Identify Fraud from Enron Dataset

By George Liu

Introduction

Once a gigantic corporation claiming to have revenues of 111 billion dollars, Enron's share price decreased from 90 dollars to just pennies in mid-2000 before it filed for bankruptcy in 2001. The Enron Scandal was a result of "creative accounting" and corporate fraud, and this fraud was linked to a small number of employees out of Enron's 20,000 staff. Given data, is it possible to detect fraud and identify "person of interest" for investigation? In this project, we will use the publicly available Enron dataset that contains financial and email data of 146 people to identify fraud using machine learning techniques.

The email data alone are about 2GB in size which includes thousands, if not tens of thousands or more emails even for a single person. The financial data contain more than 140 data points and over 20 features. As such, any human intervention to investigate manually would not be possible and machine learning is thus the best candidate to solve this problem. Aside from size consideration, the capability to use algorithms to comb through a huge amount of data, and to identify patterns is essential, since the patterns will ultimately be used to detect fraud. In particular, we will use classification algorithms to analyze financial data to identify person of interest (POI).

Data and Outliers

The dataset contains two parts: financial data with all the insider pay information and email data comprising emails of Enron employees. In this project, we will focus on the finance data. Below is a summary of the data:

- Number of data points: 146
- Number of features: 21
- Number of POI's: 18
- Number of non-POI's: 128
- Data structure: Python dictionaries

As we can see, the data is highly imbalanced in that the number of both classes vary greatly. This may be a challenge in the project and warrants more consideration, particularly for the cross-validation part.

Since the finance data are based on financial reports and many entries are naturally not available and labeled "NaN" when extracted into Python dictionaries, thus there are a lot of missing values in the data. The following is a list of NaN percentages for all the features in the dataset:

- 'bonus': '44%',
- 'deferral_payments': '73%',
- 'deferred_income': '66%',
- 'director_fees': '88%',
- 'exercised_stock_options': '30%',
- 'expenses': '35%',
- 'loan_advances': '97%',
- 'long_term_incentive': '55%',
- 'other': '36%',
- 'poi': '0%',
- 'restricted_stock': '25%',
- 'restricted_stock_deferred': '88%',
- 'salary': '35%',
- 'total_payments': '14%',
- 'total_stock_value': '14%'

In particular, if a data point has zero value for all features as NaN's, it won't contribute any information to the algorithm. Therefore, these data points are removed as an outlier.

Furthermore, the "TOTAL"' data point from financial report is not necessary in machine learning as it represents the sum of respective feature values for all insiders instead of an employee, and is thus removed as an outlier.

Feature Selection and Scaling

Features are not equal to information. In fact, more features can even result in lower model performance. As such, only a subset of the features is selected for model building in this project. As my initial try with PCA and SelectKBest provided sub-optimal results (neither one produces precision and recall values that are both greater than 0.3), I resort to human intuition, to use my subject matter expertise on the Enron Fraud case to help select features.

Before further processing, feature scaling is first conducted. Due to the fact that some of the algorithms I plan to use require non-negative values (e.g. feature selection with chi2), I decided to use the MinMaxScaler to scale the data into a positive range.

After thoroughly reading over the Wikipedia Enron article, I understood that it was the "short-term driven culture" that played an important role in the fraudulent behaviours of Enron employees – you want high stock price, I make the deals to add sales and profit to the books… This way, not only Wall Street is happy, the employees themselves can also reap hefty bonuses and stock options. As a result, fraud was rampant.

Based on this insight, I decided to use features that directly correspond to these factors, i.e., **"salary", "bonus", "total stock value"**. Further building on this thinking logic, I continued to create a feature that

was designed to amplify the 'short-term driven behavior' by implement a new variable that is of the form:

(total stock + bonus) / (total pay - bonus)

Which is essentially an employee's variable pay divided by the fixed pay – the more you get paid from variable compensation compared with the fixed pay, the higher the likelihood that you cook up the books to achieve this…I named this variable "**variable to fixed**".

After including this feature in my model, however, the performance becomes worse (see table below). This is probably due to the fact that some data points are missing total pay/total stock data and in order for the engineered feature to work, I have to substitute with arbitrary values. This kind of imputing may alter the pattern in the data, thus leading to performance deterioration. Consequently, I decided not to use the created feature.

