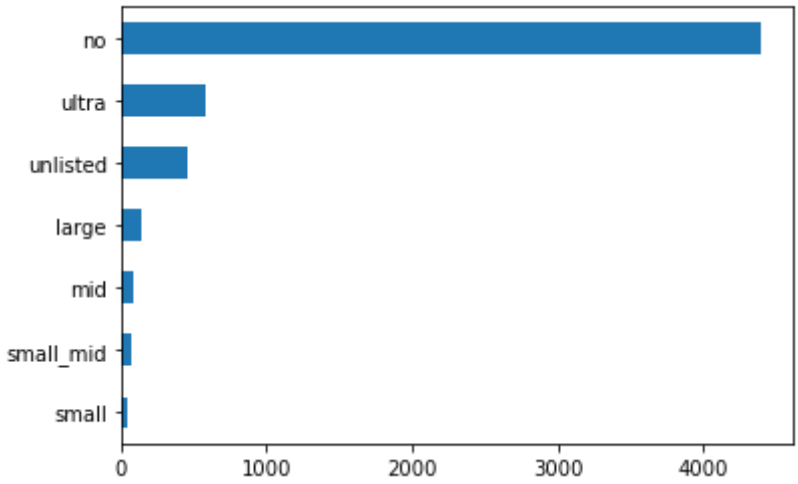
In [48]:
```
df['weights_rec'].fillna('no', inplace=True)
```
executed in 3ms, finished 13:52:24 2021-05-23

ERC20 Coins Sent

In [49]:
```
# Create column that shows marketcap in dollars for the most sent ERC20

df['cap_dollars_sent'] = df['ERC20_most_sent_token_type'].map(all_coins
```
executed in 4ms, finished 13:52:24 2021-05-23

In [50]:
```
# Create column that shows marketcap in dollars for the most sent ERC20

df['weights_sent'] = df['cap_dollars_sent'].map(market_cap_weight)
```
executed in 4ms, finished 13:52:24 2021-05-23

In [51]:
```
df['weights_sent'].value_counts(ascending=True).plot(kind='barh');
```
executed in 109ms, finished 13:52:24 2021-05-23



Compared to ERC20 tokens received, ultra is more common than unlisted to be sent from a wallet. If a wallet is sending, it is doing so for the purpose of exchanging for another cryptocurrency, gifting, or purchasing a good or service. Ultra may be more popular than unlisted because merchants are more likely to accept widely adopted cyrptocurrencies rather than low market cap value coins.

In [52]:
```
# Check for null values

df['weights_sent'].isna().sum()
```
executed in 3ms, finished 13:52:24 2021-05-23

Out[52]: 4060

In [53]:
```
# Check for null values

df['weights_sent'].fillna('no', inplace=True)
```
executed in 2ms, finished 13:52:24 2021-05-23

Same process as ERC20 tokens received

Comparing Max Value to Average Value Sent

- If a wallet has one very large transaction, that may signal that is has been hacked and the bad actor is flushing the wallet. We would see one very large transaction compared to a much smaller average

In [54]:
```
1 df.head()
```
executed in 12ms, finished 13:52:24 2021-05-23

Out[54]:

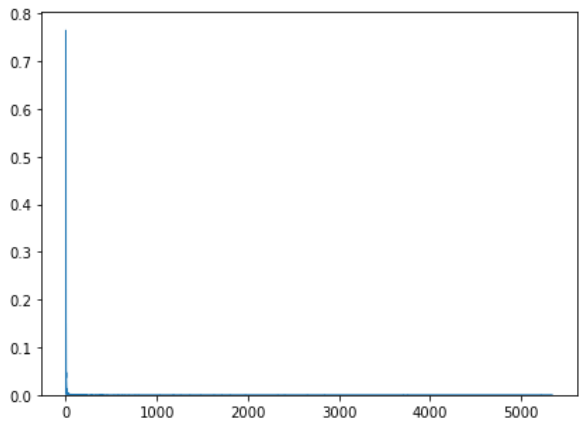
	Address	FLAG	Avg_min_between_sent_tnx	Avg_min_betwe
0	0x00009277775ac7d0d59eaad8fee3d10ac6c805e8	0	844.26000	
1	0x0002b44ddb1476db43c868bd494422ee4c136fed	0	12709.07000	
2	0x0002bda54cb772d040f779e88eb453cac0daa244	0	246194.54000	
3	0x00038e6ba2fd5c09aedb96697c8d7b8fa6632e5e	0	10219.60000	
4	0x00062d1dd1afb6fb02540ddad9cdebfe568e0d89	0	36.61000	

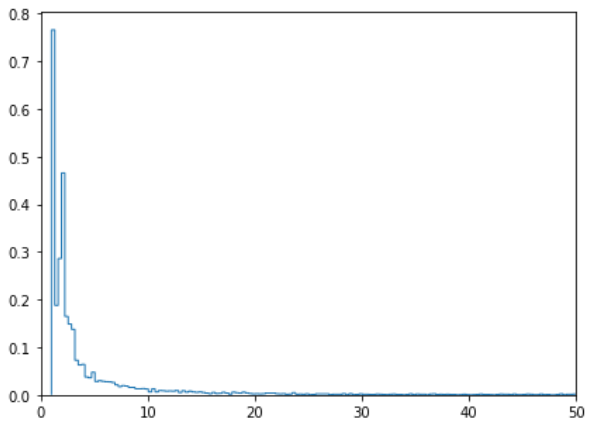
In [55]:
```
1 # Create column to show max value sent vs. avg. value sent
2
3 df['max_div_avg'] = df['max_val_sent']/df['avg_val_sent']
```
executed in 2ms, finished 13:52:24 2021-05-23

In [56]:
```
# Visualize distribution

fig, axes = plt.subplots(figsize=(15,5),ncols=2)
axes[0].hist(df['max_div_avg'], bins='auto', density=True, histtype='st

axes[1].hist(df['max_div_avg'], bins='auto', density=True, histtype='st
axes[1].set_xlim(left=0,right=50);
```

executed in 193ms, finished 13:52:24 2021-05-23

```
/Users/ethankunin/opt/anaconda3/envs/learn-env/lib/python3.8/site-package
s/numpy/lib/histograms.py:403: RuntimeWarning: invalid value encountered
in greater_equal
  keep = (a >= first_edge)
/Users/ethankunin/opt/anaconda3/envs/learn-env/lib/python3.8/site-package
s/numpy/lib/histograms.py:404: RuntimeWarning: invalid value encountered
in less_equal
  keep &= (a <= last_edge)
/Users/ethankunin/opt/anaconda3/envs/learn-env/lib/python3.8/site-package
s/numpy/lib/histograms.py:839: RuntimeWarning: invalid value encountered
in greater_equal
  keep = (tmp_a >= first_edge)
/Users/ethankunin/opt/anaconda3/envs/learn-env/lib/python3.8/site-package
s/numpy/lib/histograms.py:840: RuntimeWarning: invalid value encountered
in less_equal
  keep &= (tmp_a <= last_edge)
/Users/ethankunin/opt/anaconda3/envs/learn-env/lib/python3.8/site-package
s/numpy/lib/histograms.py:403: RuntimeWarning: invalid value encountered
in greater_equal
  keep = (a >= first_edge)
/Users/ethankunin/opt/anaconda3/envs/learn-env/lib/python3.8/site-package
s/numpy/lib/histograms.py:404: RuntimeWarning: invalid value encountered
in less_equal
  keep &= (a <= last_edge)
/Users/ethankunin/opt/anaconda3/envs/learn-env/lib/python3.8/site-package
s/numpy/lib/histograms.py:839: RuntimeWarning: invalid value encountered
in greater_equal
  keep = (tmp_a >= first_edge)
/Users/ethankunin/opt/anaconda3/envs/learn-env/lib/python3.8/site-package
s/numpy/lib/histograms.py:840: RuntimeWarning: invalid value encountered
in less_equal
  keep &= (tmp_a <= last_edge)
```





Data is very left skewed. Most of the values are below 10 but there are outliers values on the right

side of the distribution

In [57]:
```
df['max_div_avg'].sort_values(ascending=False)
```
executed in 4ms, finished 13:52:24 2021-05-23

Out[57]:
```
7206    5344.33445
7082    5221.72742
6478    3755.86854
6828    3707.75812
266     2843.00000
           ...
9832           nan
9833           nan
9835           nan
9837           nan
9839           nan
Name: max_div_avg, Length: 9816, dtype: float64
```

In [58]:
```
# Check how max_div_avg compares for fraud vs. non-fraud wallets

df.groupby('FLAG')['max_div_avg'].agg(['mean', 'median'])
```
executed in 7ms, finished 13:52:24 2021-05-23

Out[58]:

FLAG	mean	median
0	16.00695	2.31129
1	4.14258	1.06056

- Non fraud has a greater median and mean max sent divided by the average transation size
- Contrary to hypothesis

Remove Redudant Columns

In [59]:
```
df.head()
```
executed in 11ms, finished 13:52:24 2021-05-23

Out[59]:

	Address	FLAG	Avg_min_between_sent_tnx	Avg_min_betwe
0	0x00009277775ac7d0d59eaad8fee3d10ac6c805e8	0	844.26000	
1	0x0002b44ddb1476db43c868bd494422ee4c136fed	0	12709.07000	
2	0x0002bda54cb772d040f779e88eb453cac0daa244	0	246194.54000	
3	0x00038e6ba2fd5c09aedb96697c8d7b8fa6632e5e	0	10219.60000	
4	0x00062d1dd1afb6fb02540ddad9cdebfe568e0d89	0	36.61000	

In [60]:
```
1 cols_to_drop = ['cap_dollars_rec', 'cap_dollars_sent']
```
executed in 2ms, finished 13:52:24 2021-05-23

In [61]:
```
1 df = df.drop(columns=cols_to_drop, axis=1)
```
executed in 4ms, finished 13:52:24 2021-05-23

Dropping the market cap weights because those are reprsented as categorical variables

EXPLORE

- Observe how the variables are correlated with fraud
- See how the variables differ when viewing fraudulent vs non-fraudulent wallet transactions
- Analyze the distribution of key variables
- Check how ERC20 transactions compare to ETH transactions

Inspect Correlation

In [62]:
```
# Check correlation with FLAG

corr = pd.DataFrame(df.corr().iloc[0])
corr['abs'] = abs(corr['FLAG'])
corr.sort_values(by='abs', ascending=False).drop('FLAG', axis=0).rename
```

executed in 19ms, finished 13:52:24 2021-05-23

Out[62]:

	corr_value	abs
erc_missing	0.56859	0.56859
Time_diff_first_last	-0.26961	0.26961
Avg_min_between_received_tnx	-0.11864	0.11864
Total_tnx	-0.10055	0.10055
Received_Tnx	-0.07954	0.07954
Sent_tnx	-0.07819	0.07819
avg_val_sent	-0.06336	0.06336
Unique_Sent_To_Addresses	-0.04569	0.04569
max_div_avg	-0.03348	0.03348
Unique_Received_From_Addresses	-0.03207	0.03207
Avg_min_between_sent_tnx	-0.02962	0.02962
Total_ERC20_tnxs	-0.02576	0.02576
ERC20_min_val_sent	0.02536	0.02536
ERC20_avg_val_sent	0.02504	0.02504
ERC20_max_val_sent	0.02503	0.02503
ERC20_total_ether_sent	0.02476	0.02476
max_val_sent	-0.02244	0.02244
min_value_received	-0.02161	0.02161
ERC20_uniq_sent_addr	-0.02061	0.02061
max_value_received_	-0.01930	0.01930
ERC20_uniq_rec_contract_addr	-0.01844	0.01844
ERC20_uniq_rec_token_name	-0.01797	0.01797
ERC20_uniq_rec_addr	-0.01758	0.01758
total_ether_received	-0.01695	0.01695
total_Ether_sent	-0.01503	0.01503
avg_val_received	-0.01188	0.01188
ERC20_min_val_rec	0.01113	0.01113
ERC20_uniq_sent_token_name	0.00946	0.00946
ERC20_avg_val_rec	0.00786	0.00786
min_val_sent	0.00659	0.00659

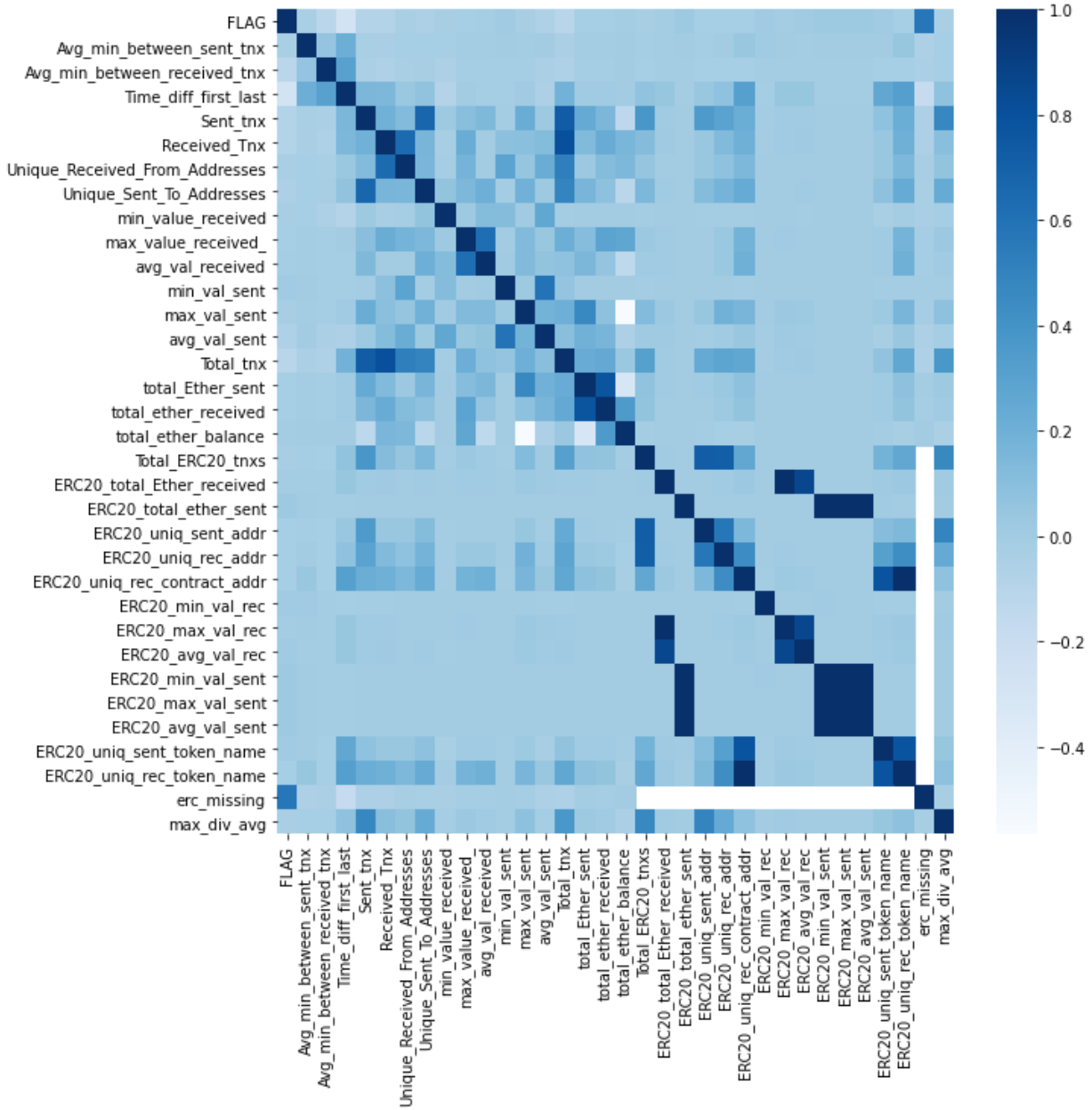
	corr_value	abs
ERC20_total_Ether_received	-0.00449	0.00449
ERC20_max_val_rec	-0.00433	0.00433
total_ether_balance	-0.00324	0.00324

Correlation only sorts through numeric data. Erc_missing has the strongest correlation with fraud. This variable indicates that the ERC20 transaction records are missing. I suspect they are missing because the owner of the wallet did not know that the transactions were being made because they were hacked. The second strongest correlation is time diff between first and last which represents the number of minutes between the first and last transaction. This represents how long the wallet has been in use. The longer the wallet has been in use, the less liklihood that it has been a victim of fraudulant transactions because it has a negative correlation. Most of the correlation values are very low.

