Project 5: Identify Fraud from Enron Email

GLOSSARY

POI	Person of Interest. It indicates an ENRON employee who was found to be involved in the fraud by investigations.
Non-POI	Enron employee that was pointed to by investigations to have taken part in the fraud.
SVM	Support Vector Machines, a machine learning algorithm.
kNN	K Nearest Neighbour, a machine learning algorithm.
NLTK	Natural Language ToolKit. A python library to work with Natural Language Processing.
NLP	Natural Language Processing

PROJECT SUMMARY

General:

The project is about identifying those who were involved in the Enron scandal of the early 2000s. The provided data is split into two, independent categories:

1- The financial data of a selected set of members
2- The emails of another selected set of members

The two sets do have some overlapping, but they do not include exactly the same employees.

For each set, we have a number of employees who are declared as Person of Interest (POI for short). These employees were the ones that were pointed to by the investigations. The whole idea of the project is to see if we can build a model that can differentiate whither an employee is a POI or not, based on his\her data.

Each dataset will be examined independently, i.e. I will not try to use a person's data from both data set to estimate their "POIness". This was tempting at first, but given that the datasets contain a lot of different people (Only 4 POI in the text data for example), this will not be a practical approach.

The final thing to mention is that all of my initial work was done over a Jupyter Notebook, so the score numbers given here in the report represent the scores that I had using a cross validation with a much lower number of folds, so numbers are substantially higher than the ones I got from the tester script. But these numbers were a good indicator, when I ran the best models that I had, I got very good results. I deem them good results based over this thread where it says that scores in the 0.4s or 0.5s order are pretty difficult to get:

https://discussions.udacity.com/t/whats-the-highest-recall-and-precision-that-youve-gotten/27271/2

Financial Data:

Overview

The dataset contains 143 useful observations (There were three discarded observations, TOTAL, THE TRAVEL AGENCY IN THE PARK and LOCKHART EUGENE E). For each observation, we had 22 features that describe them, with the most noteworthy one is the POI label, and whither this person is a POI or not.

I have split the data to POI and non-POI members, and then called the describe() function. A few interesting things became obvious:

1- POIs do not have any restricted stock deferred values. This feature will be discarded since I guess it would cause overfitting (And even the available values for non-POI count is 17, unreliable)
2- Deferred income is a negative value
3- There are some entries (20 to be exact) that have no total payments. After inspecting these, it turned out that these are sparse entries. Some of them had their stocks and email data available. It is a hard choice whither to keep them or not. Also, they were all non-POI.
4- Director fees and loan advances have low counts. They will be discarded.

Email Data:

The email data is a collection of Enron emails sent and received by some of the top Enron employees using their corporate email account. There is roughly half a million email and the unzipped data is about 2 gigabytes. The employees in the dataset are not exactly the same ones present in the financial data, therefore the processing of these datasets will be done independently.

Financial Data

Preprocessing

Data Cleaning

There were 3 observations that were discarded:
- TOTAL: The total of all samples, not really an observation
- THE TRAVEL AGENCY IN THE PARK: Not sure what is that, but definitely not a person
- LOCKHART EUGENE E: This one had no data, only NAs.

After that, the features were split into three categories:
1- ‘poi’ : That’s our classification label
2- Numeric features: These are our potential features
3- Non-Numeric features: Like the name and email, these were not really useful, so these will be discarded

Also, there are some obvious data entry errors within the data. For example, BANNANTINE JAMES M had a salary of 477$. I think the number was probably 477,000$, since he was a top executive (and seeing that he had a total payment of almost one million dollars), but I have decided not to change the value to 477,000.

So after loading the data into a Pandas dataframe, the numeric features were cast into float and the non-numeric features columns were dropped.

Outliers

Although there are some outliers within the true data, like Kenneth Lay, I have decided not to filter him out due to his relevance to the data. I do, on purpose, want my model to learn on him. His “anomaly-ness” is something that I want to be taken into consideration by the model.

