James Dellinger

Udacity Intro to Machine Learning Course

Final Project

February 22, 2018

Enron Submission Free­Response Questions

*1. 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? [relevant rubric items: “data exploration”, “outlier investigation”]*

Answer:

The goal of this project is to identify Enron employees who may have committed fraud. While certain individuals had already been marked by prosecutors as "persons of interest" (POIs), there is no guarantee that every individual who is actually guilty of fraud has already been flagged as a suspect.

Machine learning gives us tools to scrutinize the Enron financial and email dataset to discover patterns that are common to Enron fraud suspects. The hope is that we will come up with a set of features that, when used with one of our machine learning algorithms, will be able to predict with great precision and recall whether or not an Enron employee should be flagged as a POI.

If we can achieve this for known POIs, we could then conceivably use this algorithm to identify other Enron employees who, although not originally flagged as persons of interest, perhaps should have been.

The dataset I will use contains 146 entries. Each entry contains by default 21 features that contain either financial or email information (email info typically is denominated in number of emails). Of the 146 entries, 18 have already been flagged as POIs, leaving 128 entries that weren't already targeted as POIs.

There are outliers in the financial data. Most notably, there's an entry called "TOTAL," which is just sum of values for each financial feature for each of the other 145 entries in the dataset. There was also an entry titled "THE TRAVEL AGENCY IN THE PARK." This entry does not represent an Enron employee, and it received $362,096 worth of payments under the "other" category. Otherwise, values for all other features for this entry were 0. I removed both of these outliers from the dataset dictionary before proceeding further. I did find other outliers, such as the financial data for POIs like Ken Lay and Jeffrey Skilling. But because these individuals are POIs, these data points may prove useful for my algorithm, so I did not remove them.

At first glance, some financial features looked promising, such as “loan\_advances” -- I noticed, for example, that Ken Lay received an $81,525,000 loan advance. However, upon closer inspection, I saw that out of my 144 remaining entries in the dataset, there were only two other employees who had received a loan advance. This indicated to me that this feature would not be helpful in predicting a POI. Similarly, the “director\_fees” and “restricted\_stock\_deferred” features were empty for most entries in the dataset, and the individuals who did have values for these features were not on the POI names list. I thus concluded that none of these three features would be good make use of in my algorithm. Furthermore, I decided that the three email-related features, “to\_messages,” “from\_messages,” and “email\_address” would not be helpful in predicting POIs.

Therefore, in addition to the “poi” feature, I decided to start by using 14 of the 20 remaining original features, along with one of the features that I would engineer in task 3. I also planned to use univariate feature selection (SelectKBest) in task 3 to whittle these 15 features down to the most effective ones.

*2. What features did you end up using in your POI identifier, and what selection process did you use to pick them? Did you have to do any scaling? Why or why not? As part of the assignment, you should attempt to engineer your own feature that does not come ready­made 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.) 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. [relevant rubric items: “create new features”, “intelligently select features”, “properly scale features”]*

