POI Identification

Feature Selection/Data Analysis

(Task 1: Preliminary Feature Analysis)

In determining what features contain the most useful information for the data set, I decided to take every feature currently in the Enron data set and print out relevant pieces of information: the amount of 'NaN' values (i.e. unfilled observations) and the max/minimum values for each category, with email addresses as the exception. What this brought to light is that there are 6 features where the NaN count exceeds 146/2 = 73, or half of the dataset. Those features are deferral payments, loan advances, restricted stock deferred, deferred income, long-term incentive, and director fees. Since so many observations on the data set are lacking values in these categories, it is probably best to completely ignore using them entirely for the sake of accuracy.

(1) Remove the following list of features from potential analysis for having over half of the observations being 'NaN':

- Deferral payments
- Loan Advances
- Restricted stock deferred
- Deferred income
- Long-term incentive
- Director fees

As for which features to use, below is a list of important features I deemed important and justifications:

Feature Name	Justification	Empirical Information
From_poi_to_this_person	POIs may have a larger number of emails from other POIs because they need to engage in shady business practices or cover up information	1. 60 unrecorded observations total ('NaN's) 2. range(0,528)

From_this_person_to_poi	In a similar train of thought to the last feature, POIs may be more likely to email other POIs to cover up information later found in the Enron scandal	1. 60 unrecorded observations total 2. range(0,608)
Total_stock_value	POIs like Kenneth Lay have been known to sell a large amount of stock prior to its value plummeting, so its likely that POIs	1. 20 unrecorded observations total 2. Huge range: -44093 to 434509511
Salary	One explanation for a high salary could be that someone is garnishing salary from some legitimate source of Enron – or in other words, a POI	1. 51 unrecorded observations total 2. Huge range: 477 to 26704229
Shared_reciept_with_poi	If they've been getting emails that are also sent to POIs, there is a good chance they are also involved in any wrongdoing or connected to the POIs in some way	1. 60 unrecorded observations total 2. range(2, 5521)
Bonus	A large or unusual bonus to someone's pay grade could indicate a payoff of some sort for doing someone else's dirty work-i.e, make that person a POI	1. 64 unrecorded observations total 2. Huge range: 70000 to 97343619

These features were picked for potentially valuable information as well as not having too many 'NaN's in the data set. Email related data, like shared_reciept_with_poi, from_poi_to_this_person, and from_this_person_to_poi, generally has a low variation, on the order of 1e2 or 1e3, which means that a simple max-min scaler can be applied to the data without much of a hitch. Another possible method could be even done in conjunction-scaling the amount of messages to or from poi with the total number of messages to or from. This would create a much more tightly knit data set and reduce the variance immensely.

When it comes to salary, bonus total_stock_value however, there is a huge range that the values can take- from an order of 1e7 in salary and bonus to 1e8 in stock value. Outlier detection in this case is most likely important,

as it can reduce the scaling of values, but even so min max scaler would most likely have many values fall to 0.000? or somewhere near that order. In that case, mean normalization might be better, because in the case of having values fall in the order of [-1, 1] could result in unusually low values (or those less than 0) are associated with normal company figures, and unusually high values (those way above 0) are the signs of someone involved in illicit business- a POI.

(2) When implementing the 3 email-related features (from_this_person_to_poi, from_poi_to_this_person, and shared_reciept_with_poi), experiment with different ways of scaling:

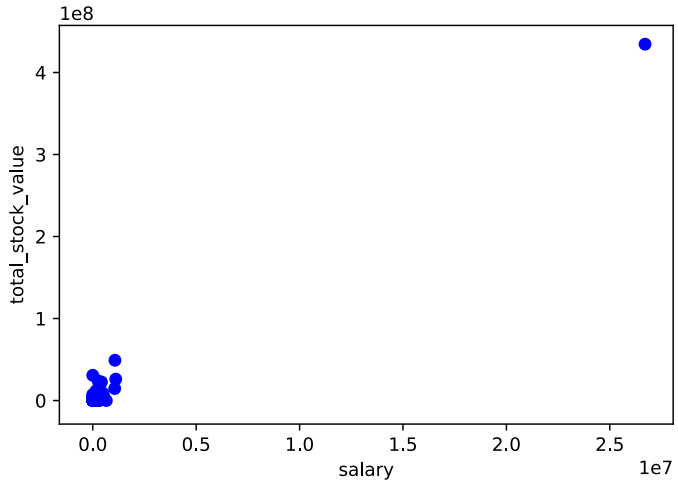
- The features as a stand-alone
- Running them through a min_max_scaler
- Taking the fraction of "from message" features to total from messages, and "to message" features to total to messages from a person
- Not only taking the fraction but applying min_max_scaler afterwards

(3) Outlier detection for the features salary, bonus and total_stock_value is critical- the ranges for all 3 features are very high. This also means scaling will be important, but a simple min max scaler might not still be desirable. Very low values, like 6615 in salary, will be scaled to 0.00023-such minute changes might be hard to distinguish and cause other features to dominate over these variables, so perhaps using mean normalization instead will be a better indicator, because positive and negative values might be mapped as a POI or non-POI indicator respectively. So using one of these methods while scaling will most likely help:

- Min_max_scaler
- Mean normalization

(Task 2: Outlier Detection and Removal)

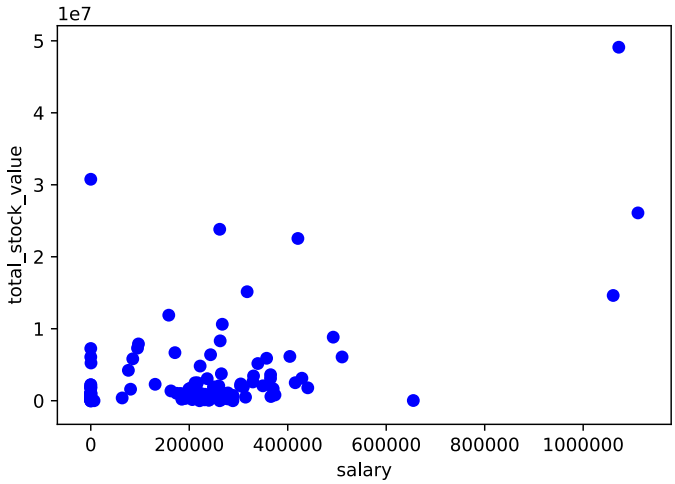
For each of these features, it's important to graph the values to get a sense of any outliers that may be present. An important case would be the outlier seen below when graphing salary and total_stock_value:



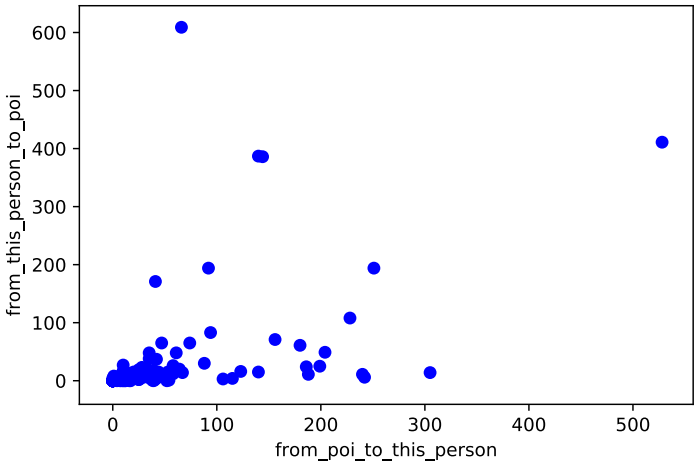
When determining that key in a previous lesson, it turned out it was the "TOTAL" key in our dictionary, the sum of all the individuals. Therefore, it is paramount to remove it because that point could sway decision making on high salary/stock owning individuals not being POI when they might very well be.

