Linh Tam Trinh
October 27, 2016

P5: Machine Learning with Enron Fraud

1. Data Exploration

The goal of this project is to use machine learning to find persons of interest (POI) in the Enron Fraud. Machine learning can be very useful in accomplishing this goal by quickly learning the trends distinguishing POI's from non-POI's. In this data set, there are 146 samples. Of these 146 people, 18 of them are POI's and 128 are non-POI's. This percentage breakdown of 88% non-POI to 12% POI could be useful for predicting the POI's. The data set contains 21 features on POI, emails, and finances. Since many features are not applicable to everyone, such as "bonus", there are many features with "NaN" for missing values. Counting the number of 'NaN's shows that POI's tend to have a lower percentage of 'NaN's than non-POI's.

Various features were also plotted to detect outliers. In a plot between 'salary' and 'bonus', there was a data point that was much higher than the rest, investigating the data point revealed that it was the "total". Since this was not reflective of the sample and was a quirk of the data collection, it was removed using the ".pop()" function. After the "total" was removed, a few other outliers were detected and investigated. Some of these outliers belonged to POI's and could be telling information, so they were left in the data set.

2. Features Selection

For the features selection, SelectKBest and PCA were both used in conjunction with MinMaxScaler. Before I selected the features, I added a created a new feature called "NaN_feature", which is the number of "NaN" in each sample. From the data exploration, the percentage of "NaN" for POI's was noticeable lower than the normal. This is an interesting trend

and may be indicative of whether or not a person is a POI. Perhaps POI's tend to have a lower percentage of "NaN"s because they are more involved in the company or simply that POI's tend to receive more benefits from the company such as "bonus" and "total_stock_options". After adding the new feature to the data, I decided to scale all of the features. Since some features range in the millions, such as "salary" and some features range double digits, such as "NaN_feature". Initially, I was using the RandomForest classifier and after scaling the features, the algorithm precision value dropped slightly (from 0.5 to 0.33), perhaps an important feature, such as "bonus" had been scaled. To improve the scores, I used Select K Best to tease out the best features. Using the f_classif parameter, I looked for feature scores above the F-critical value of 3.91. Thirteen features met this criteria, including my new "NaN_feature" (feature score 7.81). This increased the precision score back to 0.5. Afterwards, I used PCA to tease out the best components and used the top ten components because they had a much higher explained variance ratio. This kept the accuracy, precision, and recall scores the same (accuracy: 0.90, precision: 0.5, recall: 0.33).

After comparing the tuned algorithms, GaussianNB had the best metrics, therefore I switched to it for my final classifier. I wanted to double check to make sure SelectKBest and PCA worked well with GaussianNB. After plotting the feature scores (*figure 1*) and plotting the SelectKBest metrics (*figure 2*), it seems that using only one feature (k=1) had the highest metrics for the least number of features. This is a surprisingly low number of features and may be the result of poor validation, therefore I tested it on the Tester.py function. After testing it on the Tester.py function, fourteen features had the highest scores for the least number of features (*figure 3*). Therefore I decided to use k=14 for SelectKBest. After analyzing the PCA explained variance ratio (*figure 4*) and the Tester.py metrics scores for different numbers of components (*figure 5*), ten components had the highest metrics scores.