In [63]:
```
1 fig, ax = plt.subplots(figsize=(10,10))
2 sns.heatmap(df.corr(), cmap='Blues')
```

executed in 609ms, finished 13:52:25 2021-05-23

Out[63]: <AxesSubplot:>



- In general, the variables do not have much correlation with each other
- What stands out is that ERC statistics have lower than average correlation with non-ERC statistics of the same measurement

In [64]:
```
# See the difference between variables for fraudulent vs non-fraudulent

df.groupby('FLAG').agg(['mean', 'median', 'std'])
```

executed in 67ms, finished 13:52:25 2021-05-23

Out[64]:

	Avg_min_between_sent_tnx			Avg_min_between_received_tnx			Time_diff_first_last	
	mean	median	std	mean	median	std	mean	medi
FLAG								
0	5419.36464	22.75000	22482.32234	9461.69245	1403.40000	25348.31384	264682.31558	1168
1	3888.10978	0.00000	17505.36774	2874.71264	82.07000	10624.45345	55230.05795	75

In all of the values, the mean is magnitudes larger than the median values which indicates that there is significant skew in the data and there are large outlier values on the right side of the distribution. Additionally, fraud and non-fraud variables have very different characteristics. The median is similar in certain instances but the means are very different

Given that there is a large separation in summary statistics between fraud vs. non-fraud wallets, I am confident the model will be able to effectively classify transactions

Inspecting Distribution

In [65]:
```
# Separate out columns that do not contain ERC20 information

df.columns
ether_cols = [col for col in df.columns if col not in 'ERC']
ether_cols
erc = set(df.columns[df.columns.str.contains('ERC')])
non_erc = set(df.columns)
non_erc = list(non_erc.difference(erc))
non_erc.remove('Address')
non_erc.remove('FLAG')
```

executed in 3ms, finished 13:52:25 2021-05-23

In [66]:
```
df.columns
ether_cols = ['Avg_min_between_sent_tnx',
       'Avg_min_between_received_tnx', 'Time_diff_first_last', 'Sent_tn
       'Received_Tnx', 'Unique_Received_From_Addresses',
       'Unique_Sent_To_Addresses', 'min_value_received', 'max_value_rec
       'avg_val_received', 'min_val_sent', 'max_val_sent', 'avg_val_sen
       'Total_tnx', 'total_Ether_sent', 'total_ether_received',
       'total_ether_balance', 'max_div_avg']
```

executed in 3ms, finished 13:52:25 2021-05-23

In [67]:
```
# Subset data into fraudulent vs non-fraudulent wallets

df_fraud = df[df['FLAG']==1]
df_valid = df[df['FLAG']==0]
```

executed in 8ms, finished 13:52:25 2021-05-23

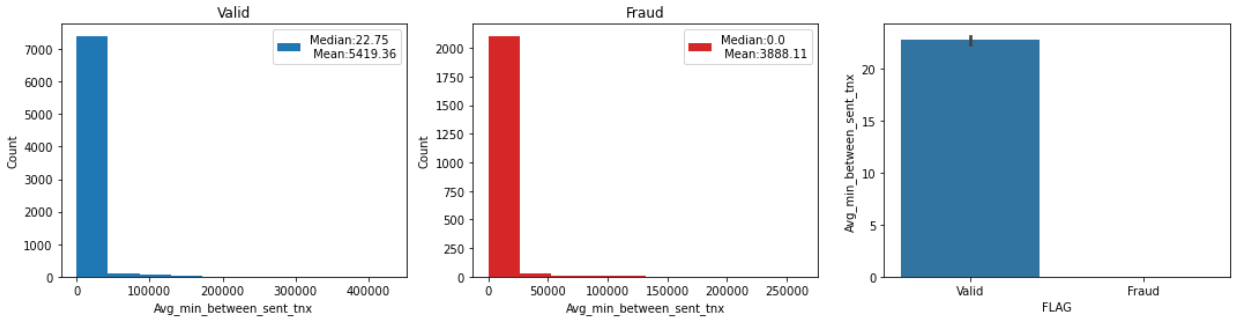
In [68]:

```
from numpy import median

def valid_fraud_comparison(df1, df2, col):
    """
    Comopare variable of valid vs. fraudulent transactions
    Parameters: DataFrame Valid and DataFrame Fraud
    Output: Two histograms displaying median and mean
    """
    fig, axes = plt.subplots(ncols=3, figsize=(15,4))


    valid_median=round(df1[col].median(),2)
    valid_mean=round(df1[col].mean(),2)
    axes[0].hist(df1[col],label=f'Median:{valid_median} \n Mean:{valid_
    axes[0].set_xlabel(col)
    axes[0].set_ylabel('Count')
    axes[0].set_title('Valid')
    axes[0].legend()

    fraud_median=round(df_fraud[col].median(),2)
    fraud_mean=round(df_fraud[col].mean(),2)
    axes[1].hist(df_fraud[col], color='C3',label=f'Median:{fraud_median
    axes[1].set_xlabel(col)
    axes[1].set_ylabel('Count')
    axes[1].set_title('Fraud')
    axes[1].legend()


    sns.barplot(data=df, y=col, x='FLAG',ax=axes[2],ci=68, estimator=me
    axes[2].set_xlabel('FLAG')
    axes[2].set_xticklabels(['Valid', 'Fraud'])
    fig.tight_layout()
    plt.show();

```

executed in 5ms, finished 13:52:25 2021-05-23

In [69]:
```
for col in ether_cols:
    print(col)
    valid_fraud_comparison(df_valid, df_fraud, col)
    print('-----------------------')

```
executed in 8.70s, finished 13:52:33 2021-05-23

```
Avg_min_between_sent_tnx
```




```
-----------------------
Avg_min_between_received_tnx
```

- All of the data is left skewed
- There are significant outliers on the right side of the distribution
- Most observations are within a tight range and then outlier values on the far right side of the distribution
- Valid transactions tend to have larger values

Inspect Relationship between ERC20 and Ethereum Transactions

In [70]:
```
# Sort data into corresponding ETH vs. ERC20 Data

ether_cols = ['Total_tnx','Unique_Received_From_Addresses',
       'Unique_Sent_To_Addresses', 'min_value_received', 'max_value_rec
       'avg_val_received', 'min_val_sent', 'max_val_sent', 'avg_val_sen
       'total_Ether_sent', 'total_ether_received']

ERC20_cols = ['Total_ERC20_tnxs', 'ERC20_uniq_rec_addr', 'ERC20_uniq_se
              'ERC20_min_val_rec', 'ERC20_max_val_rec', 'ERC20_avg_val_
              'ERC20_min_val_sent','ERC20_max_val_sent', 'ERC20_avg_val
              'ERC20_total_ether_sent','ERC20_total_Ether_received']
```
executed in 2ms, finished 13:52:33 2021-05-23

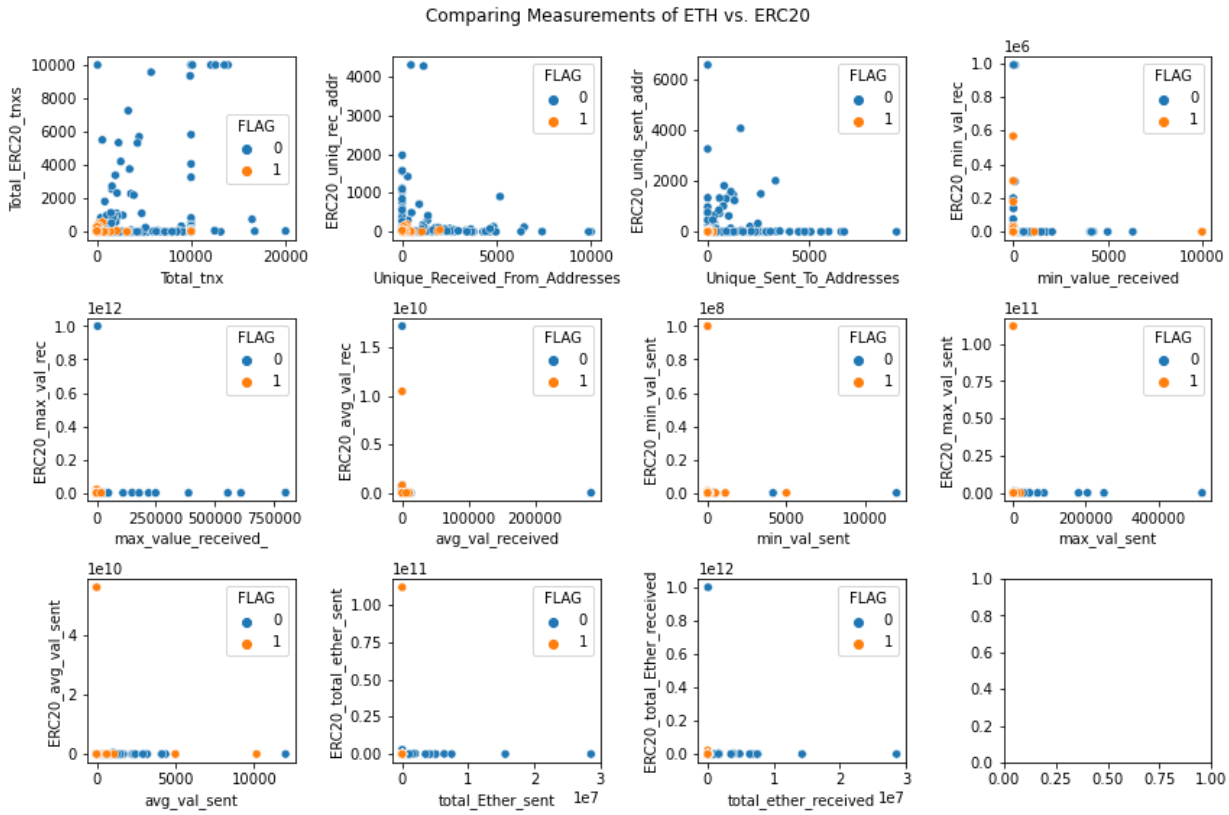
In [71]:
```
# Confirm the features between ETH and ERC20 match up

eth_erc_cols = list(zip(ether_cols, ERC20_cols))
eth_erc_cols
```

executed in 3ms, finished 13:52:33 2021-05-23

Out[71]:
```
[('Total_tnx', 'Total_ERC20_tnxs'),
 ('Unique_Received_From_Addresses', 'ERC20_uniq_rec_addr'),
 ('Unique_Sent_To_Addresses', 'ERC20_uniq_sent_addr'),
 ('min_value_received', 'ERC20_min_val_rec'),
 ('max_value_received_', 'ERC20_max_val_rec'),
 ('avg_val_received', 'ERC20_avg_val_rec'),
 ('min_val_sent', 'ERC20_min_val_sent'),
 ('max_val_sent', 'ERC20_max_val_sent'),
 ('avg_val_sent', 'ERC20_avg_val_sent'),
 ('total_Ether_sent', 'ERC20_total_ether_sent'),
 ('total_ether_received', 'ERC20_total_Ether_received')]
```

In [72]:
```
fig, axes = plt.subplots(nrows=3, ncols=4, figsize=(12,8))
axes=axes.flatten()
for ax, (eth, erc) in zip(axes, eth_erc_cols):
         sns.scatterplot(data=df, x=eth, y=erc, hue='FLAG',ax=ax)
fig.suptitle('Comparing Measurements of ETH vs. ERC20')
fig.tight_layout()
```

executed in 3.05s, finished 13:52:36 2021-05-23



- The main conclusion is that ETH non-fraud transactions seem to have the most variance
- Fraudulant transactions typically have lower values than valid transactions
- ETH transaction statistics have a greater variance than ERC20 transaction statistics

MODEL

Based on our EDA and goal of the model, I will build a K-Nearest Neighbors model and Random Forest model. The dataset has many outlier values and random forests do a good job of handling outlier values.

The purpose of the model is to accurately predict fraudulent transactions. I am prioritizing recall over other evaluation metrics because the cost of false negative is greater than the cost of false positive. If a customer receives a notification for potential fraud that is not fraud, there is a little harm. If the model suspects fraud, but does not notify the customer because it is more focused on precision, the damages are higher. Can quanitify the value of the model by seeing how it compares to a dummy model, and then calculating how much in ether it would have saved if it notified customers of fraud.

60% of the crypto market share is in either Bitcoin or Ethereum. Bitcoins controls 40% of the market share, Ethereum 20%, and the next largest coin controls 2.5% of the market share. The model predicts fraud in ETH and ERC20 transactions. Since ETH controls a large market share in the crypto space, this model is broad reaching.

Train Test Split

In [73]:
```
from sklearn.model_selection import train_test_split
y=df['FLAG']
X = df.drop(columns=['Address', 'FLAG', 'ERC20_most_sent_token_type', '
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size=0.3,
```

executed in 245ms, finished 13:52:37 2021-05-23

In [74]:
```
# Ensure split was succesful

print(X_test.shape)
print(y_test.shape)
print(X_train.shape)
print(y_train.shape)
```

executed in 3ms, finished 13:52:37 2021-05-23

```
(2945, 35)
(2945,)
(6871, 35)
(6871,)
```

Preprocessing Pipeline

- Used to quickly one-hot encode categorical variables and scale numeric variables

In [75]:
```
from sklearn.pipeline import Pipeline, make_pipeline

# Pull out numeric columns

num_cols = X_train.select_dtypes('number').columns.to_list()
```
executed in 5ms, finished 13:52:37 2021-05-23

We know that the ERC20 values have a lot of null values when cases are fraudulent. Given that they are numeric columns, they must be encoded with a number. Choosing to encode with median over mean because it is resistant to outliers. Could also choose to impute value that is on the end of the distribution but that will artifically increase or decrease the skew of the data.

In [76]:
```
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

# Create numeric pipeline with imputer

num_transformer = Pipeline(steps=[
    ('imputer',SimpleImputer(strategy='median')),
    ('scale',StandardScaler())])
```
executed in 70ms, finished 13:52:37 2021-05-23

In [77]:
```
## Pull out categorical columns

cat_cols = X_train.select_dtypes('O').columns.tolist()
```
executed in 3ms, finished 13:52:37 2021-05-23

In [78]:
```
# Create categorical imputer

cat_transformer = Pipeline(steps=[
    ('imputer',SimpleImputer(strategy='most_frequent')),
    ('encoder',OneHotEncoder(sparse=False,handle_unknown='ignore'))])
```
executed in 4ms, finished 13:52:37 2021-05-23

Using 'most_frequent' strategy because it has the least bias for filling null categorical variables when it generalizes to new data

In [79]:
```
# Create column transformer to handle numeric and categorical columns

from sklearn.compose import ColumnTransformer,make_column_transformer

scaled_preprocessing=ColumnTransformer(transformers=[
    ('num',num_transformer,num_cols),
    ('cat',cat_transformer,cat_cols)])
```
executed in 4ms, finished 13:52:37 2021-05-23

In [80]:
```
# Get X_train and X_test from column transformer
# Fit and transform on training data
# Transform on test data

X_train_p = scaled_preprocessing.fit_transform(X_train)
X_test_p = scaled_preprocessing.transform(X_test)
```
executed in 48ms, finished 13:52:37 2021-05-23

In [81]:
```
def get_column_names(train_matrix, train_df, test_matrix, test_df):
    """
    Creates dataframes with columns from original training data
    Parameters: Train data matrix, train data dataframe, test data matr
    Returns: 1 X_train dataframe with columns and 1 X_test dataframe wi
    """
    feature_names = scaled_preprocessing.named_transformers_['cat'].\
                named_steps['encoder'].get_feature_names(cat_cols).toli
    X_cols = num_cols+feature_names
    X_train_df = pd.DataFrame(train_matrix,columns=X_cols,index=train_d
    X_test_df = pd.DataFrame(test_matrix,columns=X_cols,index=test_df.i
    return X_train_df, X_test_df
```
executed in 3ms, finished 13:52:37 2021-05-23

In [82]:
```
# Run function to retrieve train and test data

X_train_s, X_test_s = get_column_names(X_train_p, X_train, X_test_p, X_
```
executed in 2ms, finished 13:52:37 2021-05-23

In [83]:
```
# Make it easy to pass in train and test data

train_test_s = [X_train_s, X_test_s, y_train, y_test]
```
executed in 2ms, finished 13:52:37 2021-05-23

SMOTE

- Given that we have class imbalance, will use SMOTE to balance target value distribution

In [84]:
```
df['FLAG'].value_counts(1)
```
executed in 4ms, finished 13:52:37 2021-05-23

Out[84]:
```
0   0.77802
1   0.22198
Name: FLAG, dtype: float64
```

For our models to provide accurate predictions, the target value be balanced 50-50

