Overview

The Enron dataset is a dataset of email and financial data obtained in the aftermath of Enron's collapse due to corporate fraud. I want to examine the dataset to uncover patterns of fraud. Machine learning is a useful tool to do this. The applications for this sort of analysis are wide-ranging; they could be adapted to identify fraud in other areas, which could have important societal benefits.

The dataset has 23 features and 146 observations. 18 people are persons of interest (POI), meaning that they were linked to the fraud committed by Enron. This means that 12 percent of the people in the dataset are POIs.

The dataset has three categories of variables. One category is of financial variables, the amount and type of compensation people received. The second is of email variables. This measures the number of emails sent and received, as well as whether these emails were sent or received by POIs. The third category has a single variable, is whether the person is a POI.

Outliers and NaNs

The dataset used 'NaN' to signal that there is no number in a certain cell. I changed NaNs to zeros so that algorithms would be able to run. One group of outliers contained the totals of each variables. I removed this row because it isn't necessary for the analysis. Other outliers, like high payments, I left in because they didn't represent errors in the dataset. I cleaned outliers from variables with the regression technique by using code adapted from P5 of Udacity' Data Analyst Nanodegree (DAND).

Feature selection

I used SelectKBest to select the best features. I used the original features of the dataset except for email_address. I created three features. poi_messages is a measure of how much of a person's direct (without cc's) email communication involved POIs. I figure that people with a higher percentage of their communication involving POIs is more likely to be a POI themselves. from_poi_fraction and to_poi_fraction are similar to the example used in the outliers section of the DAND. However, the final model performs slightly better without the new features than with them, so I left them out.

The performance of the model with the added features is below:

Accuracy: 0.84720
Precision: 0.41541
Recall: 0.35850 F1: 0.38486
F2: 0.36860

The performance of the model without the added features is below:

Accuracy: 0.84833 Precision: 0.41964
Recall: 0.35900 F1: 0.38696
F2: 0.36968

As you can see, the model's numbers for accuracy, precision, and so forth are slightly higher without the added features than with them.

I scaled features when called for by the model.

For my final model I used five features. I tried different values of SelectKBest, and the model performance peaks at k = 5.

Here is the performance of the model with k = 4, meaning that SelectKBest used the top four features:

Accuracy: 0.84827 Precision: 0.41364
Recall: 0.33050 F1: 0.36743
F2: 0.34434

With k = 5:

Accuracy: 0.84833 Precision: 0.41964
Recall: 0.35900 F1: 0.38696
F2: 0.36968

With k = 6:

Accuracy: 0.84073 Precision: 0.38332 Recall: 0.31950 F1: 0.34851 F2: 0.33051

The model's performace is best at k = 5, and begins to decline as you move away from that number.

The final features and their scores are: [('deferred_income', 20.768541617643322), ('expenses', 20.308150998047442), ('from_messages', 3.0258706360844987), ('restricted_stock_deferred', 2.9495681097983315), ('poi', 2.2317026459469886)]

Model selection and performance

I tested naïve Bayes, decision tree, support vector machine, and k nearest neighbors. I used GridSearchCV to tune the parameters of the models.

What is parameter tuning? Parameter tuning means changing the various parameters of each model in order to optimize its performance. The definition of "optimal performance" depends on the goals of the experiment. It could be to maximize accuracy, minimize precision and/or recall, or maybe maximizethe model's speed.

While tuning a model is useful, it is possible to over-tune a model. Doing this runs the risk of overfitting your model to your data. The model would show good performance on the training data but classify new points poorly. For similar reasons, you shouldn't tune parameters to training data. You want to use a testing data set to evaluate your model's performance against data it hasn't seen before.

GridSearchCV is very useful for parameter tuning because it allows you to try different combinations of parameters for a model and select the best one.

For the decision tree, I tuned the following parameters: ({'dtree**min_samples_split': np.arange(2, 10), 'dtree**max_depth' : np.arange(1, 10), 'dtree**min_samples_leaf' : np.arange(1, 10), 'dtree**splitter':['best', 'random'], 'dtree__criterion':['gini', 'entropy']})

For the SVM: ({'SVM**kernel': ['linear', 'poly', 'rbf'], 'SVM**C' : [10, 100, 1000, 10000], 'SVM__gamma' : [0.1, 0.5, 1.0, 10]})

And for the KNN: parameters = ({'knn**n_neighbors': np.arange(3, 10), 'knn**weights' : ['uniform', 'distance'], 'knn__leaf_size' : [20, 30, 40, 50]})

In training models, the naïve Bayes model consistently gave the highest accuracy, precision and recall scores.

Model validation

Validation of models is important because it is a way to test their results and verify that the algorithm is working as desired. You split the model into training and test sets, and validate on the test set because validating on the same data you trained on would give the model a perfect performance on that dataset, but it wouldn't do a good job of classifying new data. It's as if you took a test made up of past homework questions that you answered correctly. You would do well on the test, but your knowledge wasn't really tested. At the same time, you don't want the model to be too general in that it doesn't react to new data when it sees it, so machine learners have to find the correct balance.

I used StratifiedShuffleSplit to validate the data because of the dataset's relatively small size and skewed labels: only a handful of people in the dataset are POIs.

Model evaluation

After running my code, only the naïve Bayes had precision and recalls scores of greater than 30%, meaning that the model does a relatively good job of finding all the POIs and not selecting other data. Precision is the measure of how many times someone classified as a POI is a true POI. Recall measures how many times the model guessed wrong by not classifying a true POI as a POI.

Sources from which I adapted code (from outside DAND):

Learning how to use GridSearchCV: http://bit.ly/2rqPzLL

Learning how to use StratifiedShuffle: http://bit.ly/2qHESUk

In [ ]:

```

```