Figure 1. Feature Scores

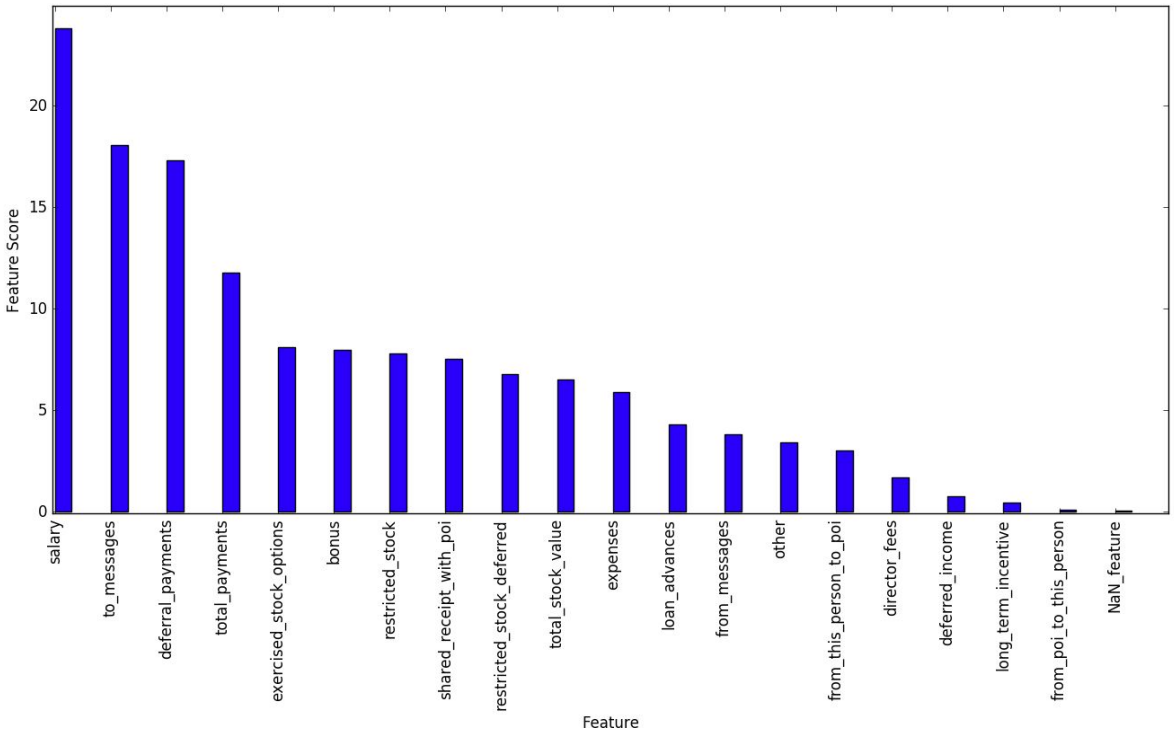


Figure 2. SelectKBest Validation Metric Scores