Preprocessing for Machine Learning:

Missing Values

The dataset is plagued with missing values. Machine learning algorithms are not designed to deal with NaN and the likes, and these cases must be handled. There exists several approaches to deal with these:
1- Substitute missing values with the feature’s mean
2- Substitute missing values with the feature’s median
3- Substitute missing values with zeros

For the first two strategies, sklearn provides the Imputer() class to perform them conveniently. The third strategy can be performed directly through Pandas, just by using the fillna() function.

When I was thinking about which strategy to use, I found that I needed more information about how the data was gathered from the first place, to figure out why we have missing data and what would be a smart assumption about the missing values. But since we have no clue about that, the choice is rather an arbitrary one. I have chosen the mean to represent the missing values, and that was based over my rational about the missing data for the stocks features:
After watching the documentary, I know that the Enron culture was aggressive towards employees who did not buy Enron stocks. So most likely these values were not zeroes, especially for top senior executives who would have never probably gotten to the top positions without being a role model according to the company’s policies. In the beginning, I thought to model the missing values over what we know from the present values, sort of Machine Learning these features as a preprocessing step for building the model, but after another thought this might be too much assumptions, too much fitting for the data we have and I feared losing the generalization. So the simpler choice, i.e. just using the mean, was chosen instead.

Scaling

Since learning algorithms do not make sense of the nature of the given values (e.g. age and salary for example, for it, these are just numbers of equivalent importance to their value), most will underperform when the features dimensions have big gaps in their ranges. To overcome that, rescaling features to be of unit size is considered a best practice. Sklearn provides two scalers, the MinMaxScaler() which scales from 0 to 1, and MaxAbsScaler() which scales from -1 to 1. My choice was to use the MinMaxScaler(). It is noteworthy though that not all classifiers will perform better after scaling. Tree based classifiers (Decision trees and random forests) are not affected by scaling the data, they will still perform the same.

A small helper function preprocess_df_for_ML() was made to do all the preprocessing, and can be found in the Jupyter notebook attached with the project.

Features

The features that will be used (i.e. the numeric ones) were further split into two categories:
1- Financial features, which describe the income and stock of an observation
2- Email features which states the number of emails sent\received by this person

New Features

After the inspection of how the features correlate with the POI label, I have tried to systematically inspect correlations between some features ratios, when this ratio makes sense. So I did not for example try to find the correlation between the number of sent emails and salary. The ratios were computed between the financial features among themselves, and the email features among themselves. Of these ratios, I have picked the following ones to be added to my list of features:

- Long Term Incentives / Total Payment, so how much this person's income from Enron was in the form of long term incentives. These correlated with POI at 0.326
- From this person to POI / from messages: This one is the percentage of emails this person sent to a POI out of all their sent messages. It correlated at 0.34 with POI

As it turns out, the first feature will be the only engineered feature to be used by the final model, and it gave a great improvement over the results (See the results section for more details).

Feature Selection

For the feature selection, I have decided to first manually pick the best features over their correlation with POI. These features were:
1- Exercised Stock Options, corr = 0.5
2- Total Stock Value, corr = 0.37
3- Bonus, corr = 0.3
4- Long Term Incentives / Total Payment, corr = 0.326
5- From this person to POI / from messages, corr = 0.34

I have done this manually just to have something to compare to when I run the automated K-Best process.

Next, I have used SelectKBest() to get me the top 10 features, trying both χ^2 and F-Score as metrics. For each metric, its rank is assigned a score (Highest = 10 points, and then decreases by one with each next one). The evaluation will be run using a crossvalidation stratified kFolds, to avoid overfitting the evaluation of the features. After running both rankings, I added their scores to get a final score over which are the best features, and these were the results:

1- Bonus
2- total_stock_value
3- shared_receipt_with_poi
4- salary
5- exercised_stock_options
6- total_payments
7- long_term_incentive

8- lti_ratio
9- sent_to_poi_ratio
10- other

So KBest and correlations have agreed over 3 out of the first 5 parameters. I have decided to have faith in the automated KBest process, and I will use its ranking for the model evaluation at first. At a later stage when I would have trained the models and know which model gives good predictions and is fast to run, I will do an exhaustive combination of the features and see if there exists a better combination.