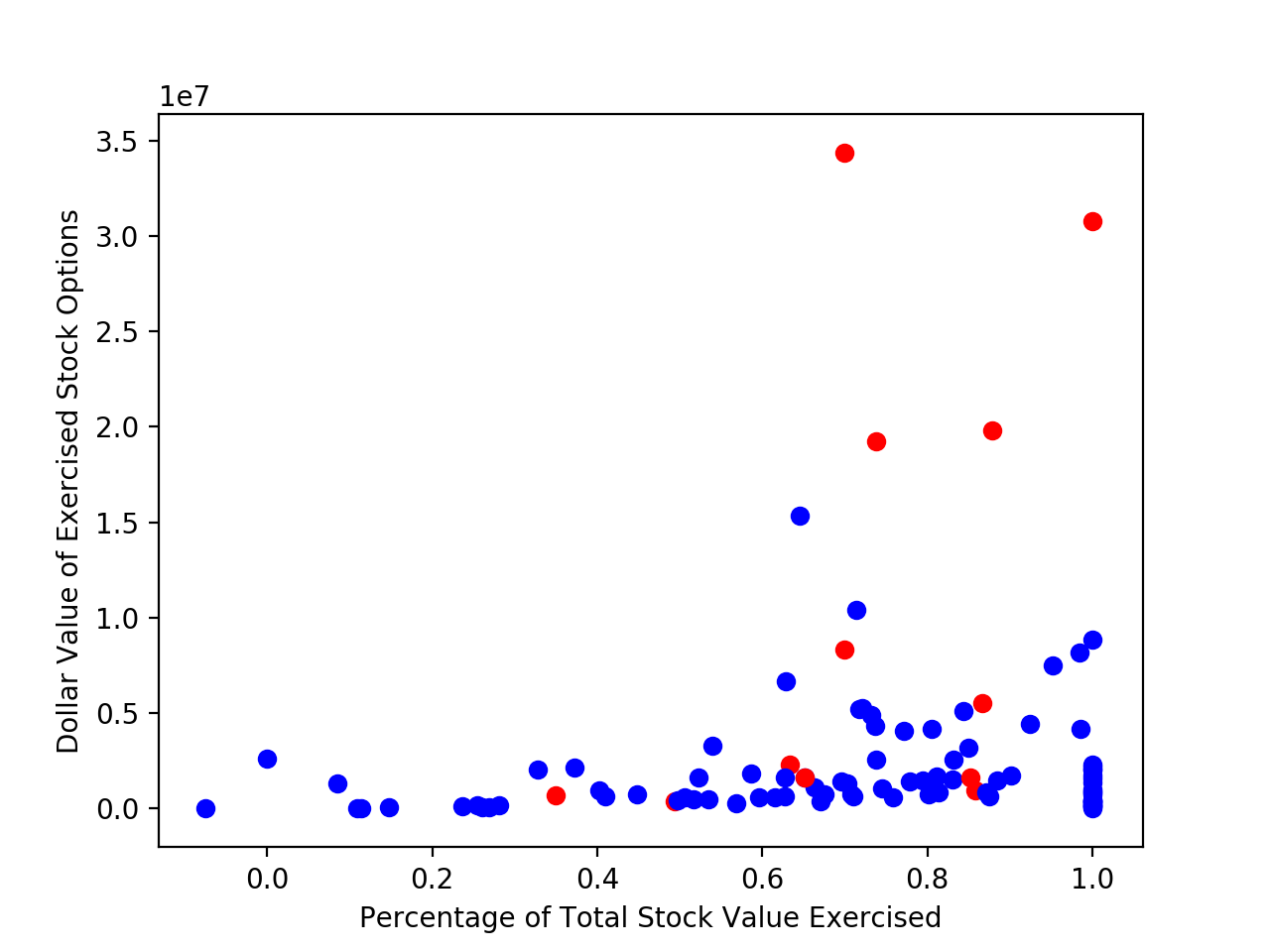
Answer:

After removing outliers, my dataset now contains only 144 data points. Because of concerns about overfitting (using a small amount of data points to train a large number of features), I will be sure not to select too large a number of features.

I engineered two new features. First, I explored whether it could be helpful to explore the percentage of total stock value that was exercised ("percentage\_total\_stock\_exercised"). I posited that a likely POI would, on the whole, have exercised a greater percentage of his/her total stock value, and would correspondingly have had a smaller percentage of this total value

remaining as restricted stock than, say, a non-POI would have had.

Here is the scatterplot of this new feature (on the X-axis), versus the dollar amount of stock options exercised (on the Y-axis). POIs are represented by red dots:

