Fraud Detection using card transaction data

- **Introduction**
- **Dataset**
  - **Explore relevant features**
  - **Pre-process**
- **Initial Modeling**
- **Model improvements**
- **Conclusion and Recommendations**

Introduction:

Credit Card fraud has been a growing problem in the digital age. In 2020, covid-19 accelerated this trend and we are seeing record numbers of credit card transactions. Along with these trends, fraudulent transactions have been increasing in size and sophistication. Companies

Dataset:

We have 5000 customer ids with a subset of transactions over the year 2016. This is a supervised learning problem containing 786,363 transactions with 12,417 of those being flagged as fraudulent. We unpack the .json file into a panda's dataframe and find 28 columns with several being blank or irrelevant for analysis. After we drop those columns and clean up the datatypes, we begin to explore the fraud vs. no fraud transactions.

```
fraud cases : 12417
valid cases : 773946
fraud case % : 0.016043754990658264
```

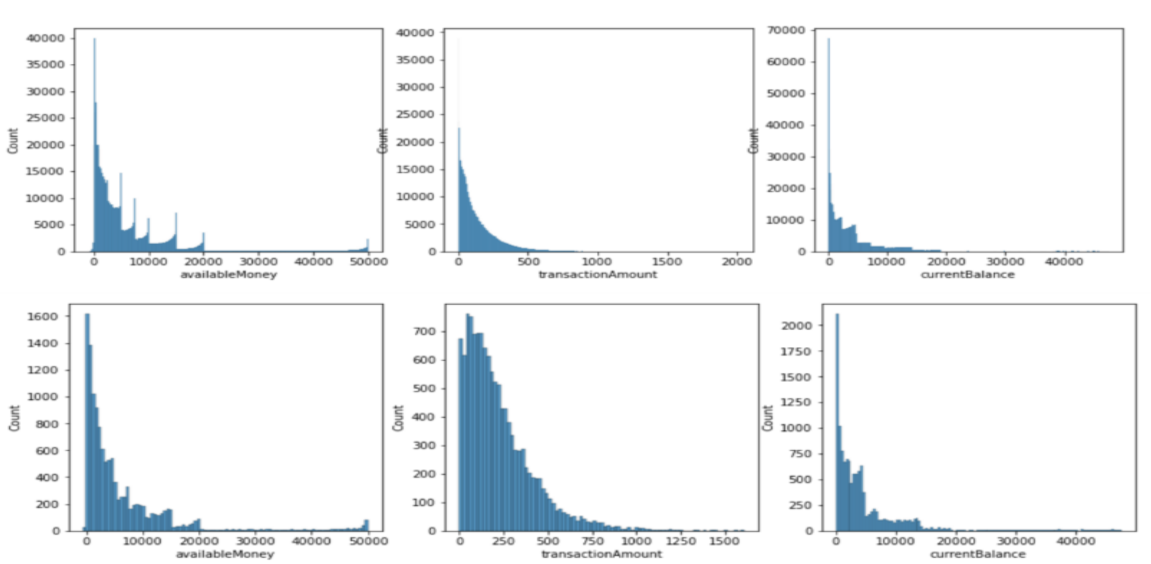
We have a very imbalanced dataset. Only a small amount of fraud cases we can use to train on. This is a common scenario in the world of fraud detection, where most transactions are legitimate.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 786363 entries, 0 to 786362
Data columns (total 29 columns):
 #   Column                    Non-Null Count   Dtype
---  ------                    --------------   -----
 0   accountNumber             786363 non-null  int64
 1   customerId                786363 non-null  int64
 2   creditLimit               786363 non-null  int64
 3   availableMoney            786363 non-null  float64
 4   transactionDateTime       786363 non-null  object
 5   transactionAmount         786363 non-null  float64
 6   merchantName              786363 non-null  object
 7   acqCountry                786363 non-null  object
 8   merchantCountryCode       786363 non-null  object
 9   posEntryMode              786363 non-null  object
 10  posConditionCode          786363 non-null  object
 11  merchantCategoryCode      786363 non-null  object
 12  currentExpDate            786363 non-null  object
 13  accountOpenDate           786363 non-null  object
 14  dateOfLastAddressChange   786363 non-null  object
 15  cardCVV                   786363 non-null  int64
 16  enteredCVV                786363 non-null  int64
 17  cardLast4Digits           786363 non-null  int64
 18  transactionType           786363 non-null  object
 19  echoBuffer                786363 non-null  object
 20  currentBalance            786363 non-null  float64
 21  merchantCity              786363 non-null  object
 22  merchantState             786363 non-null  object
 23  merchantZip               786363 non-null  object
 24  cardPresent               786363 non-null  bool
 25  posOnPremises             786363 non-null  object
 26  recurringAuthInd          786363 non-null  object
 27  expirationDateKeyInMatch  786363 non-null  bool
 28  isFraud                   786363 non-null  bool
dtypes: bool(3), float64(3), int64(6), object(17)
memory usage: 158.2+ MB
```

Exploring categorical variables further we can see that we have 5000 unique customerIds and they have 10 different creditLimit categories. The most common merchant is Uber with 2490 unique merchants.

	customerId	merchantName	acqCountry	merchantCountryCode	posEntryMode	cardPresent	expirationDateKeyInMatch	isFraud	creditLimit	posConditionCode	transactionType
count	786363	786363	786363	786363	786363	786363	786363	786363	786363	786363	786363
unique	5000	2490	5	5	6	2	2	2	10	4	4
top	380680241	Uber	US	US	05	False	False	False	5000	01	PURCHASE
freq	32850	25613	774709	778511	315035	433495	785320	773946	201863	628787	745193

We will begin exploring the numeric features of the dataset. We don't have too many columns to work with (only 3 availableMoney, transactionAmount, currentBalance) but let's take a look. We can see extreme skew in the data visually indicating the imbalanced nature of the transactions. Fraud vs. Non Fraud transactions show a similar level of skewness.