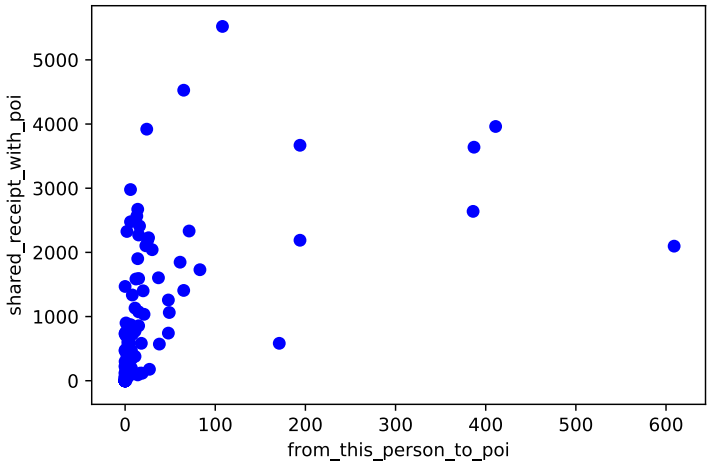
(4) Remove the key corresponding to TOTAL before passing on the dataset

The new scatterplot looks much more reasonable:



As for the other features that were determined important, there are the following graphs (3 features, but displayed on two scatterplots to see if the values are extremes relative to other:





Unlike the salary and total_stock_value, there is no real clear-cut outlier, or at least one that would seem fair. Many of the outliers in the respective variables are POIs, which confirms the inference that unusually high variable values for these features could correspond to POIs, so these values were left mostly untouched.

(Task 3: Feature Scaling and Transformation)

As for feature scaling, I decided to use the fraction of email messages to total email messages that I referenced earlier. I implemented this as below

```
def divide(value1, value2):
    """Arguments:
        value1, str of int
        value2, str of int
    Returns:
        new_val, value/value2 if executable"""
    if value1 == 'NaN' or value2 == 'NaN':
        return 'NaN'
    else:
        new_val = float(value1)/int(value2)
        return new_val

def add_feature(data,name,method):
    """Arguments:
        data, unpickled dict of dicts containing the information
        name, str of the feature name
        method, tuple of 3 elements:
            method[0] = feature_1
            method[1] = feature_2
            method[2] = function to be applied
            Ex: ("salary", "total_stock_value", divide)
    Returns:
        nothing, but name is now a feature for each key in data is added"""
    for key in data.keys():
        value_1 = data[key][method[0]]
        value_2 = data[key][method[1]]
        new_value = method[2](value_1,value_2)
        data[key][name] = new_value
    return None
```

Classifier Selection/Tuning

(Task 4: Exploring Classifier and Feature Combinations)

Below is a list of the possible combinations of features, scaling methods, classifier, and preprocessing I could do to determine the optimal combinations.

Features	Scaling Methods	Classifier	Preprocessing
Salary	Min-Max	Naïve Bayes	PCA
Total_stock_value	Standard	Decision Tree	
Frac_from_poi_to_this_person		K Neighbors	
Frac_from_this_person_to_poi		SVM	
Frac_shared_receipt_with_poi			
Bonus			

Before implementing Grid Search to find optimal parameters for each classifier, the first thing I needed to do was identify any combinations of the other 3 columns that produced optimal results. The only options for each classifier will be ones that pass/are close to 0.3 precision/recall on tester.py.

