Enron Submission Free-Response Questions

Lekhraj Sharma

Data Analyst Nanodegree

P5: Identify Fraud from Enron Email

April 2016

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"]*

The goal of this project is to identify Enron Employees who may have committed fraud (https://en.wikipedia.org/wiki/Enron_scandal) based on the public Enron financial and email dataset. A machine learning based approach is quite useful as we have good amount of data to look at. The size of dataset makes it impractical for a human to laboriously look for patterns which identify our person of interest. But machine learning based algorithms are good at shifting through large volume of data to identify possible patterns which when coupled with human intuition help build a model that can identify our person of interest with good amount of accuracy.

The original dataset used in this project came from Enron Email Dataset (https://www.cs.cmu.edu/~./enron/enron_mail_20150507.tgz) collected and distributed by the CALO project (https://www.cs.cmu.edu/~./enron/). This dataset was further scrubbed and combined with other inputs (e.g. hand generated list of persons of interest in fraud case). As preprocessing to this project, the Enron email and financial data has been combined into a dictionary (**final_project_dataset.pkl**), where each key-value pair in the dictionary corresponds to one person. The dictionary key is the person's name, and the value is another dictionary, which contains the names of all the features and their values for that person.

final_project_dataset.pkl : The dataset for the project

Number of persons (total number of data points): 146
Number of features: 21
Number of persons of interest (POI) in dataset: 18
Number of non-persons of interest (non-POI) in dataset: 128
Total Money made by Kenneth Lay, Jeffrey Skilling and Andrew Fastow: 103559793 8682716 2424083
Number of persons with valid salary in dataset: 95
Number of persons with valid email address in dataset: 111
Number of persons with not valid total_payment in dataset: 21
Percentage of persons with not valid total_payment in dataset: 14.3835616438
Number of persons of interest with not valid total_payment in dataset: 0
Percentage of persons of interest with not valid total_payment in dataset: 0.0

These 21 features in dataset fall into three major types, namely financial features, email features and POI labels.

financial features: ['salary', 'deferral_payments', 'total_payments', 'loan_advances', 'bonus', 'restricted_stock_deferred', 'deferred_income', 'total_stock_value', 'expenses', 'exercised_stock_options', 'other', 'long_term_incentive', 'restricted_stock', 'director_fees'] (all units are in US dollars)

email features: ['to_messages', 'email_address', 'from_poi_to_this_person', 'from_messages', 'from_this_person_to_poi', 'shared_receipt_with_poi'] (units are generally number of emails messages; notable exception is 'email_address', which is a text string)

POI label: ['poi'] (boolean, represented as integer, value 1 signifies person of interest)

We are interested in building an identifier which can determine POI (and non-POI) label based on selected combination of financial and email features.

As can be seen from above that out of total 146 data points, only 18 correspond to person of interest (POI). Therefore, large portion of dataset has persons of non-POI (128). This kind of imbalanced dataset presents its own challenges as we will see later on (see answer to question 6, evaluation).

In several data points, several of financial and email features have missing value (NaN). As can be seen from earlier details of dataset, several data points do not have valid values, such as salary, email address and total payment. Interestingly while only around 14% data points have total payment value but it is workable since all points of interest do have this value. For example, the person named 'BAY FRANKLIN R' does not have value for director_fees, exercised_stock_options, from_messages, from_poi_to_this_person, from_this_person_to_poi, loan_advances, long_term_incentive, shared_receipt_with_poi and to_messages. All these missing values (NaN) were replaced with zero when converting to numpy array data as part of feature formatting. We can also see big shots (Kenneth, Jeffrey and Andrew) did make good amount of money. We will explore this aspect when we think about a new feature (see answer to question 2).

Yes, there was an outlier entry in this dataset with key as "TOTAL". This possibly was artifact of spreadsheet, its total row. This was removed from data dictionary (see poi_id.py for the code to remove this outlier). From visual inspection of bonus versus salary plot, there were four more possible outliers that stood out. But on closer examination they turned out to be point of special interest as they belonged to person of special interest such as "LAY KENNETH" or "SKILLING JEFFREY" (first one the founder and second one the COO of Enron). Therefore, they were not removed.