```
Fdf.transactionAmount.sum()
```
executed in 6ms, finished 01:31:12 2020-12-29

```
2796505.89
```

```
Fdf.transactionAmount.mean()
```
executed in 6ms, finished 01:31:12 2020-12-29

```
225.2159048079239
```

```
nFdf.transactionAmount.mean()
```
executed in 6ms, finished 01:31:12 2020-12-29

```
135.57024862199447
```

We can see the total amount of Fraud transactions is about $2.8 million on 12,417 cases.

The average legitimate transaction amounts are 135.57.

While the average fraud transaction amounts are 225.21.

The average amounts between fraud and no fraud transactions is very significant. Fraud transactions average almost 100$ more than non-fraud. This would suggest we need to lookout for unusually high purchase amounts than average from regular vendors. This makes sense as fraudsters are usually trying to milk the transactions while they have the chance.

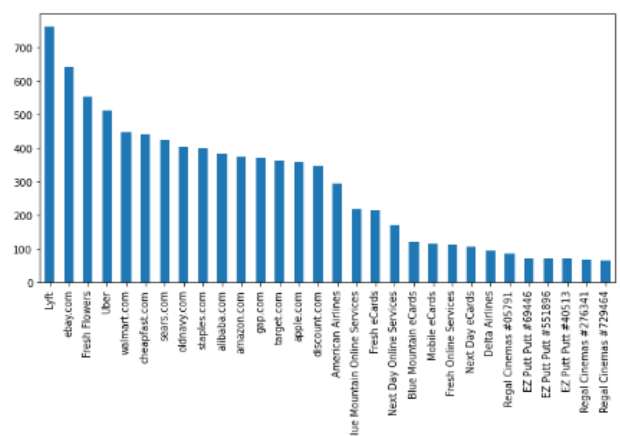
Next we compare merchants to see where the fraud is happening and if there are any insights we can glean.

What are the spending habits / patterns of the transactions? What features stand out between fraud and no-fraud transactions. Compare Distributions on merchants between fraud/no-fraud.

Most popular merchants by # of transactions

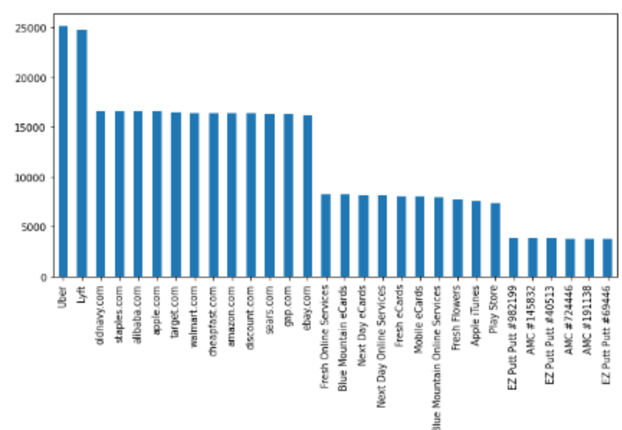
```
#Popular merchants in Fraud transactions
Fdf['merchantName'].value_counts().nlargest(30).plot
Fdf.merchantName.value_counts()
```
executed in 342ms, finished 01:31:22 2020-12-29

```
Lyft                        760
ebay.com                    639
Fresh Flowers               553
Uber                        512
walmart.com                 446
                           ...
Hyatt House #770440           1
Universe Massage #596422      1
Powerlifting #764704          1
Cinnabon #124416              1
Dunkin' Donuts #387966        1
Name: merchantName, Length: 1042, dtype: int64
```



In [24]:
```
#Popular Merchants in Valid Transactions
nFdf['merchantName'].value_counts().nlargest(30).plo
nFdf.merchantName.value_counts()
```
executed in 707ms, finished 01:31:22 2020-12-29

Out[24]:
```
Uber                       25101
Lyft                       24763
oldnavy.com                16591
staples.com                16581
alibaba.com                16576
                             ...
Boost Mobile #104815           2
Runners #383214                2
EZ Wireless #149871            1
Curves #849125                 1
TMobile Wireless #602341       1
Name: merchantName, Length: 2490, dtype: int64
```



The Fraud vs. no-Fraud vendor count is similar, however one vendor Fresh Flowers seems to stand out as it is #3 in fraud but #22 in overall transactions by count. Further investigation of this vendor and its links could yield some clues to fraudster's behavior.

```
1 Fdf.cardPresent.value_counts(normalize=True)
2
```

executed in 8ms, finished 01:31:22 2020-12-29

```
False    0.721752
True     0.278248
Name: cardPresent, dtype: float64
```

In [26]:

```
1 df.cardPresent.value_counts(normalize=True)
```

executed in 14ms, finished 01:31:22 2020-12-29

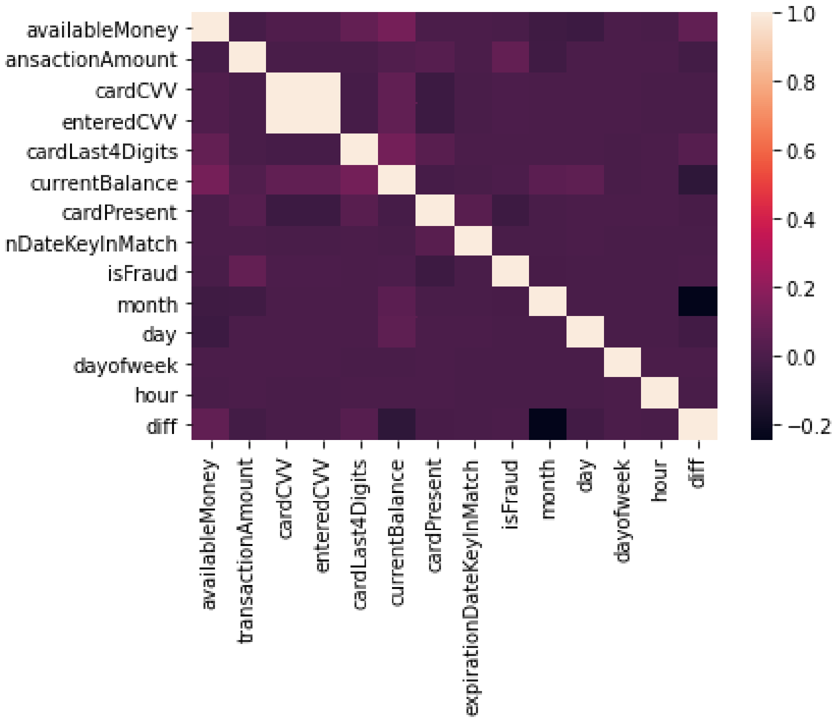
Out[26]:

