Udacity Machine Learning Project

Patrick Coakley

1. The goal of this project is to create and train a machine learning model that is able to accurately predict persons of interest in the Enron dataset. Enron was a large energy company based out of Houston, Texas, and during the 90's experience a huge growth. It became one of the largest companies of its kind in the world, but in 2001 it was revealed that there was an incredible amount of corruption in power. Eventually accusations of insider trading and corporate fraud lead to Enron's bankruptcy by 2004, leaving rippling effects on accounting firms and business regulations for years to come.

The dataset itself is a large dump of information containing the some of the personal and financial information of employees at Enron. By using this data and exploring some of the higher profile POIs, such as Kenneth Lay, we can extract the necessary information to create our model.

Prior to any data cleaning, there are 146 employees in the dataset that represent the total number of data points. There are 18 POIs in the data set, making it 18 out of 146 at the start, making POIs only about 12% (18/146) of the total number. This makes the data imbalanced, meaning that the number of positives in the data set are significantly less than negative. Each employee has a number of different features, ostensibly divided up into two categories: financial and email. Of the 21 features, email addresses were dropped because they are not relevant here.

When exploring the data, the two big outliers I found in the dataset were the 'TOTAL' and 'THE TRAVEL AGENCY IN THE PARK' entries. The first represents the total amount of information, and the second, as described in the report used in the learning materials, represents "payments that were made by Enron employees on account of business-related travel to The Travel Agency in the Park (later Alliance Worldwide), which was co- owned by the sister of Enron's former Chairman. Payments made by the Debtor to reimburse employees for these expenses have not been included." Since neither of these represent people, we need to remove them.

There were also quite a bit of ‘NaN’s, or null entries, in the dataset. Dropping them would not work as the dataset is already too small, so making them zeroes is a compromise between getting rid of them and preserving the data integrity. Dealing with them is necessary regardless of the route taken because ‘NaN’ is a string, and every other piece of data we will be working with will be a float or integer outside of the boolean ‘poi’.

2. Originally I wanted to only use the financial features. This is due to the fact that I felt fiscal numbers would be easier to compare than the email features. However, after I spent some time looking at the report included with our dataset and exploring the data, there didn't seem to be much of a relationship that I could find that definitively would be useful for manual feature creation. I then decided to include the email features as they were used in the earlier learning materials and were easier to implement.

I decided to create two simpler features based on the ratio of emails to and from a POI over their respective totals. These features make sense because it lets you measure the interaction levels of possible POIs, and it

would make sense that most POIs would interact with one another. The feature ‘from_person_to_poi_ratio' takes all of the messages sent from this person to a POI and divides it over the total amount of sent messages. Likewise, ‘from_poi_to_this_person' does the opposite, where it takes all messages received from a POI to this person and divides it by the total amount of received messages.

In my pipeline I decided to include SelectKBest and PCA as options while I tuned my parameters. I found that SelectKBest often chose more features when not using Principal Component Analysis. The only parameter I used in my pipeline for SelectKBest was 'k', which represents the top number of features to select. I started out with a lower ranger of numbers and found that when trying to use higher ranges of numbers, my recall score was going down quite a bit.

One thing to note when using PCA in conjunction with SelectKBest is that the minimum of the latter must match the maximum of the former. That is, If you want to use a wider range of component selection with PCA, you need to make sure that the maximum number of components being kept is at least equivalent to the minimum of the SelectKBest feature selection parameter. When using PCA, I found that it generally gravitated towards 3 or less components.

I decided to add feature scaling only after I was able to have PCA and SelectKBest cooperate. I decided to use MinMaxScaler because it is simple to implement and because of my experience using it in the mini-projects. When I included feature scaling to my pipeline, I noticed that I was getting much higher recall scores "for free" when using higher values on SelectKBest and PCA. However, when I lowered those values I was getting worse recall scores.

After disabling PCA and only using SelectKBest with MinMaxScaler, I was achieving better scores. It's possible that because the data is limited that PCA was favoring heavily scaled data and was biased towards certain aspects. In any case, I decided to not use PCA after a great deal of testing because of these results.

Here are the parameters I found after tuning and getting the best results for each classifier:

- GaussianNB: MinMaxScaler on, SelectKBest 'k=5', no PCA
- DecisionTreeClassifier: MinMaxScaler on, SelectKBest 'k=17', no PCA
- SVC: MinMaxScaler on, SelectKBest 'k=16', no PCA

Here are the features used by SelectKBest when I finalized my results:

- ['from_poi_to_this_person', 'from_this_person_to_poi', 'loan_advances', 'shared_receipt_with_poi', 'from_poi_to_person_ratio'] (the last one being of my created features)
- Interesting to note that financial features were not really as relevant as I was thinking at the beginning of the project

When I disabled my newly created features and ran my final classifier, SelectKBest picked the following features:

- ['from_poi_to_this_person', 'from_this_person_to_poi', 'loan_advances', 'shared_receipt_with_poi', ‘to_messages’]

When I didn’t include my extra features, my precision and recall were 0.41964 and 0.35900, respectively. However, when I did include the new features, I got a precision of 0.41686 and a recall of 0.35850. I think it’s interesting that SelectKBest chooses one of my features but my scores go down. In the end I decided to non use them, but I still found this curious nonetheless.