In [85]:
```
# Import SMOTE

from imblearn.over_sampling import SMOTE
```
executed in 90ms, finished 13:52:37 2021-05-23

In [86]:
```
sm = SMOTE(random_state=42)
```
executed in 2ms, finished 13:52:37 2021-05-23

In [87]:
```
X_train_sm, y_train_sm = sm.fit_resample(X_train_s, y_train)
```
executed in 81ms, finished 13:52:37 2021-05-23

In [88]:
```
# Ensure our model now has 50-50 balance

print(y_train_sm.value_counts(1))
print('------------------')
print(y_train_sm.value_counts())
```
executed in 4ms, finished 13:52:37 2021-05-23

```
1   0.50000
0   0.50000
Name: FLAG, dtype: float64
------------------
1    5338
0    5338
Name: FLAG, dtype: int64
```

In [89]:
```
print(X_train_sm.shape)
X_test_s.shape
```
executed in 3ms, finished 13:52:37 2021-05-23

```
(10676, 47)
```

Out[89]:
```
(2945, 47)
```

Dummy Baseline Model

- Compare to final model to see how well it performs
- Simply predicts target based on mean

In [90]:
```
# Create dummy model for baseline

from sklearn.dummy import DummyClassifier
dummy_clf = DummyClassifier(strategy='stratified', random_state=42)
dummy_clf.fit(X_train_sm,y_train_sm)
```
executed in 4ms, finished 13:52:37 2021-05-23

Out[90]:
```
DummyClassifier(random_state=42, strategy='stratified')
```

In [91]:
```
from sklearn.metrics import plot_roc_curve, auc, plot_confusion_matrix,
                            f1_score, recall_score, classification_repo



def evaluate_model(fit_model, X_train, X_test, y_train, y_test, cmap='B
    """
    Returns results of a fit model
    Parameters: fit model, training and test data, can optionally adjus
    Returns: Classification report, ROC curve, AUC score, confusion mat
    """

    y_hat_test = fit_model.predict(X_test)
    y_hat_train = fit_model.predict(X_train)
    print('********************CLASSIFICATION REPORT ******************
    print(classification_report(y_test, y_hat_test, target_names=['vali
    print('************************************************************

    fig, axes = plt.subplots(ncols=2, figsize=(10,4))
    plot_roc_curve(fit_model, X_test, y_test, ax=axes[0])
    plot_confusion_matrix(fit_model, X_test, y_test, cmap=cmap, normali
    plt.show()


    # Using F1 weighted because my data has a class imbalance
    recall_train=round(recall_score(y_train, y_hat_train, average='weig
    recall_test=round(recall_score(y_test, y_hat_test, average='weighte
    recall=round(recall_score(y_test, y_hat_test),5)

    print(f"Recall training score: {recall_train}")
    print(f"Recall test score: {recall_test}")
    if recall_train > recall_test:
        print(f"Overfit by +{round(recall_train-recall_test,5)}")
    else:
        print(f"Underfit by -{round(recall_train-recall_test,5)}")
    print('---------------')
    if dummy==True:
        pass
    else:
        print(f"Recall score improvement: +{recall-0.52}")

```

executed in 6ms, finished 13:52:37 2021-05-23

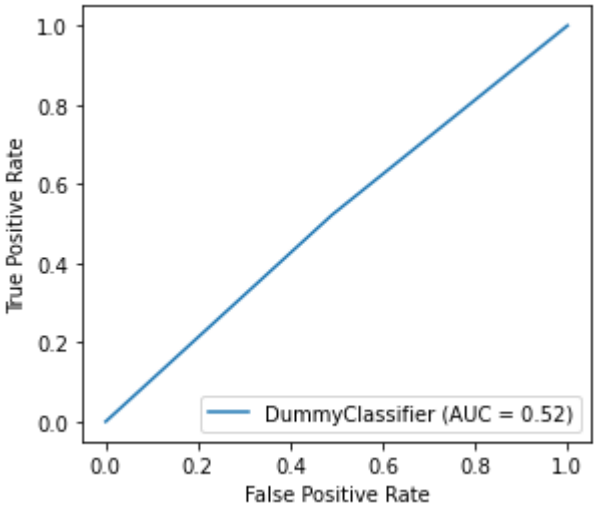
In [92]:
```
train_test_sm = [X_train_sm, X_test_s, y_train_sm, y_test]
```

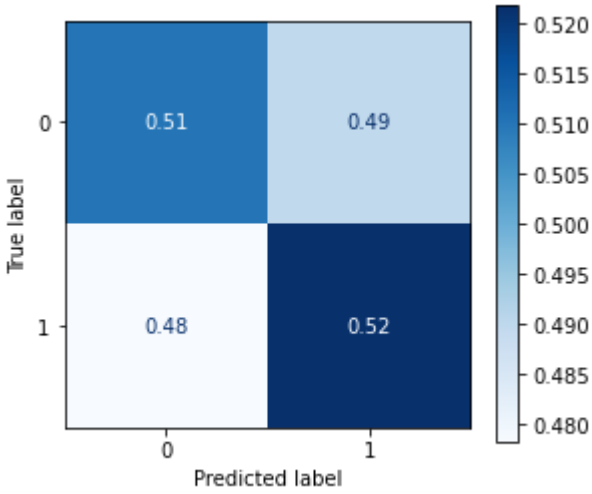
executed in 2ms, finished 13:52:37 2021-05-23

In [93]:
```
# Evaluate how the dummy model performs

evaluate_model(dummy_clf, *train_test_sm, dummy=True)
```

executed in 195ms, finished 13:52:37 2021-05-23

```
********************CLASSIFICATION REPORT ***************************
              precision    recall  f1-score   support

       valid       0.79      0.51      0.62      2299
       fraud       0.23      0.52      0.32       646

    accuracy                           0.51      2945
   macro avg       0.51      0.52      0.47      2945
weighted avg       0.67      0.51      0.55      2945

************************************************************************
```






```
Recall training score: 0.50178
Recall test score: 0.51273
Underfit by --0.01095
---------------
```

- Fraudulent recall score was 0.52, this is our most important classifier
- Out of all fraud cases, the model accurately predicted 52% of them
- Goal of future models to boost recall score of fraud (1)
- The model is underfit

K-Nearest Neighbors Model

- Assumptions: Data is scaled
- Begin with vanilla model and then tune using feature selection and GridSearchCV

Vanilla KNN

In [94]:
```
# Set up training dataset shortcut

train_sm = [X_train_sm, y_train_sm]
```

executed in 2ms, finished 13:52:37 2021-05-23

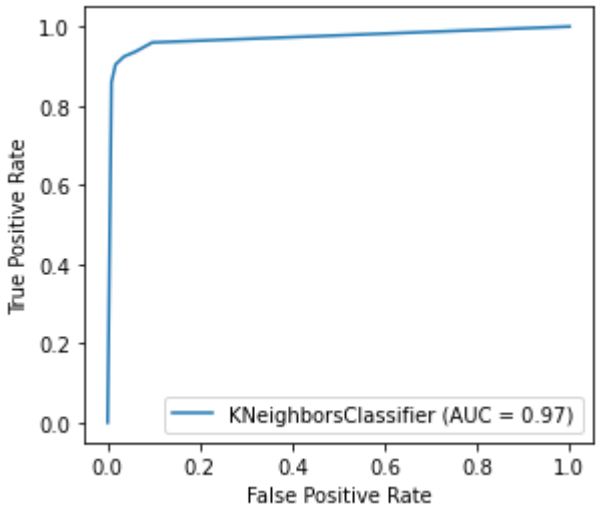
In [95]:
```
# Fit a vanilla KNN model

from sklearn.neighbors import KNeighborsClassifier

knn_clf = KNeighborsClassifier()
knn_clf.fit(*train_sm)
```

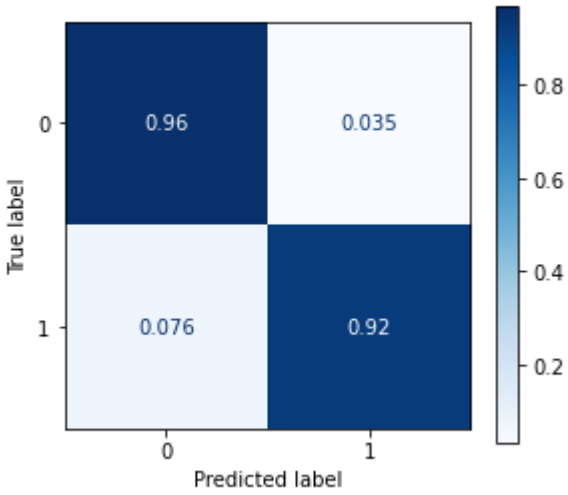
executed in 74ms, finished 13:52:37 2021-05-23

Out[95]:
```
KNeighborsClassifier()
```

In [96]:
```
1 evaluate_model(knn_clf, *train_test_sm)
```
executed in 3.60s, finished 13:52:41 2021-05-23

```
********************CLASSIFICATION REPORT ***************************
              precision    recall  f1-score   support

       valid       0.98      0.96      0.97      2299
       fraud       0.88      0.92      0.90       646

    accuracy                           0.96      2945
   macro avg       0.93      0.94      0.94      2945
weighted avg       0.96      0.96      0.96      2945

******************************************************************************
```






```
Recall training score: 0.98089
Recall test score: 0.95586
Overfit by +0.02503
---------------
Recall score improvement: +0.40415
```

- The KNN F1 scores and recall scores jumped significantly from the dummy model
- AUC score is 0.97, almost at 1.00, the max
- Out of all fraud cases, the model accurately predicted 92% of them
- **Next:** Perform feature selection to try and optimize recall

Testing for Feature Selection

In [97]:
```
1 # Bring in Variance Threshold
2
3 from sklearn.feature_selection import VarianceThreshold
```
executed in 9ms, finished 13:52:41 2021-05-23

In [98]:
```
## Check for uniform features

selector = VarianceThreshold(threshold=0.00)
selector.fit(X_train_sm)
```
executed in 8ms, finished 13:52:41 2021-05-23

Out[98]:
```
VarianceThreshold()
```

In [99]:
```
# Check columns for uniform features

keep_features = selector.get_support()
keep_features
```
executed in 3ms, finished 13:52:41 2021-05-23

Out[99]:
```
array([ True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True])
```

In [100]:
```
# Confirm no columns have uniform features

keep_features.sum()==len(X_train_sm.columns)
```
executed in 2ms, finished 13:52:41 2021-05-23

Out[100]:
```
True
```

Our model does not contain any uniform values, these would be useless for modeling purposes

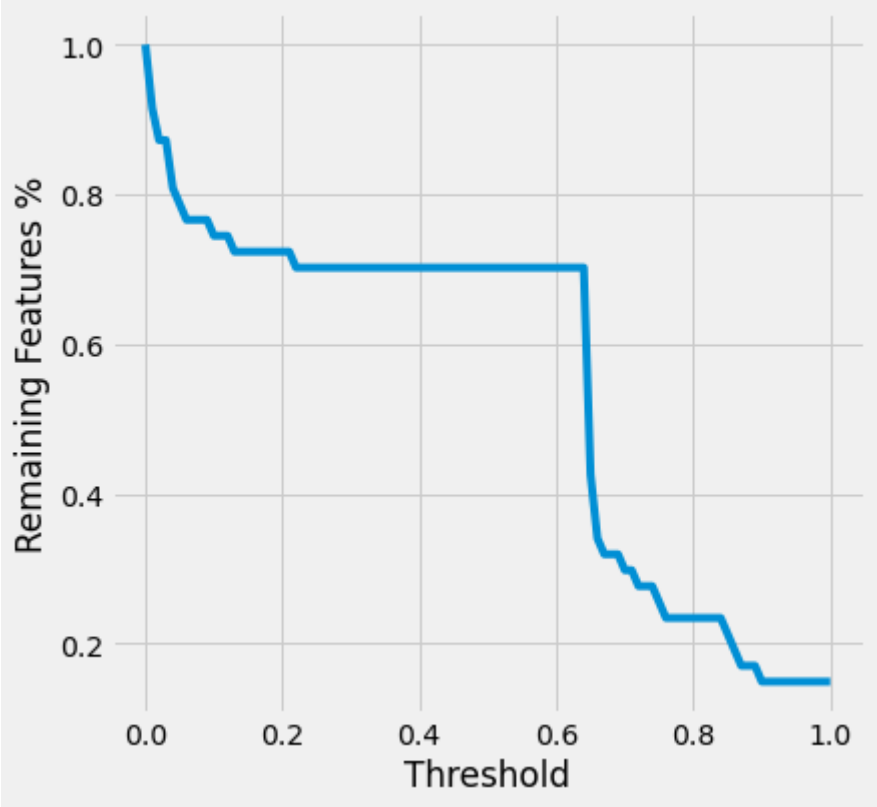
In [101]:
```
# Original number of columns

keep_features.sum()
```
executed in 2ms, finished 13:52:41 2021-05-23

Out[101]:
```
47
```

In [102]:
```
from sklearn.feature_selection import VarianceThreshold

def variance_thresh(X_train, thresh):
    """
    Produces number of remaining columns for given threshold using skle
    Parameters: X_train, threshold
    Returns: % of remaining features
    """
    selector = VarianceThreshold(threshold=thresh)
    selector.fit(X_train)
    keep_features = selector.get_support()
    remaining = (keep_features.sum()/len(X_train.columns))
    return remaining
```
executed in 2ms, finished 13:52:41 2021-05-23

In [103]:
```
# Create loop to show remaining number of columns at certain threshold

rem = []
for thresh in np.linspace(0,1,101):
    rem.append(variance_thresh(X_train_sm, thresh))
```
executed in 224ms, finished 13:52:41 2021-05-23

In [104]:
```
with plt.style.context('fivethirtyeight'):
    fig, ax = plt.subplots(figsize=(6,6))
    sns.lineplot(x=np.linspace(0,1,101), y=rem)
    ax.set_xlabel('Threshold')
    ax.set_ylabel('Remaining Features %');
```
executed in 78ms, finished 13:52:41 2021-05-23



A significant amount of my features are low variance because it takes a high threshold (compared to rule of thumb of 0.01) to remove just 30% of my features (0.2 threshold would achieve this). Looks like the optimal tradeoff between feature removal and threshold is between 0.01-0.05

In [105]:
```
# Instantiate Variance threshold

selector = VarianceThreshold(threshold=0.01)
selector.fit(X_train_sm)
```
executed in 5ms, finished 13:52:41 2021-05-23

Out[105]:
```
VarianceThreshold(threshold=0.01)
```

In [106]:
```
# Check number of columns remaining

keep_features = selector.get_support()
keep_features.sum()
```

executed in 2ms, finished 13:52:41 2021-05-23

Out[106]:
```
43
```

In [107]:
```
# Manipulate DF to include new columns after feature selection

X_train_sel = X_train_sm.loc[:,keep_features]
X_test_sel = X_test_s.loc[:,keep_features]
```

executed in 3ms, finished 13:52:41 2021-05-23

In [108]:
```
# Fit model with new columns

knn_clf.fit(X_train_sel, y_train_sm)
```

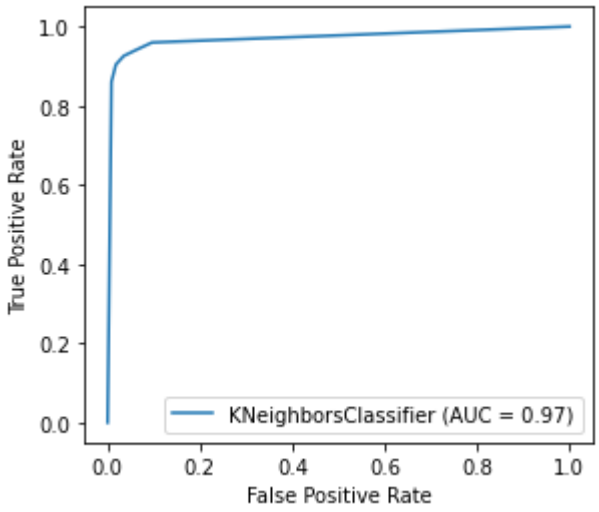
executed in 82ms, finished 13:52:41 2021-05-23