```
False    0.551266
True     0.448734
Name: cardPresent, dtype: float64
```

(cardPresent = False) during Fraud transactions was 72% of time vs only 55% for all transactions.

This feature seems relevant as it shows a large difference when comparing fraud/no-fraud transactions. It makes sense that the card isn't present more in fraud as it is one less protective layer they don't have to go through. This feature will be important in helping us predict fraud.

Next we will check if there are any correlations or relationships between features we can visualize.



Not too much information here but we see slight correlations to fraud in the currentBalance and transactionAmount columns

Initial Modeling:

For our initial modeling we deploy sci-kit learn modules to help us model and identify important features. After selecting the relevant columns(12) we used pd.get dummies on the categorical features and binarized the true and false columns ending up with 26 columns total. We split the data 75/25 for training/testing.

Given the unbalanced nature of the dataset, a lot of techniques involve re-sampling or rebalancing the data so that it looks more normal. This usually helps improve the models but tree models not as much. We first compare a few tree and ensemble classifiers to see how they perform. We are using AUC/ROC in order to score our model. Accuracy is not as useful for these imbalanced fraud datasets.

These models are ranging from .50-.68 AUC/ROC which is alright but not amazing.

Modern techniques include using models like XGBoost which has been shown to outperform almost every other model in regards to binary classification.

https://ieeexplore.ieee.org/abstract/document/9214206

https://www.e3s-conferences.org/articles/e3sconf/abs/2020/74/e3sconf_ebldm2020_02042/e3sconf_ebldm2020_02042.html

Let's run XGBClassifier start with some values to get a baseline. Then get feature importance. Then tune hyper parameters.

```
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, gamma=0, gpu_id=0,
              importance_type='gain', interaction_constraints='',
              learning_rate=0.300000012, max_delta_step=0, max_depth=6,
              min_child_weight=1, missing=nan,
              monotone_constraints='(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)',
              n_estimators=100, n_jobs=-1, num_parallel_tree=1,
              random_state=123, reg_alpha=0, reg_lambda=1, scale_pos_weight=1,
              subsample=1, tree_method='gpu_hist', validate_parameters=1,
              verbosity=None)
```

After training we get these results.

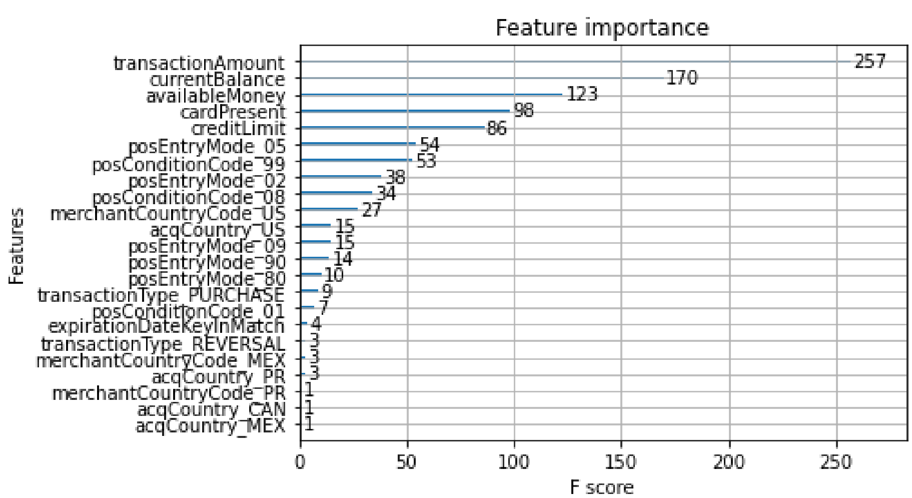
```
[[580424     28]
 [  9314      7]]
              precision    recall  f1-score   support

       False       0.98      1.00      0.99    580452
        True       0.20      0.00      0.00      9321

    accuracy                           0.98    589773
   macro avg       0.59      0.50      0.50    589773
weighted avg       0.97      0.98      0.98    589773
```

```
train-auc-mean  train-auc-std  test-auc-mean  test-auc-std
      0.718580       0.005469       0.717199      0.007960
      0.730219       0.002662       0.726934      0.008008
      0.733076       0.001862       0.729231      0.006477
      0.735265       0.001461       0.731955      0.004387
      0.736865       0.000939       0.733383      0.004755
      0.737928       0.000686       0.733999      0.005266
      0.739243       0.000843       0.734293      0.005492
      0.740862       0.000750       0.734764      0.005259
      0.742556       0.000597       0.735202      0.005318
      0.743936       0.000718       0.735746      0.005357
.7357459
```

The AUC is about .73 much better than the previous models. We can also explore which features the model used in order to get these results.



The top 4 features used by the model contain all 3 numeric features as well as the cardPresent feature which we identified earlier in our data exploration.

Nest we attempt to tune the hyper parameters of the model to see if we can improve the score. We used a RadnomziedSearch method on the several hyper parameters of the model. (min_child_weight, max_depth, gamma)

```
#tune some hyperparameters.

param_grid = {

    'max_depth':range(3,10,1),
 'min_child_weight':range(1,6,1)

}
```

```
#tune gamma

param_grid = {

   'gamma':[i/10.0 for i in range(0,5)]

}
xgb2 = xgb.XGBClassifier(objective="binary:logistic",max_depth=4,
                         min_child_weight=1,
                         tree_method='gpu_hist',n_jobs=-1)

xgb2_cv = RandomizedSearchCV(estimator=xgb2,
                             param_distributions=param_grid,
                             scoring='roc_auc',
                             verbose=1,random_state=123,n_jobs=-1)
```

