Enron Project Report

Description of the Dataset

The dataset for this project is comprised of financial data from individuals who were required to file information as part of the Enron bankruptcy case, which began in late 2001 and concluded in November, 2004. In addition to the financial information, there is data regarding the number of email sent and received, both in total and in relation to individuals designated as “persons of interest”. In this context, a person of interest is someone who was indicted, or received immunity in return for testimony. The dataset as provided is a Python dictionary with 146 entries.

Structure of the data:

Each entry in the dictionary supplied is supposed to have as its key the name of the referenced individual, and the value is a dictionary containing data values for that individual. A sample of 10% of the names was printed and a Google search done on each. All appear to have been senior executives at Enron or its affiliates, or a member of the Board of Directors at Enron. Two entries were identified which were not for individuals, labeled “TOTAL”, and “THE TRAVEL AGENCY IN THE PARK”. These were removed from the dictionary before further analysis. The data values include 14 financial data items, 5 email items, and a string giving the individual’s email address at Enron.

Exploratory Analysis:

One striking feature of this dataset is the amount of missing information. Out of a potential 2736 (144 * 19) data points, there is information for 1435, or just over 52%. The missing information is not distributed evenly between “poi’s” and “non-poi’s”. There are values for about 78.6% of the items referencing the 18 poi’s, but only 50.2% of the items for the 126 non-poi’s in the database.

Medians were calculated for each of the numeric data items, faceted by poi/non-poi status. This was done with an initial load of the dictionary into a Pandas dataframe, where missing values were denoted by np.nan. It was repeated after substituting “0” for the missing values, as done by the imputation strategy in the supplifed featureFormat function.

These are the medians for non-poi’s, and poi’s, and the ratio of poi/non_poi (limited to columns with data in more than 50% of the rows):

	non_poi	poi	ratio
other	12961.0	149204.0	11.511766
exercised_stock_options	1030329.0	3914557.0	3.799327
shared_receipt_with_poi	594.0	1589.0	2.675084
from_this_person_to_poi	6.0	15.5	2.583333
restricted_stock	413586.5	985032.0	2.381683
from_poi_to_this_person	26.5	62.0	2.339623
total_stock_value	1030329.0	2206835.5	2.141875
to_messages	944.0	1875.0	1.986229
bonus	700000.0	1275000.0	1.821429
total_payments	1057548.0	1754027.5	1.658580
expenses	44601.0	50448.5	1.131107
salary	251654.0	278601.0	1.107080
from_messages	41.0	44.5	1.085366

Medians for poi/non-poi with 0 substituted for missing values:

```
                          non_poi        poi       ratio
other                       540.0   149204.0  276.303704
shared_receipt_with_poi      54.0     1195.0   22.129630
bonus                    100000.0  1225000.0   12.250000
to_messages                 220.5     1543.0    6.997732
expenses                   9795.5    50448.5    5.150171
restricted_stock         315068.0   927126.0    2.942622
total_stock_value        878950.5  2206835.5    2.510762
from_messages               15.5       37.5    2.419355
exercised_stock_options  596344.0  1288766.0    2.161112
total_payments           867948.0  1754027.5    2.020890
salary                   186410.5   276788.0    1.484831
```

The default behavior of the provided featureFormat function is to assign "0" to missing values. As shown here, this strategy will change, sometimes drastically, the difference between the medians for poi's and non-poi's. This might make a certain feature more effective in a classifier, but probably doesn't reflect the reality of the information. For example, salary information is missing for 32 of the 128 individuals in the dataset who are not directors. All of these people were senior executives, and certainly did not have a salary of "0".

Distributions of Values:

Histograms of financial features with data in > 50% of rows were generated. None of these appeared to have normal-shaped pattern. After applying a log-transform, many appear to be normal shape. Those features which appear to have enough data and appear normal by visual inspection after log transform are: bonus, exercised stock options, restricted stock, salary, total payments, and total stock value. This may, or may not, be important to consider when using certain classifiers.

