Identify Fraud from Enron Email

[Intro to Machine Learning]

Aishwarya Venketeswaran

Table of Contents

Analyzing the dataset

In this project, I seek to build a machine learning algorithm that can identify Enron employees who might have committed fraud from the Enron financial and email dataset. Enron was an energy trading company that existed in 2000. By 2002, the company was bankrupt. There was a lot of fraud occurring in the company. The dataset of Enron contains real emails from employees. In this project, I will be using machine learning algorithms to use the patterns in emails to find people who might have committed fraud.

Machine learning can be used to accomplish the given task in the following way. Firstly, the data has to be explored and outliers have to be removed. Then, features have to be carefully and intelligently selected. After testing several algorithms, the algorithm the suits the needs of our application and has the best performance has to be chosen. Then, the well-chosen features have to be provided to the algorithm so that it can be trained. Finally, the algorithm can be used to make predictions about the dataset.

Data Exploration

Firstly, I analyzed some of the features in the dataset and determined their data types:

Salary - numerical
Job title - categorical
Timestamp - time series
Contents of emails - text
Number of emails sent by a give person - Numerical
To/From fields of emails – text

Then, I examined the data to determine some important features of the dataset such as the following:

- total number of data points = 146
- allocation across classes (POI/non-POI)
  - Number of POIs in the sample = 18
  - Number of non-POIs = 146-18 = 128
  - Number of POIs totally in the population = 35
- number of features per person = 21

Hence, it can be noted that all the POIs are not there in the dataset we are analyzing – we have only 18 of the 35 POIs. The unavailability of complete, accurate data poses a problem - it is difficult to understand the overall issue and analyze the dataset. While performing analysis, we might get incorrect results. Due to incomplete data, we may not able to properly train our model to produce the correct results.

Enron was involved in several schemes including:
- selling assets to shell companies at the end of each month, and buying them back at the beginning of the next month to hide accounting losses.

- causing electrical grid failures in California.
- a plan in collaboration with Blockbuster movies to stream movies over the internet.

People who occupied important positions during much of the time the fraud occurred:

- Jeffrey Skilling was the Enron's CEO.
- Kenneth Lay was the chairman of the Enron board of directors.
- Andrew Fastow was the chief financial officer.

Of these three people, Kenneth Lay took home the most money - $103559793

Missing data

In the dataset, NaN is used to denote an unfilled feature. Only 95 people have a known salary and only 111 people have a known email address.

A total of 14.4% of people have Nan for their total payments. Only 0% of the POIs have a NaN for their total payments. Because 0% of the POIs have Nan for their total payments, if a machine learning algorithm used total_payments as a feature, I would expect it to associate a NaN value with non-POIs.

Outlier Investigation

While the dataset has a couple of outliers, one outliers is obvious and significant. The name of the dictionary key of this data point is "TOTAL". When the machine learning algorithm is run, this data point should be taken out because it is an error caused by the spreadsheet.

Besides this major outlier, there are almost four more outliers including LAY KENNETH L and SKILLING JEFFREY K. Since these are valid data points, they should not be removed.

```
### Task 2: Remove outliers
my_dataset= {}
for i,j in zip(data_dict.keys(),data_dict.values()):
  if i!='TOTAL':
    my_dataset[i] = j
```

The data exploration phase allowed me to recognize important characteristics about the dataset such as size, percentage of POIs, etc. This information can be used to understand what kind of features can be used in the machine learning algorithm. Also, I was able to conclude the limitations of the dataset such as missing data and outliers.