Features Used	Accuracy	Precision	Recall	F1
['salary', 'bonus', 'total_stock_value']	0.84146	0.47589	0.30100	0.36876
['salary', 'bonus', 'total_stock_value', **'variable_to_fixed'**]	0.29154	0.17287	0.95250	0.29263
['salary', 'deferral_payments', 'total_payments', 'exercised_stock_options', 'bonus','restricted_stock', 'restricted_stock_deferred', 'total_stock_value', 'expenses', 'loan_advances', 'other', 'director_fees', 'deferred_income', 'long_term_incentive']	**0.85160**	**0.42653**	**0.32800**	**0.37083**
['salary', 'deferral_payments', 'total_payments', 'exercised_stock_options', 'bonus','restricted_stock', 'restricted_stock_deferred', 'total_stock_value', 'expenses', 'loan_advances', 'other', 'director_fees', 'deferred_income', 'long_term_incentive', **'variable_to_fixed'**]	0.85160	0.42643	0.32750	0.37048

Note: Model performance is tested using the provided tester.py script. Algorithm used is GaussianNB.

To further improve the model performance, automatic feature selection algorithms were also employed. Specifically, I used the univariate feature selection method SelectKBest with f_classif as the score function. Further more, the scores based on cross-validation for all possible feature number choices were calculated as shown below:

[mean: 0.22178, std: 0.13585, params: {'selectkbest__score_func': <function f_classif at 0x0000000008C154A8>, 'selectkbest__k': 1},

mean: 0.31894, std: 0.10561, params: {'selectkbest__score_func': <function f_classif at 0x0000000008C154A8>, 'selectkbest__k': 2},

mean: 0.31497, std: 0.13175, params: {'selectkbest__score_func': <function f_classif at 0x0000000008C154A8>, 'selectkbest__k': 3},

mean: 0.33073, std: 0.12074, params: {'selectkbest__score_func': <function f_classif at 0x0000000008C154A8>, 'selectkbest__k': 4},

mean: 0.30127, std: 0.13971, params: {'selectkbest__score_func': <function f_classif at 0x0000000008C154A8>, 'selectkbest__k': 5},

mean: 0.32214, std: 0.13395, params: {'selectkbest__score_func': <function f_classif at 0x0000000008C154A8>, 'selectkbest__k': 6},

mean: 0.30978, std: 0.12701, params: {'selectkbest__score_func': <function f_classif at 0x0000000008C154A8>, 'selectkbest__k': 7},

mean: 0.27514, std: 0.12297, params: {'selectkbest__score_func': <function f_classif at 0x0000000008C154A8>, 'selectkbest__k': 8},

mean: 0.28593, std: 0.12138, params: {'selectkbest__score_func': <function f_classif at 0x0000000008C154A8>, 'selectkbest__k': 9},

mean: 0.28125, std: 0.11681, params: {'selectkbest__score_func': <function f_classif at 0x0000000008C154A8>, 'selectkbest__k': 10},

mean: 0.29443, std: 0.12494, params: {'selectkbest__score_func': <function f_classif at 0x0000000008C154A8>, 'selectkbest__k': 11},

mean: 0.32823, std: 0.09530, params: {'selectkbest__score_func': <function f_classif at 0x0000000008C154A8>, 'selectkbest__k': 12},

mean: 0.29890, std: 0.06591, params: {'selectkbest__score_func': <function f_classif at 0x0000000008C154A8>, 'selectkbest__k': 13},

mean: 0.27966, std: 0.06677, params: {'selectkbest__score_func': <function f_classif at 0x0000000008C154A8>, 'selectkbest__k': 14}]

And the scores for all the features are:

[('bonus', 30.075694443555427),

('salary', 19.025227695471706),

('total_stock_value', 15.96401432946872),

('exercised_stock_options', 15.823481308433996),

('long_term_incentive', 11.364867050642424),

('deferred_income', 10.301967367552212),

('restricted_stock', 9.3637197414396827),

('total_payments', 8.6907982205238916),

('loan_advances', 6.4194276901454597),

('expenses', 4.9966617373568853),

('other', 3.7726750859659743),

('director_fees', 1.8736737620636614),

('restricted_stock_deferred', 0.82708554313644589),

('deferral_payments', 0.08904109736862395)]

Based on this result, four features were finally chosen. These features are:

['**salary', 'exercised_stock_options', 'bonus', 'total_stock_value**']

This is very interesting as it largely matches previous manual feature selection result, proving that human intuition based on subject matter expertise can be very useful in identifying the best features for machine learning.