Dimensionality Reduction Attempt

I tried to see if squeezing the data would show a hidden latent variable that can be useful for the prediction. I am disregarding the use of dimensionality reduction as a way to simplify the data as the dataset is not massive anyway. It is very manageable in terms of size and processing power needs. So unless the dimensionality reduction will enhance the results, I will not bother implementing it.

So the latent variables I suspect are divided into two main categories:

1- Financial
2- Email Data

The financial data is further divided into two categories:

A) Income
B) Stocks data (Will not be used, see explanation below)

So given my vision for the possible latent variables, I applied PCA only over each category alone, so no combination between variables coming from separate categories. I think doing it all combined does not make sense, even if it yields good results (I have not tried it). In that hypothetical situation that a PCA applied over all the data gives a good intuition, I think this would be only due to chance, as I cannot think of a hidden logic about a relationship between email counts and the amount of stocks owned for example.

Also, the “total”s variables were discarded, as a way to avoid their dominance over the data.

For the stocks data, we do not have all that much variables to use from the first place, so we would try to reduce them. There are some variables that were discarded already (due to their sparsity), a variable that is already chosen to be used as is so we are left with only one variable, the restricted stock. So, no PCA here.

Now that I have presented a global overview of how I viewed the application of PCA, now I will present the implementation and the results.

1- Over each category, build an exhaustive combinations of its variables.
2- Build the PCA for each combination
3- Measure the correlation of the first component and POI

There was no interesting correlation within the groups, but nevertheless there was some interesting variance explanation. For example, 'from_poi_to_this_person', 'from_messages' and 'from_this_person_to_poi' can be compacted into a single variable and still hold 99% of the variance. I am not sure about how to interpret this, is it a latent variable or not, but for this project I am going to leave this as is and move on with my model building.

Metrics

The metrics that will be used for judging the performance are the F1 scores, recall and precision. The F1 score is just the harmonic mean of recall and precision, but I have decided to look at the individual constituents when I examine my results. The use of these metrics, and not just an overall score about how many correct answers our model gave back, is that we can judge better if the model is actually biased towards a specific answer. In one of my attempts, the model was always predicting that answer is 'non-POI'. And because of the imbalance between the numbers of non-POI versus POIs (There is a lot more non-POIs), the model's score was very acceptable. So this potential pitfall is the main reason why we go for precision and recall, and not just a simple percentage of correct answers.

As how to interpret the precision and recall for this project, we can think of the precision as how well was the estimator able to identify true POIs out of all its POI predictions. In other words how many of the predicted POIs were correct. It is the ratio between correct POI predictions out of all its POI predictions (Whither it is correct or not). Higher precision means that the estimator did not incorrectly label a non-POI as a POI. Its equation is:

$$Precision\ = \frac{True\ Positives}{True\ Positives + False\ Positives}$$

For recall, it is about how many POIs were found out of all POIs fed to the estimator. In other words, how many correct answers were found. Its equation is:

$$Recall\ = \frac{True\ Positives}{True\ Positives + False\ Negatives}$$

There is no right or wrong when it comes about favouring recall or precision over another, it is application dependent. We should favour precision when the cost of misidentification is high, like how a judge thinks when he\she examines a criminal case. In case of doubt, they would favour setting a criminal loose instead of incarcerating an innocent person. To favour recall over precision, then that means that the cost of slipping a true positive is high. An example to this is the medical field, we would rather misidentify a healthy person to have cancer and perform more investigations rather than declaring that an actual cancer patient is not sick.

For this project, I do not think that there is a specific bias towards one or another. If the output of this project has the final word, like a judge, then we should optimize the solution for a higher precision at the expense of recall, but if it is used for preliminary investigations, we should favour recall and then do more investigations over the potential POIs. For now, I will aim for the highest balanced scores of both, not one too much higher than the other.

Algorithm

My initial approach for choosing the algorithm was rather a surveying one than having a clear choice in mind. It was more of a brute-forcing approach, taking advantage that the dataset is not that big, so it is computationally affordable to try different approaches (More or less. Training a random forest with an elaborate features grid took forever, and I have halted the training after three hours and reduced the parameters grid).