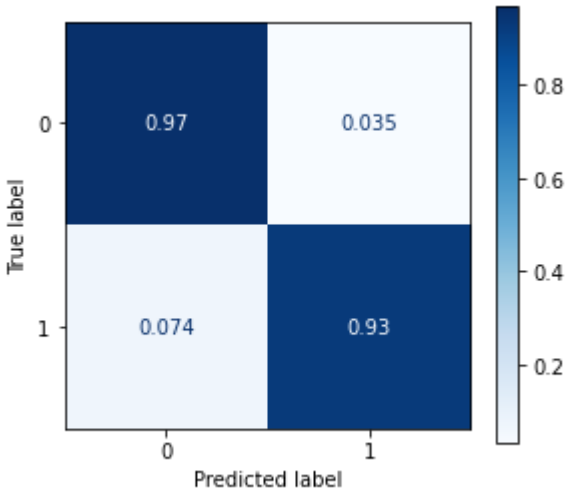
Out[108]:
```
KNeighborsClassifier()
```

In [109]:
```
# Evaluate model performance with feature selection

evaluate_model(knn_clf, X_train_sel, X_test_sel, y_train_sm, y_test)
```

executed in 3.35s, finished 13:52:45 2021-05-23

```
********************CLASSIFICATION REPORT ***************************
              precision    recall  f1-score   support

       valid       0.98      0.97      0.97      2299
       fraud       0.88      0.93      0.90       646

    accuracy                           0.96      2945
   macro avg       0.93      0.95      0.94      2945
weighted avg       0.96      0.96      0.96      2945

**************************************************************************
```






```
Recall training score: 0.98089
Recall test score: 0.95654
Overfit by +0.02435
---------------
Recall score improvement: +0.40569999999999995
```

- Features have been removed and recall score has gone up which is the metric we are looking to optimize
- Recall score improved from 0.92 to 0.93

In [110]:
```
# Models dropped from original to feature selection training data

set(X_train_sm.columns).difference(set(X_train_sel))
```

executed in 2ms, finished 13:52:45 2021-05-23

Out[110]:
```
{'weights_rec_mid',
 'weights_sent_mid',
 'weights_sent_small',
 'weights_sent_small_mid'}
```

- Proceed with this model because improved score

- Dropping the smaller market cap values from the feature that we engineered
- Unlike linear regression, it is acceptable to drop certain columns because not concerned with multicollinearity

KNN Tuning

- Prioritizing recall because cost of false positive is low but cost of false negative is very high
- Sacrafising a balance of F1 to achieve a higher recall score
- Test out various hyperparamaters in an interative fashion to find the model that produces the highest recall score

In [111]:
```
from sklearn.model_selection import GridSearchCV

def grid_search(clf, param_grid, X_train, y_train,cv=3, scoring='recall
    """
    Returns dictionary containing dataframe and best parameters
    Parameters: classifcation model, parameter_grid for grid search, X_
    Output: Dictionary containing sorted dataframe and best paramaeters
    Keys: dataframe, best_params
    """
    results_dict={}

    grid_search = GridSearchCV(clf, param_grid, cv=cv, scoring=scoring)
    grid_search.fit(X_train, y_train)
    grid_search_df = pd.DataFrame(grid_search.cv_results_)
    cols_to_drop = ['mean_fit_time', 'std_fit_time', 'mean_score_time',
    grid_search_df = grid_search_df.drop(columns=cols_to_drop, axis=1)

    results_dict['dataframe'] = grid_search_df.sort_values(by='rank_tes
    results_dict['best_params'] = grid_search.best_params_

    return results_dict
```

executed in 3ms, finished 13:52:45 2021-05-23

In [112]:
```
# Set up parameter grid

knn_param_grid = {
    'n_neighbors': [2,5,10,15,30],
    'weights': ['uniform', 'distance'],
    'metric':['euclidean', 'manhattan']}
```

executed in 2ms, finished 13:52:45 2021-05-23

In [113]:
```
# Set up lists to make parameters easier to fill in

train_test_sel = [X_train_sel, X_test_sel, y_train_sm, y_test]
train_sel = [X_train_sel, y_train_sm]
```

executed in 1ms, finished 13:52:45 2021-05-23

In [114]:
```
# Run function to create dictionary of results
# Optimizing for recall

knn_res = grid_search(knn_clf, knn_param_grid, *train_sel, scoring='rec
```

executed in 28.5s, finished 13:53:13 2021-05-23

In [115]:
```
knn_res['dataframe']
```

executed in 7ms, finished 13:53:13 2021-05-23

Out[115]:

	param_metric	param_n_neighbors	param_weights	split0_test_score	split1_test_score	split2_tes
11	manhattan	2	distance	0.98426	0.99270	
1	euclidean	2	distance	0.98482	0.99045	
13	manhattan	5	distance	0.97976	0.98989	
3	euclidean	5	distance	0.97695	0.98652	
15	manhattan	10	distance	0.97527	0.98652	
17	manhattan	15	distance	0.96965	0.98371	
5	euclidean	10	distance	0.96740	0.98483	
12	manhattan	5	uniform	0.96571	0.98258	
2	euclidean	5	uniform	0.96178	0.98258	
10	manhattan	2	uniform	0.95728	0.98315	
19	manhattan	30	distance	0.96571	0.97865	
7	euclidean	15	distance	0.96178	0.97865	
0	euclidean	2	uniform	0.95278	0.97809	
14	manhattan	10	uniform	0.95559	0.97472	
16	manhattan	15	uniform	0.95559	0.97247	
9	euclidean	30	distance	0.95391	0.96854	
6	euclidean	15	uniform	0.94829	0.96404	
4	euclidean	10	uniform	0.94435	0.96573	
18	manhattan	30	uniform	0.94660	0.95337	
8	euclidean	30	uniform	0.93367	0.94831	

- Distance seems to be the most effective weight as the top 7 results all use that metric
- The top neighbors are between 2-10
- Manhattan is slightly better than euclidean because when using the same measurement, manhattan performs better

In [116]:
```
1 knn_res['best_params']
```
executed in 2ms, finished 13:53:13 2021-05-23

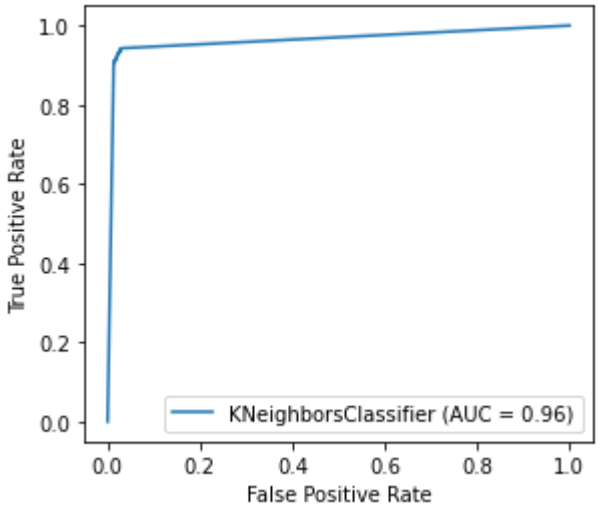
Out[116]:
```
{'metric': 'manhattan', 'n_neighbors': 2, 'weights': 'distance'}
```

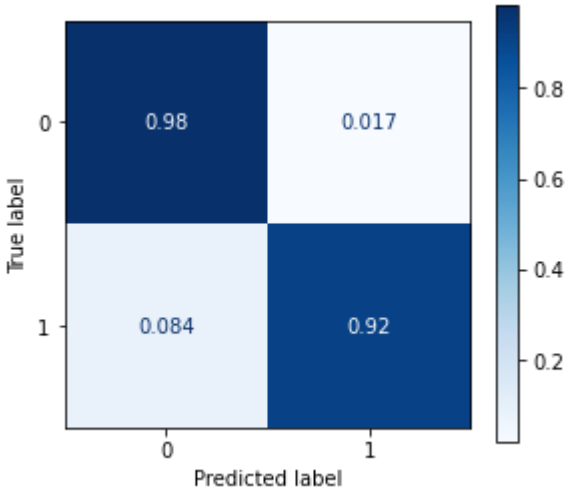
In [117]:
```
1 knn_clf2 = KNeighborsClassifier(n_neighbors=2, weights='distance', metr
2 knn_clf2.fit(X_train_sel, y_train_sm)
```
executed in 84ms, finished 13:53:13 2021-05-23

Out[117]:
```
KNeighborsClassifier(metric='manhattan', n_neighbors=2, weights='distanc
e')
```

In [118]:
```
1 evaluate_model(knn_clf2, X_train_sel, X_test_sel, y_train_sm, y_test)
```
executed in 2.66s, finished 13:53:16 2021-05-23

```
********************CLASSIFICATION REPORT ***************************
              precision    recall  f1-score   support

       valid       0.98      0.98      0.98      2299
       fraud       0.94      0.92      0.93       646

    accuracy                           0.97      2945
   macro avg       0.96      0.95      0.95      2945
weighted avg       0.97      0.97      0.97      2945

**********************************************************************
```






```
Recall training score: 1.0
Recall test score: 0.96808
Overfit by +0.03192
---------------
Recall score improvement: +0.39640999999999993
```

- Recall score has descreased from 0.93 to 0.92
- AUC score has decreased as well

In [119]:
```
# Manhattan is the best metric, test out different k values and weights

knn_param_grid = {
    'n_neighbors': [2,3,4,5,6,7,8,9,10],
    'weights': ['distance'],
    'metric':['manhattan']}
```
executed in 2ms, finished 13:53:16 2021-05-23

In [120]:
```
knn_res2 = grid_search(knn_clf, knn_param_grid, X_train_sel, y_train_sm
```
executed in 11.9s, finished 13:53:28 2021-05-23

In [121]:
```
knn_res2['dataframe']
```
executed in 5ms, finished 13:53:28 2021-05-23

Out[121]:

	param_metric	param_n_neighbors	param_weights	split0_test_score	split1_test_score	split2_test_
0	manhattan	2	distance	0.98426	0.99270	0.
1	manhattan	3	distance	0.98426	0.99157	0.
2	manhattan	4	distance	0.98201	0.99045	0.
3	manhattan	5	distance	0.97976	0.98989	0.
4	manhattan	6	distance	0.97976	0.98876	0.
6	manhattan	8	distance	0.97695	0.98764	0.
5	manhattan	7	distance	0.97639	0.98876	0.
7	manhattan	9	distance	0.97527	0.98652	0.
8	manhattan	10	distance	0.97527	0.98652	0.

In [122]:
```
knn2 = knn_res2['best_params']
```
executed in 1ms, finished 13:53:28 2021-05-23

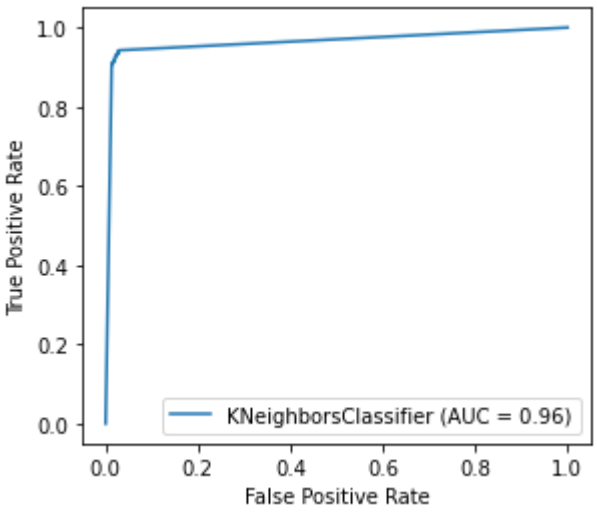
In [123]:
```
# Instantiate KNN with best params
# Train on filtered data

knn3 = KNeighborsClassifier(**knn_res2['best_params'])
knn3.fit(X_train_sel, y_train_sm)
```
executed in 82ms, finished 13:53:28 2021-05-23

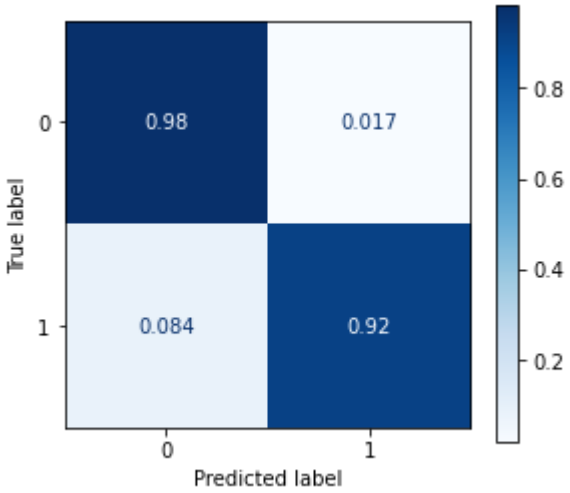
Out[123]:
```
KNeighborsClassifier(metric='manhattan', n_neighbors=2, weights='distanc
e')
```

In [124]:
```
# Evaluate performance

evaluate_model(knn3, X_train_sel, X_test_sel, y_train_sm, y_test)
```

executed in 2.58s, finished 13:53:31 2021-05-23

```
********************CLASSIFICATION REPORT ***************************
              precision    recall  f1-score   support

       valid       0.98      0.98      0.98      2299
       fraud       0.94      0.92      0.93       646

    accuracy                           0.97      2945
   macro avg       0.96      0.95      0.95      2945
weighted avg       0.97      0.97      0.97      2945

**********************************************************************
```






```
Recall training score: 1.0
Recall test score: 0.96808
Overfit by +0.03192
---------------
Recall score improvement: +0.39640999999999993
```

- Score is the same as prior
- Seems we have reached a ceiling with KNN as it is overfitting
- 2 seems to be the optimal number of neighbors
- **Model Considerations:** Tried feature selection and hyperparameter tuning. Model is overfit by about 3%

KNN Conclusions

- Overall, compared to the dummy, the KNN model performed very well on the training and test set
- In the dummy, the model detected fraud 52% of the time, in the final KNN model, it was able to accurately detect fraud 92% of the time, a 40% improvement
- Out of all valid transactions, it accurately predicted 98%
  - Of the 2% that are misclassified, the only harm is that the customer will receive a notification saying that the transaction they are initiating may be fraudulent
  - They can confirm or deny this and proceed to send the funds
- For fraudulant cases, the median of the average value sent in a transaction is 0.5 ETH. The 200 moving day average (the average price of ETH the past 200 days) is $1,528
  - While there is no official number, BTC and ETH have clearly been the target of hacks and we shall say that 5% of all wallets will be hacked at some point in existance
  - If a wallet service has 10,000 users, it can expect 500 of them to experience some sort of fraud. Given that the median average value sent out per accounts that experience fraud is 0.5 ETH, these customers are losing $764
  - The dummy model can predict fraud 50% of the time. So 250 people will experience fraud costing them $764 each for a total loss of $191,000
  - Of course, this is bad business for the wallet company, the users are highly likely to transfer their funds to another wallet service after experiencing fraud
  - If the company were to use the KNN model that we have tuned, it would detect fraud 92% of the time
    - Out of the 500 customers who would hypothetically experience fraud, only 40 customers would experience fraud because the model would notify the customer before sending the suspicious transaction
    - 40 customers causes a total loss of $30,560
- **Conclusion**: Out of 10,000 customers, the company can expect that 500 of them will be targeted for fraud. Using the model, rather than 250 of them being impacted by fraud, only 30 of them will be
  - This reduces potential losses from $191,000 to $30,560, an 84% decrease in costs
  - Additinally, 88% fewere customers will be affected by fraud
  - The average expected loss by fraud is reduced from $19.11 to $3.05

Random Forest Model

- Using the Random Forest model because it is more resistant to variance and outliers
- Ensemble method so in theory it should provide strong results
- Still interpretable and can provide conclusions

Preprocessing Pipeline

- Data does not need to be scaled
- For the Random Forest Model, reprocessing the data with out the Standard Scaler
- Prefer not to use the Standard Scaler because it is less interpretable

In [125]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import BaggingClassifier, RandomForestClassifier

print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
```

executed in 2ms, finished 13:53:31 2021-05-23

```
(6871, 35)
(2945, 35)
(6871,)
(2945,)
```

Data does not need to be scaled and RandomForestClassifier object accepts a 'class weight' balance so will not need to use smote

In [126]:
```
# Pull out numeric columns
# Pull out categorical columns

num_cols = X_train.select_dtypes('number').columns.to_list()
cat_cols = X_train.select_dtypes('O').columns.tolist()
```

executed in 4ms, finished 13:53:31 2021-05-23

In [127]:
```
# Use median as default strategy

num_transformer = Pipeline(steps=[
    ('imputer',SimpleImputer(strategy='median'))])
```

executed in 2ms, finished 13:53:31 2021-05-23

In [128]:
```
# Create categorical imputer
# Do not need to drop because RF can handle multicollinearity

cat_transformer = Pipeline(steps=[
    ('imputer',SimpleImputer(strategy='most_frequent')),
    ('encoder',OneHotEncoder(sparse=False,handle_unknown='ignore'))])
```

executed in 2ms, finished 13:53:31 2021-05-23

In [129]:
```
# Create column transformer to handle numeric and categorical columns

preprocessing=ColumnTransformer(transformers=[
    ('num',num_transformer,num_cols),
    ('cat',cat_transformer,cat_cols)])
```

executed in 2ms, finished 13:53:31 2021-05-23

In [130]:
```
# Fit transform training data
# Transform testing data