Algorithm

Using the Machine Learning Map on Sklearn, I tried a variety of algorithms including LinearSVC, KNeighborsClassifier, SVC, RandomForestClassifier, and GaussianNB. Although some algorithms may provide very high precision or recall rate, it is GaussianNB that gives a balanced performance on both metrics, and therefore a higher f1 score. The following is a summary of different algorithms' performances (ranking by f1 score).

Algorithm Used	Accuracy	Precision	Recall	F1
GaussianNB	0.85080	0.42372	0.33050	0.37135
KNeighborsClassifier	0.81373	0.30615	0.31350	0.30978
RandomForestClassifier	0.85240	0.40901	0.24050	0.30290
LinearSVC	0.86687	0.50241	0.15650	0.23866
SVC (linear kernel)	0.86507	0.47115	0.09800	0.16225

Note: To reduce run time, k = 4 in SelectKBest was used. Results obtained with the tester.py script.

The above result also shows that the accuracy metric is not reliable compared with f1 score in scenarios like this project where data are imbalanced.

Parameter Tuning

Parameter tuning is very important as it provides an opportunity to adjust the model to specific situation, thus allows for optimal performance. Without proper parameter tuning, even a suitable model for a particular situation can result in a suboptimal performance. Since the GaussianNB algorithm doesn't provide parameter tuning, here we will cover an example with the KNeighborsClassifier model.

The KNeighborsClassifier algorithm includes a total of 8 parameters available for tuning. In this project, we will evaluate "n_neighbors", "weights", "algorithm" and "leaf_size". Below is the input for parameters in the pipeline:

- 'kneighborsclassifier__n_neighbors': [1, 5],
- 'kneighborsclassifier__weights': ['uniform'],
- 'kneighborsclassifier__algorithm': ['auto', 'ball_tree'],
- 'kneighborsclassifier__leaf_size': [1, 10],

And the following is the tuning result given by GridSearchCV:

[mean: 0.21693, std: 0.15386, params: {'kneighborsclassifier__weights': 'uniform', 'kneighborsclassifier__leaf_size': 1, 'kneighborsclassifier__algorithm': 'auto', 'kneighborsclassifier__n_neighbors': 1, 'selectkbest__k': 4, 'selectkbest__score_func': <function f_classif at 0x0000000008EE34A8>},

mean: 0.12987, std: 0.15362, params: {'kneighborsclassifier__weights': 'uniform', 'kneighborsclassifier__leaf_size': 1, 'kneighborsclassifier__algorithm': 'auto', 'kneighborsclassifier__n_neighbors': 5, 'selectkbest__k': 4, 'selectkbest__score_func': <function f_classif at 0x0000000008EE34A8>},

mean: 0.21693, std: 0.15386, params: {'kneighborsclassifier__weights': 'uniform', 'kneighborsclassifier__leaf_size': 10, 'kneighborsclassifier__algorithm': 'auto', 'kneighborsclassifier__n_neighbors': 1, 'selectkbest__k': 4, 'selectkbest__score_func': <function f_classif at 0x0000000008EE34A8>},

mean: 0.12987, std: 0.15362, params: {'kneighborsclassifier__weights': 'uniform', 'kneighborsclassifier__leaf_size': 10, 'kneighborsclassifier__algorithm': 'auto', 'kneighborsclassifier__n_neighbors': 5, 'selectkbest__k': 4, 'selectkbest__score_func': <function f_classif at 0x0000000008EE34A8>},

mean: 0.21693, std: 0.15386, params: {'kneighborsclassifier__weights': 'uniform', 'kneighborsclassifier__leaf_size': 1, 'kneighborsclassifier__algorithm': 'ball_tree', 'kneighborsclassifier__n_neighbors': 1, 'selectkbest__k': 4, 'selectkbest__score_func': <function f_classif at 0x0000000008EE34A8>},