```
[[580445      7]
 [  9317      4]]
              precision    recall  f1-score   support

       False       0.98      1.00      0.99    580452
        True       0.36      0.00      0.00      9321

    accuracy                           0.98    589773
   macro avg       0.67      0.50      0.50    589773
weighted avg       0.97      0.98      0.98    589773
```

	train-auc-mean	train-auc-std	test-auc-mean	test-auc-std
0	0.711783	0.001592	0.710731	0.003505
1	0.724598	0.002766	0.723307	0.003699
2	0.728424	0.001012	0.726780	0.002156
3	0.729534	0.000621	0.727336	0.002523
4	0.730829	0.000750	0.729158	0.001920
5	0.731245	0.000556	0.729726	0.002076
6	0.731701	0.000385	0.730045	0.002211
7	0.732536	0.000568	0.730303	0.002380
8	0.733083	0.000652	0.730710	0.002063
9	0.734193	0.000588	0.731335	0.002047
10	0.735131	0.000526	0.732209	0.001450
11	0.735766	0.000520	0.732290	0.001212
12	0.736149	0.000630	0.732719	0.001407
13	0.736960	0.000552	0.733351	0.001439
14	0.737884	0.000591	0.733942	0.001494
15	0.738739	0.000497	0.734500	0.001491
16	0.739621	0.000606	0.735049	0.001472
17	0.740552	0.000847	0.735888	0.001351
18	0.741888	0.001006	0.736801	0.001441
19	0.742814	0.000819	0.737253	0.001620

```
0.7372528
```

However we got no overall improvement in score after running all these hyper parameter optimizations. What we now need to gain performance is feature engineering. Next we try an automated tool (FeatureTools) to create features that will boost our model performance.

FeatureTools Modeling:

FeatureTools is a module that you can create relationships between various databases and features in order to automatically create more features. This can be a bit of a brute force approach. As it takes a significant amount of computation time to build the matrices and run the computations on new features. It took time to build the feature and also to encode the final feature matrix before modeling.

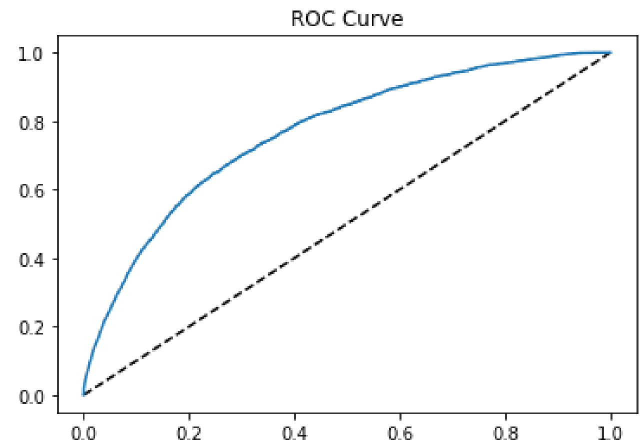
Using FeatureTools we managed to create an additional 154 features (130 after pruning) that we then used to again run our xgboost model. Featuretools uses one-hot encoding on categorical data and we end up with 464 columns in our new auto feature dataset. This time we get better results of around .80 AUC on the test data.

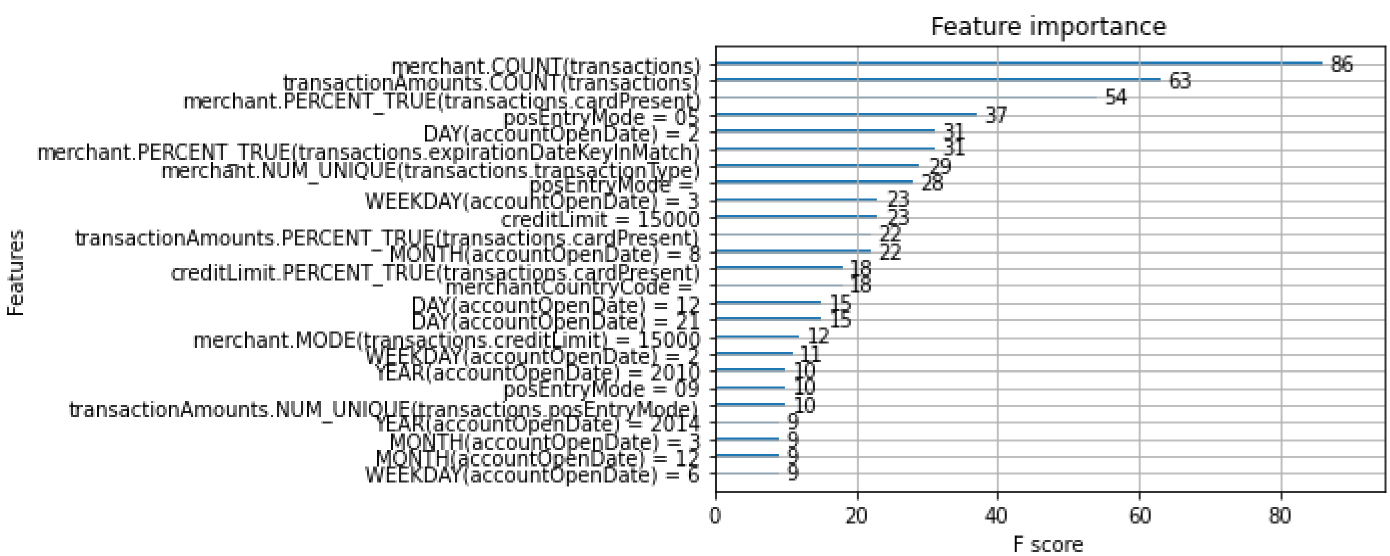
```
42          0.895460          0.002776          0.799664          0.005854
43          0.897176          0.002663          0.799369          0.006005
44          0.898786          0.002793          0.799527          0.005749
45          0.900555          0.002185          0.800119          0.006348
46          0.902005          0.002793          0.800535          0.006130
47          0.902597          0.002895          0.800514          0.006346
48          0.904275          0.003136          0.801255          0.006103
49          0.905095          0.003309          0.801117          0.006141
0.8011172
```