X_train_p = preprocessing.fit_transform(X_train)
X_test_p = preprocessing.transform(X_test)
```

executed in 40ms, finished 13:53:31 2021-05-23

In [131]:
```
def get_column_names(column_transformer, train_matrix, train_df, test_m
    """
    Creates dataframes with columns from original training data
    Parameters: Train data matrix, train data dataframe, test data matr
    Returns: 1 X_train dataframe with columns and 1 X_test dataframe wi
    """
    feature_names = preprocessing.named_transformers_['cat'].\
                named_steps['encoder'].get_feature_names(cat_cols).toli
    X_cols = num_cols+feature_names
    X_train_df = pd.DataFrame(train_matrix,columns=X_cols,index=train_d
    X_test_df = pd.DataFrame(test_matrix,columns=X_cols,index=test_df.i
    return X_train_df, X_test_df
```

executed in 2ms, finished 13:53:31 2021-05-23

In [132]:
```
# Insert names into matrices and transform into DataFrame

X_train_ns, X_test_ns = get_column_names(preprocessing, X_train_p, X_tr
```

executed in 2ms, finished 13:53:31 2021-05-23

In [133]:
```
# Make sure column names appear

X_test_ns.head()
```

executed in 28ms, finished 13:53:31 2021-05-23

Out[133]:

	Avg_min_between_sent_tnx	Avg_min_between_received_tnx	Time_diff_first_last	Sent_tnx	Rece
7743	0.00000	339.31000	193813.48000	1.00000	
3919	3.50000	3886.85000	7780.70000	2.00000	
9339	0.00000	261.82000	23289.00000	1.00000	
3618	154.92000	0.00000	309.83000	2.00000	
5863	0.00000	0.00000	1.57000	1.00000	

In [134]:
```
# Make sure column names appear

X_train_ns.head()
```

executed in 18ms, finished 13:53:31 2021-05-23

Out[134]:

	Avg_min_between_sent_tnx	Avg_min_between_received_tnx	Time_diff_first_last	Sent_tnx	Rec
5584	161.45000	0.82000	324.53000	2.00000	
3828	2222.71000	7491.53000	137398.70000	18.00000	
3400	53.34000	823.96000	275042.98000	306.00000	
1952	0.00000	2151.61000	200099.62000	0.00000	
555	4790.91000	4690.94000	81145.63000	14.00000	

Vanilla RFM

- Evaluate performance of Vanilla RFM to get a baseline how the model is performing
- KNN vanilla performance was ~91% recall on the fraud (1) cases

In [135]:
```
# Instantiate Vanilla RFM
# Fit the model

rf_clf = RandomForestClassifier(random_state=42, class_weight='balanced
rf_clf.fit(X_train_ns, y_train)
```

executed in 817ms, finished 13:53:31 2021-05-23

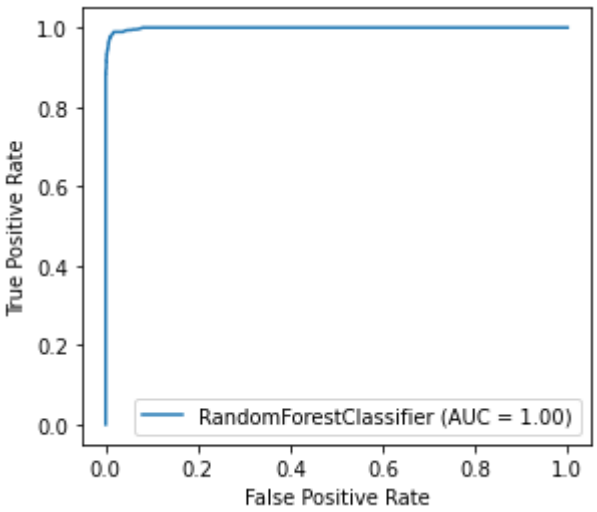
Out[135]:
```
RandomForestClassifier(class_weight='balanced', random_state=42)
```

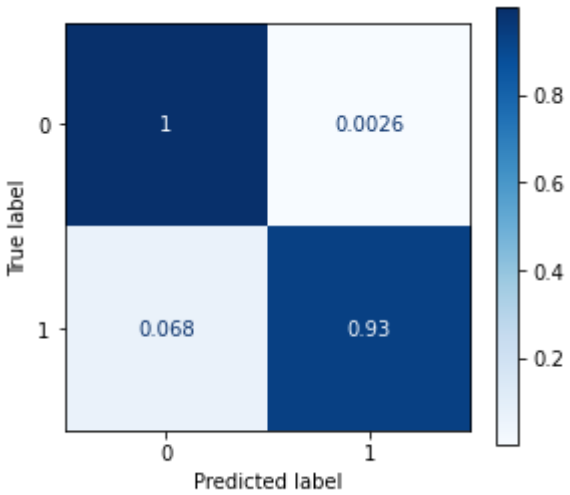
Rather than using SMOTE to create synthetic observations, can you use RF built in class_weight metric

In [136]:
```
evaluate_model(rf_clf, X_train_ns, X_test_ns, y_train, y_test)
```

executed in 302ms, finished 13:53:32 2021-05-23

```
********************CLASSIFICATION REPORT ***************************
              precision    recall  f1-score   support

       valid       0.98      1.00      0.99      2299
       fraud       0.99      0.93      0.96       646

    accuracy                           0.98      2945
   macro avg       0.99      0.96      0.97      2945
weighted avg       0.98      0.98      0.98      2945

**************************************************************************
```






```
Recall training score: 1.0
Recall test score: 0.98302
Overfit by +0.01698
---------------
Recall score improvement: +0.41189
```

- The model had a recall on the fraud test data of 93%
  - Out of 100 fraudulent transactions, the model was able to accurately detect 93% of them

- The model is very slightly slightly overfit, about 1.5%
- Recall score moved up about 41% compared to the dummy variable
- Optimizing for Fraud Recall, because very high cost of false negatives
- AUC score is 1.0 which is the maximum score
- Vanilla RFM is already performing as well as the top KNN model
- **Next Steps:** Use Feature Selection to try and optimize recall score

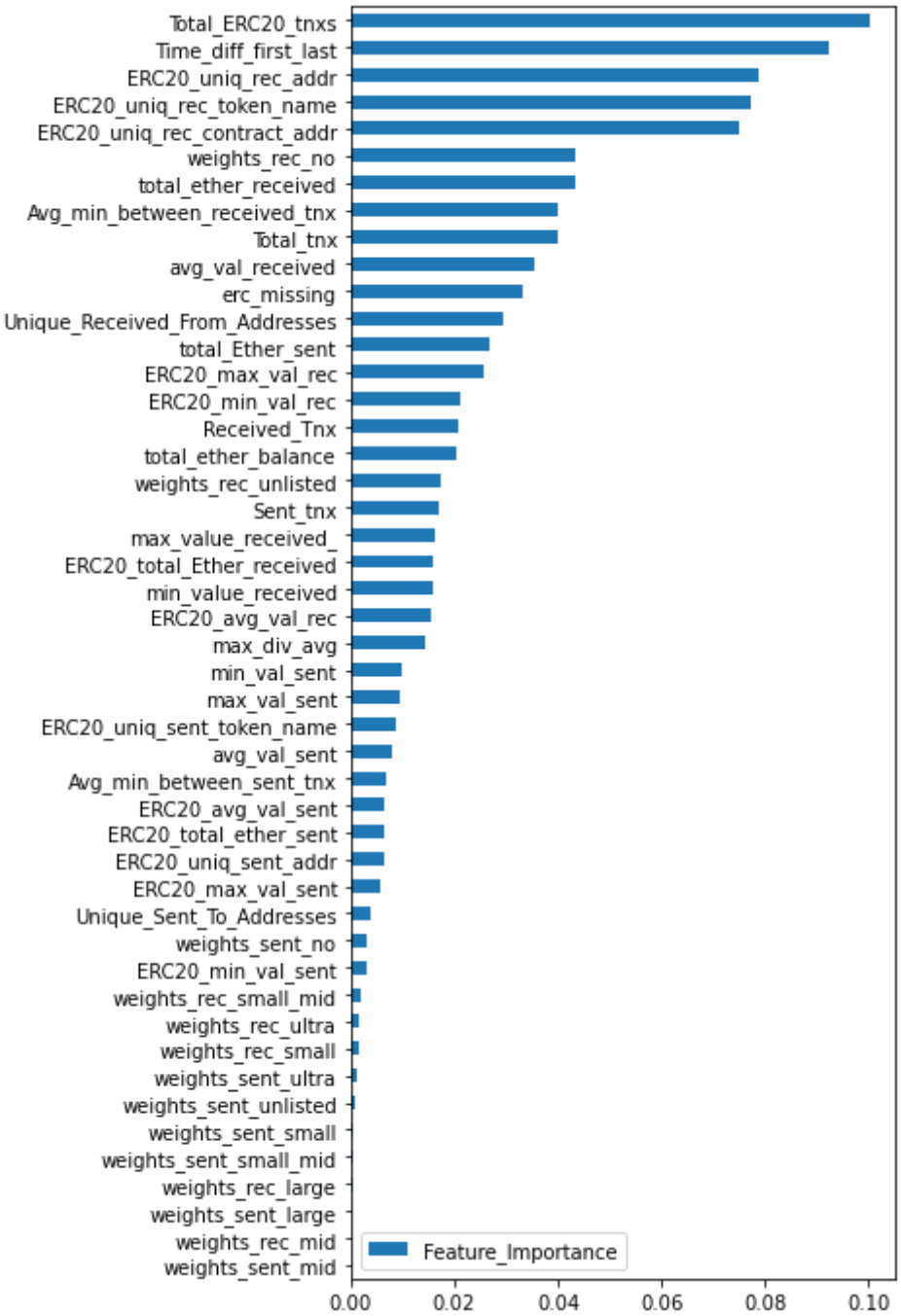
RFM Feature Selection

- Use a combination of feature selection and Permuation Importance to filter out unncessary columns
- Permutation Importance is not a tool for feature selection but it classifies which variables have a large impact on recall (if specified)
- Before dropping columns based on Feature Importance, ensure that columns are not making a signficant impact on recall score
- Filtering out columns may improve recall score because there may be less confounding variables

In [137]:
```
# Plot feature importance
# Reference: https://stackoverflow.com/questions/44101458/random-forest

feat_importances = pd.Series(rf_clf.feature_importances_, index=X_train
feat_importances = feat_importances.rename(columns={0:'Feature_Importan
feat_importances.sort_values(by='Feature_Importance',ascending=True).pl
```

executed in 370ms, finished 13:53:32 2021-05-23

Out[137]: <AxesSubplot:>



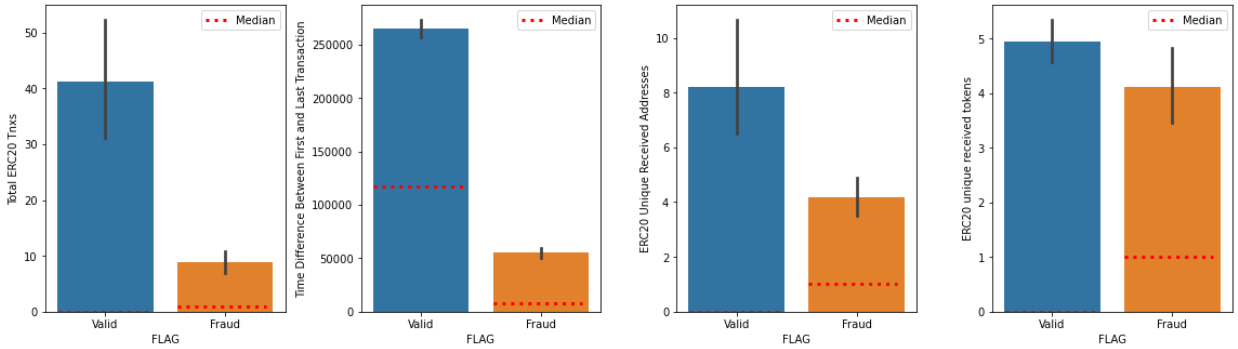
- Feature important tells us how important a variable is for predicting a target
- The maximum feature importance is 0.10
- Total_ERC_20_tnxs, time_diff_first_last, ERC20_uniq_rec_token_name, ERC20_uniq_rec_contract_address
- After ERC20_uniq_rec_contract_address the feature importance dips from aprox 0.08 to aprox 0.04

In [138]:

```
# Evaluate how the most important features compare in fraudent vs valid

fig, axes = plt.subplots(ncols=4, figsize=(16,5))

fig.suptitle('Comparing Most Important Features')

# Total ERC20 tnx
v_t_med = df_valid['Total_ERC20_tnxs'].median()
f_t_med = df_fraud['Total_ERC20_tnxs'].median()

# Time Difference Between First and Last Transaction
v_tdif_med = df_valid['Time_diff_first_last'].median()
f_tdif_med = df_fraud['Time_diff_first_last'].median()

# ERC20 Unique Addresses
v_ua_med = df_valid['ERC20_uniq_rec_addr'].median()
f_ua_med = df_fraud['ERC20_uniq_rec_addr'].median()

# ERC20 Unique Received Tokens
v_rt_med = df_valid['ERC20_uniq_rec_token_name'].median()
f_rt_med = df_fraud['ERC20_uniq_rec_token_name'].median()

sns.barplot(data=df, x='FLAG', y='Total_ERC20_tnxs', ax=axes[0])
axes[0].set_xticklabels(['Valid', 'Fraud'])
axes[0].set_ylabel('Total ERC20 Tnxs')
axes[0].axhline(v_t_med, xmin=0.053, xmax=0.445, color='red', ls=':', l
axes[0].axhline(f_t_med, xmin=0.55, xmax=0.92, color='red', ls=':', lin
axes[0].legend()

sns.barplot(data=df, x='FLAG', y='Time_diff_first_last', ax=axes[1])
axes[1].set_xticklabels(['Valid', 'Fraud'])
axes[1].set_ylabel('Time Difference Between First and Last Transaction'
axes[1].axhline(v_tdif_med, xmin=0.053, xmax=0.445, color='red', ls=':'
axes[1].axhline(f_tdif_med, xmin=0.55, xmax=0.92, color='red', ls=':',
axes[1].legend()

sns.barplot(data=df, x='FLAG', y='ERC20_uniq_rec_addr', ax=axes[2])
axes[2].set_xticklabels(['Valid', 'Fraud'])
axes[2].set_ylabel('ERC20 Unique Received Addresses')
axes[2].axhline(v_ua_med, xmin=0.053, xmax=0.445, color='red', ls=':',
axes[2].axhline(f_ua_med, xmin=0.55, xmax=0.92, color='red', ls=':', li
axes[2].legend()

sns.barplot(data=df, x='FLAG', y='ERC20_uniq_rec_token_name', ax=axes[3
axes[3].set_xticklabels(['Valid', 'Fraud'])
axes[3].set_ylabel('ERC20 unique received tokens')
axes[3].axhline(v_rt_med, xmin=0.053, xmax=0.445, color='red', ls=':',
axes[3].axhline(f_rt_med, xmin=0.55, xmax=0.92, color='red', ls=':', li
axes[3].legend()

fig.tight_layout()
```

executed in 563ms, finished 13:53:33 2021-05-23

Comparing Most Important Features



- Total ERC20 Tnxs: As we can see, the mean for total ERC20 tnxs is about 4x higher for valid versus fraudulent wallets
- Time difference between first and last transaction: On average, valid transactions have about a 5x higher time difference between first and last transactions
  - Equates to about 150 days to 30 days
- ERC20 Unique Received Addresses: About 2x as high for valid versus fraudulent wallets
- ERC20 Unique Received Tokens: Roughly the same, valid is slightly higher
- Overall, interesting to note that many of the top features for model classification are regarding ERC20 transactions as opposed to standard Ethereum transactions
- The median gives us a sense that the distribution contains lots of outliers

In [139]:

```
# Fit permuation_importance to see which features have the highest affe

from sklearn.inspection import permutation_importance
r = permutation_importance(rf_clf, X_test_ns, y_test, scoring='recall',
```

executed in 5.23s, finished 13:53:38 2021-05-23

In [140]:
```
# Turn values into DataFrame so we can plot it

importance_df = pd.Series(r['importances_mean'], index=X_test_ns.column
importance_df = importance_df.rename(columns={0:'PI Mean'})
importance_df.sort_values(by='PI Mean', ascending=False)
```

executed in 5ms, finished 13:53:38 2021-05-23

Out[140]:

	PI Mean
Unique_Received_From_Addresses	0.03189
Time_diff_first_last	0.01517
Total_ERC20_tnxs	0.01517
ERC20_uniq_rec_contract_addr	0.01053
ERC20_uniq_rec_addr	0.01022
ERC20_uniq_rec_token_name	0.00743
Avg_min_between_received_tnx	0.00712
avg_val_received	0.00372
total_ether_received	0.00279
Received_Tnx	0.00248
ERC20_uniq_sent_token_name	0.00248
max_div_avg	0.00186
min_value_received	0.00186
min_val_sent	0.00186
weights_sent_unlisted	0.00155
ERC20_uniq_sent_addr	0.00155
ERC20_total_Ether_received	0.00124
weights_rec_unlisted	0.00124
ERC20_max_val_rec	0.00093
ERC20_min_val_rec	0.00093
weights_rec_ultra	0.00062
ERC20_avg_val_sent	0.00062
max_val_sent	0.00031
Unique_Sent_To_Addresses	0.00031
weights_rec_large	0.00000
weights_rec_mid	0.00000
weights_sent_ultra	0.00000
weights_sent_small_mid	0.00000
weights_sent_small	0.00000
weights_sent_mid	0.00000

	PI Mean
weights_sent_large	0.00000
weights_rec_small_mid	0.00000
weights_rec_small	0.00000
weights_rec_no	0.00000
Avg_min_between_sent_tnx	-0.00031
ERC20_max_val_sent	-0.00031
total_Ether_sent	-0.00031
Sent_tnx	-0.00062
max_value_received_	-0.00093
avg_val_sent	-0.00093
ERC20_total_ether_sent	-0.00093
erc_missing	-0.00124
ERC20_min_val_sent	-0.00124
ERC20_avg_val_rec	-0.00124
total_ether_balance	-0.00155
weights_sent_no	-0.00155
Total_tnx	-0.00155

In [141]:
```
# Merge feature importance with Permutation Importance

importance_df = importance_df.reset_index(drop=True)
feat_importances = feat_importances.reset_index(drop=True)
df_importance = pd.concat([importance_df, feat_importances],axis=1)
```

executed in 2ms, finished 13:53:38 2021-05-23

In [142]:
```
# Add indices

df_importance=df_importance.set_index(X_train_ns.columns)
df_importance.head()
df_importance.reset_index()

```

executed in 8ms, finished 13:53:38 2021-05-23

Out[142]:

	index	PI Mean	Feature_Importance
0	Avg_min_between_sent_tnx	-0.00031	0.00681
1	Avg_min_between_received_tnx	0.00712	0.03998
2	Time_diff_first_last	0.01517	0.09219
3	Sent_tnx	-0.00062	0.01704
4	Received_Tnx	0.00248	0.02074
5	Unique_Received_From_Addresses	0.03189	0.02949
6	Unique_Sent_To_Addresses	0.00031	0.00381
7	min_value_received	0.00186	0.01567
8	max_value_received_	-0.00093	0.01629
9	avg_val_received	0.00372	0.03562
10	min_val_sent	0.00186	0.00987
11	max_val_sent	0.00031	0.00945
12	avg_val_sent	-0.00093	0.00785
13	Total_tnx	-0.00155	0.03992
14	total_Ether_sent	-0.00031	0.02675
15	total_ether_received	0.00279	0.04351
16	total_ether_balance	-0.00155	0.02023
17	Total_ERC20_tnxs	0.01517	0.10015
18	ERC20_total_Ether_received	0.00124	0.01575
19	ERC20_total_ether_sent	-0.00093	0.00646
20	ERC20_uniq_sent_addr	0.00155	0.00641
21	ERC20_uniq_rec_addr	0.01022	0.07868
22	ERC20_uniq_rec_contract_addr	0.01053	0.07495
23	ERC20_min_val_rec	0.00093	0.02110
24	ERC20_max_val_rec	0.00093	0.02572
25	ERC20_avg_val_rec	-0.00124	0.01548
26	ERC20_min_val_sent	-0.00124	0.00287
27	ERC20_max_val_sent	-0.00031	0.00552
28	ERC20_avg_val_sent	0.00062	0.00647

	index	PI Mean	Feature_Importance
29	ERC20_uniq_sent_token_name	0.00248	0.00886
30	ERC20_uniq_rec_token_name	0.00743	0.07720
31	erc_missing	-0.00124	0.03313
32	max_div_avg	0.00186	0.01419
33	weights_rec_large	0.00000	0.00023
34	weights_rec_mid	0.00000	0.00015
35	weights_rec_no	0.00000	0.04351
36	weights_rec_small	0.00000	0.00138
37	weights_rec_small_mid	0.00000	0.00197
38	weights_rec_ultra	0.00062	0.00140
39	weights_rec_unlisted	0.00124	0.01748
40	weights_sent_large	0.00000	0.00015
41	weights_sent_mid	0.00000	0.00007
42	weights_sent_no	-0.00155	0.00301
43	weights_sent_small	0.00000	0.00036
44	weights_sent_small_mid	0.00000	0.00029
45	weights_sent_ultra	0.00000	0.00121
46	weights_sent_unlisted	0.00155	0.00065

In [143]:
```
# Melt so we can include hue value for feature importance ranking

df_melted = pd.melt(df_importance.reset_index(), value_vars=['PI Mean',
```

executed in 5ms, finished 13:53:38 2021-05-23

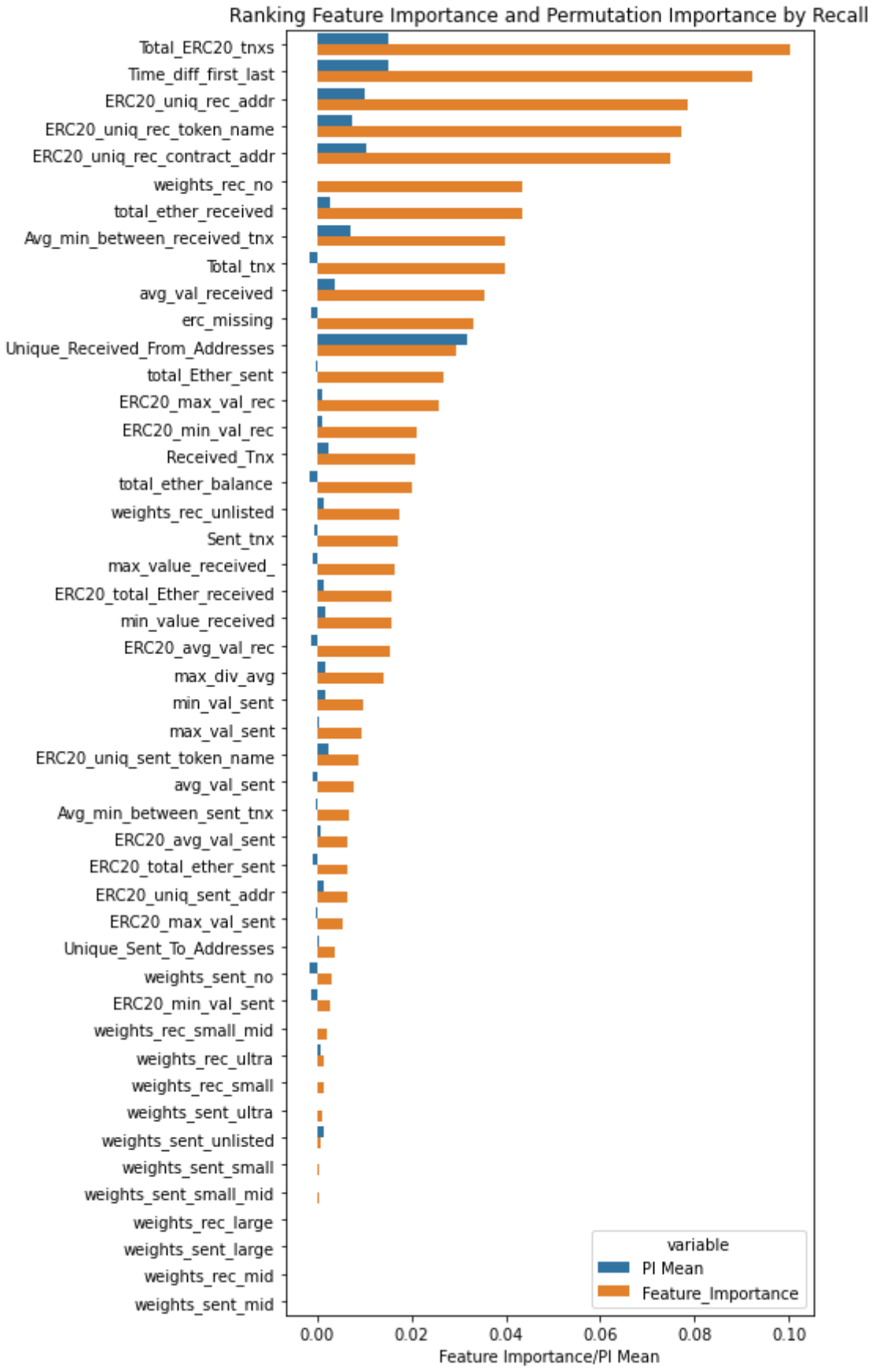
In [144]:
```
# Order by feature importance

ordered_list=df_importance.sort_values(by='Feature_Importance', ascendi
```

executed in 2ms, finished 13:53:38 2021-05-23

In [145]:

```
fig, ax = plt.subplots(figsize=(6,15))
sns.barplot(data=df_melted, y='index', x='value', hue='variable', orien
ax.set_ylabel('')
ax.set_xlabel('Feature Importance/PI Mean')
plt.title('Ranking Feature Importance and Permutation Importance by Rec
```

executed in 650ms, finished 13:53:39 2021-05-23



- The top columns in feature selection are also the top columns for Permutation Importance when focusing on recall
- Unique received from address has the most disparity between importance levels
- A few of the features have a negative impact on recall

- In general, the magnitude of PI is much lower than the RF Feature Importance
- Out of 47 columns, will try dropping 20% and see if my model has a major difference in performance
  - Equates to 10 columns
  - Compare score before and after drop

In [146]:
```
# Drop 10 lowest feature importance columns

cols_to_drop = list(df_importance.iloc[-10:].index)
cols_to_drop
```

executed in 3ms, finished 13:53:39 2021-05-23

Out[146]:
```
['weights_rec_small_mid',
 'weights_rec_ultra',
 'weights_rec_unlisted',
 'weights_sent_large',
 'weights_sent_mid',
 'weights_sent_no',
 'weights_sent_small',
 'weights_sent_small_mid',
 'weights_sent_ultra',
 'weights_sent_unlisted']
```

- They are all OHE collumns that I have feature engineered
- Weights no which signifies if a wallet holds ERC20 tokens is maintained in the dataset
- This is a strong indicator
- Interesting to note that if a wallet has received an ERC20 token is a very strong indicator but if a column has sent an ERC20 is not cosidered a strong indicator

In [147]:
```
# Remove columns from feature selection

X_train_fil = X_train_ns.drop(cols_to_drop, axis=1)
X_test_fil = X_test_ns.drop(cols_to_drop, axis=1)
```

executed in 4ms, finished 13:53:39 2021-05-23

In [148]:
```
# Confirm columns are being removed

print(X_train_fil.shape)
X_test_fil.shape
```

executed in 2ms, finished 13:53:39 2021-05-23

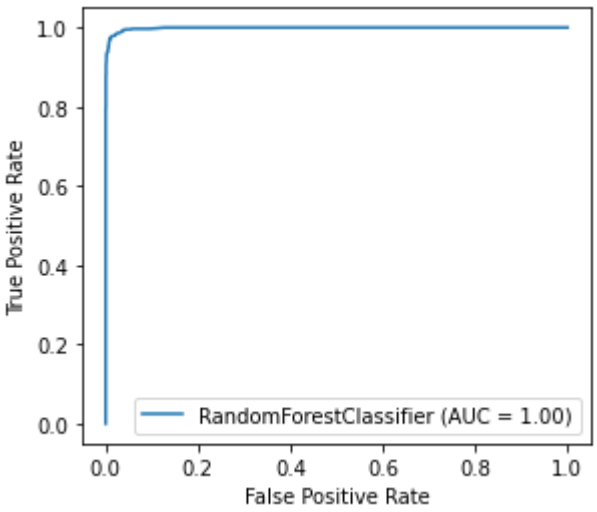
```
(6871, 37)
```

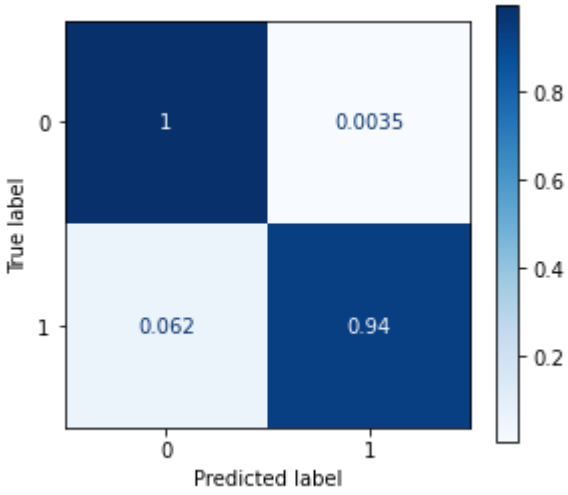
Out[148]:
```
(2945, 37)
```

In [149]:
```
# Evaluate model with selected features removed

rf_clf.fit(X_train_fil, y_train)
evaluate_model(rf_clf, X_train_fil, X_test_fil, y_train, y_test)
```

executed in 1.11s, finished 13:53:40 2021-05-23

```
********************CLASSIFICATION REPORT ***************************
              precision    recall  f1-score   support

       valid       0.98      1.00      0.99      2299
       fraud       0.99      0.94      0.96       646

    accuracy                           0.98      2945
   macro avg       0.98      0.97      0.98      2945
weighted avg       0.98      0.98      0.98      2945