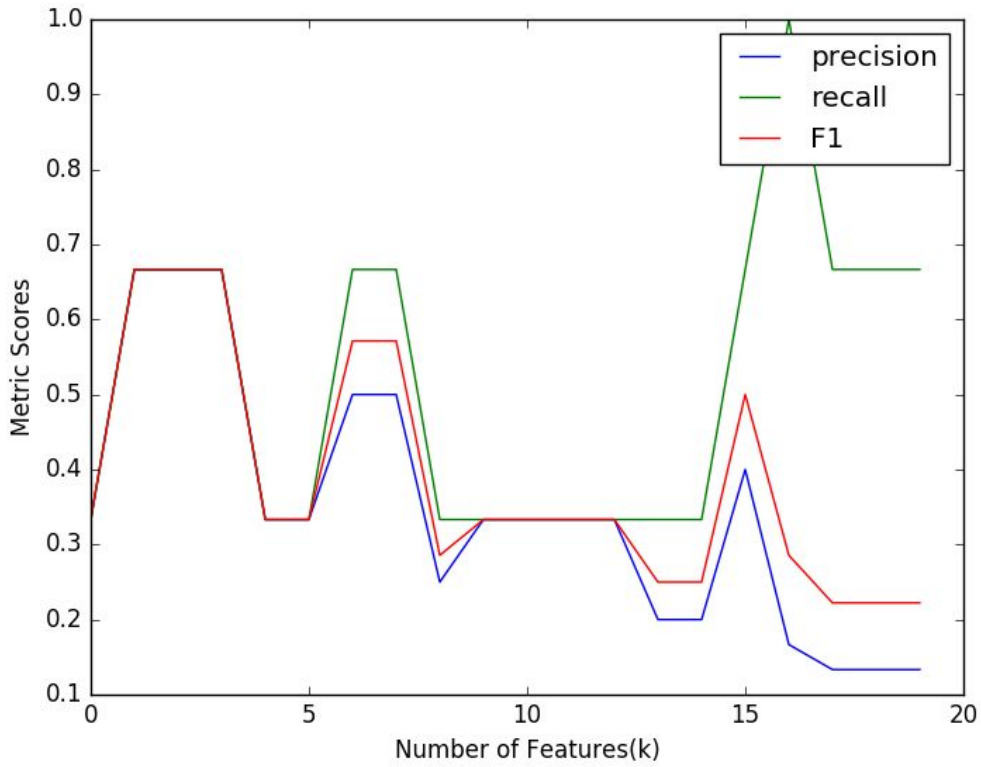


Figure 3. SelectKBest Tester.py Metrics Scores

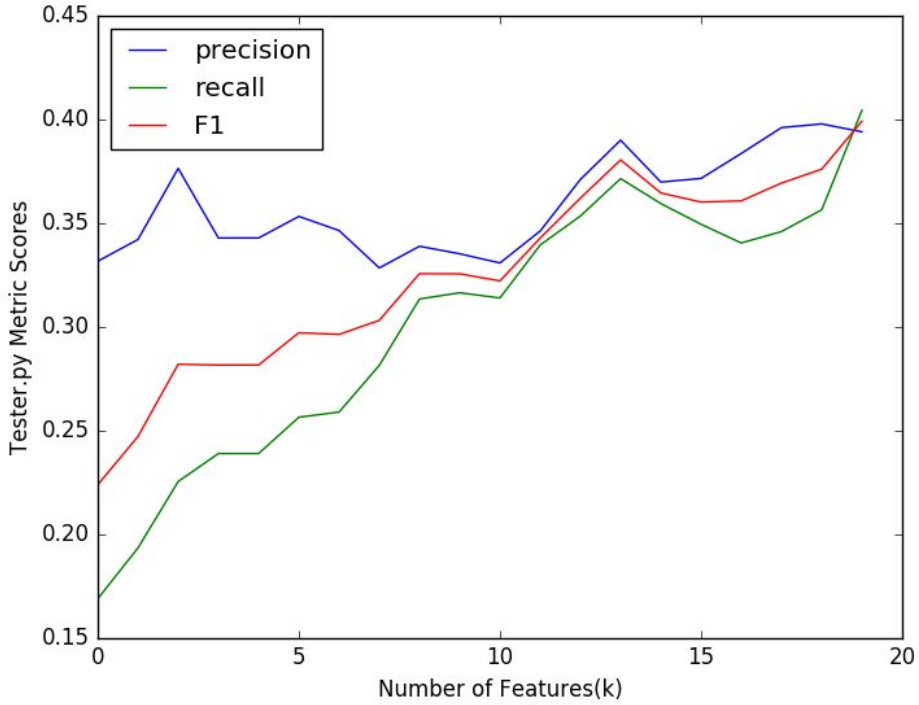