```
[[232120     55]
 [  3678     56]]
              precision    recall  f1-score   support

       False       0.98      1.00      0.99    232175
        True       0.50      0.01      0.03      3734

    accuracy                           0.98    235909
   macro avg       0.74      0.51      0.51    235909
weighted avg       0.98      0.98      0.98    235909
```





The most interesting part of this model is that it is using entirely different features than our previous xgb default model. It is using counts and num_unique amongst other features that have been auto generated. We can use these features to tell us where we can look for further improvements or analysis. For example, it seems there are some date times that we can look into as the model is using day month and year features. There might be some patterns or datetime correlations that can help us understand the fraud vs. no fraud better.

With an AUC of .80 we are capturing a strong majority of fraud cases in our model, enough to consider deploying this model.

PyCaret Modeling:

We could stop here, but I also decided to run this whole process again through a new ML tool PyCaret.

"*PyCaret is an open-source, low-code machine learning library in Python that aims to reduce the cycle time from hypothesis to insights. It is well suited for seasoned data scientists who want to increase the productivity of their ML experiments by using PyCaret in their workflows or for citizen data scientists and those new to data science with little or no background in coding.* " – PyCaret Homepage

We can setup various environments with different amounts of pre-processing, feature generation, model selection, blending, hyper parameter tuning, etc. I found using this tool very handy, however my memory and computer were bogging down very hard.   But we are able to achieve the best results using this module.

First we use the same features from our initial model but also include a few of our own date time features. We also create a few ratios from the numeric features.

```python
#Time features
df['month'] = df.transactionDateTime.dt.month
df['day'] = df.transactionDateTime.dt.day
df['hour'] = df.transactionDateTime.dt.hour

#Create a few features that combine matching columns into True/False

df['CVVMatch'] = (df.cardCVV == df.enteredCVV)
df['accountDiff'] = (df.accountOpenDate - df.dateOfLastAddressChange).dt.day
df['acqCountry_merchant_match'] = (df.merchantCountryCode == df.acqCountry)


#Numeric Ratios
df['creditRatio'] = df['creditLimit'] / df['availableMoney']
df['transactionRatio'] = df['transactionAmount'] / df['availableMoney']
df['balanceRatio'] = df['currentBalance'] / df['availableMoney']
df['balanceCreditRatio'] = df['currentBalance'] / df['creditLimit']
```

```
 0   creditLimit                786363 non-null  category
 1   availableMoney             786363 non-null  float32
 2   transactionDateTime        786363 non-null  datetime64[
 3   transactionAmount          786363 non-null  float32
 4   merchantName               786363 non-null  category
 5   posEntryMode               786363 non-null  category
 6   posConditionCode           786363 non-null  category
 7   merchantCategoryCode       786363 non-null  category
 8   currentExpDate             786363 non-null  category
 9   accountOpenDate            786363 non-null  datetime64[
 10  dateOfLastAddressChange    786363 non-null  datetime64[
 11  transactionType            786363 non-null  category
 12  currentBalance             786363 non-null  float32
 13  cardPresent                786363 non-null  bool
 14  expirationDateKeyInMatch   786363 non-null  bool
 15  isFraud                    786363 non-null  bool
 16  month                      786363 non-null  category
 17  day                        786363 non-null  category
 18  hour                       786363 non-null  category
 19  CVVMatch                   786363 non-null  bool
 20  accountDiff                786363 non-null  float32
 21  acqCountry_merchant_match  786363 non-null  bool
 22  creditRatio                786363 non-null  float32
 23  transactionRatio           786363 non-null  float32
 24  balanceRatio               786363 non-null  float32
 25  balanceCreditRatio         786363 non-null  float32
dtypes: bool(5), category(10), datetime64[ns](3), float32(8)
memory usage: 54.9 MB
```

```
dtypes: bool(5), datetime64[ns](3), float64(7), int64(5), object(6)
memory usage: 129.7+ MB
```

We also changed the data types from float64 to 32 and categories to objects which reduced the memory usage by about half. PyCaret allows you to setup your environments but we used mostly the default parameters. There is a lot of room for experimentation with the environment parameters.

0	session_id	6173	29	Normalize	False
1	Target	isFraud	30	Normalize Method	None
2	Target Type	Binary	31	Transformation	False
3	Label Encoded	False: 0, True: 1	32	Transformation Method	None
4	Original Data	(786363, 26)	33	PCA	False
5	Missing Values	False	34	PCA Method	None
6	Numeric Features	8	35	PCA Components	None
7	Categorical Features	14	36	Ignore Low Variance	True
8	Ordinal Features	False	37	Combine Rare Levels	True
9	High Cardinality Features	False	38	Rare Level Threshold	0.100000
10	High Cardinality Method	None	39	Numeric Binning	False
11	Transformed Train Set	(550454, 707)	40	Remove Outliers	False
12	Transformed Test Set	(235909, 707)	41	Outliers Threshold	None
13	Shuffle Train-Test	True	42	Remove Multicollinearity	True
14	Stratify Train-Test	False	43	Multicollinearity Threshold	0.950000
15	Fold Generator	StratifiedKFold	44	Clustering	False
16	Fold Number	5	45	Clustering Iteration	None
17	CPU Jobs	7	46	Polynomial Features	False
18	Use GPU	True	47	Polynomial Degree	None
19	Log Experiment	False	48	Trignometry Features	False
20	Experiment Name	clf-default-name	49	Polynomial Threshold	None
21	USI	5226	50	Group Features	False
22	Imputation Type	simple	51	Feature Selection	False
23	Iterative Imputation Iteration	None	52	Features Selection Threshold	None
24	Numeric Imputer	mean	53	Feature Interaction	False
25	Iterative Imputation Numeric Model	None	54	Feature Ratio	True
26	Categorical Imputer	constant	55	Interaction Threshold	0.010000
27	Iterative Imputation Categorical Model	None	56	Fix Imbalance	False
28	Unknown Categoricals Handling	least_frequent	57	Fix Imbalance Method	SMOTE

.

