1. This is a fantastic public dataset for an important piece of US business and financial history. The dataset features financial data and emails for many people involved in the Enron financial scandal that saw a top 10 global company descend into bankruptcy within 2 years due to their shady practices. There were a number of features to work with, many of them being financial as those were the simplest to quantify which range from salary and bonus to stock. Other important features were those found by scanning the emails. While I could have created new features by analyzing ‘bag of words’ frequencies for specific word lists and tying them to individuals, I decided to keep the analysis simple by analyzing the number of emails send from one person of interest to another. The most important part of the dataset are the labels which come from the actual outcome of the investigation whom were individuals who were indicted, found guilty, or testified for immunity. Without those labels this exercise would have been a case of unsupervised learning and the best we may have been able to do would be to group certain patterns together and infer guilt using those outcomes. There were a few outliers in the dataset. I removed a total of three outliers and elected to leave in a few others that perhaps some of the other student removed. I used the ‘pop’ function to remove the ‘TOTAL’ dictionary key as we had done throughout the lectures. I also removed two individuals:

data\_dict.pop('BELFER ROBERT', 0)

data\_dict.pop('BHATNAGAR SANJAY', 0)

The data for these two individuals were mis-categorized as they were placed in incorrect columns. I found them during feature selection as there were negative numbers in unexpected columns. Since they didn’t have much other information other than deferred pay, I removed them. I could have also removed the ‘non-individuals’ such as the Travel company, but elected to leave it in as there were already so few data points in the set as a whole (142).

1. It seemed like there were many different measurable features, but only a handful of latent features that contain the real information. To help decide which carried the most value I split up the features by unit type ($, # of emails, fraction of emails). I created three new features that were not contained in the project dataset by writing a script similar to the one at the beginning of the ‘Feature Selection’ lesson. I modified the dictionary by adding the fraction of emails sent to and received from a POI, and I created a new fractional feature for the fraction of emails with ‘shared\_receipt\_with\_poi’ to the total of messages the individual receives. I figured that people with a higher fraction of their total emails received that are also sent to other POIs would also be a POI and that the fractional value would help separate more from non-POI.

I used SelectKBest combined with the Chi2 function on the financial and stock data starting with 5 selections. The scores (in order of listing from the financial data PDF) are thus:

3.32175323e+06 4.03860512e+07 1.28500313e+07 6.06347927e+05 1.73236159e+07 3.30975525e+05 1.88131500e+05 3.10511704e+08 2.69763730e+08

Out of these I chose to keep the top three features (in order of importance): Total Payments, Total Stock Value, and Bonus as I felt these financial features best expressed the others.

I used the Lasso function to gauge the importance of the email features, with an alpha of 0.00001. The coefficients of the email statistics were all very small, so I elected not to use ‘To messages’, ‘From messages’, or the others. The Lasso function on the fraction of emails was more indicative giving coefficients of (‘fraction to poi’, ‘fraction from poi’, ‘fraction shared with poi’):

0.4863963791, -0.2762912358, 0.1617155927

Each of these were used in my final feature list:

features\_list = ['poi', 'bonus', 'total\_payments', 'total\_stock\_value', 'fraction\_to\_poi',

'fraction\_from\_poi', 'fraction\_shared\_with\_poi']

After sending these latent features into FeatureFormat and TargetFeatureSplit, I needed to scale the features since they had such a difference in magnitudes (and units). I used the MinMaxScaler to normalize all of the values to range between 0 and 1.

1. Since this exercise is more of a classification exercise than regression, I needed to try those we learned towards the beginning of the course. Since I was curious on how many different algorithms would fare and to get more practice using what I learned in the course, I decided to code up and test the following: Naïve Bayes, SVM, Decision Trees, KNearestNeighbors, Random Forest, and Adaboost.

I elected to use the Train\_Test\_Split feature we had been using most of the course with a test size of 0.25 on the labels and scaled features with a defined random state. I also measured the time it took to evaluate each classification algorithm since I would be running each in series on the same run of code. To measure the performance of each run, I had the code output the accuracy score, recall score, and precision score of each algorithm. The Naïve Bayes was the easiest to use since the default features were enough and it gave the same results each time. The others I began using a GridSearchCV parameter scan and printout the best estimator properties. For the SVC, I scanned the kernel, C, and gamma. For the DecisionTreeClassifier I scanned criterion, splitter, max\_features, and min\_samples\_split. For KNN I scanned n\_neighbors, weights, algorithm, and leaf\_size. For Random Forest I scanned n\_estimators, criterion, max\_features, and min\_samples\_split. Lastly for AdaBoost I scanned n\_estimators, learning\_rate, and algorithm. As far a time was concerned, Random Forest took the longest (41 s) followed by AdaBoost (18 s) and then SVM (8 s).

At first none of the precision or recall values were above 0.2 despite the accuracy approaching 0.9. I then tried to use PCA with each algorithm fed through a pipeline (still using grid search) and scanning the number of PCA components to use which varied for each algorithm. After finding the best combination of tuned parameters for each algorithm, I hard-coded those parameters for each one to measure and save the classifiers for the testing algorithm.