Parameter Tuning

Before going into the details of each algorithm, I will talk about parameter optimization. Each estimator is controlled by a certain set of parameters. For example, SVM's parameters are C, gamma, kernel type and if it is a polynomial kernel, its degree. These parameters are the real deal, an estimator by its own is not a fire-and-forget tool, how it is going to perform depends over the values of these parameters. To continue with the SVM example,the C parameter controls the "smoothness" of the separation surface. A higher C will result a wiggly surface, it will try to make the separation surface more optimized to solve the training set correctly. But this can result an overfitting as well, so going as high as possible while keeping the generalization is how we would want to optimize the C value.

Of course this process is tedious to be performed manually, so sklearn provides the GridSearchCV tool. This tool takes in a dictionary of an estimator's parameters with a list of the chosen values to try out, and it will try all different combinations of parameters values and return the best combination. The best practice for choosing the possible values for a certain parameter is to go in exponential steps, so that GridSearchCV would pinpoint the range of the best values. Then we can reiterate around that range, sort of doing a manual gradient descent.

At first, I have used the GridSearchCV to find the best parameters, here goes the initial list of estimators along of their different parameters:
Linear Regression

<u>Solver</u>: liblinear
<u>C</u>: 0.001, 0.01, 0.1, 1, 10, 100, 1000
<u>Penalty</u>: L1 and L2

<u>Solver</u>: newton-cg, lbfgs, sag
<u>C</u>: 0.001, 0.01, 0.1, 1, 10, 100, 1000
<u>Penalty</u>: L2

k-Nearest-Neighbors
<u>k</u>: 1 to 30
<u>distance</u>: Eucledian, Manhatten and Chebyshev

Support Vector Machines
<u>Kernels</u>: Linear, RBF, Poly and Sigmoid
<u>degrees (For Poly and Sigmoid)</u>: 1, 2 and 3
<u>C</u>: 0.001, 0.01, 0.1, 1, 10, 100, 1000
<u>Gamma</u>: 'auto', 0.001, 0.0001

Decision Trees
<u>Criterion</u>: GINI and Entropy
<u>Max Depth</u>: None, 1, 2, 3, 4, 5, 6, 7, 8, 9
<u>Max Features</u>: None, 'sqrt', 'log2', 1 to number of features (Changes according to which KBest set we're working

with

Ensembles

AdaBoosst

Number of Estimators: 10 to 100, with steps of 10
Learning rate: 0.1, 0.5, 1.0

Random Forest

Criterion: GINI and Entropy
Max Depth: None, 1, 2, 3, 4, 5, 6, 7, 8, 9
Max Features: None, 'sqrt', 'log2', 1 to number of features (Changes according to which KBest set we're working with
Number of Estimators: 10 to 100, with steps of 10
Learning rate: 0.1, 0.5, 1.0

Each of these classifiers combinations were trained over all KBest combinations. After training and validating, the results seemed nice, even after testing the models over the dataset the result was acceptable. But when I ran the SkLearn's report method, there was a problem: The results never identified any POI. In fact, it always gave an output of "non poi", extremely biased! After inspecting a bit, I think the problem was the heavy unbalance in number of training points, there are much more Non-POI than POIs. So I have decided to redo the training, but this time I will include in the grid a class_weight option, keeping non-POI weight constant at 1, and varying the POI weight from 1 to 7. That seemed good at the beginning, then there was another problem: the scoring method in SkLearn takes the average performance for both classes, and even when I used f1_weighted, that problem persisted. The general score became lower, but the training model returned by GridSearchCV always, somehow, returned a model that simply ignored the POI training model. My response was to develop my own training\scoring code.

This training\scoring method simply starts by doing what GridSearchCV does, i.e. loop over all combinations of an estimator's parameters, and also it will loop over different features combinations; a very exhaustive method. After training, it is used to predict the test set. But then instead of getting the best score, it gets the confusion matrix of its prediction. From the confusion matrix, I check if the confusion matrix contains correct predictions for both categories, and if during ths KFold cross validation the same applied for a minimum of 6 times.