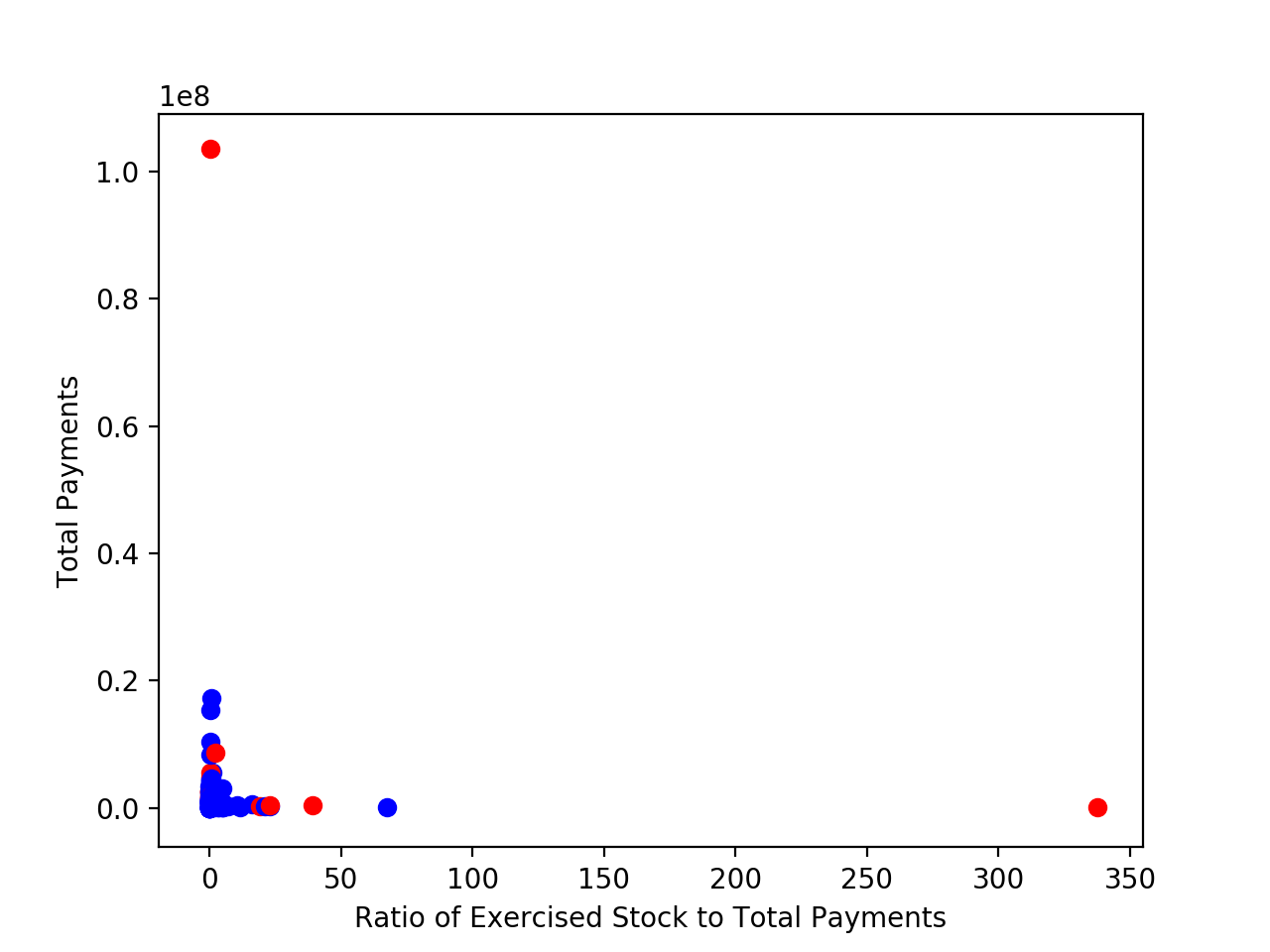


I noticed that there was a cluster of four outlier POIs who both exercised a large absolute dollar value of stock options, and also exercised the biggest majority of their total stock value. This was helpful as it indicates that there may a trend of POIs having both larger amounts of total stock and exercising a correspondingly greater proportion of that stock. However, at the same time I was concerned by the POIs interspersed amongst all the non-POI blue dots along the bottom of the scatterplot. The trend of having more stock and exercising a greater proportion of it seems to be more pronounced in POIs. However, as the graph above indicates, non-POIs also tend exercise a slightly larger proportion of their stock as they get more of it, but to a perhaps to lesser extent than the handful of POIs depicted above.