************************************************************************
```






```
Recall training score: 1.0
Recall test score: 0.9837
Overfit by +0.0163
---------------
Recall score improvement: +0.41808
```

- Recall score on the testing data has improved to 94%, this is the metric we are trying to optimize
- Vanilla test recall on the fraud column was at 93%
- Removing columns improved the score
- These columns may have negatively impacted recall score
- Additionally, with less columns, we are improving computational efficiency

RFM Tuning

- Use GridSearch CV to optimize for recall
- Iterate this process until optimal model is found

In [150]:
```
# Set up parameter grid
# Include default and several other options
# Only use balanced to account for class weight imbalance

rf_param_grid = {
    'n_estimators': [80,100,120],
    'criterion': ['gini', 'entropy'],
    'max_depth':[10,15,20],
    'min_samples_leaf':[1,2,5,10],
    'class_weight':['balanced']}
```

executed in 2ms, finished 13:53:40 2021-05-23

In [151]:
```
# Run grid search

rf_grid = grid_search(rf_clf, rf_param_grid, X_train_fil, y_train)
```

executed in 1m 46.1s, finished 13:55:26 2021-05-23

In [152]:
```
rf_grid['best_params']
```

executed in 3ms, finished 13:55:26 2021-05-23

Out[152]:
```
{'class_weight': 'balanced',
 'criterion': 'entropy',
 'max_depth': 10,
 'min_samples_leaf': 10,
 'n_estimators': 120}
```

In [153]:
```
# Observe Top 10 Ranked Hyperparameters

df_rf_grid = rf_grid['dataframe']
df_rf_grid[df_rf_grid['rank_test_score']<=11]
```

executed in 8ms, finished 13:55:26 2021-05-23

Out[153]:

	param_class_weight	param_criterion	param_max_depth	param_min_samples_leaf	param_n_estin
57	balanced	entropy	15	10	
69	balanced	entropy	20	10	
47	balanced	entropy	10	10	
42	balanced	entropy	10	5	
66	balanced	entropy	20	5	
70	balanced	entropy	20	10	
58	balanced	entropy	15	10	
34	balanced	gini	20	10	
22	balanced	gini	15	10	
33	balanced	gini	20	10	
54	balanced	entropy	15	5	
21	balanced	gini	15	10	
46	balanced	entropy	10	10	
44	balanced	entropy	10	5	
43	balanced	entropy	10	5	
45	balanced	entropy	10	10	

- Class weight: Balanced
- Criterion: entropy
- Max Depth: explore further
- Min Sample Leaf: explore further
- N_estimators: explore further

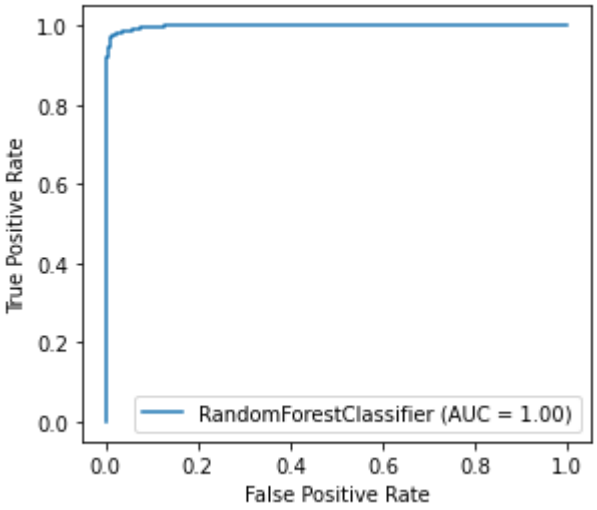
In [154]:
```
# Fit model with results from GridSearch

rf2_clf = RandomForestClassifier(n_estimators=80,class_weight='balanced
```

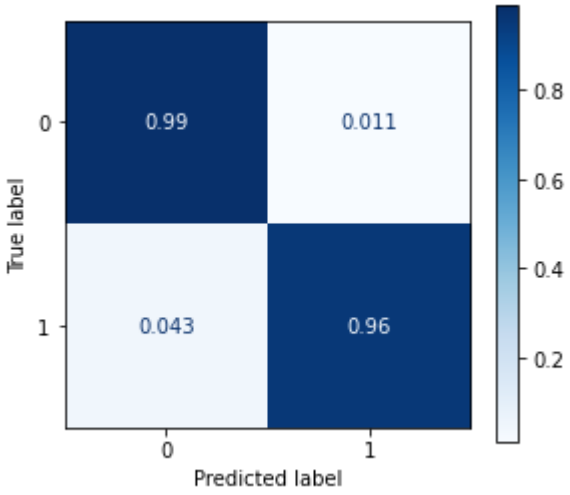
executed in 2ms, finished 13:55:26 2021-05-23

In [155]:
```
# Fit and evaluate new model performance

rf2_clf.fit(X_train_fil, y_train)
evaluate_model(rf2_clf, X_train_fil, X_test_fil, y_train, y_test)
```

executed in 944ms, finished 13:55:27 2021-05-23

```
********************CLASSIFICATION REPORT ***************************
              precision    recall  f1-score   support

       valid       0.99      0.99      0.99      2299
       fraud       0.96      0.96      0.96       646

    accuracy                           0.98      2945
   macro avg       0.97      0.97      0.97      2945
weighted avg       0.98      0.98      0.98      2945

************************************************************************
```






```
Recall training score: 0.99389
Recall test score: 0.982
Overfit by +0.01189
---------------
Recall score improvement: +0.43665999999999994
```

- Recall score improved from 0.94 to 0.96
- It also appears less overfit
- **Next:** Try another grid search using results from first search

In [156]:
```
# Set up parameter grid

rf_param_grid2 = {
    'n_estimators': [70, 80,120, 130],
    'criterion': ['entropy'],
    'max_depth':[15,20, 25],
    'min_samples_leaf':[4,5,6],
    'class_weight':['balanced']}
```

executed in 2ms, finished 13:55:27 2021-05-23

- Some of the initial GridSearch results were close to the tails of the distribution so trying a new parameter grid with more extreme values

In [157]:
```
# Fit the GridSearch

rf_grid2 = grid_search(rf2_clf, rf_param_grid2, X_train_fil, y_train)
```

executed in 57.4s, finished 13:56:24 2021-05-23

In [158]:
```
# Observe results

df_rf_grid2 = rf_grid2['dataframe']
df_rf_grid2[df_rf_grid2['rank_test_score']<=11]
```

executed in 9ms, finished 13:56:24 2021-05-23

Out[158]:

	param_class_weight	param_criterion	param_max_depth	param_min_samples_leaf	param_n_estim
34	balanced	entropy	25	6	
9	balanced	entropy	15	6	
4	balanced	entropy	15	5	
11	balanced	entropy	15	6	
35	balanced	entropy	25	6	
28	balanced	entropy	25	5	
23	balanced	entropy	20	6	
17	balanced	entropy	20	5	
31	balanced	entropy	25	5	
30	balanced	entropy	25	5	
29	balanced	entropy	25	5	

- Mean test score higher than first param grid
- N_estimators is still optimized on the end of the distribution will try another search with lower n_estimators
- Min samples leaf is optimzed at the end of the distribution so will try some higher values
- Max depth is centered so I believe it is close to the optimal value
- **Next:** Run GridSearch with new parameters

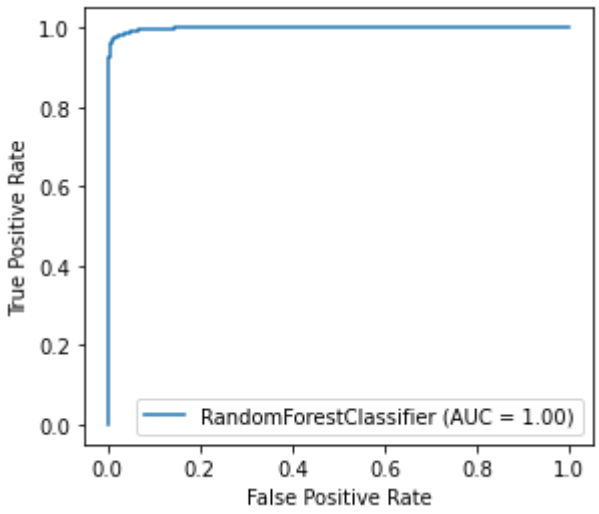
In [159]:
```
1 rf_grid2['best_params']
```
executed in 2ms, finished 13:56:24 2021-05-23

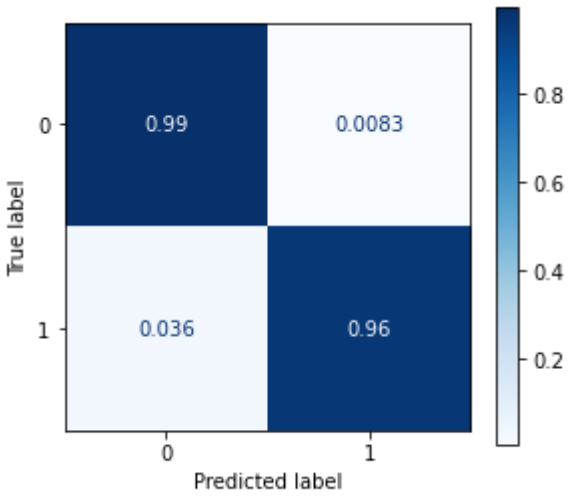
Out[159]:
```
{'class_weight': 'balanced',
 'criterion': 'entropy',
 'max_depth': 25,
 'min_samples_leaf': 6,
 'n_estimators': 120}
```

In [160]:
```
1 rf3_clf = RandomForestClassifier(class_weight='balanced',criterion= 'en
2                                  min_samples_leaf= 6, n_estimators= 80,
```
executed in 2ms, finished 13:56:24 2021-05-23

In [161]:
```
1 rf3_clf.fit(X_train_fil, y_train)
2 evaluate_model(rf3_clf, X_train_fil, X_test_fil, y_train, y_test)
```
executed in 963ms, finished 13:56:25 2021-05-23

```
********************CLASSIFICATION REPORT ***************************
              precision    recall  f1-score   support

       valid       0.99      0.99      0.99      2299
       fraud       0.97      0.96      0.97       646

    accuracy                           0.99      2945
   macro avg       0.98      0.98      0.98      2945
weighted avg       0.99      0.99      0.99      2945

******************************************************************************
```






```
Recall training score: 0.9952
Recall test score: 0.98574
Overfit by +0.00946
---------------
Recall score improvement: +0.4444
```

- Score is continuing to improve
- AUC curve is still at max value of 1.0

In [162]:
```
# Set up new parameter grid based on findings from previous grid

rf_param_grid3 = {
    'n_estimators': [55, 60,65, 70],
    'criterion': ['entropy'],
    'max_depth':[18,20, 22],
    'min_samples_leaf':[6,7,8,9],
    'class_weight':['balanced']}
```
executed in 3ms, finished 13:56:25 2021-05-23

In [163]:
```
rf_grid3 = grid_search(rf3_clf, rf_param_grid3, X_train_fil, y_train)
```
executed in 48.0s, finished 13:57:13 2021-05-23

In [164]:
```
# Observe results

df_rf_grid3 = rf_grid2['dataframe']
df_rf_grid3[df_rf_grid2['rank_test_score']<=11]
```
executed in 7ms, finished 13:57:13 2021-05-23

Out[164]:

	param_class_weight	param_criterion	param_max_depth	param_min_samples_leaf	param_n_estim
34	balanced	entropy	25	6	
9	balanced	entropy	15	6	
4	balanced	entropy	15	5	
11	balanced	entropy	15	6	
35	balanced	entropy	25	6	
28	balanced	entropy	25	5	
23	balanced	entropy	20	6	
17	balanced	entropy	20	5	
31	balanced	entropy	25	5	
30	balanced	entropy	25	5	
29	balanced	entropy	25	5	

In [165]:
```
rf_grid3['best_params']
```
executed in 3ms, finished 13:57:13 2021-05-23

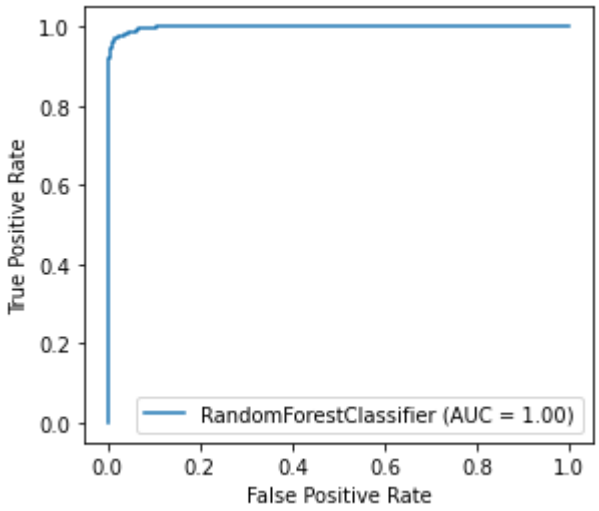
Out[165]:
```
{'class_weight': 'balanced',
 'criterion': 'entropy',
 'max_depth': 18,
 'min_samples_leaf': 9,
 'n_estimators': 55}
```

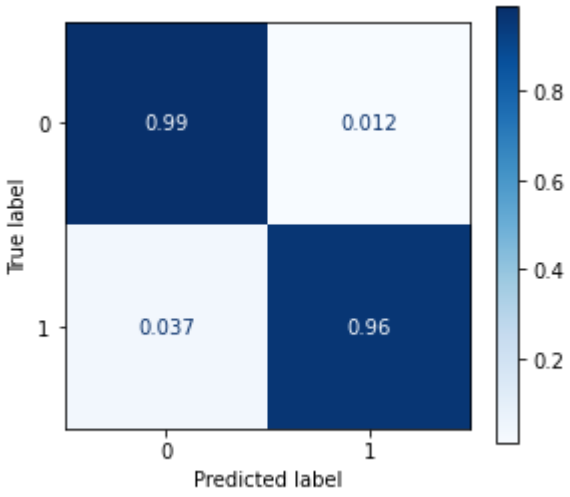
In [166]:
```
rf4_clf = RandomForestClassifier(class_weight='balanced',criterion= 'en
                                 min_samples_leaf= 9, n_estimators= 55,
```
executed in 2ms, finished 13:57:13 2021-05-23

In [167]:
```
rf4_clf.fit(X_train_fil, y_train)
evaluate_model(rf4_clf, X_train_fil, X_test_fil, y_train, y_test)
```

executed in 711ms, finished 13:57:14 2021-05-23

```
********************CLASSIFICATION REPORT ***************************
              precision    recall  f1-score   support

       valid       0.99      0.99      0.99      2299
       fraud       0.96      0.96      0.96       646

    accuracy                           0.98      2945
   macro avg       0.97      0.98      0.97      2945
weighted avg       0.98      0.98      0.98      2945

************************************************************************
```






```
Recall training score: 0.99185
Recall test score: 0.98234
Overfit by +0.00951
---------------
Recall score improvement: +0.44284999999999997
```

- Recall score for fraudulent cases decreased so will work with the third iteration of the Random Forest
- False negative test results are slightly higher (0.036 to 0.037)

Imputation Strategies

- Observe how different imputation strategies impact recall score
- Will only need numeric columns because categorical columns have already been one hot encoded

In [168]:
```
# Observe original pipeline

scaled_preprocessing.named_transformers_
```
executed in 10ms, finished 13:57:14 2021-05-23

Out[168]:
```
{'num': Pipeline(steps=[('imputer', SimpleImputer(strategy='median')),
                        ('scale', StandardScaler())]),
 'cat': Pipeline(steps=[('imputer', SimpleImputer(strategy='most_frequen
t')),
                        ('encoder',
                         OneHotEncoder(handle_unknown='ignore', sparse=Fals
e))])}
```

In [169]:
```
# Create pipeline and add our best model for assessing scores

grid_pipe2 = Pipeline(steps=[('ct',scaled_preprocessing),
                             ('clf',RandomForestClassifier(class_weight='balanced'
                                                           min_samples_leaf= 6, n_estimators= 80))
```
executed in 2ms, finished 13:57:14 2021-05-23

In [170]:
```
# Set up SimpleImputer with different types of imputation strategies

params2 = {'ct__num__imputer__strategy':['median','mean','most_frequent
           'clf__n_estimators': [80,100,120],
             'clf__criterion': ['gini', 'entropy'],
             'clf__max_depth':[10,15,20, None],
             'clf__min_samples_leaf':[1,2,5,10],
             'clf__class_weight':['balanced']}
```
executed in 4ms, finished 13:57:14 2021-05-23

- Not using a constant value because it will make a larger difference on the distribution of data than median, mean, or mode

In [171]:
```
# Set up grid search and optimize for recall score
# Fit the grid search on training data

# Takes 30 mins to run so saved the results as a DataFrame in the repo

# gridsearch2 = GridSearchCV(grid_pipe2,params2, cv=5, scoring='recall'
# gridsearch2.fit(X_train,y_train)
```
executed in 1ms, finished 13:57:14 2021-05-23

In [172]:
```
grid_search_df2 = pd.read_csv('gridsearch2df')
```
executed in 21ms, finished 13:57:14 2021-05-23

In [173]:
```
1 grid_search_df2.sort_values(by='rank_test_score')
```

executed in 23ms, finished 13:57:14 2021-05-23

Out[173]:

	Unnamed: 0	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_clf__class_weig
285	285	0.55907	0.01031	0.01769	0.00060	balanc
177	177	0.55306	0.00658	0.01740	0.00012	balanc
279	279	0.38306	0.00643	0.01317	0.00006	balanc
213	213	0.56052	0.01164	0.01768	0.00008	balanc
168	168	0.56679	0.01165	0.01736	0.00020	balanc
...	...	...	...	...	...	
116	116	0.61445	0.00389	0.01945	0.00007	balanc
38	38	0.41756	0.00349	0.01459	0.00016	balanc
74	74	0.41560	0.00617	0.01449	0.00017	balanc
218	218	0.46611	0.00307	0.01435	0.00009	balanc
110	110	0.41473	0.00455	0.01441	0.00010	balanc

288 rows × 20 columns

- According to the DataFrame, median performed the best on the cross validaded training data
- Followed by most frequent and then mean
- Median was the original imputation method
- Most frequent had the lowest standard deviation but all of the results fell within a very tight range
- **Conclusion:** Continue to use median as imputation method for missing values

In [174]:
```
# Instantiate model using results from DataFrame
# Fit model

rf5_clf = RandomForestClassifier(class_weight='balanced', criterion='en
                                 min_samples_leaf=10, n_estimators=120)
rf5_clf.fit(X_train_fil, y_train)
```

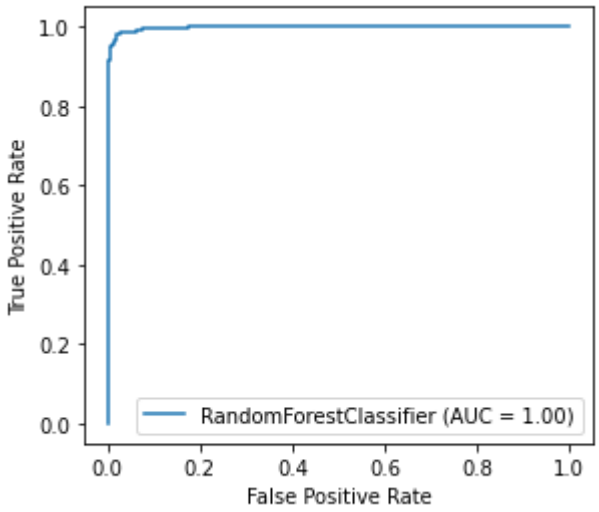
executed in 1.06s, finished 13:57:15 2021-05-23

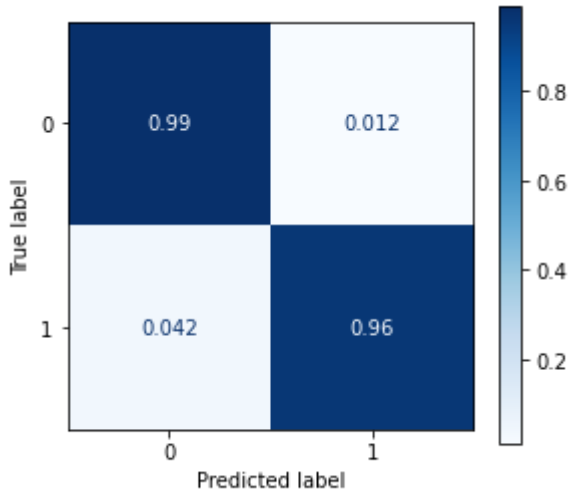
Out[174]:
```
RandomForestClassifier(class_weight='balanced', criterion='entropy',
                       min_samples_leaf=10, n_estimators=120)
```

In [175]:
```
# Check performance

evaluate_model(rf5_clf, X_train_fil, X_test_fil, y_train, y_test)
```

executed in 282ms, finished 13:57:15 2021-05-23