Optimize Feature Selection

Create New Features

The rate at which persons of interest [POIs] send emails to other POIs is higher than the rate at which POIs send emails to the public. So, rather than simply considering the total number of emails (which does not present an accurate picture), I computed features to represent percentages of communications. Just considering the total number of messages is not helpful because some people have a habit of writing more emails than others. On the other hand, considering what percentage of their communications were with a POI tells us how closely a person is associated with persons of interest. Hence, I created three new features to represent information more clearly:

fraction_from_poi

A fraction between 0 and 1 representing what fraction of messages were from a poi out of all the messages.
Code snippet:

```
from_poi_to_this_person = data_point["from_poi_to_this_person"]
to_messages = data_point["to_messages"]
fraction_from_poi = computeFraction( from_poi_to_this_person,to_messages )
my_dataset[i]["fraction_from_poi"] = fraction_from_poi
```

fraction_to_poi

A fraction between 0 and 1 representing what fraction of messages were addressed to a poi out of all the messages.
Code snippet:

```
from_this_person_to_poi = data_point["from_this_person_to_poi"]
from_messages = data_point["from_messages"]
fraction_to_poi = computeFraction( from_this_person_to_poi,from_messages)
my_dataset[i]["fraction_to_poi"] = fraction_to_poi
```

fraction_all_poi

A fraction between 0 and 1 representing the extent of interactions of a poi and a person out of all the messages (considering the ‘from’ and ‘to’ fields).
Code snippet:

```
my_dataset[i]["fraction_all_poi"] = (fraction_to_poi*0.5) + (fraction_from_poi*0.5)
```

Intelligently Select Features

I tried several features before finalizing a final set of features. I used TfIdfVectorizer along with a focus on feature importance values [with DecisionTreeClassifier] to choose features.

The features I used in my algorithm finally are:

```
features_list = ['poi','expenses', 'other']
```

- POI was used because it is a required Boolean feature. Hence, it was included in the list.
- 'expenses' and 'other' were used because by a process of trying various combinations of features, examining their importance and eliminating features with low importance values, revealed that using these features provided high accuracy, precision and recall values. Additionally, the decision to choose these two features can be backed up logically. People with high values for 'expense' and 'other' could be POI because people who commit could engage in activities with abnormal expenses and miscellaneous money transactions [i.e. 'other'].

Other features were ignored due to one or more of the following reasons:
- it is noisy
- it caused overfitting
- it is strongly correlated with another feature that is already being used
- usage of too many features slows down the process.

TfIdfVectorizer can be used to select features efficiently.While performing feature selection, an important point has to be considered - when few features are used, there is high bias thereby causing high error on training set and oversimplification due to little attention paid to the data. One the other hand, carefully minimized SSE can lead to high variance which pays too much attention to the data and causes overfitting. This could lead to higher error on the test set than the training set because the algorithm is doing overfitting. When overfitting occurs, the accuracy on the training set is high but the accuracy on the test set is low.
Also, the graph of the number of features vs. the quality of the model follows the normal distribution. Hence, there a point [maximum of the parabola] where the number of features is optimal and the quality of the model is at its maximum.

The following code snippets demonstrate how I finally chose the features that I chose:

Firstly, I tried all the financial features -

```
features_list = ['poi','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']
```

After running the program, I found the following importance values. By investigating the importance values, it can be noted that the following features: director_fees, long_term_incentive, deferred_income, restricted_stock_deferred,loan_advances,deferral_payments are not important.

```
('exercised_stock_options', 0.24740486725663738)
('restricted_stock', 0.17837389380530974)
('salary', 0.13148704171934261)
('total_payments', 0.11891592920353983)
('expenses', 0.096341186319062355)
('other', 0.091422566371681488)
('bonus', 0.078042825537294455)
('total_stock_value', 0.058011689787132198)
('director_fees', 0.0)
('long_term_incentive', 0.0)
('deferred_income', 0.0)
('restricted_stock_deferred', 0.0)
('loan_advances', 0.0)
('deferral_payments', 0.0)
```

Then, I tried the following the email features:

```
features_list = ['poi','to_messages', 'from_poi_to_this_person',  'from_messages',
'from_this_person_to_poi',  'shared_receipt_with_poi']
```