Outlier Investigation:

As noted above, 2 rows with keys of "TOTAL" and "THE TRAVEL AGENCY IN THE PARK", clearly not representing individual persons, were removed from the dataset. After removing these rows, the top 2 values, and the associated names and poi status, were examined with the following code:

```
for feature in FINANCIAL_FEATURES:
    print df_feat[[feature, "poi"]].sort_values(feature,ascending=False).head(2)
    median_feature = df_feat[feature].median()
    mean_top_2 = df_feat[feature].sort_values(ascending=False).head(2).mean()
    print "Mean for top 2 is %3.1f times median for the feature." %(mean_top_2 / median_feature)
    print "\n"
```

```
With output such as:
                 loan_advances    poi
LAY KENNETH L      81525000.0   True
FREVERT MARK A      2000000.0  False
Mean for top 2 is 20.9 times median for the feature.
```

These values, although clearly "outliers" related to the majority of the data, probably represent the reality of the extraordinary greed and corruption taking place at Enron. Ken Lay, for example, was in fact alleged in an SEC complaint to have sold over $70,000,000 in Enron stock back to the company "to repay cash advances on an unsecured Enron line of credit." Thus, I've chosen not to remove outliers such as these in the analysis.

Feature Selection and Classifier Development:

First a few comments regarding evaluation and validation. Evaluation is the process of measuring the performance of a model, and is a key part of the process of developing and improving it. The goal, after all, is to develop the best possible model for use in the real world. Validation refers to testing a model's performance using novel data, meaning data different from that which was used to develop and train the model. This step is particularly important in machine learning. Automated processes (machine learning) can develop models which perform extremely well on the training data but do not generalize well, and perform poorly with novel information. This is called overfitting, and avoiding it is a key part of developing a useful prediction tool.

Evaluation requires using metrics which are meaningful in understanding how well a model (a classifier, in this case) performs the task for which it was developed. The problem here is to identify those individuals in the dataset who are known to be so-called persons of interest -"POI's" (and by extension – in the real world - find additional POI's, as yet unknown). Because the number of target cases (POI's) is a small minority of all cases, a simple metric such as accuracy can be misleading. Accuracy simply measures the proportion of correct predictions (positive or negative) by the model. The problem posed here is to find POI's in the dataset, so the important aspect of performance is how well the model does in making a positive identification of a POI. Two related, but different, aspects of positive performance are important: precision and recall. Precision refers to how often a positive identification is correct, that is how likely is it that an identified POI is a true POI? In other contexts, this is called positive predictive power. Recall is a measure of how likely a true positive in the dataset will be picked up by the model. What proportion of true POI's are detected? This measure is also known as sensitivity. These 2 measures can be combined in so-called "f scores". These are essentially (weighted) averages of precision and recall. The f1 score gives them equal weight, and the f2 score gives recall more importance. (One can also calculate f(1/2) giving precision more importance.) In all cases discussed below, in addition to accuracy, I report precision, recall, and f1 and f2 scores. When using the automated classifier evaluation tool GridSearchCV, a single score is required to rank parameter combinations, and the f1 score is used for this purpose.

I think it's important to note that metrics such as timing and memory usage are not included in this problem. These would, in fact, be very important factors in developing a model for deployment in any real-world situation.

To validate the model, an appropriate evaluation strategy is used to measure performance on a set of data different from that used to develop and train it. In an optimal situation, a model would be validated using data completely different from those used to develop it. In this case, there is only a very small dataset available for the entire process. A simple approach is to split the dataset in two, using one subset for the training step and the other for testing. However this may miss important aspects of the data which could be useful for model development, and may avoid testing the model on difficult or challenging cases. A better way to make optimal use of the data is to slice it in multiple different ways, and do the testing/training process on all the different permutations and average the results in some way. The StratifiedShuffleSplit algorithm in sklearn provides exactly this capability. This module will return multiple lists of randomly selected training and testing sets, and it ensures that the proportion of classes (POI's and non-POI's in this case) in the training and testing sets are the same as in the dataset as a whole.

