Detecting POI's In Enron Data

Joby John

`Summarize for us the goal of this project and how machine learning is useful in trying to accomplish it. As part of your answer, give some background on the dataset and how it can be used to answer the project question. Were there any outliers in the data when you got it, and how did you handle those?`

We are given a dataset (a dictionary) of people with their names as 'keys' and a bunch of financial and email features. They are also identified as being a Person Of Interest (poi) or not. The goal is to train a supervised machine learning algorithm that can identify whether or not a person is a poi given the financial and email features alone and the chosen classifier is required to provide a *precision* and *recall* of over 0.3.

Why Machine Learning

There are many features (14 financial and 7 email) makes this an intractable problem to just plot figures to figure out clusters or identify trends that separate poi's from non-poi's. The fact that we have labels identifying the poi's from the non-poi's makes this a problem suited for supervised machine learning. The assumption we are making is that the financial and email features are sufficient to tell whether a person is a poi or not; therefore, by training a classifier and given an unlabeled set of features, we should be able to figure out whether the person is a poi or not.

Technically we could also use an unsupervised approach (like a k-means with k=2), but such an approach is not appropriate as it would be be tantamount to throwing away information in the case of this particular dataset where we know if a person is a poi or not.

Data Exploration:

The dictionary has 146 entries. Of these, one is the TOTAL which is addressed below. There are 18 poi's and the rest 127 are non-poi's. As mentioned before we have 14 financial and 7 email features.

- Many of the financial features have more than 50% missing values:
  - deferral_payments (73% values are missing);
  - 'loan_advances' (97% missing values)
  - 'restricted_stock_deferred' (85% missing values)
  - 'deferred_income' (66% missing values)
  - 'long_term_incentive' (54% missing values)
  - 'director_fees' (88% missing values)

  This is indicating that these features would have poor predictive power and should not be considered.

Outliers:

To check if there are outliers is an important step if one is performing regression or support vector machine (svm) based classification as the outliers can greatly affect the outcome of the classifier. For a decision tree, for example, this does not impact as much, as the algorithm functions on information-gain based splitting and will not be affected by the distance of one point from the mean of the data. The primary method used for detection of outliers was plotting the data in 2 and 3 dimensions (*plot_features.py*). It was

observed that the TOTAL entry in the dictionary is an outlier and therefore needs to be removed (using data_dict.pop('TOTAL',0))

Two other employees are also removed for very high '*exercised_stock_options*' and high '*total_payments*' respectively: LAY KENNETH L, SKILLING JEFFREY K. A few more outliers were identified but it was found that the performance is better with those entries left in the data set.

To spot an outlier, once the plot is made, the approximate value of the outlier is estimated and then the dictionary is checked iteratively to find all the keys that meet the condition (i.e. choose and delete keys with feature greater than relevant value.). See Fig.1 below.