1. Tuning the parameters of the algorithm means to optimize the input parameters for the classifier functions as some inputs will out-perform others. If you don’t do this well, the worst thing that will happen is that the algorithm won’t converge on a solution. If it does converge, non-optimal parameters will show poor performance results. I mentioned how I tuned my algorithms in question 3 above using GridSearchCV, Pipelines, and PCA before running those identified parameters by themselves. Here is the output for a few of the best estimators:

SVM Best Estimators: Pipeline(steps=[('reduce\_dim', PCA(copy=True, n\_components=6, whiten=False)), ('clf\_svm', SVC(C=1000, cache\_size=200, class\_weight='balanced', coef0=0.0, decision\_function\_shape=None, degree=3, gamma=1.0, kernel='rbf', max\_iter=-1, probability=False, random\_state=None, shrinking=True, tol=0.001, verbose=False))])

Decision Tree Best Estimators: Pipeline(steps=[('reduce\_dim', PCA(copy=True, n\_components=2, whiten=False)), ('clf\_tree', DecisionTreeClassifier(class\_weight=None, criterion='entropy', max\_depth=None, max\_features='sqrt', max\_leaf\_nodes=None, min\_samples\_leaf=1, min\_samples\_split=2, min\_weight\_fraction\_leaf=0.0, presort=False, random\_state=None, splitter='best'))])

KNN Best Estimators: KNeighborsClassifier(algorithm='auto', leaf\_size=5, metric='minkowski',

metric\_params=None, n\_jobs=1, n\_neighbors=8, p=2, weights='uniform')

1. Validation is essentially measuring the performance of ones algorithm, but in more ways than just the accuracy of the predictions. If all we are chasing is accuracy then it can become easy to violate the bias-variance balance and overfit the data. Accuracy will not be a useful metric for skewed classes (perhaps as is the case in the Enron corpus), setting a default class by taking caution, or when the performance is tuned for one specific type of error. The useful metrics here are the recall score and precision score which I mentioned in my code in question 3. I also created a scoring metric on my own by taking the product of the accuracy, recall, and precision in order to provide a rank for each of the algorithms.

|  |
| --- |
| Algorithm Type Accuracy Recall Precision Score Metric Time Taken (s) |
| Naive Bayes 0.89 0.20 1.00 0.178 0.00 |
| Support VM 0.81 0.60 0.38 0.181 0.42 |
| D. Trees 0.83 0.60 0.43 0.214 0.80 |
| K. Nearest 0.86 0.00 0.00 0.000 0.00 |
| R. Forest 0.86 0.00 0.00 0.000 0.08 |
| AdaBoost 0.86 0.00 0.00 0.000 0.11 |

**GaussianNB()**

Accuracy: 0.86027 Precision: 0.45960 Recall: 0.27300 F1: 0.34253 F2: 0.29713

Total predictions: 15000 True positives: 546 False positives: 642 False negatives: 1454 True negatives: 12358

**DecisionTreeClassifier**(class\_weight=None, criterion='entropy', max\_depth=None,

max\_features='sqrt', max\_leaf\_nodes=None, min\_samples\_leaf=1,

min\_samples\_split=4, min\_weight\_fraction\_leaf=0.0,

presort=False, random\_state=None, splitter='best')

Accuracy: 0.82873 Precision: 0.34325 Recall: 0.31150 F1: 0.32661 F2: 0.31737

Total predictions: 15000 True positives: 623 False positives: 1192 False negatives: 1377 True negatives: 11808

**RandomForestClassifier**(bootstrap=True, class\_weight=None, criterion='entropy',

max\_depth=None, max\_features='sqrt', max\_leaf\_nodes=None,

min\_samples\_leaf=1, min\_samples\_split=5,

min\_weight\_fraction\_leaf=0.0, n\_estimators=40, n\_jobs=1,

oob\_score=False, random\_state=5, verbose=0, warm\_start=False)

Accuracy: 0.86747 Precision: 0.50696 Recall: 0.21850 F1: 0.30538 F2: 0.24656

Total predictions: 15000 True positives: 437 False positives: 425 False negatives: 1563 True negatives: 12575

**AdaBoostClassifier**(algorithm='SAMME', base\_estimator=None, learning\_rate=0.1,

n\_estimators=50, random\_state=None)

Accuracy: 0.87067 Precision: 0.59091 Recall: 0.09750 F1: 0.16738 F2: 0.11705

Total predictions: 15000 True positives: 195 False positives: 135 False negatives: 1805 True negatives: 12865

The SVM and KNN classifiers in the tester.py gave divide by zero errors that I didn’t take the time to tune out since the decision tree classifier gave a satisfactory result. I am going with the decision tree results.

1. My evaluation metrics are listed above. I will describe the Decision Tree classifier since it gave both a precision and recall score above the requisite 0.3. The precision is the measure of how many items were labelled positive and how many truly belonged there. The recall is the measure of how many items labelled positive were correctly classified. In other words, the algorithm correctly labelled 623 items as being a POI, falsely accused 1192 of being a POI, and missed 1377 items that should have been called a POI. The algorithm correctly claimed 11808 as being innocent. Of the algorithms I tested, the DecisionTreeClassifier, without more tweaking of features, was able to correctly identify more POI and fewer incorrectly labelled POI than the others I tried.