```
clf = setup(data=df,
            target='isFraud',
            use_gpu=True,
            fold=5,
            remove_multicollinearity=True,
            multicollinearity_threshold=.95,
            combine_rare_levels=True,
            ignore_low_variance=True,
            feature_ratio=True,
            n_jobs=7)
```

We get a slightly different dataset that our previous models with the transformed dataset having many more columns(707 vs 26/464) than our other models.

Next we will compare all of the classification models available and see which gives us the best AUC. Pycaret makes it very easy to run this comparison and we an include/exclude models if we want. Here were the comparative results.

```
best = compare_models(sort = 'AUC', exclude=['gbc'])
```

executed in 1h 34m 24s, finished 06:38:58 2020-12-29

	Model	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC	TT (Sec)
lightgbm	Light Gradient Boosting Machine	0.9850	0.8232	0.0577	0.7599	0.1072	0.1052	0.2066	9.3820
xgboost	Extreme Gradient Boosting	0.9851	0.8185	0.0562	0.8890	0.1056	0.1040	0.2209	23.4380
catboost	CatBoost Classifier	0.9852	0.8094	0.0559	0.9904	0.1058	0.1043	0.2332	21.4400
et	Extra Trees Classifier	0.9853	0.7759	0.0651	0.9869	0.1221	0.1204	0.2514	198.3500
rf	Random Forest Classifier	0.9847	0.7747	0.0223	1.0000	0.0436	0.0430	0.1474	106.5780
ada	Ada Boost Classifier	0.9843	0.7624	0.0001	0.1000	0.0003	0.0003	0.0037	145.0700
lda	Linear Discriminant Analysis	0.9811	0.7548	0.0380	0.1363	0.0593	0.0528	0.0642	64.4220
nb	Naive Bayes	0.6816	0.7189	0.5913	0.0356	0.0629	0.0351	0.0811	7.7880
lr	Logistic Regression	0.9843	0.7001	0.0004	0.3500	0.0009	0.0008	0.0115	69.9860
dt	Decision Tree Classifier	0.9726	0.5531	0.1200	0.1215	0.1207	0.1068	0.1069	137.7320
knn	K Neighbors Classifier	0.9843	0.5163	0.0001	0.0400	0.0003	0.0001	0.0014	214.6640
qda	Quadratic Discriminant Analysis	0.9824	0.5000	0.0020	0.0084	0.0028	0.0001	-0.0001	41.8720
svm	SVM - Linear Kernel	0.9796	0.0000	0.0103	0.0519	0.0135	0.0076	0.0117	74.0800
ridge	Ridge Classifier	0.9843	0.0000	0.0000	0.0000	0.0000	-0.0000	-0.0001	7.3160

As we discovered previously the gradient boosting models are giving us the best AUC scores for this classification project. Also, the speed of top 3 models is something to note.

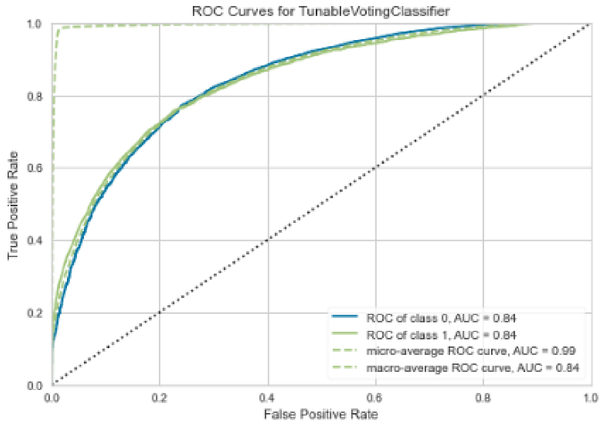
We are already getting slightly better AUC scores with a few different models compared to our previous attempts. We can also use a blend feature in PyCaret to blend our models and see if we can get better results. We took the top 4 models and tried to blend them. We were able to achieve the best AUC using this method.

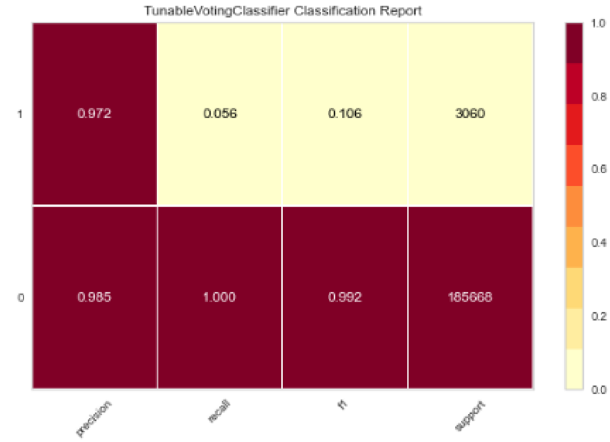
```
#blend top 4 base models
blender = blend_models(blends, choose_better = True, optimize='AUC')
```

xecuted in 54m 4s, finished 08:13:07 2020-12-29

	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC
0	0.9852	0.8351	0.0558	0.9872	0.1056	0.1041	0.2329
1	0.9850	0.8421	0.0428	0.9672	0.0819	0.0807	0.2017
2	0.9850	0.8340	0.0435	0.9677	0.0833	0.0820	0.2035
3	0.9850	0.8406	0.0486	0.9437	0.0924	0.0910	0.2122
4	0.9853	0.8555	0.0645	0.9780	0.1210	0.1193	0.2492
Mean	0.9851	0.8415	0.0510	0.9688	0.0968	0.0954	0.2199
SD	0.0001	0.0077	0.0082	0.0145	0.0148	0.0146	0.0184

Hovering around .84 AUC this blended model is the best performing. Take a look at the resulting ROC curve, confusion matrix and classification report.