mean: 0.12987, std: 0.15362, params: {'kneighborsclassifier__weights': 'uniform', 'kneighborsclassifier__leaf_size': 1, 'kneighborsclassifier__algorithm': 'ball_tree', 'kneighborsclassifier__n_neighbors': 5, 'selectkbest__k': 4, 'selectkbest__score_func': <function f_classif at 0x0000000008EE34A8>},

mean: 0.21693, std: 0.15386, params: {'kneighborsclassifier__weights': 'uniform', 'kneighborsclassifier__leaf_size': 10, 'kneighborsclassifier__algorithm': 'ball_tree', 'kneighborsclassifier__n_neighbors': 1, 'selectkbest__k': 4, 'selectkbest__score_func': <function f_classif at 0x0000000008EE34A8>},

mean: 0.12987, std: 0.15362, params: {'kneighborsclassifier__weights': 'uniform', 'kneighborsclassifier__leaf_size': 10, 'kneighborsclassifier__algorithm': 'ball_tree', 'kneighborsclassifier__n_neighbors': 5, 'selectkbest__k': 4, 'selectkbest__score_func': <function f_classif at 0x0000000008EE34A8>}]

The best parameters are:

{'kneighborsclassifier__weights': 'uniform', 'kneighborsclassifier__leaf_size': 1, 'kneighborsclassifier__algorithm': 'auto', 'kneighborsclassifier__n_neighbors': 1, 'selectkbest__k': 4, 'selectkbest__score_func': <function f_classif at 0x0000000008EE34A8>}

Model Validation and Performance

Once the model is available, we need to evaluate it in order to be able to predict its performance on unknown data. A traditional way is to split the data into training and testing sets, after fitting the model with the training set, then evaluate the model using the test set. However, this can give an overly optimistic result since the split of training and testing data can be in such a way that even though the model may not generalize well, a good performance result may still be returned.

To avoid this, I used both the training-testing way of evaluating and cross validation. Cross validation is first implemented as part of GridSearCV in the process of parameter tuning. Once the best parameters are chosen based on f1 score through StratifiedShuffleSplit cross-validation, the model is tested using the held out test data which are unseen by the model.

StratifiedShuffleSplit combines StratifiedKFold and ShuffleSplit to randomize test-set sampling and average the results. “Stratify” is important in this case, as the data are heavily imbalanced with a small number of POI class. In order to ensure precision and recall both provide meaningful results, we need to make sure classes are well represented in each sample. “ShuffleSplit” is necessary as we have a relatively small number of data points, so in order not to waste any data and to prevent overfitting, we need to reuse through this randomization process.

In particular, I looked at the precision, recall and f1 metrics – the precision rate is the proportion of real positive cases out of predicted positive cases, while the recall rate is the proportion of predicted positive cases out of given real positive cases, while f1 being the harmonic mean of both rates which is offers “the best of both worlds”.

For the final model the performance results are (based on evaluation with the test set with my own script):

	precision	**recall**	**f1-score**	**support**
Non-POI	0.96	0.96	0.96	27
POI	0.50	0.50	0.50	2
Avg / Total	0.93	0.93	0.93	29

In the context of this dataset, the precision rate refers to the percentage of real POI’s out of the predicted POI cases (50%), and the recall rate is the opposite – given the known real POI cases, what proportion does the model correctly identify (50%)?

A final note is that should one look at the train-test validation result, a suboptimal model can very likely be chosen since for some algorithms the rates for POI class is not available. Therefore, cross validation is of critical importance in building machine learning models.

In summary, we can get both precision and recall rates as good as random guess solely based on the Enron financial data. This definitely needs to and can be improved. One possible way is to use text learning on the email data included in this dataset and perform classification correspondingly. We can either combine the new email text features with existing financial ones or use them independently.

Reference:

1. Enron Scandal (Wikipedia)
2. Crime in the Suites
3. Red Flags in Enron's Reporting of Revenues and Key Financial Measures
4. Choosing the right estimator
5. Machine Learning Map
6. Applying PCA to test data for classification purposes

Integrity Statement:

I hereby confirm that this submission is my work. I have cited above the origins of any parts of the submission that were taken from Websites, books, forums, blog posts, GitHub repositories, etc.