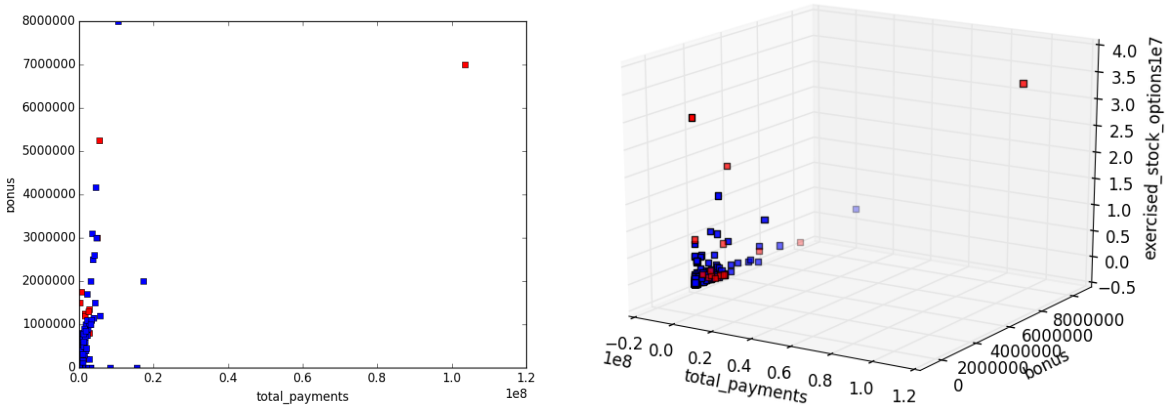

Figure 1: An example of an outlier: total_paymenst is too high

`What features did you end up using in your POI identifier, and what selection process did you use to pick them?`

The features that were used (in the best performing DecisionTree) along with feature_importances

1. 'total_stock_value', 0.0187664
2. 'expenses', 0.14192093
3. 'exercised_stock_options', 0.12212516
4. 'other', 0.15013123
5. 'restricted_stock', 0.07819769
6. 'shared_receipt_with_poi', 0.21464075
7. 'Topoi2frm', 0.16603503 (Newly introduced feature)
8. 'poi_mail_ratio' 0.1081828 (Newly introduced feature)

The features that were used (in the best performing SVM) with χ^2 score:

1. 'bonus', 1.83493206e+00
2. 'total_stock_value', 2.45619681e+00
3. 'expenses', 1.36882835e+00
4. 'exercised_stock_options', 2.64265719e+00
5. 'shared_receipt_with_poi', 1.76833959e+00
6. 'Topoi2frm', 3.91460933e+00
7. 'bon2sal', 2.31739139e+00

The χ^2 statistic and the associated p-values (less than 1.5e-1 for chosen features) are a measure the dependence of the feature and the outcome (i.e. poi/non-poi). If there is *not* enough evidence for the *independence*, then chances are that the feature selected could actually predict the outcome. It was possible to use this measure also because all features are positive quantities as required by the χ^2 score.

`In your feature selection step, if you used an algorithm like a decision tree, please also give the feature importances of the features that you use, and if you used an automated feature selection function like SelectKBest, please report the feature scores and reasons for your choice of parameter values.`

See above for the *feature_importance* and the *feature scores*.
Initially the following steps were used iteratively:

1. Using intuition and checking if variables like "other" and "from_messages" alone can affect the classifier (and confirming this intuition with following steps)
2. Seeing how much information content is there in *each* feature (if the number of NaN values was greater than 50 % of the cases then that feature was not used)
3. Plotting the features (see plot_features.py): Checking to make sure features are not extremely correlated (for example: having gross and net salary would not add much information). No such correlations were spotted in the few plots that were attempted.

However, the **final approach** for feature selection relied on a quantitative assessment of feature contribution. Using a criterion like **selectKBest** for SVM (based on χ^2 score) or **feature_importance** for DecisionTree to choose different number of features and for each case assessing the performance as determined by *tester.py*. The final choice was based on $\mathbf{F}_1$-score , i.e, the number of features that maximized $\mathbf{F}_1$-score as determined by *tester.py*. In other words, the number of features ($\mathbf{k}$) was varied till the $\mathbf{F}_1$-score was maximized.

`Did you have to do any scaling? Why or why not?`

Yes, features were scaled. SVM's are sensitive to the scale of the data (i.e. scale of the different features). Especially in this dataset since we are combining two different classes of features (financial and email) it is necessary to scale the features so that they can be compared in a consistent manner. The email features are in a few thousands and the financial features can run in to the millions. Also, the newly introduced features are all normalized ratios and by definition lie between 0 and 1. It then becomes necessary to use a min-max scaler in order to restrict all features to this range between 0 and 1. Other approaches like DecisionTree and Adaboosted-DecisionTree did not need scaled features. However the scaled features were used in testing these algorithms also. It is important to note that the *fit* of the *scaling* is done on training data and in the *tester.py* the same scaler is used. See line 26 in *tester.py*:

```
data = scaler.transform(featureFormat(dataset, feature\_list, sort\_keys= True))
```