After running the program, it can be concluded that all the features in this list have at least 10% importance. So, I am going to keep them in the next iteration.

```
('shared_receipt_with_poi', 0.31767905571992117)
('from_this_person_to_poi', 0.23673742240979229)
('from_messages', 0.2296674679487179)
('from_poi_to_this_person', 0.12547134238310706)
('to_messages', 0.090444711538461495)
```

Then, I combined the important features from both the categories by considering the above results and tried the following features:

```
features_list = ['poi','to_messages', 'from_poi_to_this_person', 'from_messages',
‘from_this_person_to_poi',  ‘shared_receipt_with_poi',’exercised_stock_options',
'restricted_stock','salary','total_payments', 'expenses', 'other','bonus', 'total_stock_value']
```

I got the following output from running the above list. It can be noted that total_stock_value, total_payments, from_messages,to_messages, and from_poi_to_this_person have low importance values. Hence, I can eliminate these features in the next iteration. I think logically, to_messages is a feature that must be important. Hence, I am going to continue to keep it in the feature list.

```
('from_this_person_to_poi', 0.2386750804505228)
('exercised_stock_options', 0.22600000000000017)
('other', 0.19335050568900133)
('salary', 0.1167538213998391)
('expenses', 0.070052292839903385)
('bonus', 0.064452433628318442)
('shared_receipt_with_poi', 0.059457964601769907)
('restricted_stock', 0.031257901390644785)
('total_stock_value', 0.0)
('total_payments', 0.0)
('from_messages', 0.0)
('from_poi_to_this_person', 0.0)
('to_messages', 0.0)
```

In the next iteration, only the important [i.e. features with importance>0] features from the previous iteration are going to be kept in the list.

```
features_list = ['poi', 'from_this_person_to_poi',  'shared_receipt_with_poi',
'exercised_stock_options','restricted_stock','salary', 'expenses', 'other', 'bonus','to_messages' ]
```

The features_importances list generated by the previous makes us conclude that salary, restricted_stock and shared_reciept_with_poi can be eliminated.

```
('other', 0.24129172917444439)
('bonus', 0.19831880651773121)
('expenses', 0.16861679861679865)
('exercised_stock_options', 0.16784274193548393)
('from_this_person_to_poi', 0.14050274657836476)
('to_messages', 0.083427177177177167)
('salary', 0.0)
('restricted_stock', 0.0)
('shared_receipt_with_poi', 0.0)
```

The list has been narrowed down significantly at this point and consists of only 6 actual features [excluding poi].

```
features_list = ['poi', 'from_this_person_to_poi', 'exercised_stock_options','expenses','’other',
'bonus','to_messages' ]
```

From the importance values, it can be concluded that to_messages can be removed at this point because it has an importance value<0.2 .

```
('exercised_stock_options', 0.26423667120341454)
('bonus', 0.26308699423665005)
('other', 0.23984484715449525)
('from_this_person_to_poi', 0.10900468272171245)
('expenses', 0.096948937711250585)
('to_messages', 0.026877866972477064)
```

After removing ‘to_messages’, this is the new features list -

```
features_list = ['poi', 'from_this_person_to_poi', 'exercised_stock_options','expenses',
'other','bonus' ]
```

Because 'from_this_person_to_poi' has a low value, this feature is going to be removed.

```
('expenses', 0.31489963512256097)
('exercised_stock_options', 0.27207185020655639)
('bonus', 0.26344038469090025)
('other', 0.1055009379663885)
('from_this_person_to_poi', 0.044087192013593866)
```

New feature list after removing 'from_this_person_to_poi' was run and the following output snippet shows the importance values.

```
features_list = ['poi', 'exercised_stock_options','expenses', 'other','bonus']
```

```
('expenses', 0.31487729325551977)
('bonus', 0.28996456930040648)
('other', 0.25383494212584623)
('exercised_stock_options', 0.14132319531822751)
```

After removing the feature with the least importance at this point, I ran the algorithm with the following features:

```
features_list = ['poi','expenses', 'other','bonus']
```

```
('expenses', 0.45246405251161115)
('other', 0.2909755967737897)
('bonus', 0.25656035071459921)
```