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 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.) 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”, “properly scale features”, “intelligently select feature”]

The final list of features used in POI identifier are these five features:

'salary', 'total_stock_value', 'shared_receipt_with_poi', 'bonus', 'expenses'

Started with initial guess of salary as a feature and used a simple and fast Gaussian Naïve Bayes based identifier to evaluate the accuracy. The Gaussian Naïve Bayes is quite fast and allowed to quickly iterate over several features to see which features gave better accuracy. After several such iteration settled on above set of features. Here is quick sample of these iterations:

```
features_list = ['poi','salary']
#Accuracy: 0.71875 Precision: 0.25 Recall: 0.142857142857

features_list = ['poi','salary', 'total_payments', "bonus", "exercised_stock_options", "total_stock_value", "long_term_incentive",
"expenses", "from_this_person_to_poi",  "from_poi_to_this_person", "shared_receipt_with_poi"  ]
#Accuracy: 0.895833333333 Precision: 0.6 Recall: 0.5

features_list = ['poi','salary', 'total_payments', "from_this_person_to_poi","from_poi_to_this_person"  ]
#Accuracy: 0.863636363636 Precision: 0.0 Recall: 0.0

#features_list = ['poi','salary', 'total_stock_value', "from_this_person_to_poi", "from_poi_to_this_person"  ]
#Accuracy: 0.818181818182 Precision: 0.2 Recall: 0.2

features_list = ['poi','salary', 'total_stock_value', "from_this_person_to_poi", "from_poi_to_this_person",
"shared_receipt_with_poi"  ]
#Accuracy: 0.840909090909 Precision: 0.4 Recall: 0.333333333333

features_list = ['poi','salary', "bonus", "from_this_person_to_poi","from_poi_to_this_person", "shared_receipt_with_poi"  ]
#Accuracy: 0.763157894737 Precision: 0.166666666667 Recall: 0.2

features_list = ['poi','salary', "bonus", "shared_receipt_with_poi"  ]
#Accuracy: 0.763157894737 Precision: 0.25 Recall: 0.4

features_list = ['poi','salary', 'total_stock_value', "from_this_person_to_poi", "from_poi_to_this_person",
"shared_receipt_with_poi"  ]
#Accuracy: 0.840909090909 Precision: 0.4 Recall: 0.333333333333

features_list = ['poi','salary', 'total_stock_value', 'shared_receipt_with_poi', 'bonus' ]
#Accuracy: 0.840909090909 Precision: 0.428571428571 Recall: 0.5

features_list = ['poi','salary', 'total_stock_value', 'shared_receipt_with_poi', 'bonus', 'expenses', 'to_messages' ]
#Accuracy: 0.934782608696 Precision: 0.5 Recall: 0.666666666667

# Finally settled on the following feature list as it has right balance
# with minimum number of features with higher performance
#
features_list = ['poi','salary', 'total_stock_value', 'shared_receipt_with_poi', 'bonus', 'expenses' ]
#Accuracy: 0.913043478261 Precision: 0.4 Recall: 0.666666666667
```

Did try initially scaling of features using *MinMaxScaler* as we had substituted value of zero for several missing values. But later on removed it when settling down on Decision Tree classifier. Decision tree algorithm is not impacted due to scaling since each feature is only compared against itself and not traded off against other features.

Did engineer a new feature (called '**high_worth'**) as proxy to identifying high net worth individuals. Hypothesis was that a person committing fraud is more likely to have financial gains, therefore he/she likely to be a high net worth individual (e.g. having more than 1 million in salary plus bonus). It simply had boolean value, 1 denoting that this individual had high net worth. But did NOT end up using it since it did not make significant impact on performance of POI identifier.

Performance with new feature 'high_worth" with GaussianNB classifier:

```
# Accuracy: 0.913043478261 Precision: 0.4 Recall: 0.666666666667
```

The above performance is not different from without adding this new feature. Therefore, did not use this new feature.