`As part of the assignment, you should attempt to engineer your own feature that does not come ready-made in the dataset -- explain what feature you tried to make, and the rationale behind it. (You do not necessarily have to use it in the final analysis, only engineer and test it.)`

Some of the features that were introduced/engineered:

1. **'bon2sal'** The ratio of bonus to salary. If someone receives a high bonus relative to his salary, it could indicate some hanky-panky.
2. **'stock2sal'** The ratio of stocks to salary. Same logic as above, inordinatesly high stock to salary ratio could be suspicious.

3. **'Topoi2frm'** Ratio of mails from a person to the poi to the total number of mails sent. If the person is sending too many emails to the poi relative to the total emails he sends out, it could imply that they are also a poi. It is necessary to normalize the emails sent out so that we can compare two subjects with completely different email volumes.
4. **'frmpoi2To'** Ratio of mail from a poi to a person to the total number of mails received. Same reason as the above feature. More emails coming in from a poi can indicate the recepient is also a poi.
5. **'poi_mail_ratio'** Ratio of total number of mail to and from a poi to the total number of mails to and from a person. This feature is also designed using the same reason as above. But instead of using the To and From emails separately, here the ratio of all emails to and from a poi to the total emails to and from a person is considered. This is the feature that finally helped meet the performance criterion (0.3 precision and recall).
6. **'is_enron_emp'**: weather or not a person is an enron employee It is observed from the data that all poi's have an *@enron.com* email address. So it is safe to say that if some one is not an enron employee, he is not a poi. The hope was that adding this feature will bring down the number of False Positives and hence bring up the precision.

`What algorithm did you end up using? What other one(s) did you try? How did model performance differ between algorithms?`

Both the DecisionTree and SVM were able to meet the required critertion of precision and recall>0.3. However the Decision Tree gave the best precision and recall after the right number of features were provided as input. See **Model Performance** below.

`What does it mean to tune the parameters of an algorithm, and what can happen if you don't do this well? How did you tune the parameters of your particular algorithm? (Some algorithms do not have parameters that you need to tune -- if this is the case for the one you picked, identify and briefly explain how you would have done it for the model that was not your final choice or a different model that does utilize parameter tuning, e.g. a decision tree classifier).`

Tuning Parameters
Every algorithm has a set of parameters. These are typically the number of features, regularization constants, soft margin parameters, kernel free parameters, max-depth of a tree, no: of leaves, etc. Typically we identify a set of parameters that we would use to fit the training data. Once the algorithm with a given parameter is fit to the training data, the performance (accuracy, precision, recall, $\mathbf{F}_1$-score etc) of the algorithm on *test* data is compared with the performance with other parameters to determine which choice gives the best performance on the untested data. If we do not choose the parameter that maximizes performance then, theoretically there is a better choice that can improve performance. The main of purpose of tuning the parameter (of which *validation*, addressed below, is an integral part) is to ensure that the performance is maximized and we get an algorithm that can ensure similar performance on new data. Once the model-class has been fixed, a sub-optimal choice of parameters will lead to either high bias or high variance. One of the purposes of parameter tuning is to find the sweet-spot in the bias-variance trade-off.

The parameters in the Support Vector Classifier are C and γ (for *rbf* kernel). They are tuned using the **GridSearchCV** function which essentially trains a SVM on every possible outcome of the paramter combinations and returns the combination that maximizes some performance metric. In our case, since we want to maximize both precision and recall, **'$\mathbf{F}_1$'** score (the harmonic mean of *precision* and *recall*) is chosen as the metric to be maximized in GridsearchCV. We find that C=5e4 and gamma = 0.1 are the best parameters for this dataset with this performance metric.

Model Performance: According to *tester.py*
Final choice in each case is highlighted with bold F1-score.