At this point, all the features are contributing significantly. However, in order to find a feature list with the minimum number of features, I tried eliminating ‘bonus’ –

```
features_list = ['poi','expenses', 'other']
```

It was noted that the accuracy, precision and recall values were higher after removing bonus. Hence, this is the feature list I used finally.

```
Accuracy: 0.79827       Precision: 0.44924       Recall: 0.48450
('other', 0.75985212626115917)
('expenses', 0.24014787373884086)
```

Features I selected are: 'other'(importance_value = 0.75985212626115917) and 'expenses'(importance_value = 0.24014787373884086). Also, ‘poi’ is a required feature.

Properly Scale Features

Decision tree is not an algorithm that would be affected by feature rescaling. Hence, feature rescaling was not performed. Decision trees use vertical and horizontal lines and hence, there is no trade off. Also, only algorithms in which two dimensions affect the outcome are affected by rescaling.

Algorithm

I attempted using different algorithms such as Naïve Bayes, Support Vector Machines and Decision Tree Classifier, and compared their performances. Finally, I chose the algorithm with the best performance.

Pick an algorithm

Naïve Bayes

Firstly, I tried applying Naïve Bayes. In general, Naïve Bayes algorithms apply Bayes formula by assuming that all the features are independent.

$$P(y \mid x_1, \ldots, x_n) = \frac{P(y)P(x_1, \ldots x_n \mid y)}{P(x_1, \ldots, x_n)}$$

In this case, I applied Gaussian Naïve Bayes which is a specific type of Naïve Bayes. In GaussianNB, the probability of each feature is Gaussian -

$$P(x_i \mid y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right)$$

Code Snippet:

```
from sklearn.naive_bayes import GaussianNB
clf = GaussianNB()
```

Output:

```
Got a divide by zero when trying out: GaussianNB()
Precision or recall may be undefined due to a lack of true positive predictions
Total time taken 2.60199999809 seconds.
```

The advantage of Naïve Bayes is that it is easy to implement and very efficient. However, its weakness is that it might break sometimes. It is also important to note that Naïve Bayes executes very quickly [unlike Support Vector Machines].

Support Vector Machines

Then, I tried applying SVM. Support Vector Machines is a supervised learning method. It can be used efficiently for high dimensional spaces. Hence, I tried applying it to this particular case. It is memory efficient but takes a lot of time.

Code Snippet:

```
from sklearn.svm import SVC
clf = SVC()
```

Output:

```
Got a divide by zero when trying out: SVC(C=1.0, cache_size=200,
class_weight=None, coef0=0.0, degree=3, gamma=0.0,   kernel='rbf', max_iter=-1,
probability=False, random_state=None,   shrinking=True, tol=0.001, verbose=False)
Precision or recall may be undefined due to a lack of true positive predictions
Total time taken 4.53500008583 seconds
```

The prediction times were higher compared to naïve Bayes [double the time taken by Naïve Bayes]. Additionally, it did not work in this particular case as shown in the output above.

Random Forest Classifier

Then, I tried using Random forest classifier. Random forest classifier uses many decision tree classifiers [that work on various subsets of the data] to find produce predictions. This algorithm also took a long time to compute – 14 seconds. In this case, the accuracy was 0.80473, the precision was 0.43236 and the recall was 0.23650.