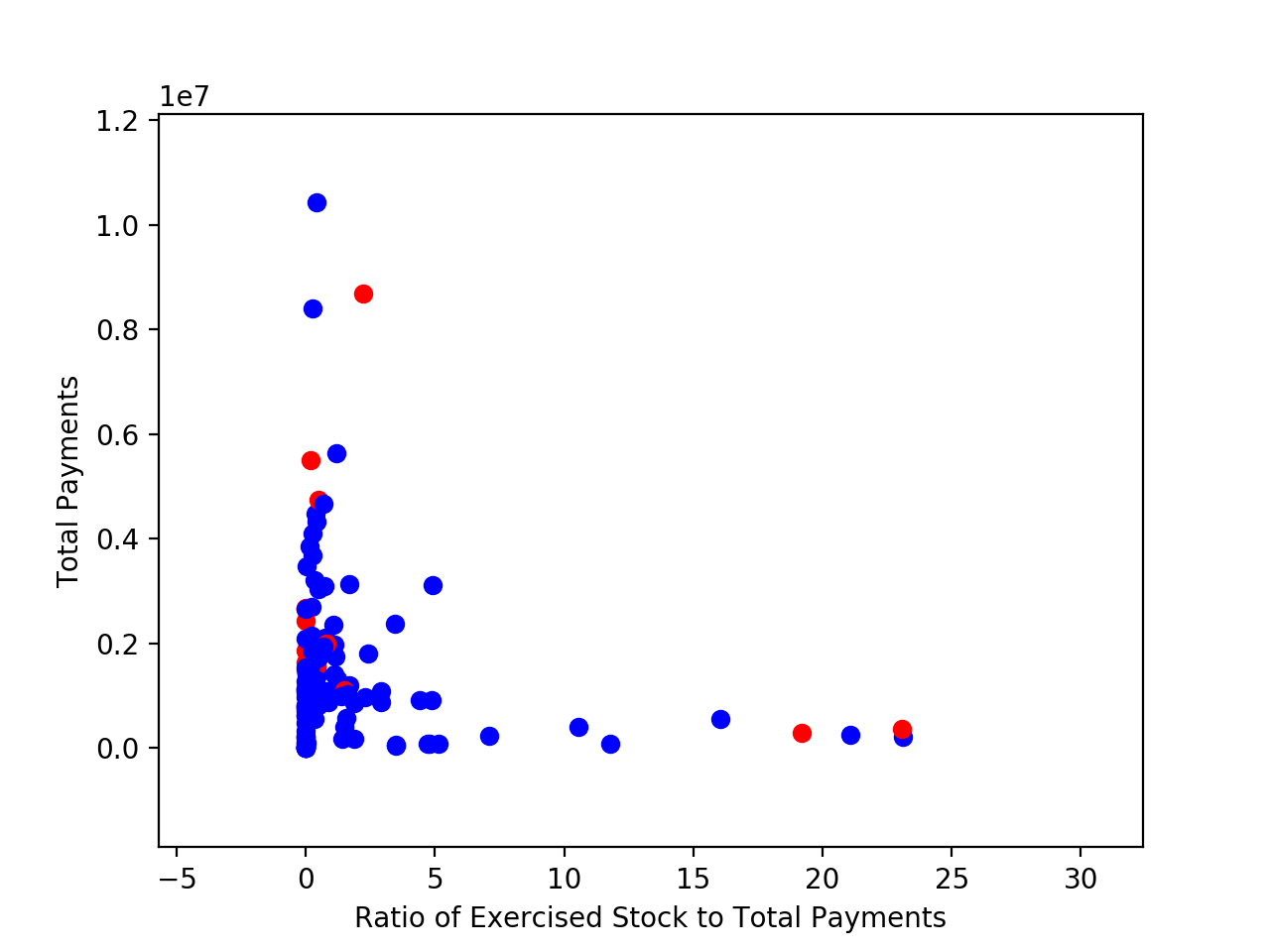
I will find out just how valuable this new feature relative to the other finance and email features already present when I run SelectKBest. I wouldn’t be surprised if the simple dollar value of exercised stock options feature (exercised\_stock\_options) ends up being a better predictor of POIs according to SelectKBest.

I decided to make an attempt to engineer a second new feature: the ratio of exercised stock options to total payments ("ratio\_of\_exercised\_stock\_to\_payments"). Here I hypothesized that a POI or a possible POI would be more likely to have exercised a dollar value of stock options that, when compared to a run-of-the-mill non-POI, would be an exponentially larger multiple of the individual’s total payments. In other words, I have a hunch that POI’s stock amounts increase exponentially relative to an increase in their salaries, while non-POI’s stock would not be a similarly large multiple of their total payments.

Unfortunately, things looked even worse for this second attempt. Here is the scatterplot of this "ratio\_of\_exercised\_stock\_to\_payments" feature (on the X-axis), versus the dollar amount of total payments (on the Y-axis). POIs are represented by red dots:



Things also unfortunately aren’t any better when we ignore the extreme POI outliers on each axis:



There’s a lot of red POIs mixed in with the blue dots. And to make matters worse, POIs extend to the extremes of each axis.

