1. The project goal was to use machine learning to identify POI's (Persons of interest) in the Enron data set. The data set included 146 data points with 21 features for each individual. Some of the financial features were salary, bonuses, and stock options. There were also email features that included number of emails sent to and received from POIs. There were 18 POIs identified in the dataset. Numerous employees were hard to make decisions about due to having "NaN" values.

   Several employees had extremely high bonuses and salaries along with negative stock values and holdings. These employees were the outliers. While exploring the data set there were two obviously phony data points - "Total" was included for all employees, and one entry labeled "THE TRAVEL AGENCY IN THE PARK" which proved to not be an employee. Both entries were removed. When creating the new features, I looked at the minimum and maximum values before and after.

   data_dict.pop('THE TRAVEL AGENCY IN THE PARK')

   data_dict.pop('TOTAL')

2. At first, I used features_list_n, which included many of the financial and email counts. I was in the hopes that SelectKBest would narrow this down, but it appeared that SelecKBest instead decreased the target thresholds. I then tried different ways to leverage SelectKBest. When I used features_list_n, which included all financial and email features. When I ran SelectKBest I used some different values for K and came up with a smaller list.

   To look at the number of emails sent and received from known POIs I created a feature that would calculate a percentage. I was assuming that people with a higher percentage of emails to and from known POIs would be more likely to also be a POI.

```
'SelectKBest scores: '
[('exercised_stock_options', 24.815079733218194),
 ('total_stock_value', 24.18289867856688),
 ('bonus', 20.792252047181535),
 ('salary', 18.289684043404513),
 ('percent_to_poi', 16.40971254803579),
 ('deferred_income', 11.458476579280369),
 ('long_term_incentive', 9.922186013189823),
 ('restricted_stock', 9.2128106219771),
 ('total_payments', 8.772777730091676),
 ('shared_receipt_with_poi', 8.589420731682381),
 ('loan_advances', 7.184055658288725),
 ('expenses', 6.094173310638945),
 ('from_poi_to_this_person', 5.243449713374958),
 ('other', 4.187477506995375),
 ('percent_from_poi', 3.128091748156719),
 ('from_this_person_to_poi', 2.382612108227674),
 ('director_fees', 2.1263278020077054),
 ('to_messages', 1.6463411294420076),
 ('deferral_payments', 0.2246112747360099),
```

```
('from_messages', 0.16970094762175533),
('restricted_stock_deferred', 0.06549965290994214)]
```

To look improve accuracy and precision I used SelectKBest to scale the features to the ones listed below.

```
['poi',
 'exercised_stock_options',
 'total_stock_value',
 'bonus',
 'salary',
 'percent_to_poi',
 'deferred_income',
 'long_term_incentive',
 'restricted_stock',
 'total_payments',
 'shared_receipt_with_poi']
```

3. Using an idea from one of my references I created functions that could be easily called and modified to test the data. The algorithm I chose to use was Decision Tree. I also tried using Adaboost, SVM, and Naïve Bayes. SVM took too long to run. The other algorithms were either above or below the .3 threshold for precision and the accuracy was closer to 80%. The final output for all algorithms is:

   Adaboost

   training time: 0.054 s

   predicting time: 0.004 s

   accuracy = 0.8604651162790697

   Decision Tree

   training time: 0.003 s

   predicting time: 0.001 s

   accuracy = 0.9069767441860465

   NaiveBayes

   training time: 0.003 s

   predicting time: 0.001 s

   accuracy = 0.8837209302325582

SVM_CLF

ran a about 1.5 minutes with no output

4. Algorithm tuning is critical to model performance. Limitations can be set on particular behavior of the algorithm, such as the maximum height of a building, or the maximum numbers of estimators. Adaboost terminates boosters. SVM can be used to define kernel types, which allows for the selection a particular separation of groups that can be more or less linear.

I leveraged SKLearn to tune my decision tree algorithm. I used the "criterion=entropy" as my parameter. This measures the split decision tree as it runs. I tried to use "min_samples_split" and "max_depth" and was unable to get better performance for either accuracy, precision or recall.

The decision tree algorithm, with the features and modified data set, had the following outcome.

```
Accuracy: 0.82560
Precision: 0.35333
Recall: 0.37100
F1: 0.36195
F2: 0.36733

Total predictions: 15000
True positives:  742
False positives: 1358
False negatives: 1258
True negatives: 11642
```

5. Validation in Machine Learning is the process used to check the validity of the training model by testing the model on an unseen dataset. A classic error when validating data would be to train you model on all of the data at one time. To validate my data, I used the stratifiedshufflesplit found in the tester.py script provided by Udacity.

6. Using the stratifiedshufflesplit my final model had these average performance metrics:

Accuracy: 0.83053

This accuracy means the model was 83.1% accurate in the prediction of whether or not a person was a POI.

Precision: 0.36382

This means that 36.4% of people the model classified as POIs were really POIs.

Recall: 0.36200

This means that the model correctly identified 36.2% of the POIs in the data.