Code Snippet:

```
from sklearn.ensemble import RandomForestClassifier
clf   = RandomForestClassifier()
```

Output:

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
            max_depth=None, max_features='auto', max_leaf_nodes=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators=10, n_jobs=1,
            oob_score=False, random_state=None, verbose=0,
            warm_start=False)
           Accuracy: 0.80473      Precision: 0.43236     Recall: 0.23650        F1: 0.30
575        F2: 0.26006
           Total predictions: 11000          True positives:  473   False positives:  6
21         False negatives: 1527  True negatives: 8379

Time taken to complete 14.3120000362
```

Decision Tree Classifier

Finally, I tried decision tree classifier. Decision tree classifier are non-parametric classifiers that construct trees to represent decisions at each juncture and finally allocate a classification. Because the tree might be too complex, decision tree classifiers could lead to overfitting.

Code Snippet:

```
from sklearn.tree import DecisionTreeClassifier
clf = DecisionTreeClassifier()
```

Output:

```
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
            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: 0.80100     Precision: 0.45599     Recall: 0.48950        F1: 0.47
215        F2: 0.48241
           Total predictions: 11000          True positives:  979   False positives: 11
68         False negatives: 1021  True negatives: 7832

Time taken to complete 1.88499999046
```

It must be noted that the accuracy is higher than the other algorithms. Additionally, the precision and recall values are also better. Total time taken was 1.91499996185 seconds. This algorithm executed faster than both Support Vector Machines and Naïve Bayes. The accuracy was close to 0.80100, precision was 0.45599 and the recall value was 0.48950.

Final table summarizing performance of various algorithms:

Algorithm	Accuracy	Precision	Recall	Total time (seconds)
Naïve Bayes	-	-	-	2.601
Support Vector Machines	-	-	-	4.535
RandomForestClassifier	**0.80473**	0.43236	0.23650	14.312
DecisionTreeClassifier	0.80100	**0.45599**	**0.48950**	**1.885**

Therefore, considering the performance metrics from three algorithms that I tested, Decision Tree Classifier performed the best. Hence, I am picking this algorithm.

Tune the algorithm

Tuning the parameters of an algorithm means choosing the particular parameters of the algorithm so that is works well for the dataset we are working with. Each algorithm provides a different set of parameters to work with. These parameters usually have default values. The algorithm works decently with all the default values. But, as demonstrated by this project (i.e. this section), tuning the parameters will allow us to have better performance.

If the algorithm is not tuned well there are many issues associated with it. Entropy is how the tree splits data. It is also a measure of impurity. We need to determine ways to minimize impurity. Additionally, it is important to consider the amount of information gain. We have manage to maximize the information gain and minimize impurity. In decision tree classifiers, it is important that we tune the parameters available to us to ensure that we can achieve our goals i.e. maximize the information gain and minimize impurity and to there is no overfitting.

I tuned the DecsisionTreeClassifier algorithm in the following method –

- Firstly, I tuned the max_depth parameter. This parameter represents that maximum number of depth that the tree can grow to. So, having a high number for max_depth can lead to overfitting. Each additional level contains double the number of nodes in the previous level leading to drastically increasing the memory usage. I tried different values for the max_depth such as 3, 7, 9, 12, 8, etc.

  Code snippet [template]:

  ```
  clf =  DecisionTreeClassifier(max_depth=n)
  ```

  This tables summarizes my observations:

max_depth	Accuracy	Precision	Recall
Default	0.80000	0.45396	0.49300
3	0.74182	0.17290	0.11100
7	0.80709	0.46741	0.43750
9	0.80227	0.45871	0.48600
12	0.80055	0.45488	0.48900
8	0.80700	0.47019	0.48500

  I finally found that max_depth=8 gives the best performance.

- Then, I tuned the max_features parameter. This parameter represents the maximum number of features that have to be considered when making a split. There are several options including sqrt, log2, auto, etc. For example, if “sqrt” option is chosen, max_features=sqrt(n_features). I tried several options as represented in the table below.
  Code snippet [template]:

  ```
  clf = DecisionTreeClassifier(max_features="x")
  ```

  This tables summarizes my observations:

max_features	Accuracy	Precision	Recall
Default	0.80000	0.45396	0.49300
sqrt	0.77400	0.38655	0.41400
log2	0.78182	0.40449	0.42350
auto	0.77600	0.39098	0.41600

I concluded that using None [i.e the default option] results in best performance.

- Another parameter that I tuned is random_state. Random_state represents the seed used by the random number generator. I tried applying various values for random_state such as 42, 49, 60, 44, 41, etc. as shown in the table.
  Code snippet [template]:

```
clf = DecisionTreeClassifier(random_state=n)
```

This tables summarizes my observations:

random_state	Accuracy	Precision	Recall
Default	0.80000	0.45396	0.49300
42	0.80309	0.46114	0.49250
49	0.79918	0.45178	0.48950
60	0.79827	0.44924	0.48450
44	0.79845	0.44965	0.48450
41	0.79991	0.45345	0.48950

I concluded that using random_state = 42 gives the best performance.

Finally, considering the various conclusions I made previously, I combined the values of the tuned parameters to obtain the final tuned classifier –

Code snippet:

```
clf = DecisionTreeClassifier(random_state=42,max_depth=8)
```

Output:

```
('other', 0.75985212626115917)
('expenses', 0.24014787373884086)

DecisionTreeClassifier (class_weight=None, criterion='gini',
max_depth=8,max_features=None, max_leaf_nodes=None,
min_samples_leaf=1,min_samples_split=2,
min_weight_fraction_leaf=0.0,random_state=42, splitter='best')

Accuracy: 0.80736     Precision: 0.47085    Recall: 0.48050
F1: 0.47562      F2: 0.47854
Total predictions: 11000   True positives:  961
False positives: 1080
False negatives: 1039 True negatives: 7920
```

It can be noted that the values of accuracy, precision and recall of the tuned algorithm is better than the original algorithm.

Features I selected are: 'other'(importance_value = 0.75985212626115917) and 'expenses'(importance_value = 0.24014787373884086). Also, ‘poi’ - a required feature is used.

Validation and Evaluation

Usage of Evaluation Metrics and Algorithm Performance

The three evaluation metrics that I used to evaluate the algorithm's performance are:

Accuracy

Accuracy is the number of the data points that are correctly identified divided by the total number of data points identified. Some problems associated with accuracy are that it may not be ideal for skewed classes, it may want to err on the side of guessing innocent and it may want to err on the side of guessing guilty.
My final algorithm has an accuracy of **0.80736**.

Precision

Precision is the probability of correctly identifying a data point as positive when the data point is identified as positive. Mathematically, precision is equal to the number of true positives divided by the number of positives (true and false). In our case, the precision rate represents the rate of a person being a POI actually when a person is identified as a POI by the algorithm.
My final algorithm has a precision of **0.47085**.

Recall

Recall rate is the probability of correctly identifying a data point as true when in fact the data point is in fact true. Mathematically, recall is equal to the number of true positives divided by the sum of true positives and false negatives. In our case, the recall rate represents the rate of correctly identifying a POI when a person is in reality a POI.
My final algorithm's recall value is **0.48050**.

Validation Strategy

Validation i.e. having a testing data allows to gives an estimate of performance on an independent dataset and serves as a check on overfitting. It is important because if the algorithm is not validated, we cannot check how well our algorithm is performing. If the algorithm is not validated, we cannot be sure that the algorithm is performing well. When cross validation is done by splitting the dataset into a training set and a testing set, there is a tradeoff happening. When a data point is added to the testing set, it means that that point is no longer available for training. One thing that can go wrong with validation is not properly splitting the dataset. If there is a small training set and a large testing set, the algorithm will not be trained well. On the other hand, if there is a large training set with a small testing set, the algorithm might be trained well but we will not have the opportunity to test it properly.

The validation strategy used was cross validation with sklearn.cross_validation. It allows us create a split of the data into two sets - one for training and another for testing. After training the algorithm with the training data, we use our algorithm to make predictions for the testing features. Then, to understand how well our algorithm is performing we compare the predictions (made by our algorithm) and the test labels (correct/expected results). Using this validation strategy helps in getting the minimum training time and minimum run time. However, the accuracy may not be the best possible accuracy.