To sum it up, while I’m still open to the possibility of using the first feature I engineered in my algorithm, this second feature is a definite no-go. Thus, I added only the first of my engineered features to the other features in the features\_list I had created in task one. Here’s what the full list looks like:

['poi','deferral\_payments','expenses','deferred\_income','long\_term\_incentive','shared\_receipt\_with\_poi','loan\_advances','other','bonus','total\_stock\_value','from\_poi\_to\_this\_person','from\_this\_person\_to\_poi','restricted\_stock','percentage\_total\_stock\_exercised','salary','total\_payments','exercised\_stock\_options']

Aside from “poi,” which will serve as our labels, there are now 16 features in this list that I could choose to include in my algorithm. If I were to use all 16 of these features, my concern is that I would end up with too high a variance, which could lead to overfitting and result in a much higher error on my test set than on my training set.

To compensate for this, I chose to use SelectKBest univariate feature selection with k=7. However, before implementing and tuning my machine learning algorithms in tasks four and five, I had originally started out by running SelectKBest with k=3. Once I had tuned and tested my algorithms, I found that holding my classifier parameters constant, my accuracy, precision, and recall scores from running tester.py increased as I increased the value of k to 7. Once I got to k=8 and greater, all three scores began to decrease. This led me to conclude that using the top seven features according to SelectKBest gave me the best balance between bias and variance.

Here are the seven highest scoring features according to SelectKBest:

1. total\_payments (feature score: 24.8150797332 )
2. bonus (feature score: 24.1828986786 )
3. other (feature score: 20.7922520472 )
4. percentage\_total\_stock\_exercised (feature score: 18.2896840434 )
5. expenses (feature score: 11.4584765793 )
6. deferred\_income (feature score: 9.92218601319 )
7. from\_this\_person\_to\_poi (feature score: 9.21281062198 )

I was pleasantly surprised to see that when using k=7, the first feature that I engineered (percentage\_total\_stock\_exercised) made the cut!

(Note: According to the Enron insider pay spreadsheet, the “other” feature represents miscellaneous payments an insider received that could include items such as severance, consulting services, relocation costs, tax advances and allowances for employees on international assignment, etc.)

Since I intended to use Decision Trees and Naïve Bayes as candidates for my preferred machine learning algorithm, I did not foresee a need to deploy feature scaling on my features because neither of these algorithms pay attention to actual distances between data samples in the way that, say, SVM would.

*3. What algorithm did you end up using? What other one(s) did you try? How did model performance differ between algorithms? [relevant rubric item: “pick an algorithm”]*

Answer:

I implemented both Decision Tree and Naïve Bayes algorithms. I ultimately ended up using a tuned Decision Tree algorithm. Naïve Bayes doesn’t have parameters that need to be tuned.

Before testing each algorithm in tester.py, I first found that by cross-validating using KFold (n\_splits=3, shuffle=True), my Naïve Bayes algorithm tended to have the highest accuracy most of the time, followed by my tuned and un-tuned Decision Tree algorithms, which tended to swap second and third place with each other from test to test.

I was thus in for a surprise when running Naïve Bayes on my dataset in tester.py: precision and recall never surpassed 0.3, no matter how many attempts I made. Since there was no tuning I could do to improve upon the performance of Naïve Bayes, I decided that my best bet would be to try and tune my Decision Tree algorithm.

*4. 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? What parameters did you tune? (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). [relevant rubric items: “discuss parameter tuning”, “tune the algorithm”]*

Answer:

Tuning the parameters of an algorithm is the process of finding the combination of values for those parameters that result in the best possible performing classifier. It’s essentially answering the question of how we can best customize a classifier, using the various options built into the classifier’s algorithm, such that our classifier will consistently make the best possible predictions.

To give an example, the Decision Trees algorithm contains one parameter called min\_samples\_split, which is the number of samples that must be present if the algorithm is to split an internal node in the tree. The default value is 2, however, greater values can be used. So, in this case, parameter tuning would be the process of fitting, making predictions, and scoring the performance of the Decision Trees classifier, using different values for min\_samples\_split, and choosing the value that resulted in classifier with the highest scores for accuracy, precision, and recall.