Gaussian Naïve Bayes:

I chose to start with a simple classifier, Gaussian Naïve Bayes. I first used a pipeline of SelectKBest and GaussianNB, evaluated with a GridSearchCV where the k parameter of SelectKBest was varied from 1 to len(features_list):

```
skb = SelectKBest(score_func = f_classif)
gnb = GaussianNB()
pl = Pipeline([('skb', skb), ('gnb', gnb)])
### Use SSS cross-validator. To save time while testing, n_iter is 100, not 1000 as
### in the Udacity tester.py code.
```

```
cv = StratifiedShuffleSplit(labels, n_iter = 100, random_state = 42)
param_grid = {'skb__k' : [x for x in range(1, len(base_features_list))]}
gs = GridSearchCV(pl, param_grid, scoring = 'f1', cv = cv)
gs.fit(features, labels)
print "Best Estimator:", gs.best_estimator_
print "F1 score:", gs.best_score_
```

This returned a value of k=5 for the best estimator, with f1 = 0.3986. Feeding this combination into the tester which used the StratifiedShuffleSplit with n_iter = 1000, gave the following performance metrics:

```
Accuracy        Precision       Recall          F1              F2
0.849           0.422           0.359           0.388           0.370
```

This process yielded the optimum number of features, but not the names of the best features. It is possible to rank feature importance by fitting the SelectKBest selector on the entire dataset, but this doesn't seem quite right as the selection should be done on the training data only. I chose to set up a loop iterating through all of the cross-validation training sets returned by StratifiedShuffleSplit, fitting SelecKBest(k=5) on each training set, and counting how many times each feature was selected in this process. The top feature names and counts, ranked by count, were:

```
total_stock_value :    1000
exercised_stock_options :      998
bonus : 997
salary :        989
deferred_income :      506
long_term_incentive :  190
restricted_stock :     133
shared_receipt_with_poi :      109
total_payments :       58
expenses :     13
from_poi_to_this_person :      7
```

All other features had a count of zero.
Taking the top five features in this list, and using those features with a GaussianNB classifier in the testing process gave the following results:

```
Testing GaussianNB with features: ['total_stock_value', 'bonus',
'exercised_stock_options', 'salary', 'deferred_income']
Accuracy        Precision       Recall          F1              F2
0.855           0.489           0.381           0.428           0.398
```

This gives some improvement over the previous result.

Feature engineering and new features:

My next step was to explore whether applying a log-transform to some of the financial features would improve the results. **I thought this might be useful since one of the assumptions of the Naïve Bayes classifier is that features have a normal distribution, and several of the financial features definitely had a more normal appearing histogram following a log-transform.** However, trying 2 variations of substituting these transformed features did not improve on the performance of just the top 5 unmodified financial features.

After trying and abandoning the log-transform step, I explored adding the email information to the mix in some way. None of the email features made it into the "top 5" identified above. However, one of them, "shared_receipt_with_poi", did get included in about 10% of the cross-validation iterations. When I simply added this feature to the top-5 financial features, the performance scores were:

```
Testing GaussianNB with features: ['bonus', 'deferred_income', 'exercised_stock_options',
'salary', 'total_stock_value', 'shared_receipt_with_poi']
accuracy        precision       recall          f1              f2
 0.852          0.474           0.363           0.411           0.381
```

This didn't improve the classifier's performance. **As I understand this feature, it is the count messages, received by an individual, which were simultaneously sent to a POI. Since the total number of messages received by individuals varied dramatically ("to_messages" max = 15,149, min = 57), I decided to try a type of scaling on this feature. I created a new feature which is the ratio "shared_receipt_ratio" = "shared_receipt_with_poi" / "to_messages".** This is the proportion of any one individual's received messages which were also sent to a POI. Using this modified feature gave:

```
Testing GaussianNB with features: ['bonus', 'deferred_income', 'exercised_stock_options',
'salary', 'total_stock_value', 'shared_receipt_ratio']
accuracy        precision       recall          f1              f2
0.861           0.516           0.386           0.441           0.406
```

This is the best performance so far for this classifier. I tried one other email ratio feature, which was the sum of all POI-related email features divided by total emails ("from_messages" + "to_messages). Using this with the top 5 financial features gave exactly the same performance.