Decision Tree Performance
# of Features 8: Accuracy: 0.85680 Precision: 0.46060 Recall: 0.43250 $\mathbf{F}_1$**: 0.44611** F2: 0.43784
#of Features 11: Accuracy: 0.83553 Precision: 0.37289 Recall: 0.34250 $\mathbf{F}_1$: 0.35705 F2: 0.34818
#of Features All: Accuracy: 0.81067 Precision: 0.28283 Recall: 0.27350 $\mathbf{F}_1$: 0.27809 F2: 0.27532

SVM Performance

#of Features 7: Accuracy: 0.83014 Precision: 0.39418 Recall: 0.35200 $\mathbf{F}_1$**: 0.37190** F2: 0.35970
#of Features 9: Accuracy: 0.82893 Precision: 0.38746 Recall: 0.34000 $\mathbf{F}_1$: 0.36218 F2: 0.34854
#of Features 10: Accuracy: 0.82093 Precision: 0.34323 Recall: 0.27750 $\mathbf{F}_1$: 0.30688 F2: 0.28855
#of Features 8: Accuracy: 0.82836 Precision: 0.38209 Recall: 0.32650 $\mathbf{F}_1$: 0.35212 F2: 0.33629

`What is validation, and what's a classic mistake you can make if you do it wrong? How did you validate your analysis?`

Crossvalidation or just validation is a method of using separate "test" data to test the validity of the model trained on the training data. It provides a reliable estimate (a lower bound) of the performance of the trained classifier out in the "wild" on completely new data.

If we don't validate our model, we run the risk of overfitting. i.e. we can tune the parameters such that it fits the training data very well but the model will not generalize on untested data and will return poorer performance compared to the training data.

In this project, in addition to the testing using a StratifiedShuffleSplit in *tester.py*, validation is also done by splitting the data in to a test set and training set and also using a stratifiedKfold validation. However, for the decision tree, the precision and recall reported by the stratifiedKFold was found to be unreliable; this might be because of the small data size (which necessitates StratifiedShuffleSplit rather than). However in each of the cases tested, *tester.py* was able to verify that both precision and recall met the required conditions.

Used both *precision* and *recall* to test performance. **Precision** is the ratio of True Positive(TP) to True Positive + False Positive (TP+FP). $Pr = \frac{TP}{TP+FP}$; i.e. it tells us how often the algorithm labels a non-poi as a poi (i.e. FP's). The closer the precision is to 1, the fewer false positives are being detected by the algorithm.

Recall is the ratio of TP to (True Positive (TP)+False Negative(FN)); $Re = \frac{TP}{TP+FN}$, i.e. it tells us how often the classifier misses to catch a true poi (i.e. FN). The closer the recall is to 1, the fewer the False Negatives and rarely will the classifier mislabel a poi as a non-poi.

With a precision of, say, 0.3, the number of True Positive (i.e. detection of a poi who is actually a poi) was only 30% of all cases detected as poi's.

The algorithm erroneously classifies a lot of people who are actually poi's as not poi's. (i.e there are quite a few False Negatives). With a *recall* (or sensitivity) of 0.3, the number of True Positives detected was about 30 % of all the cases that are actually poi's.

In this problem, both precision and recall are required to be greater than 0.3. There were some cases of SVM parameters that lead to a high precision (0.57) and high accuracy (87%) but in these cases recall suffered and fell to around 0.1. There is a trade-off between precision and recall and we need to find parameters that maximize *both* metrics. For this we purpose and the fact that we can pass only one score to GridSearchCV, we used the $\mathbf{F}_1$ score, which is the *harmonic mean* of precision and recall.

It is also noted that here we cannot use accuracy as a good measure of our performance because, the number of poi's to non-poi's is 18 to 127. By calling every one a non-poi we would have an error rate of less than 15%; clearly this algorithm does not accomplish much and hence accuracy is generally not a good idea when the ratio of positive cases to negative cases (or vice versa) is imabalanced.