```
plot_model(blender, plot = 'confusion_matrix')
```

executed in 30.0s, finished 08:57:25 2020-12-29

In [25]:

```
pred_holdout_blender = predict_model(blender)
```

executed in 23.2s, finished 08:27:04 2020-12-29

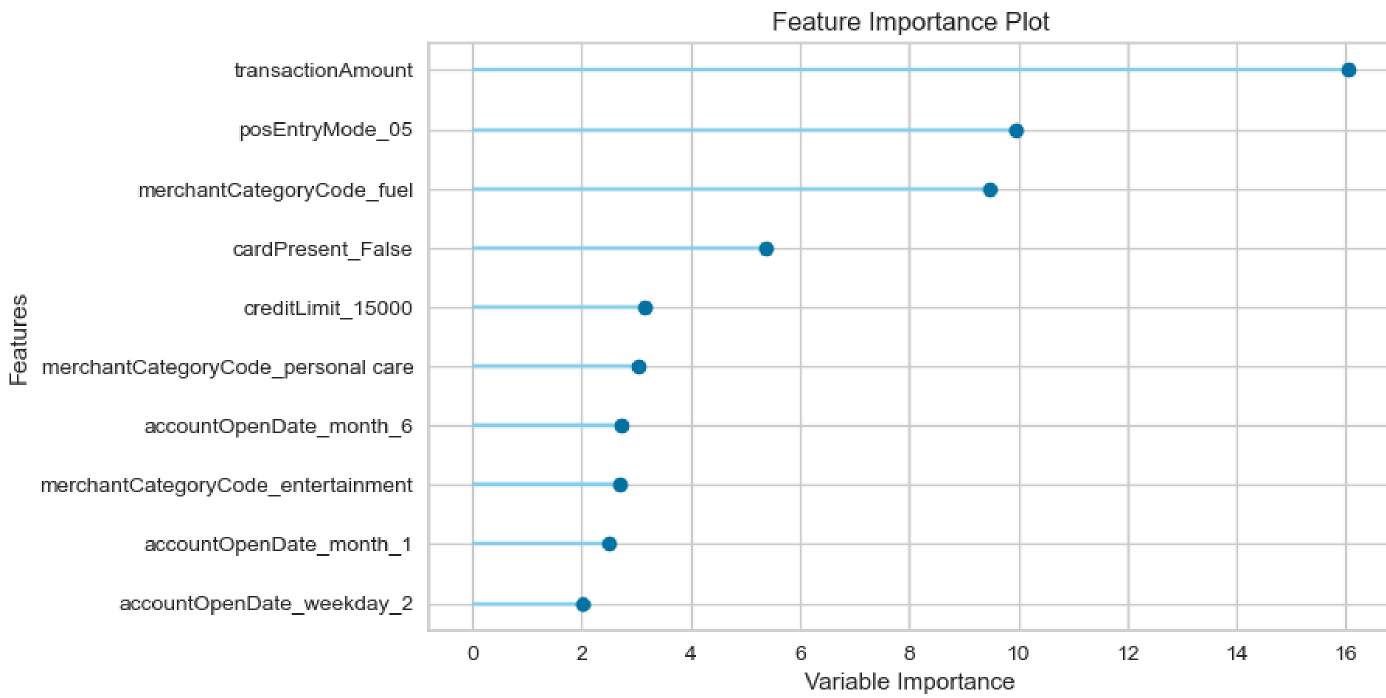
	Model	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC
0	Voting Classifier	0.9847	0.8429	0.0562	0.9718	0.1063	0.1047	0.2318

This blended model yields the best AUC but one important thing is we can't access the feature importances so it's a bit of a blackbox. The best way to improve the model would still be to try out different feature engineering based on the feature importance returned from the models.

Lets run all 3 top gradient models individually and see what features they used. (xgboost, lightgbm, catboost)

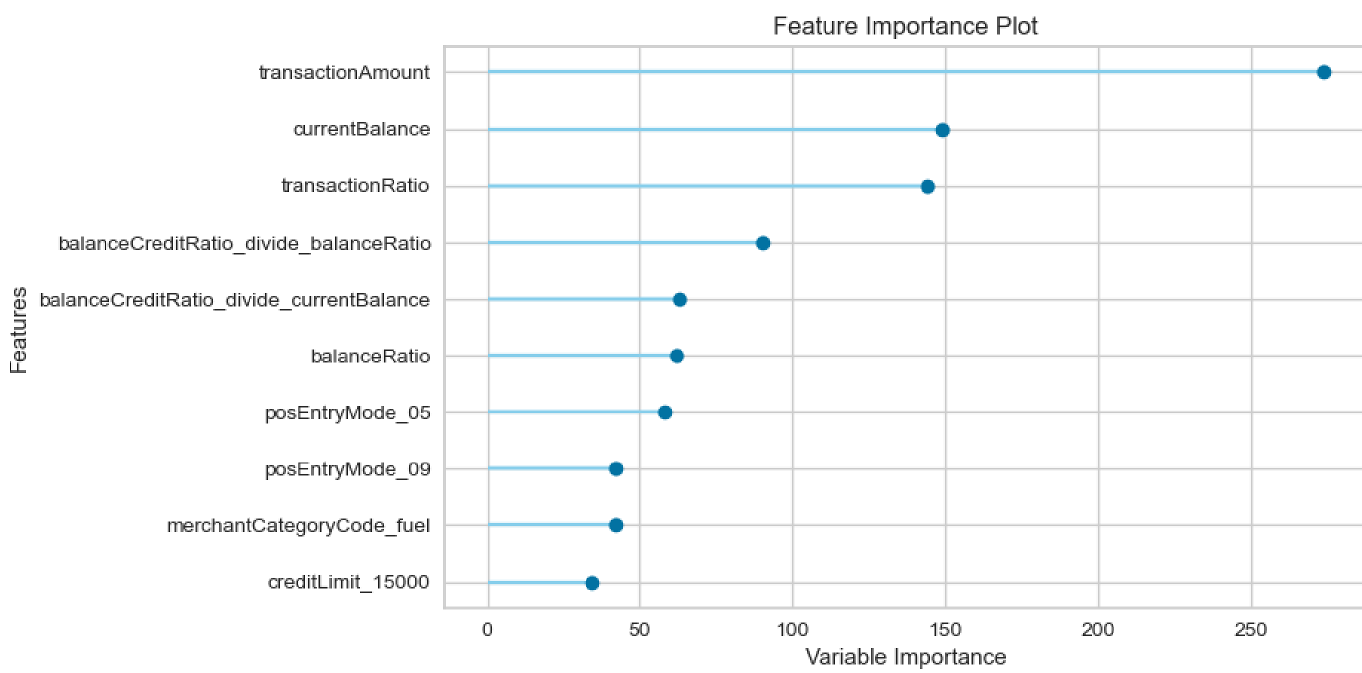
```
1  plot_model(catboost, plot = 'feature')
```

executed in 2.65s, finished 11:24:34 2020-12-29



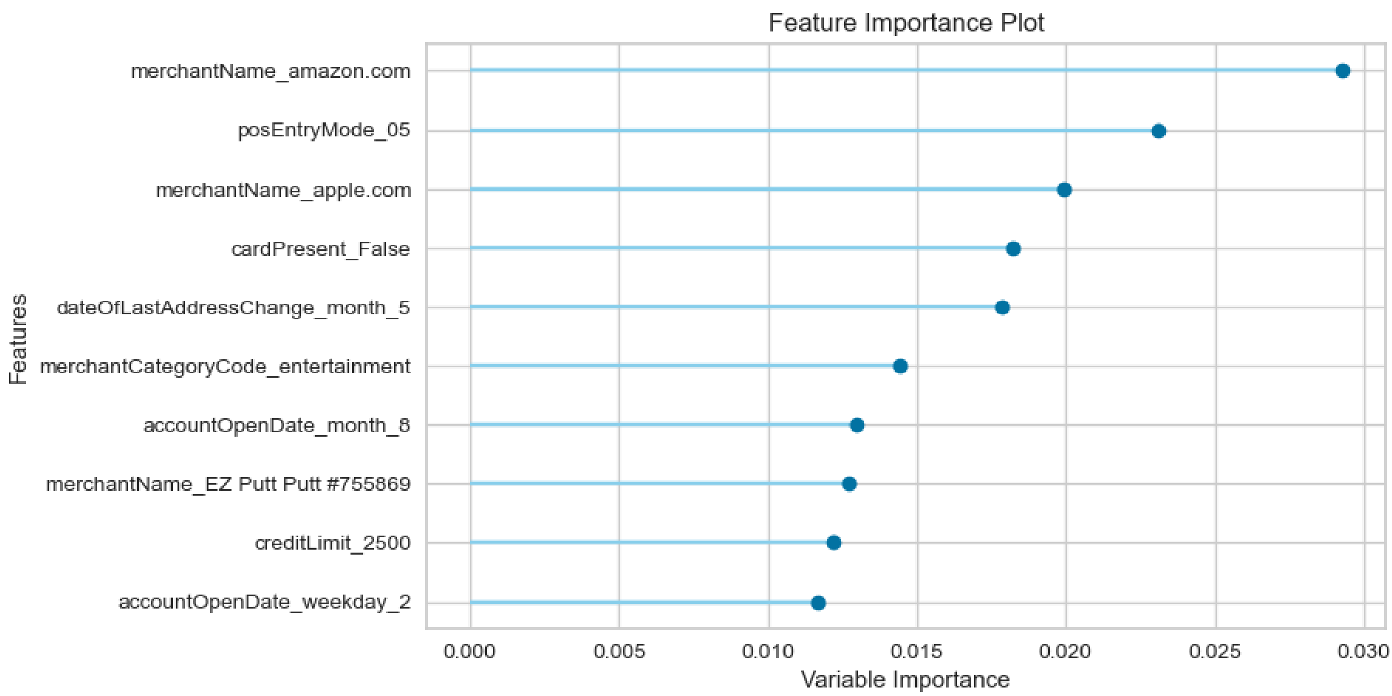