3. For my three algorithms, I decided to use Naive-Bayes by way of GaussianNB, Support Vector Machines by way of SVC, and Decision Trees by way of DecisionTreeClassifier. Originally I was only going to do SVM and Decision Trees, but I included Naive-Bayes because it made for much easier testing due to how simple it is to implement, and it allowed me to have a baseline to compare the other two since they both have a wide array of different parameters.

In the end I decided pick GaussianNB because it simply rendered the best scores with the least amount of performance time. This also includes the incredible amount of time I spent trying to tweak the parameters for the other two classifiers. I did not expect that GaussianNB would give me better results than SVC or DecisionTreeClassifier, so it was kind of surprising that after all of the time taken to try and figure out the optimal parameters that the one with the least amount of setup worked best. Of DecisionTreeClassifier and SVC, the former performed better when tuned, and SVC was easily the slowest of the three, especially during the tuning process.

GridSearchCV allows you to input multiple sets of parameters and it will automatically run iterative searches to find the best values for you instead of having to type them in each time you want to try a different approach. This in conjunction with Pipeline saves a great deal of time and allows for a leak-free environment to add and change different parts of your classifier model, such as feature scaling or feature selection.

4. To tune to parameters of an algorithm simply means to experiment and find the best possible settings in order to get the desired results. In this case, it meant changing each classifier's parameters to include a range of different values and finding the best ones based on the outcome of the scores. Part of the reason I picked GauassianNB is because it has no parameters, making it very good for testing different settings outside of the classifiers, such as PCA or SelectKBest.

In tuning the DecisionTreeClassifier settings, I found that the less parameters I set manually, the better the recall score I was getting in my GridSearchCV results. I found this interesting as I had spent a great deal of time trying to find the best range of values for each of the other parameters and by simply ignoring them I got much better results. After taking this approach, I was able to achieve closest I had gotten with this classifier to a precision score and recall score over 0.30 by simply tweaking the SelectKBest values, as it favored much higher ones than GaussianNB.

For SVC, I only used 'kernel' and 'C'. 'Kernel' had two options, 'rbf', which is the default, and 'linear'. Of the two classifiers that do have parameters, I found SVC to be the more difficult. Higher 'C' values resulted in better recall scores, but it also performed slower and slower as the number went up. When printing the best estimator results from GridSearchCV, I noticed that it never picked anything other than the 'linear' option for 'kernel'. If forced to use 'rbf', my precision score went down quite a bit. Much like a higher 'C' value, using 'linear' slowed my performance down quite a bit but improved the results I wanted.

5. Validation is necessary in order to make sure your algorithm is being tested against data that only represents a small part of what you will eventually use it for on the real data. By using testing and training data, we are able to get a good estimate of the performance of the algorithm without having to use the entire dataset, and it allows us to check for overfitting, which is a common problem with some classifiers, such as Decision Trees. This means that you have to spend a great deal of time making sure the right amount of parameters are used, otherwise the machine learning model might overreact when the new data is introduced.

For this project, since the data in this project is limited I implemented StratifiedShuffleSplit, which takes train and test indices to split data in train and test sets, making sure that the validation set preserves the same percentage of samples for each class; StratifiedShuffleSplit is also used by the tester.py validation.

6. Recall and precision scores were used to evaluate the algorithms, as well as f1. While accuracy is often used to test the classifier, it doesn't work well here due to the unbalanced data. Essentially, because we have a low number of POIs, even if the classifier didn’t detect any of them, it would still have a high accuracy. Instead, precision and recall make more sense to use. Precision is the measure of result relevancy and recall is the measure of the relevant results being returned.

That is, precision is the number of relevant items being returned divided by the total number of relevant and irrelevant items (in this case, actual POIs identified by the classifier divided by both actual and misclassified POIs), and recall is the number of relevant items being

returned divided by the total number of relevant records in the dataset (actual POIs identified by the classifier divided by the actual POIs in the dataset).

When you have a low recall and high precision, your classifier will be overly conservative and might miss some POIs, but will more accurately predict a POI accurately. On the other hand, a high recall and low precision will make your classifier more liberally try and predict a POI, leading to more POIs being correctly identified overall, but also likely leading to more false positives.

The f1 score is simply the average of those two scores normalized so that you can compare it to other algorithms. When you have a high f1 score, it means your false positives and false negatives are low, making it much more likely your classifier will predict a POI accurately. F1 score was what was used in GridSearchCV to pick the best parameters.

After running my final test.py run with the final parameters tuned for the different algorithms, I received scores of the following:

Algorithm	Accuracy	Precision	Recall	F1	Time (in sec)
GaussianNB	0.84833	0.41964	0.35900	0.38696	1.616
DecisionTreeClassifier	0.80947	0.26583	0.24350	0.25418	1.72
SVC	0.86407	0.46108	0.11550	0.18473	2.988

With a higher precision score than recall score, this current model would have a higher likelihood of each prediction being a POI, but it would also miss quite a few POIs due to its conservative nature. While not great

results, due to the limited scope of the project and the dataset, I am satisfied with these numbers, especially since the final model did not require as much effort in tweaking parameters as the other classifiers I tested did.