Taking into consideration that for the current data set all models performed more or less the same, I have decided to reduce this search down to four estimators only:

1- Support Vector Machines
2- Logistic Regression
3- K Nearest Neighbours
4- Decision Tree

As mentioned earlier, when evaluating the performance I looked at the f1-score and its constituents (i.e. precision and recall). And I paid a special care for those of the POI testing units, not the overall performance. It is similar to the example of cancer detection, that we favour false positives over false negatives and we bias our algorithm as such.

In the beginning, the results were plagued with the same problem: most estimators gave a constant answer to the test set, always 'non-POI'. At this point I did not think yet that the imbalance between the number of POI and non-POI could be the reason, thus I did not touch the weights for each category. The thing worth mentioning here is that I was able to still get acceptable results by changing the training-set to test-set ratio. When I have increased, a lot, the size of the testing set (It became 40% of the whole data), things seemed to be well. I had excellent results for both training and testing results. But part of me thinks that this is a coincidence, not a reliable technique, but at the end I will remember that when everything else fails, I will try experimenting with the ratio of the train-test split.

Validation

As required by the rubric, I am including a part about explaining model validation. The big problem about just taking scores as a metric for an estimator's performance is that it does not show if the model is over or underfit. So we have to spare a part of our input data, so that the model would be tested against unseen inputs. This is the first split for validation.

Next, while training the model, we further split training data into folds, each fold is a different combination of the training data split into training and validation sets, this step is called cross-validation. I used cross validation all over the project, even when I was doing feature selection, i.e. have taken the scores of features from a cross-validation split, computed the score of the feature on each fold and then use an average score for the feature.

Results

I have run each estimator with different splits, and compared the results. The iterations were too many, so without adding unnecessary details here are the quick results:

1- With weights between the two categories (non-Poi\POI) equal 1:1:
   a. Support Vector Machine Poly Kernel was remarkably performing well. By well I mean that feeding it different splits, it always had a model that performed well. Third degree kernels were slightly better than second degree ones.
   b. In the second place, comes decision trees.
   c. Logistic Regression, SVM with other kernels and kNN were not up to the challenge. I had to customize very specific splits and custom kfolds (apparently with k = 10, it was not good. K=5 or less was not good neither.) to get results.

2- After changing the weights (Most effective weights were around 1:6 for a cross validation of 10 folds):
   a. Decision Trees were excellent. They performed the best.
   b. SVM Poly Kernel seemed to give around the same performance, as with equal weights. But unlike unweighted trials, here second degree kernels were noticeably better than third degree ones.

   c. Logistic Regression started to give better models. What is noteworthy is that its results were very accurate for non-POI. This may be bias, I am not sure.

3- In terms of time taken to perform the training, Decision trees and Logistic Regression were fastest.

4- Decision trees gave the best results when given the fewest number of features. Other classifiers needed more features to match (Actually exceed in the case of equal weights) the decision tree classifier's performance.