Figure 4. Principal Component Variance Ratio

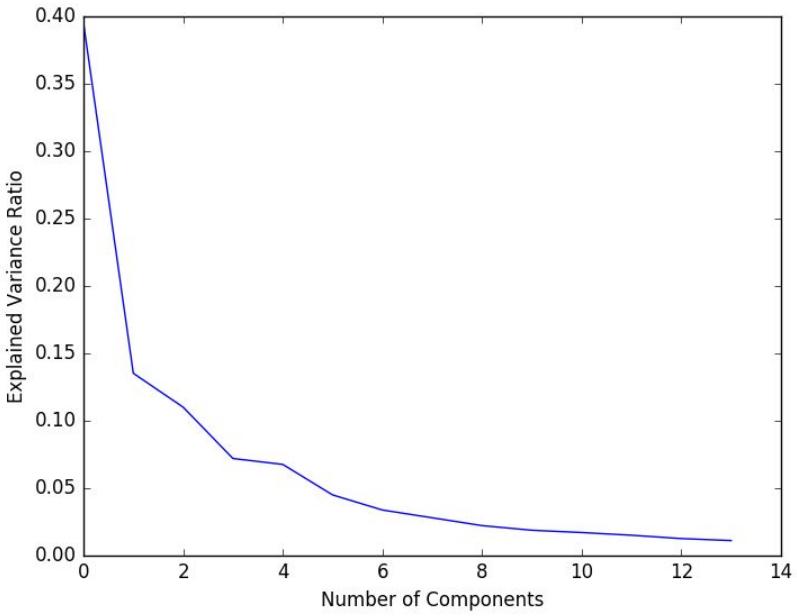
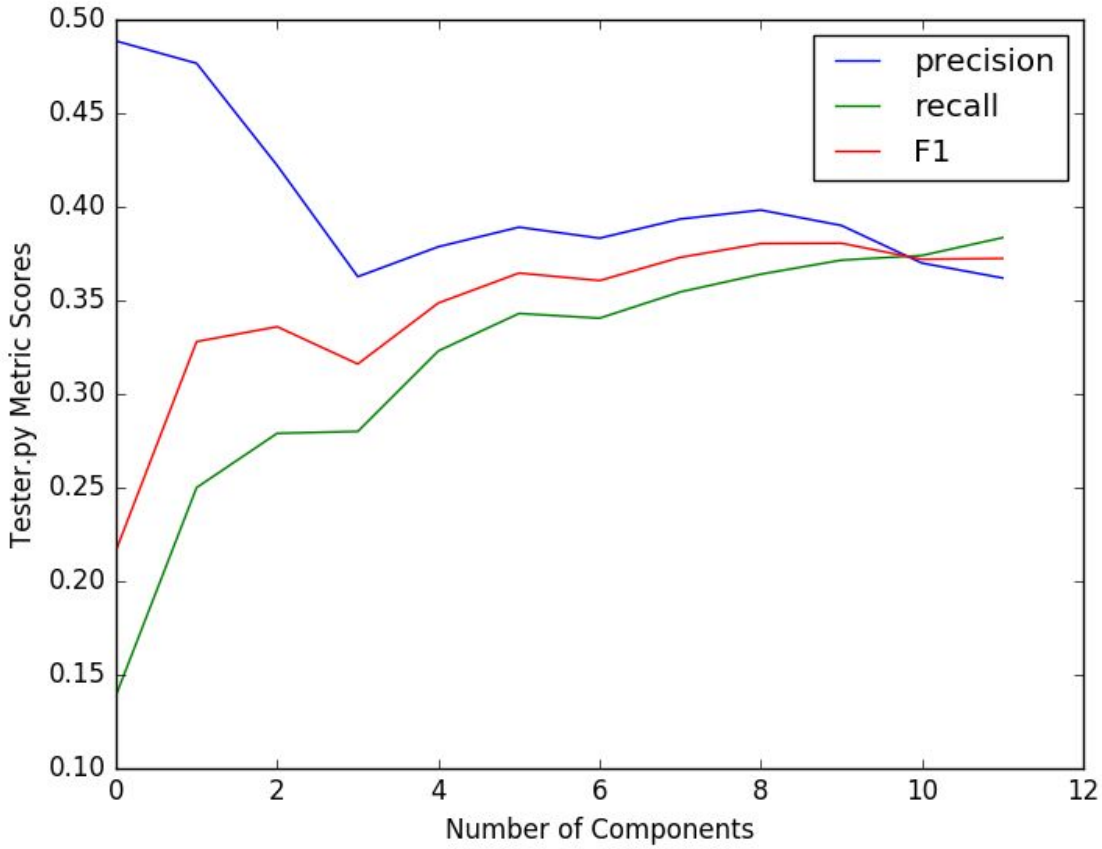