Finally, since the Naïve Bayes algorithm assumes feature independence, I decided to try using primary component analysis to see if that would improve performance. To prevent the PCA transformation from being dominated by features with higher variance, the features need to be scaled before the transformation. I tried 4 different combinations: financial features alone, all base features, best performing financial features, and best performing financial features + shared_email_ratio. The pipline used for testing was:

MinMaxScaler -> PCA -> GaussianNB

with the PCA n_components ranging from 1 to len(test_features_list). The best estimator from this process used only the top five financial features, with PCA n_components = 2. Running this through the full testing algorithm yielded:

```
accuracy        precision       recall          f1              f2
0.877           0.616           0.371           0.463           0.403
```

Interestingly, this very simple classifier gives the best performance yet for this algorithm.

In summary, the Naïve Bayes classifier achieved surprisingly good results. Using unmodified features, the best performance was achieved with the 5 top-rated financial features plus a new email ratio feature. However, using PCA on the top five financial features alone produced the best results with this algorithm.

KNeighborsClassifier:

The next classifier I chose to examine was the KNeighborsClassifier. Unlike the Naïve Bayes algorithm, this one has parameters which can be (or need to be) adjusted to optimize performance. Many machine learning algorithms will perform very differently on different types of data. To improve performance, one can adjust, or tune, different aspects of the algorithm. As an example: this classifier uses the average of the classes of the closest neighbors to the test point, mapped in multi-dimensional parameter space, to make its prediction. One of the adjustable parameters is the number of neighbors to include in the decision. If a test point is close to the decision surface, one might need to include several neighbors to correctly identify it. If the decision surface is irregular, however, too large a setting could easily include examples of the wrong class. I examined adjusting n_neighbors, the number of neighboring points included in the calculation, as well as the distance weighting used ('uniform' or 'distance'). Since this classifier uses distances in parameter space to calculate probabilities of the different classes, the features need to be scaled. I tried both the MinMaxScaler and RobustScaler, and the latter appeared to be the better choice. I set up a grid search with SelectKBest, RobustScaler, and KNeighbors. When I used all of the financial features plus an email ratio feature as described above, the returned best estimator was:

SelectKBest(k = 4), KNeighbors(n_neighbors = 3, weights = 'uniform')

This estimator only included the top 4 financial features, and gave performance of:

accuracy	precision	recall	f1	f2
0.870	0.521	0.346	0.416	0.371

When the features list was set to the top 4 financial features only, the performance was:

accuracy	precision	recall	f1	f2
0.872	0.645	0.379	0.477	0.413

Similar to the GaussianNB classifier, specifying the feature list, as opposed to using the SelectKBest selector on each iteration of the cross-validation loop, improves performance. This classifier also is quite simple, and is slightly better than the best GaussianNB I was able to find.

As a final step in developing this classifier, I added the custom imputation step of substituting the median salary for all non-directors in the dataset. Interestingly, running the grid search described above on this version of the dataset resulted in a best estimator of:

SelectKBest(k=3), KNeighbors(n_neighbors = 4, weights = 'distance')

Selecting 3 top-ranked features of ['total_stock_value','bonus', 'exercised_stock_options'], and using a KNeighbors classifier as specified results in performance of:

accuracy	precision	recall	f1	f2
0.875	0.638	0.438	0.519	0.467

This very simple classifier has, overall, the best performance of any!

Resources Used:

1. Sklearn documentation.

2. Stack exchange for a few pointers.

That's it.