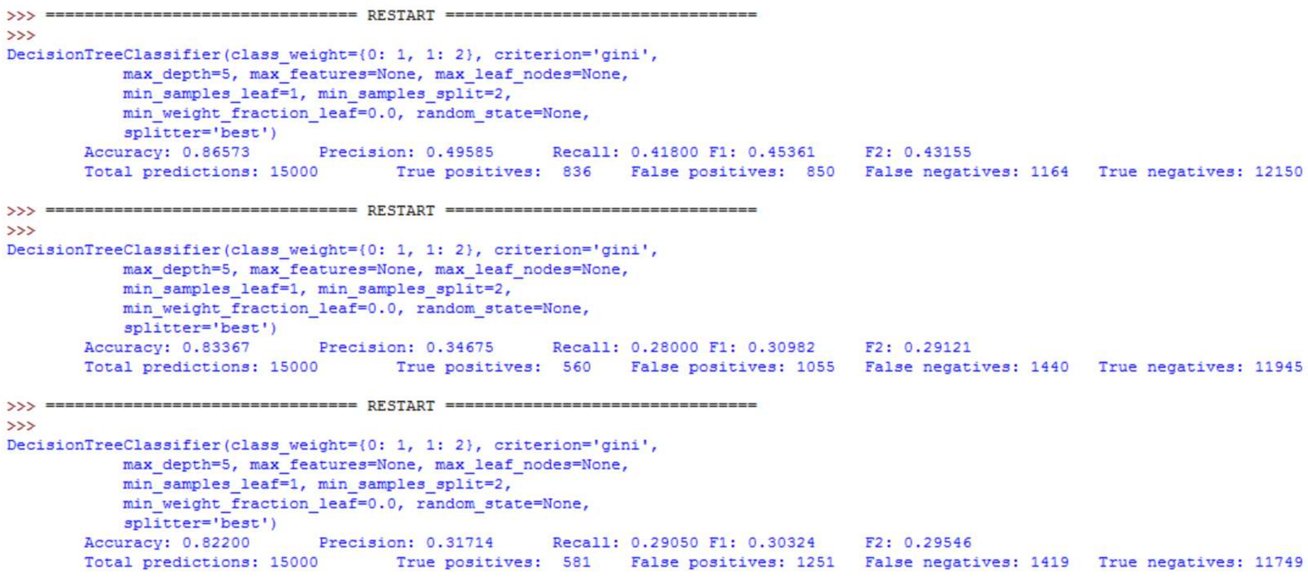
5- All classifiers except decision trees performed best when they used a large number of features. In one of the iterations, I have decided to use a reduced set of features: only the top 6 ones. Decision trees' best performance came using a combination of only 4 features, while all other classifiers gave their best when they used all the features.

6- For SVM Poly:
   a. Gamma = 1000 sounds like a good choice. Higher influence from far points seems to cause poor results, so a high gamma is required.
   b. C value: Apparently low numbers did not work best, we needed a wigglier surface to start getting good results. Starting from C = 100 and higher, results seemed promising, and they did not change in quality with much higher C's. Because of this, I think a C = 100 seems the good choice, the smoothest possible value that gives acceptable values.
   c. With high C and gamma values, training was significantly slower.

7- When I ran my decision tree classifier over the tester.py script, initially I got results about 0.36. The results changed a bit with each run, but they were always near that number. I was not satisfied by the result, and have manually changed some of the parameters to see how they affected the tester.py script. When I reduced the POI weight from 6 to 2, the performance sky-rocketed, becoming 0.47 for precision and 0.42 for recall. So I am going for this adjustment, not the one I came up with using the Notebook.

8- I went an extra step, and looped over different values of max_depth , max_features and class_weight. Here are the effect of each one:
   a. Class_weight:
      i. at ratio 1:1, the model was not all that good. Precision\Recall were in early 0.3s
      ii. at ratio 1:2 : An excellent ratio, and returns very good precision and slightly less recall score.
      iii. at ratio 1:3 The scores were still very good, but in this case the recall was the slightly higher score. So depending over our intentions, if we are willing to sacrifice some extra false positives to catch more POIs, ratio 1:3 is the one to be used.
      iv. At higher ratios, the recall score was very high (reaching as high as 0.7 for a weight of 1:8, with precision = 0.2). This one is useless, as we have a ridiculous number of false positives.
   b. max_depth: Maximum depth is the depth of the tree after which the algorithm will stop splitting the data into new branches. Theoretically, we can go down as much as the number of samples in the data, but naturally this maximum is not something we see in practice. Forcing the tree to stop at depths 5 or 6 improved the results, the precision went up from about 0.42s to 0.47s
   c. max_features controls how many features to take into consideration while finding a split for a certain node. The tree performed best when this was None (It was also validated by trying out different numbers of max_features, max_features = 6 performed best, and we had 6 features).