```
1  plot_model(lgbm, plot = 'feature')
```

executed in 2.67s, finished 11:09:19 2020-12-29



```
1 plot_model(xgboost, plot = 'feature')
```

executed in 2.85s, finished 11:09:15 2020-12-29



What's interesting is that these models top 10 features are mostly different yet arrive at a similar evaluation score.

PosEntryMode_05 is the only feature in the top 10 between all 3 models.

XGboost features indicate there are times of the day and certain merchants to look into more closely. We have Amazon, Apple, but also EZ-Putt which seems peculiar, would warrant further investigation. The feature merchantCategoryCode_entertainment also shows up in 2 models again warranting further scrutiny.

LGBM is also interesting in that it was the only model that utilized our manual engineered numeric ratios. We also ran feature_ratios=True in our PyCaret setup environment so we see our manual ratios put against other ratios.

The 4 standout features are transactionRatio, balanceCreditRatio/balanceRatio, balanceCreditRatio/currentBalance and balanceRatio. All these suggest a strong relationship between with currentBalance feature. Perhaps fraudsters are monitoring when money is available in various accounts. The feature merchantCategoryCode_fuel also appears to be one to lookout for as gas cards are a well-known vehicle for fraud.

Catboost and LGBM had transactionAmount as the top feature and it makes sense. During our initial EDA, we saw the average amount per fraud transaction is almost 33% above "normal" so you definitely want to watch out for the amounts. Catboost also has merchantCategoryCode_fuel as its 3rd most important feature, again signaling us to be more alert around these types of transactions.

Both XGboost and Catboost cardPresent_False as their 4th most important feature. Our previous EDA highlighted the discrepancy in this feature between fraud/no-fraud. This is a common sense that a stolen or fraudulent card would be absent or hidden if possible during a fraud transaction and that idea bears out in this dataset. Catboost also utilizes some time of day features that would be good to analyze/explore further.

Overall the gradient boosting models have shown to be superior when it comes to binary classification of rare/infrequent events often with highly skewed distributions. We found that we can eke out a little bit of performance by blending these models together in PyCaret. The most effective way to improve our prediction power is by analyzing the features utilized by the models to help inform our next direction of analysis or inquiry. This in turn will help us create better features and further refine the model.

It's true what they say, model optimization is more often than not a form of feature engineering. We have a model that is strong to start and ready to be deployed, while we also have a few new lines of inquiry to explore further.