Note 1: I did not try every combination, but I did try dozens for each classifier in terms of possible feature combinations

Note 2: If PCA and no PCA versions of a combination passed 0.3 recall/precision, then I only considering which option returned the best values

Note 3: Same for note 2 applies to scaling, but in many cases it made minute changes

Classifier: Naïve Bayes
Note: Invariant to feature scaling

Option A

Precision: 0.52320
Recall:0.30450

Option B

Precision: 0.51020
Recall: 0.30000

Option C

Precision: 0.52761
Recall: 0.31050

Option D

Precision: 0.53400
Recall: 0.32200

Option E

Precision: 0.51020
Recall: 0.30000

Features	Scaling Methods	Classifier	Preprocessing
Salary (A)		Naïve Bayes	PCA (A/B/C/D/E)
Total_stock_value (A/B/C/D/E)			
Frac_from_poi_to_this_person (D/E)			
Frac_from_this_person_to_poi (E)			
Frac_shared_receipt_with_poi (B/E)			
Bonus (A/B/C/D/E)			

What's interesting to note here is that I believed all 3 of the new features would contain some insight on the data as a whole, but evidence here suggests otherwise. Options B and E give the same results, but E includes all 3 new variables while B only includes one of them, suggesting that they all follow a similar pattern to each other.

Classifier: Decision Tree

Notes:

- Inherent Randomness (Set random_state to be 2)
- Mostly Invariant to Scaling (changes are very minute, around 0.1%)

Option A

Precision: 0.31250
Recall: 0.33250

Option B

Precision: 0.32342
Recall: 0.34800

Option C

Precision: 0.48603
Recall: 0.39150

Option D

Precision: 0.37064
Recall: 0.32950

Features	Scaling Methods	Classifier	Preprocessing
Salary (A/B/D)			PCA (C/D)
Total_stock_value (A/B)		Decision Tree	
Frac_from_poi_to_this_person (A)			
Frac_from_this_person_to_poi (A)			
Frac_shared_receipt_with_poi (A/B/C/D)			
Bonus (A/B/C/D)			

Interesting that scaling had some effect, but not by any practical means- one notable example being recall decreasing from 0.333 to 0.3250 to get Option A, but precision increased, so it was more of a subjective choice to pick A rather than one objectively performing better. Preprocessing often had a large effect on the precision and recall, wildly changing the results.

Classifier: KNN
Note: Very dependent on feature scaling

Option A

Precision: 0.60734
Recall: 0.28150

Features	Scaling Methods	Classifier	Preprocessing
Salary	Min-Max (A)		PCA (A)
Total_stock_value	Standard		
Frac_from_poi_to_this_person		K Neighbors	
Frac_from_this_person_to_poi			
Frac_shared_receipt_with_poi			
Bonus (A)			

KNN performed extremely poorly, mostly in regards to recall rate. The problem I believe with this dataset is that there are several instances of someone having similar pay grade, getting the same emails, etc. as a POI but not be a POI, and that can mess with the algorithm. This can be especially true given the rather imbalance between the two classes: non-POIs vastly outnumber POIs, so a POI can be surrounded by non-POIs and thus be classified as a non-POI by KNN. Given the tunable nature of the algorithm, I decided to leave the closest option: only using the bonus. In addition to Naïve Bayes and Decision Trees, KNN shows that bonus is a robust way of detecting POIs.

Classifier: Support Vector Machine

Not even worth writing up a report for this method- kept getting the warning that meant no true positive identifications for POIs existed in tester.py. The accuracy was not bad, in the 0.80s, but the precision and recall were always zero-I can't fully explain the poor performance of SVM but it was disheartening that nothing I tweaked seemed to improve it

Classifier: Random Forest

Notes:

- Largely Independent to scaling
- Inherent randomness (set random_state to be 2)

Option A

Precision: 0.41452
Recall: 0.30550

Option B

Precision: 0.48887
Recall: 0.29650

Features	Scaling Methods	Classifier	Preprocessing
Salary			PCA (A/B)
Total_stock_value			
Frac_from_poi_to_this_person			
Frac_from_this_person_to_poi			
Frac_shared_receipt_with_poi (B)		(Random Forest)	
Bonus (A/B)			

RandomForest wasn't my original idea for classifier testing, but seeing how the Decision Tree classifier performed made me want to try it out in retrospect. The run time made running tester.py annoying, so I made sure to try and screen for good values in my own tests before running it there. Overall getting recall high enough was a challenge, and it definitely performed poorly the more features it was exposed to. Given the ability to tune, I decided to record Option B anyway for further testing.