9- The engineered feature had a great effect over the overall result. I have tried to run my code once with it, once without it and a third time without it but substituting it with a good feature (shared_receipt_with_poi). After each change, I ran the tester.py script. Here are the results:



```
>>> ================================ RESTART ================================
>>>
DecisionTreeClassifier(class_weight={0: 1, 1: 2}, criterion='gini',
            max_depth=5, max_features=None, max_leaf_nodes=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, random_state=None,
            splitter='best')
        Accuracy: 0.86573       Precision: 0.49585      Recall: 0.41800 F1: 0.45361     F2: 0.43155
        Total predictions: 15000        True positives:  836    False positives:  850   False negatives: 1164   True negatives: 12150

>>> ================================ RESTART ================================
>>>
DecisionTreeClassifier(class_weight={0: 1, 1: 2}, criterion='gini',
            max_depth=5, max_features=None, max_leaf_nodes=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, random_state=None,
            splitter='best')
        Accuracy: 0.83367       Precision: 0.34675      Recall: 0.28000 F1: 0.30982     F2: 0.29121
        Total predictions: 15000        True positives:  560    False positives: 1055   False negatives: 1440   True negatives: 11945

>>> ================================ RESTART ================================
>>>
DecisionTreeClassifier(class_weight={0: 1, 1: 2}, criterion='gini',
            max_depth=5, max_features=None, max_leaf_nodes=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, random_state=None,
            splitter='best')
        Accuracy: 0.82200       Precision: 0.31714      Recall: 0.29050 F1: 0.30324     F2: 0.29546
        Total predictions: 15000        True positives:  581    False positives: 1251   False negatives: 1419   True negatives: 11749
```

The first run was when the lti_ratio was included in the features set, and the second was without it and the third is substituting it. A substantial drop in the performance can be seen when comparing the first result against the latter two. So, this feature is definitely a success!

Conclusion

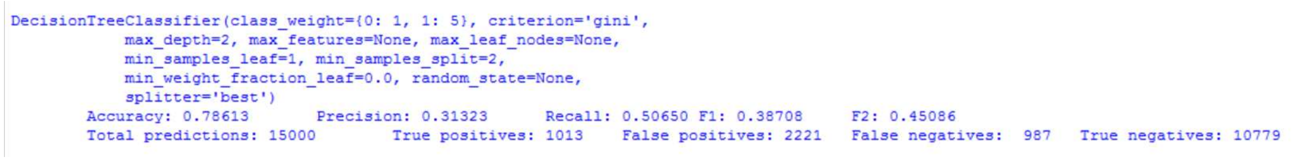
My final classifier is a decision tree classifier with weighted classes. The weight is 1 for non-POI to 2 for POI.

- Weight: Non-POI = 1, POI = 2
- Max Depth = 5
- Max Features = None
- Features = Bonus, Total Stock Values, Salary, Exercised Stock Options, Total Payments, Long Term Incentive by Total Payments ratio.

Note: If we wanted to favour recall over precision, then the model that would still pass the project's criteria using the same features is is:

- Weight: Non-POI = 1, POI = 5
- Max Depth = 2
- Max Features = None

This last one gave:



```
DecisionTreeClassifier(class_weight={0: 1, 1: 5}, criterion='gini',
            max_depth=2, max_features=None, max_leaf_nodes=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, random_state=None,
            splitter='best')
        Accuracy: 0.78613       Precision: 0.31323      Recall: 0.50650 F1: 0.38708     F2: 0.45086
        Total predictions: 15000        True positives: 1013    False positives: 2221   False negatives:  987   True negatives: 10779
```

Lessons Learned

1- The process of feature selection is really tricky and requires a lot of work. SelectKBest is good as a starting point, but we have to accept that it will not give the best possible results.

2- We must always keep in mind the goal of the machine learning training process, and this is to find the best model that will perform “on average” better than others. So we have to try the model on different training\testing sets to see that it consistently works. We must not get fooled by a model that gave excellent results on a certain split, the best model might be one that never performed with the highest scores, but the one that gave consistently acceptable results across different sets.

3- Automated features provided by sklearn like GridSearchCV are nice, but they can lead to false beliefs about the model’s performance. I think that before settling on a final model, trying to do the learning process manually and inspecting the results of each step is the best way to gain confidence about the model.

4- When everything else fails, remember the little details about the model and the data, like the size of the training\testing splits. Changing this slice gave me, counterintuitively, better results.

5- A small dataset is really a challenge. The model performance changes a lot depending over which part was passed for training and validation and which part was passed for testing. So the lesson is: working on small dataset will not give a reliable model. The best thing to be done is just trying to see which estimator seems to give consistent results, and which parameters work best with this estimator but that in no way means that we should take the training\testing results as a performance guarantee.

