UDACITY MACHINE LEARNING PROJECT:
IDENTIFY FRAUD FROM ENRON EMAIL

JOSEPH JASPER
11/21/2017

SECTION 1.

The purpose of this project is to leverage the principals taught to us through Udacity's Intro to Machine Learning course to create a machine learning algorithm to classify if an individual related to a company should be investigated as a person of interest for fraud. We have the wealth of email and financial data left to us from the Enron investigation in the early 2000's to create an algorithm that could hopefully be leveraged for other companies in the future.

In exploring the Enron data provided, there were some records that could be classified as outliers. There was a record for the total line of the dataset, which definitely needed to be removed; but there were a few individuals in the dataset that exceeded the other's by far. Unfortunately, many of these were records that belonged to high ranking individuals in the company including the CEO Ken Lay. I decided that the limited data and the fact that many of these potential outliers necessitated that I keep the records in the data set. I would hope that creating modified features, using principal component analysis or scaling many of the features would help to offset the fact that these outliers were in the dataset. There are only 145 valid records and 18 poi's identified.

SECTION 2.

In my initial attempts, I decided to see how far I could go on the features that were reviewed in the Udacity course leading to this project as they seemed to yield interesting results. At first, I used bonus, salary and two modified features (to poi rate and from poi rate) based on some of the email features that came included with the data set. I initially had botched the creation of these features at first by swapping the denominators for their creation, which lead to a deceivingly high precision score (.72) but I was not able to achieve a sufficient recall score.

I went back to reevaluate the 20 original features (not counting poi). Through a series of trials and errors, I eventually decided to leverage as many features as possible and use a combination of Select K Best and Grid Search CV to find the right features to (hopefully) get the best results. I removed the loan advances feature, as there were only 3 records that had a value for this feature and they were all classified as poi's; and I removed the email features that went into the creation of the from_poi_rate and to_poi_rate. Select K Best filtered to 5 remaining features: bonus, exercised_stock_options, salary, tp_poi_rate and total_stock_value. From my human intuition, this seemed reasonable as they struck me as features that could show irregularities in financial information and the users that were most potential to gain or lose based on how the company fairs as a whole. I was a happy to see that one of my modified features made the cut.

SECTION 3.

The algorithm I finally settled on was a pipeline leveraging select k best (to get the 5 best features) and Gaussian Naïve Bayes.

I had originally tried to leverage the k neighbors classifier with the original features I mentioned above and found that I was able to get some decent precision values at a pretty high cost to recall. I then found this interesting graphic that I used as a general guideline to inform a grid search where I gave k neighbors another try as well as SVC combined with PCA and scaling. K neighbors yielded a quick fitting and prediction time though it regularly underperformed on the recall score. SVC was just outside of the .3 threshold for the primary metrics of this project and could likely have been tuned to meet the minimum metric scores needed for this project, but the run time was considerably worse for the modest gains to these metrics and it felt like this comment on the forum may have been a hint to look elsewhere.

SECTION 4.

Each algorithm comes with parameters that can be adjusted so that it can provide better prediction based on the specific aspects of the dataset it is being run against. Most algorithms aren't smart enough to look at the data and decide what is needed. These parameters can be so finely tuned for a specific dataset that you end up with over fitting which can hurt the algorithm's performance when predicting. However, if you are not careful with the parameters, the algorithm can be so general that it basically ignores the training data which also impact's the algorithm's usefulness.

At first, I tried to leverage a grid search algorithm to decide the best parameters for me. However, the version of grid search used on this project can't optimize for more than one score. Every time I optimized for precision, I would lose ground in recall and vice versa. I then tried to optimize for f1, which seemed to yield mediocre results for both. Once I moved on from grid search, I reevaluated the different algorithms I had used so far. With k neighbors, I mostly tuned for k and the leaf size. With SVC, I was really interested in the differences between the kernels and the C value. Once, I finally looked at Gaussian Naïve Bayes, there weren't any parameters to tune except the number of features to receive from select k best which I tried by hand ranging from 3 through 7, I found that 5 seemed to maximize for precision while keeping recall above .3.

SECTION 5.

Validation in supervised machine learning refers to reserving a subset of the available data to test your algorithm once it has been fit to a training dataset. One obvious benefit is that it can make it easier to identify if your algorithm is overfitting for the training set. However, if you do not leverage the random_state parameter and your data is in any way sorted (especially by your classifier), you can drastically skew your results.

I initially used train_test_split, but felt I could benefit from learning additional tools. I took the recommendation from the commented-out code and leveraged StratifiedShuffleSplit. At first, I was using it incorrectly, by only pulling the train and test data from the final cycle. After looking at tester.py, I corrected my mistake and in the end received the full benefit. I move back and forth between tester.py and the splitter on my code to evaluate. This provides 1000 separate tests of the same data to get the resulting scores which is truly beneficial with how small the data set is.

SECTION 6.

My average precision score is .41. This means that 41% percent of the time that my algorithm classified an individual as a poi, it was correct. My average recall score is .32, which means that my algorithm properly classified about 32% of the poi's when tested. Both of these scores are higher on tester.py, but that is likely because it is not holding as much data back for testing.

I may be willing to sacrifice some recall for a higher precision score (if .3 had not been the lower bound for this project). This would mean that the algorithm would identify fewer of the poi's but there would be more certainty in the poi's it identified. In my mind, a project like this would be to control the scope of who is initially investigated for the sake of managing investigative resources. It is likely that investigation of one poi will identify others through investigating these leads or as the result of plea bargains, so the upfront sacrifice could be worth the long-term gains.