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"]*

Finally ended up using "Decision Tree" algorithm as it gave the best result. Did try multiple other algorithms starting with simple and fast Gaussian Naïve Bayes, SVM, KNeighborsClassifier, to more complex ensemble algorithms such as RandomForestClassifier, GradientBoostingClassifier , AdaBoostClassifier,. Here is quick summary of the results with these algorithms:

```
#clf = GaussianNB()
#Accuracy: 0.913043478261 Precision: 0.4 Recall: 0.666666666667

#clf = SVC()
#Accuracy: 0.934782608696 Precision: 0.0 Recall: 0.0

#clf = KNeighborsClassifier(n_neighbors=15, weights='distance')
#Accuracy: 0.913043478261 Precision: 0.0 Recall: 0.0

#clf = RandomForestClassifier()
#Accuracy: 0.934782608696 Precision: 0.0 Recall: 0.0

#clf = GradientBoostingClassifier()
#Accuracy: 0.869565217391 Precision: 0.2 Recall: 0.333333333333

#clf = AdaBoostClassifier()
#Accuracy: 0.869565217391 Precision: 0.2 Recall: 0.333333333333

#clf = tree.DecisionTreeClassifier()
#Accuracy: 0.847826086957 Precision: 0.25 Recall: 0.666666666667
```

While initial performance of Gaussian Naïve Bayes was higher than finally chosen Decision Tree based identifier, did not end up choosing it due to little better results observed during evaluation phase based on performance metrics, especially F1 score (see evaluation question answer later on). As can be seen from above numbers, Decision Tree classifier did much better in balanced combination of accuracy, precision and recall scores.

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? (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 item: "tune the algorithm"]*

We tune parameters of an algorithm with the goal of optimizing the performance of our algorithm on an independent dataset. We sometime call our algorithm parameters as hyperparameters since coefficients found by machine learning algorithm itself are referred as parameters. Like any other optimization, tuning of parameters is generally seen as search problem where we look for set of parameters that give the best accuracy. Our search can be either manual, exhaustive, random or some greedy approach.

Each of our algorithm (e.g. SVM, KNeighbors, AdaBoost, Decision Tree) have parameters that impact both speed and accuracy of algorithm. For example, in Decision Tree algorithm, it has parameter called 'criterion' which determines how split will be made, either using gini impurity or entropy information gain. It is important for us to tune these parameters to optimize the algorithm to give us best results for our dataset. Each dataset and its questions have their unique characteristics. The default parameter settings may not give us the best results for all of our problem datasets. We need to search for parameter values that allow us to improve the performance of our algorithm. It can also sometime help us to speed up running time or run it in reduced memory environment especially for big datasets.

Initially did try to tune parameters of various algorithms (e.g. KNeighbors, AdaBoost, Decision Tree, etc.) by hand by reading their documentation. For example, tried to very number of estimators and/or learning rate for AdaBoost and saw if it helped in boosting the performance. Tried similar tuning with other algorithms as well. Since Decision Tree started to give better results even with default settings, spent more time tuning its parameters. Based on documentation, tried to vary by hand its parameters such as max_depth, min_samples_split and random_state.

```
#clf = tree.DecisionTreeClassifier()
#Accuracy: 0.847826086957 Precision: 0.25 Recall: 0.666666666667

#clf = tree.DecisionTreeClassifier(criterion="gini", max_features=None,  max_depth=None, min_samples_split=1,
random_state=0, splitter ="best")
#Accuracy: 0.847826086957 Precision: 0.166666666667 Recall: 0.333333333333

#clf = tree.DecisionTreeClassifier(random_state=0)
#Accuracy: 0.847826086957 Precision: 0.166666666667 Recall: 0.333333333333

clf = tree.DecisionTreeClassifier(random_state=0, min_samples_split=2)
#Accuracy: 0.847826086957 Precision: 0.166666666667 Recall: 0.333333333333
#
# Following is performance as reported by 'tester.py':
#Accuracy: 0.81871       Precision: 0.37970      Recall: 0.42450 F1: 0.40085
```

Later on did use GridSearchCV to see if it could help in finding the values which better the above performance gained by hand selection.

```
#parameters = {'criterion':('gini', 'entropy'), 'splitter':('best', 'random'), 'random_state':[0,10], 'min_samples_split':[1,8]}
```

```
#clf_grid = tree.DecisionTreeClassifier()
#clf = grid_search.GridSearchCV( clf_grid, parameters)
#
#Accuracy: 0.869565217391 Precision: 0.285714285714 Recall: 0.666666666667
# clf.best_params_ : {'min_samples_split': 8, 'splitter': 'random', 'random_state': 10, 'criterion': 'entropy'}
#
# Following performance was reported by 'tester.py':
#Accuracy: 0.82579	Precision: 0.36081	Recall: 0.28450 F1: 0.31814
```

While initially it did report little better performance during validation, but when tested with ‘tester.py’, hand tuned parameters reported better performance as well as ran much faster! So ended up using hand tuned parameters only.