6- The other thing to mention about feature selection is that it is not really a universal process, a one-size-fit-all process for all estimators. When I bruteforce-searched for the best features for different estimators, the highest scoring features were a little bit different between one another. True that they agreed over most of the features, but when we talk about the best set of features, we have to consider which algorithm we are going to use.

7- Training time varies widely between estimators. Logistic Regression (liblinear) was very fast to train. Random forests were the slowest.

8- Do not settle for the best score parameter or even the prediction scores on the testing set to evaluate the model, looking at the confusion matrix is the thing to be done. That reminds about looking only at averages when trying to understand some data statistically. True that averages can be a good indicator, but the possibility of having outliers can ruin any insight we get from it.

9- Decision trees performed better than other estimators when the number of features was low.

10- I am personally biased for support vector machines! I will have to work on that and keep an open mind about other estimators. Funny that while I want to reduce bias in my models I myself had a personal bias. So the key lesson here is that before attempting to explore the problem, one must remove their preconceived ideas out of the way and start learning the data with an open mind.

11- It takes time to build an intuition about guessing the best parameters for an estimator. But grid-searching is there for the help in the beginning, it shows a range of parameters that work best for the given data.

12- Lastly, since I have talked a lot about the most optimum solution and features to use, I understand that it is not always practical to look for it. We have to define what a “good enough” solution is and aim for it. The combinations of features can be endless, and there could be hidden somewhere within the features a truly optimized combination (Especially if we take into consideration the creation of new features out of existing ones, the possibilities could be infinite). Aiming for that is not practical. Considering that I could have finished the project much earlier if I went for a simple solution and take the first acceptable results I came by, and knowing that there is still room to search for an even better solution since I have not covered all the possibilities, it is important to know when to stop. This might be dictated by the deadlines imposed by the customer or maybe just the available processing power, but it is important to set goals before starting the process.

Email

My work over the email data was more of an exploratory work about the capabilities of the NLTK library, and the Natural Language capabilities of sklearn. I have attached a Jupyter Notebook of a simple project using sklearn's NLP, which I must say is quite spectacular in terms of results.

In the beginning I used NLTK a lot to explore the data and play a bit, but later I have decided to use sklearn directly as I as things progressed and I had a clear idea about what I want to do, I found out that I will not do more than tf-idf as I wanted to keep things simple.

My approach was that if I want to identify people, I had a better chance looking at their sent emails than their received ones. So all of my work was over their sent items.

In the beginning, I created a function called build_corpus(). This function looped over all users, extracting emails that are in any folder that was named ["_sent_mail", "sent", "_sent", "sent_items"]. For each email, it extracts its body, filters out if there is any forwarded part and leaves only the text this person have written. Then it adds this text to a list, and in another list it adds the person's name as the label to be used for classification. Later, this person's name was transformed into a Boolean value that corresponds to their POIness.

Next, stop words were filtered out of the text list. I used sklearn's ENGLISH_STOP_WORDS. The result is then used to build the Term Frequency-Inverse Document Frequency (tf-idf) and this will be the input for the classifier. The result had 126,462 observations, and 94,185 features.

For the classifier, I used Naïve Bayes, specifically sklearn's MultinomialNB. The train-test split was 0.8 to 0.2 with a stratified shuffle. Then I cross validated the training using a 10-fold training-validation split. Here are the results of the 10 folds:

1- 0.9726
2- 0.9731
3- 0.9731
4- 0.9733
5- 0.9732
6- 0.9729
7- 0.9728
8- 0.9732
9- 0.9733
10- 0.9730

When I used the classifier over the test data, the result was 0.9730. Quite amazing!

In the beginning, I was quiet skeptical about the result of this trial, since we have only 4 POI out of 150 people, but it seems that the massive number of samples was just enough to clear any confusion. In a future work, I plan to explore more what makes things so obvious for the classifier, what words were so unique? What were the topics they talked about? Even more, what are the topics that a POI talked about to other POIs and never to non-POIs?

So this concludes this part, it is a short but satisfactory work for me.