Figure 5. PCA Tester.py Metrics Scores



3. Algorithm Comparison

First, I started out the algorithm comparison by testing the accuracy, precision, and recall scores for RandomForest, GaussianNB, and DecisionTree on the unprocessed data before feature scaling and feature selection. RandomForest had the best scores (accuracy: 0.90, precision: 0.5, recall: 0.33), therefore it was chosen over GaussianNB (accuracy: 0.83, precision: 0.25, recall: 0.33) and DecisionTree (accuracy: 0.83, precision: 0.33, recall: 0.67). Before the final algorithm was selected, I retested the algorithms with parameter tuning on the feature scaled and feature selected data. After the data processing, GaussianNB yielded the

highest scores (accuracy: 0.84, precision: 0.29, recall: 0.5, F1: 0.36). RandomForest (accuracy: 0.91, precision: 0.5, recall: 0.25, F1: 0.33) and DecisionTree (accuracy: 0.91, precision: 0.5, recall: 0.25, F1: 0.33) yielded similarly lower scores. This led me to use GaussianNB as the final algorithm.

4. Parameter Tuning

Parameter tuning can be very important for an algorithm. The default algorithm settings may not be relevant to the data set, therefore it is important to adjust the parameters to help cater the algorithm to the particular trends in the data set. Tuning an algorithm well can help it become robust and achieve higher metrics scores, while tuning an algorithm badly can make it overfit, overlook data, or otherwise achieve lower metrics scores. For the GaussianNB algorithm, there was only one parameter to tune, the “priors” parameter (known information on the classes). By looking at the distribution of POI’s to non-POI’s, I was able to find the distribution between the classes in the training data set and in the tester.py data set. These distributions were labeled under variable names “priors_training” and “priors_tester”. Using GridSearchCV to find the best parameter between “priors: None, priors_training, priors_tester”, I found that the best parameter was “priors=None”. The parameter tuning was not needed in this case.

5. Validation Strategy

When creating an algorithm it is essential to be able to validate its performance. Validating an algorithm involves using training data to train the algorithm, then testing it on a separate testing data for validation on how well the algorithm performed. A simple validation method is to split the data into a training set and a testing set (oftentimes in larger data sets 90% training to 10% testing is used). The algorithm would then be trained on the training set

and tested on the testing set. A better performance method would be to use K-fold cross-validation. This involves dividing the data into groups called folds. All of the folds would then take turns being used as testing data and the average scores across the folds are reported. A classic validation mistake would be to overfit the data. By using too much data on the training set and not enough in the testing set, the data can be overfitted to the training and perform poorly on the testing set.

In my algorithm selection, I used a simple split with “test_size=0.2”. Since the data sample was small, I needed to use more than 10% test size to differentiate between performances. I also wanted to use as much data for training as possible, so I went with a 20% test size. Since the data set was so small, I would often get the same performance for different algorithm tunings. Looking back, I should have used K-fold cross-validation to better validate my algorithm decisions.

6. Evaluation Metrics

To evaluate my algorithm, I initialled used accuracy, precision, and recall. Accuracy is the likelihood the algorithm predicts correctly. Precision is the likelihood the algorithm predicts correctly, given that it predicts POI (true pos./(true pos. + false pos.)). Recall is the likelihood the algorithm predicts POI out of the total instances that it is POI (true pos./(true pos. + false neg.)). Since my data set was small and I used a simple validation method, I often received similar accuracy, precision, and recall scores for several tunings. A mistake I made was not using the F1 score from the start. Since precision and recall can often trade-off on each other, the F1 score is a good metric since it is a weighted combination of both precision and recall. My final tuned GaussianNB algorithm achieved accuracy 0.84, precision 0.29, recall 0.50, and F1 score 0.36.