I felt that doing a manual guess-and-check process would be cumbersome. Thankfully, using GridSearchCV helped me to be able to test multiple combinations of different values for different parameters for my Decision Trees classifier, all in one go. GridSearchCV is similar to KFold cross-validation in that it cross-validates different combinations of parameter values, all while still working off of one dataset.

I used GridSearchCV to find the best performing combination out of the following parameters for my Decision Tree classifier:

{'criterion':('gini', 'entropy'), 'splitter':('best', 'random'), 'min\_samples\_split':[2,4,6,8]}

Interestingly, I found that when I ran GridSearchCV to tune my Decision Tree classifier using the above parameter grid, GridSearchCV didn’t consistently confirm the exact same combination of parameter values as being ideal — it tended to vary. After several attempts, however, I did spot some patterns:

More often than not, GridSearchCV chose a lower value (2, or 4) for the best value for min\_samples\_split. Furthermore, I saw that “gini” and “best” were more often chosen as being the best values for the criterion, and splitter parameters, respectively.

Perhaps if the dataset had been substantially larger, GridSearchCV on its own would have been sufficient. In this case, I found that GridSearchCV as a tool helped me to narrow the range of values for classifier parameters that I would need to guess and check, but it didn’t completely eliminate the necessity of my spending some time doing some manual guessing and checking.

Indeed, after some further guessing and checking, using lower values for min\_samples\_split, I found that the following combination of parameter values resulted in the consistently strongest performing precision and recall scores for my Decision Tree classifier when it was run in tester.py:

clf = tree.DecisionTreeClassifier(min\_samples\_split=2, splitter='best', criterion='gini')

Here are the feature importance scores for this classifier, calculated after fitting and cross-validating with KFold:

1. poi (Feature Importance score: 0.3239824324 )
2. other (Feature Importance score: 0.209474919841 )
3. deferred\_income (Feature Importance score: 0.156114255014 )
4. bonus (Feature Importance score: 0.133677298311 )
5. from\_this\_person\_to\_poi (Feature Importance score: 0.117338961851 )
6. percentage\_total\_stock\_exercised (Feature Importance score: 0.0594121325829 )
7. expenses (Feature Importance score: 0.0 )

*5. What is validation, and what’s a classic mistake you can make if you do it wrong? How did you validate your analysis? [relevant rubric items: “discuss validation”, “validation strategy”]*

Answer:

Validation is the process of testing my machine learning algorithm to verify that it performs well and will serve as a useful prediction tool. In this project, my algorithm’s purpose is to predict whether or not an individual is a person of interest (POI). I use validation to confirm that my classifier actually does do this as well.

A classic mistake in validating an algorithm is training it and testing it on the exact same set of data points. This results in an algorithm that almost surely overfits its training data, and the reason this is bad is that I have no way of knowing how well my classifier is actually able to make predictions.

If I train and test my algorithm on the exact same set of data points, the best I can confirm is that I have made a decent model for the data I’ve already observed. I have no way of actually knowing whether this model will predict anything useful if/when it is used out in the wild, which ironically, is really the entire point of creating a machine learning algorithm in the first place: to be able to make decent predictions.

To avoid overfitting, we use cross-validation when measuring the predictive performance of our algorithms. Cross-validation takes the dataset and splits it into two segments: the training segment and the test segment. I train my algorithm on the training segment of data points, and then test it out using the test segment, which is a completely independent set of data points.

For this project, testing my algorithm means that my algorithm looks at each data point inside the test segment, and makes a prediction whether or not that data point (an individual) should be classified as a POI. A performance score, such as accuracy, precision, or recall, then compares my algorithm’s predictions with the known classifications (whether the individual is actually a POI or a non-POI) for each data point inside the test segment. This is how I can gauge how effective my algorithm would actually be at predicting whether any former Enron employee not originally in the dataset should have been classified as a POI.

On my first pass, I used simple cross-validation that split the dataset into one training segment and one testing segment:

features\_train, features\_test, labels\_train, labels\_test = train\_test\_split(features, labels, test\_size=0.3, random\_state=42)

While this kind of cross-validation did prevent overfitting and allowed me to compare the accuracies of my various classifiers against each other, I found that the high accuracy scores here did not translate to high accuracy scores when running my classifiers in tester.py. My classifier also failed to meet the minimum scores of 0.3 for precision and recall.