(Task 4b: Exploring Classifier and Feature Combinations)

How I screened a combination before running tester.py:

In order to have my own testing so I didn't have to run tester.py for every single possible permutation, I decided to test using both train/test splitting and 10-fold cross validation. The function is called classifier_cross_validation, and here is an example output:

```
PCA Random Forest Classifier

     Result of 30% Test set size:
Training time (s): 0.02139
Testing time (s): 0.0012
Accuracy: 0.76
Recall: 0.5
Precision: 0.6667
     Result of 10-Fold cross validation (avg):
Training time (s): 0.015
Testing time (s): 0.0013
Accuracy: 0.7653
Recall: 0.2833
Precision: 0.2333
```

Doing this gave me some intuition of judging whether it was worth to running tester.py to get a possibly passing value for both precision and recall. Since there was still a decent amount of plugging and chugging in terms of feature selection, I only left the best option for each classifier (if it exists) for Task 5

(Task 5. Tuning Best Classifiers):

By using GridSearchCV, I tested parameters on the optimal values from Decision Tree, KNN and RandomForest-Bayes doesn't really have inputs. What's interesting to note is that instead of using a regular scoring function like f_1, I decided to use a custom scoring function. Using the code

from test_classifier, I modified it to return the f1_score, and maximized my classifier's results on that value. This does mean that with a large parameter space, it would take a couple minutes to tune each of the classifiers, especially random forest. It took over 10 minutes to optimize 12 possible permutations-over 3 times as long as KNN and Decision Tree tuning, and I gave those even more combinations. Random Forest's run time made me realize that tuning it to such a value was most likely not very worth it, and is definitely not scalable to larger datasets. Below are the results of tuning what I determined the best option for each classifier.

Naïve Bayes:

Precision: 0.52761
Recall: 0.31050

Tuned KNN

Precision: 0.48006
Recall: 0.31300

Tuned Decision Tree

Precision: 0.46745
Recall: 0.42000

Tuned Random Forest

Precision: 0.45359
Recall: 0.32500

RandomForest and KNN do fairly similarly, while Naïve Bayes and the Tuned Decision Tree optimize precision and recall respectively. Here, the answer lies in what each statistic means: High precision indicates most of the POI identifications are true, while high recall means most of the POIs have been identified. Even though it would be desirable to be as precise as possible in making sure that no innocent people are accused of being POIs, I personally think that its more valuable to ensure that most of the suspicious individuals are caught. After all, my stance is that this is best suited as an identification algorithm for further questioning of individuals, not the best evidence possible. Therefore, my final classifier is the tuned Decision Tree.

NOTE: I realized that for frac_shared_receipt_with_poi, I had a person with a value over 1, which did not make sense to me because I divided the number of times he was a receipt with a poi by the total number of messages sent to him. I searched up the key with intentions to remove it, but ended up not, and just realized near the end of my report when double-checking. My decision tree's precision jumps to 0.48667 after removing it, so it looks like I should have removed it.

Conclusion of methodology:

- Remove key “TOTAL” because it corresponds to the sum of the other observations
- Add feature frac_shared_receipt_with_poi, which takes shared_receipt_with_poi and divides it by to_messages.
- Use following features:
  - Frac_shared_receipt_with_poi
  - Bonus
- Decision Tree with the following parameters (excluding defaults):
  - Criterion = ‘entropy’
  - Random_state = 2
- Using pipeline to apply PCA to the features (default parameters), then the Decision Tree
- Adding all 3 new variables were most likely not worthwhile, because they seemed to encapsulate repeat information when multiple were used for a classifier as opposed to just one (i.e. classifier did not do significantly better or worse).
- Plugging and chugging feature combinations got slightly tedious at times, and I could have used SelectKBest or another feature-reducing algorithm, but considering the low feature space I limited myself to I thought individually testing was worth it.