```
********************CLASSIFICATION REPORT ***************************
              precision    recall  f1-score   support

       valid       0.99      0.99      0.99      2299
       fraud       0.96      0.96      0.96       646

    accuracy                           0.98      2945
   macro avg       0.97      0.97      0.97      2945
weighted avg       0.98      0.98      0.98      2945

************************************************************************
```






```
Recall training score: 0.99083
Recall test score: 0.98132
Overfit by +0.00951
---------------
Recall score improvement: +0.43820000000000003
```

- Performance has leveled to 0.96 on the test case
- Recall training score is slighly greater
- Essentially not overfit at all from a macro avg recall perspective
- Out of 100 fraud cases, the model will accurately detect 97% of them

- This is the third best model in recall test performance
- **Median imputation and rf3_clf are the model and imputation method that performed best for recall**

RFM Conclusion

- The Random Forest Model is the best model I have created for optimizing for recall for fraudulant transactions
- The Random Forest outperforms the KNN model by about 4% on the test case
- The Random Forest outperforms the dummy model by about 45%
- In final performance out of all fraudulant cases, only ~3% would pass through the model undetected. Out of all valid transactions, the model would only miscategorize 1% of them fraud, this cost is low
  - For fraudulant transactions, it accurately classifies ~96% of them
  - For valid transactions, it accurately classifies ~99% of them
- If a wallet company implemented this model into its infrastructure, it would greatly improve customer satisfaction because customers know that their marginal transaction has a very liklihood of being accurately classified as fraudulant or valid. This gives them more confidence in the wallet
- Similar to the KNN interpretation, say a crypto wallet start up has 10,000 users and knows that over the course of the wallets lifetime, 500 will be targeted for fraud. The average fraudulant Ethereum transaction is 0.5 ETH (~$764)
  - In the dummy model, 500 customers are targeted for fraud and 250 are succesfully defrauded costing customers a total of $191,000
  - In the KNN model, 500 customers are targeted for fraud and 40 are succesfully defrauded costing customers a total of $30,560
  - In the RF model, 500 customers are targeted for fraud and only 19 are succesfully defrauded costing customers a total of $14,516
    - For 10,000 customers, fraud's **expected cost per customer is reduced from $19.11 to $1.55**

In [176]:
```
# Overview of model performance and business cost

model_comparison = {'Model': ['Dummy', 'KNN', 'RF'], 'Total Customers':
     'Expected Valid Tnx':[9500, 9500,9500], 'Expected Fraud Tnx':[500,
    'Defrauded Tnx':[250, 40, 19], 'Total Cost':[191000,30560,14516],
    'Avg Expected Loss':[19.11, 3.27, 1.55]}
pd.DataFrame(model_comparison)
```

executed in 7ms, finished 13:57:15 2021-05-23

Out[176]:

	Model	Total Customers	Expected Valid Tnx	Expected Fraud Tnx	Defrauded Tnx	Total Cost	Avg Expected Loss
0	Dummy	10000	9500	500	250	191000	19.11000
1	KNN	10000	9500	500	40	30560	3.27000
2	RF	10000	9500	500	19	14516	1.55000

iNTERPRET

In [177]:
```
# Bring in SHAP and JS
# conda install -c conda-forge shap

import shap
print(shap.__version__)
shap.initjs()
```

executed in 681ms, finished 13:57:16 2021-05-23

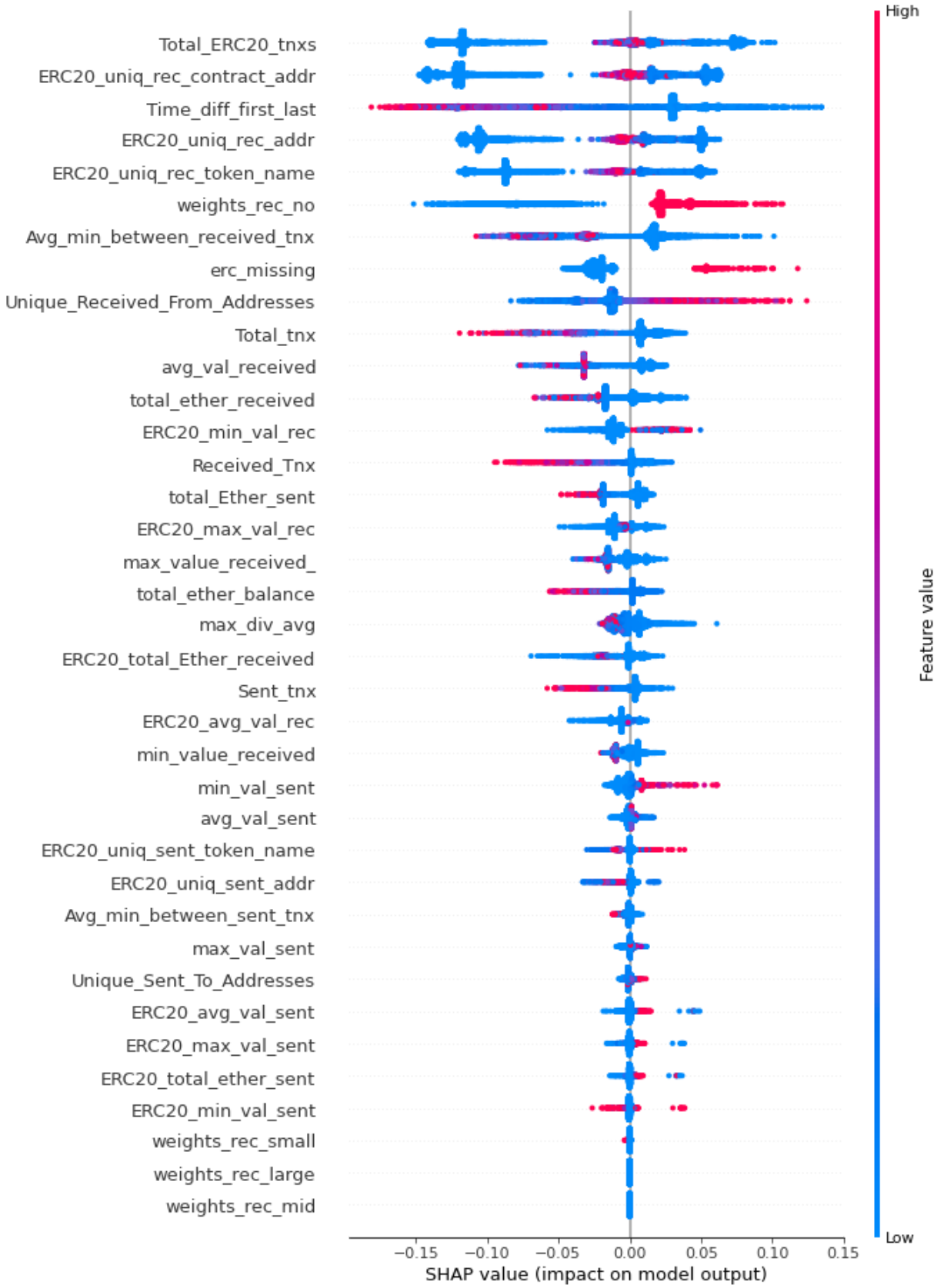
```
0.37.0
```

In [178]:
```
# Calculate SHAP values for test data

explainer = shap.TreeExplainer(rf5_clf)
shap_values = explainer.shap_values(X_test_fil,y_test)
```

executed in 5.20s, finished 13:57:21 2021-05-23

In [179]:
```
# Show summary plot

shap.summary_plot(shap_values[1],X_test_fil,max_display=40)
```

executed in 1.03s, finished 13:57:22 2021-05-23



- Total ERC20_tnx, ERC20_unique_rec_token_name, ERC20_unique_rec_address are difficult to tell which direction they are moving the prediction based on the SHAP Model
  - From EDA and cross-referencing LIME, it appears that:
    - The more active the user is in making transactions, the higher the liklihood that the wallet will be defrauded
- Time_diff_first_and_last tend to be associated with mixed values
  - Low values seem to be more correlated with fraud
- If ERC values are missing, there is a very high liklihood of fraud
  - Makes sense because the user may not know what transactions transpired in their wallet
- If a user does use ERC20 tokens, they are less likely to experience fraud
  - This is very surpising considering many ERC20 tokens are associated with fraud
- Higher the total ether balance, less likely to report fraud
  - Wallets with higher balances may be more heavily targeted but they the owners may be more vigilant

In [180]:
```
# !pip install lime
from lime.lime_tabular import LimeTabularExplainer
```

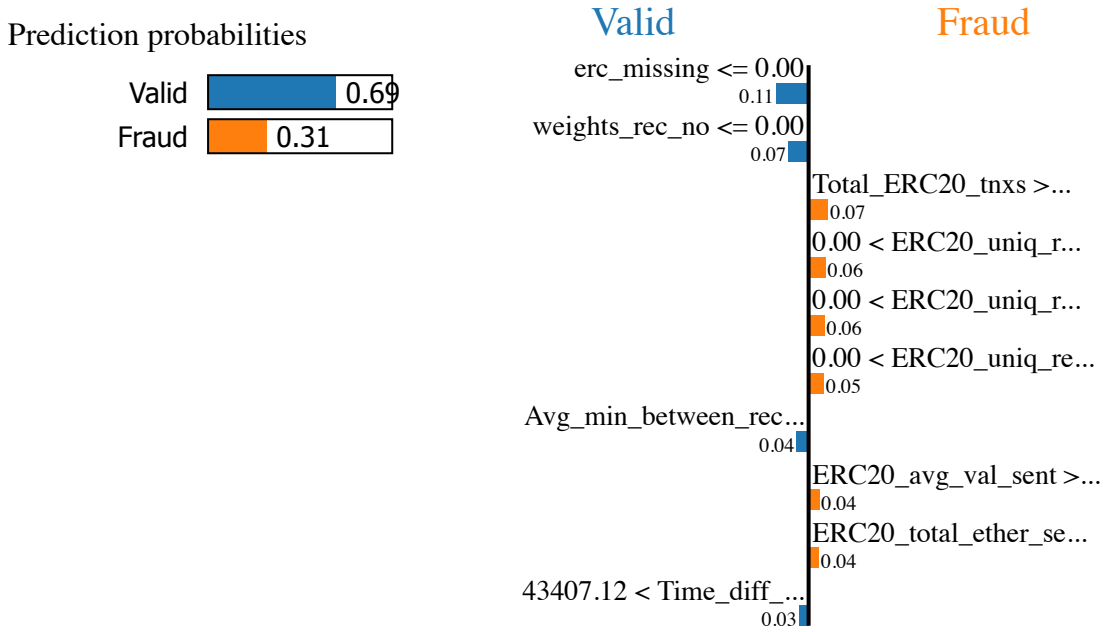
executed in 1ms, finished 13:57:22 2021-05-23

In [181]:
```
lime_explainer =LimeTabularExplainer(
    training_data=np.array(X_test_fil),
    feature_names=X_test_fil.columns,
    class_names=['Valid', 'Fraud'],
    mode='classification'
)
```

executed in 33ms, finished 13:57:22 2021-05-23

In [182]:
```
row = np.random.choice(range(len(X_test_fil)))
exp = lime_explainer.explain_instance(X_test_fil.iloc[row], rf3_clf.pre
exp.show_in_notebook(show_table=True, show_all=True)

```

executed in 5.53s, finished 13:57:28 2021-05-23



Feature	Value
Avg_min_between_sent_tnx	3146.35
Avg_min_between_received_tnx	6168.94
Time_diff_first_last	49846.55
Sent_tnx	8.00
Received_Tnx	4.00
Unique_Received_From_Addresses	2.00
Unique_Sent_To_Addresses	6.00
min_value_received	0.01
max_value_received_	4.08
avg_val_received	1.54

- A greater time difference between first and last transaction results in a higher probability of a valid transaction
  - This is a proxy for how long the wallet has been in existance, the longer the wallet has been available, the less liklihood that it will be defrauded. This may be biased by the fact

that once a wallet is defrauded, users are likely to get rid of it
- If ERC information is missing, the wallet is more likely to be defrauded
  - After a fraudulant transaction, data may not be available for the transaction that took place
- If a wallet holds ERC20 tokens, it is less likely to be defrauded
- The more ERC20 tnxs, the most unique received tokens, and the more unique received from addresses,the high likelihood for fraud

CONCLUSIONS & RECOMMENDATIONS

The cryptocurrency industry feels like it is close to making a full break through into the mainstream. The market has evolved from fringe useage in the early 2010s (selling 10,000 BTC for 2 large Dominoes pizzas) to total retail hype in 2017. The market was relatively quite from 2017 to 2019 and absolutely exloded in value in 2020 when institutions began to adopt Bitcoin as a store of value (similar to gold). There have been moments of hype scattered between and plenty of interesting stories but that's the most basic gist.

From the retail perpsective, the two majors complaints against crypto are, 'what can I buy with it?' and 'why is it worth something?'. In my opinion, once retail useage picks up as a method of payment and store of value, the question becomes, 'how do I know it's safe?'

The purpose of the model I have built is to make retailers feel safe about their transactions. People feel comfortable using credit cards to spend money because they know if something unintended happens, they can probably get their money back after speaking with customer service for 30 minutes. Unfortunately, in the crypto space, once your money is sent out of your wallet, there's no getting it back. This is exactly why technologies need to advance so customers can feel comfortable using their crypto for transactions.

Since liability is purely on the user, the next best solution is to build a model that can predict with a high level of accuracy whether the marginal wallet transactions is valid or fraudulant. If the model feels the wallet is at risk, it can ping the user and let them know to be careful before releasing the funds. Safeguards like these will never be perfect, but as more data is released, they can get smarter and smarter. The model learns from features like time between first and last transaction, the size of the transactions, and the types of coins that the wallet holds.

Currently, the Random Forest Model that I have produced has an accuracy of 98%. More importantly, it has a recall of ~96%. For every 100 fraudulant transaction, it correctly classifies 96 of them. For every 100 valid transactions, it correctly classifies 99 of them. The median average transaction for fraudulant transactions is 0.5 ETH ($764). Customers are putting a lot of faith into wallet providers and money is on the line. If the wallet company has to compete for market share, this type of service would be a crucial value-add.

Ideally, wallet providers will be able to easily integrate their products with a production level model to improve security and customer satisfaction. It is a win-win.

https://www.coindesk.com/bitcoin-pizza-10-years-laszlo-hanyecz (https://www.coindesk.com/bitcoin-pizza-10-years-laszlo-hanyecz)

In [ ]:
```
1
```