I decided to abandon simple Train/Test split and use KFold cross-validation instead:

kf = KFold(n\_splits=3, shuffle=True)

Although my KFold implementation was not quite as stringent as the Stratified Shuffle Split used in tester.py, I did find that it graded the accuracies of my classifiers harshly enough so that I was pushed to tune them further, which enabled me to eventually achieve the required minimum 0.3 scores for precision and recall when running tester.py.

*6. Give at least 2 evaluation metrics and your average performance for each of them. Explain an interpretation of your metrics that says something human­understandable about your algorithm’s performance. [relevant rubric item: “usage of evaluation metrics”]*

Answer:

I evaluated my algorithm’s performance using accuracy, precision, and recall. Here are the individual and average scores for each after testing my classifier in tester.py twenty times:

Classifier: clf = tree.DecisionTreeClassifier(min\_samples\_split=2, splitter='best', criterion='gini')

|  |  |  |  |
| --- | --- | --- | --- |
| **Attempt** | **Accuracy** | **Precision** | **Recall** |
| 1. | 0.82100 | 0.32763 | 0.32550 |
| 2. | 0.81953 | 0.32191 | 0.31950 |
| 3. | 0.82093 | 0.32624 | 0.32200 |
| 4. | 0.82207 | 0.32925 | 0.32250 |
| 5. | 0.81887 | 0.31940 | 0.31700 |
| 6. | 0.82193 | 0.32856 | 0.32150 |
| 7. | 0.82047 | 0.32258 | 0.31500 |
| 8. | 0.82113 | 0.32603 | 0.32000 |
| 9. | 0.82107 | 0.32371 | 0.31400 |
| 10. | 0.82153 | 0.32913 | 0.32600 |
| 11. | 0.82247 | 0.33095 | 0.32450 |
| 12. | 0.82047 | 0.32526 | 0.32250 |
| 13. | 0.82193 | 0.32995 | 0.32550 |
| 14. | 0.82027 | 0.32424 | 0.32100 |
| 15. | 0.82233 | 0.33079 | 0.32500 |
| 16. | 0.82273 | 0.33350 | 0.33000 |
| 17. | 0.82080 | 0.32485 | 0.31900 |
| 18. | 0.82200 | 0.32926 | 0.32300 |
| 19. | 0.81967 | 0.32134 | 0.31700 |
| 20. | 0.82160 | 0.32790 | 0.32200 |
| **Average** | **0.82114** | **0.32662** | **0.32163** |

Over 20 attempts, my algorithm scored an accuracy of 0.82114, a precision of .32662, and a recall of .32163 when run in tester.py.

My average accuracy score means that on the whole, my algorithm correctly guesses the label (POI or non-POI) 82.1% of data points fed to it. This is a nice, basic metric to begin with, but it’s not the best measure of success for my algorithm in this Enron project. After all, my non-POI class is much larger than my POI class, and I expect that when my algorithm is fed any appreciably large dataset of Enron employee information, that most will actually be non-POIs.

Indeed, out of the 144 entries in my dataset, only 18 are POIs. Accuracy gives my algorithm too much credit for marking a data point as a non-POI — something that any random, un-tuned classifier might be able to do pretty easily, thanks to nothing more than the fact that the odds of randomly guessing that a non-POI is indeed a non-POI are pretty favorable.

To properly evaluate my algorithm, I want to be able to properly penalize it for failing to properly identify a POI when it should have done so. Accuracy won’t help with this, but the precision and recall metrics will.

My algorithm’s average precision score of 32.7% and its average recall score of 32.2% tell a very different story than the comparatively higher accuracy score did. In the case of precision, this means that when my algorithm predicted that a data point/individual was a POI, that there was actually only a 32.7% chance that the individual was actually a POI. The recall score indicated that of all the individuals who were actually POIs, my algorithm was only able to find 32.2% of them, the rest were incorrectly predicted to be non-POIs.

These precision and recall scores paint a much more sobering picture of my algorithm’s predictive ability — my algorithm is more often wrong than right when it guesses someone is a POI. And even when it does flag one or more individuals as POIs, there are still twice as many POIs that my algorithm completely misses.