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 item: “validation strategy”]*

Validation is a process where we use an independent dataset to get an idea of how our identifier will perform when faced with new data. One way to achieve this is to split our original dataset in random way into two parts, training and test data, such that training dataset we use to develop and tune our identifier gives best learning results on test data set. Our training dataset should be chosen carefully to allows us to quickly iterate our features, algorithm and its parameters and check for any potential issues such overfitting, scaling of parameter issues, outliers, etc before we get to test it with our test dataset. One popular technique is called cross-validation, where one estimates how our predictive identifier will perform with new data. We define a dataset to "test" the model in the training phase (i.e., the validation or training dataset), in order to limit problems like overfitting, get an insight on how the model will generalize to an independent dataset (i.e., test or unseen dataset, for instance from a real problem). To reduce variability, multiple rounds of cross-validation are performed using different partitions, and the validation results are averaged over the rounds.

One classic mistake is of **overfitting** to training data. This can happen when we either end up training with test data (use test data in fit for classifier algorithm instead of training data) or end up choosing much larger training dataset (compared to test dataset) or select too many features. When we end up overfitting, our identifier will not generalize well to real test data and end up delivering bad performance on test data (unseen data).

Ended up using **cross validation** to split 1/3rd of dataset randomly as test data and rest as training data.

```
#from sklearn.cross_validation import train_test_split
#features_train, features_test, labels_train, labels_test = train_test_split(features, labels, test_size=0.33, random_state=42)
```

Ended up using the training dataset to fit to classifier and used test dataset to predict the performance of identifier.

```
#clf = clf.fit(features_train, labels_train)
#pred = clf.predict( features_test )
#print "Accuracy:", accuracy_score(labels_test, pred), "Precision:", precision_score(labels_test, #pred),  “Recall:",
#recall_score(labels_test, pred)
```

Following is the performance reported during validation of Decision Tree based identifier:

```
#Accuracy: 0.847826086957 Precision: 0.166666666667 Recall: 0.333333333333
```

6. *Give at least 2 evaluation metrics and your average performance for each of them. Explain an interpretation of your metrics that says something human-understandable about your algorithm's performance. [relevant rubric item: "usage of evaluation metrics"]*

The evaluation metrics corresponding to our final identifier (built on Decision Tree algorithm) are accuracy, precision, recall and F1-score.

#Accuracy: 0.81871 Precision: 0.37970 Recall: 0.42450 F1: 0.40085

While **accuracy** of around **81%**, therefore 81% of total data points were classified correctly, is good, it may not give true picture of performance on our kind of skewed dataset. In our dataset, there are only few persons of interest (POI) compared to large number of non-POI. In these kind of skewed distribution, it can be easy to predict non-POI and still get good accuracy but still not solve the problem of identifying the real POIs (e.g. an identifier which just returns non-POI always can still have high accuracy!).

The metrics **precision** and **recall** are better measure of performance for imbalanced dataset like ours. A **precision score** of around **37%** means we have reasonable confidence that identified POI is likely to be a real POI and not a false alarm. A **recall score** of around **42%** means we have reasonable confidence that our identifier will correctly identify a POI if it is given a real POI, and it is less likely to flag a non-POI wrongly as POI.

Having reasonable **F1 score**, around **40%**, indicates our identifier is balanced identifier not overly skewed to either reporting non-POIs as POI or missing a real POI. Therefore, if our identifier finds a POI than that person is almost certainly be a POI, and if the identifier does not identify someone as POI, then they are almost certainly not a POI.

In our case of identifying persons of interest who may have committed the fraud in Enron fraud case, we may want to prefer a higher precision score as we wrongly do not want to identify someone as POI. It is Ok to let go of few real POIs rather than raise a false alarm by wrongly identifying a doubtful case as POI. As they say "a person is innocent unless proven guilty"!