P5: Identifying Fraud from the Enron Dataset

Author: William Stevens

Project Overview

In 2000, Enron was one of the largest companies in the United States. By 2002, it had collapsed into bankruptcy due to widespread corporate fraud. In the resulting Federal investigation, a significant amount of typically confidential information entered into the public record, including tens of thousands of emails and detailed financial data for top executives. In this project, you will play detective, and put your new skills to use by building a person of interest identifier based on financial and email data made public as a result of the Enron scandal. To assist you in your detective work, we've combined this data with a hand-generated list of persons of interest in the fraud case, which means individuals who were indicted, reached a settlement or plea deal with the government, or testified in exchange for prosecution immunity.

Overview and Outliers

The goal of this project was to utilize the financial and email data from Enron to build a predictive, analytic model that could identify whether an individual could be

considered a "person of interest" (POI). Since the dataset contained labeled data--culpable persons were already listed as POIs--the value of this model on the existing dataset is limited. Rather, the potential value such a model may provide is in application to other datasets from other companies, to potentially identify suspects worth investigating further (assuming fraud at other companies occurs similarly to this).

The dataset contains 146 records with 20 features (14 financial features, 6 email features, and 1 labeled feature (POI)). Of the 146 records, 18 were labeled as persons of interest. Through exploratory data analysis I was able to identify 3 data points for removal.

- Lockhart Eugene E → Only NaN Data
- The Travel Agency in the Park → Data Entry Error (no individual represented)
- TOTAL → Sum of all the other data points (included extreme outliers)

Dataset Remodeling

Dataset included two records that look awfully funny (‘Belfer Robert’ & ‘Bhatnagar Sanjay’). Glancing over the PDF ‘enron61702insiderpay.pdf’ I found that the data entered into final_project_dataset.pkl was shifted over one column to the right relative to the PDF. Total stock value was actually the deferred stock value column in the PDF. I remedied both of their datasets using ‘fix_records.py’ in order to assure our analysis was as thorough as possible.

Optimize Feature Selection/Engineering

Feature	**Score**	**Frequency in Data**
Total Stock Value	22.51	123
Exercised Stock Options	22.35	98

Bonus	20.79	79
Salary	18.29	92
Stock and Payments (Total)	17.39	105
Deferred Income	11.42	48
Long Term Incentive	9.92	63
Total Payments	9.28	121
Restricted Stock	8.83	108
Shared Receipt with POI	8.59	84
Loan Advances	7.18	2

Using scikit-learn's **SelectKBest** algorithm using ‘selection_and_tuning.py’, the top eleven features were selected. Surprisingly, loan advances only appears three times in the data yet made it to the top eleven features list. Moreover, only one e-mail feature (shared receipt with poi) made it to the list.

I then engineered and tested three **new** features using 'add_features.py' …

- stock_and_payments → Sum of total stock value and total payments of that employee
- to_poi_ratio → Proportion of e-mails sent to POIs
- from_poi_ratio → Proportion of e-mails received from POIs

The k-best approach is an automated **univariate** feature-selection algorithm, and thus when using it, I was concerned with the lack of email features in the resulting dataset. Consequently, the 'to_poi_ratio' & 'from_poi_ratio' features were constructed (however neither of the two scored high enough to make it to the final cut). In addition, a financial aggregate feature was engineered, 'stock_and_payments', in order to capture how much total wealth (both liquid & invested) an individual had at the energy company, producing a good indicator of overall authority. Unlike the email features, our financial feature scored relatively well, producing a value of 17.39.

Prior to training the machine learning algorithm classifiers, I scaled all features using a min-max scaler. This feature-scaling ensured that for the applicable classifiers, all of the features would be weighted *evenly*. Each feature was scaled using **MinMaxScaler,** scored using **SelectKBest**, and then run under *various classification algorithms*. This was all done in a **pipeline** as to avoid information leakage.

Pick & Tune Our Algorithm

I chose both **decision tree classifiers** and **logistic regression** as my two primary algorithms. Out of the two, logistic regression performed higher, which makes sense as logistic regression performs ideally in environments where it has to classify binary data, which in this case equates to either being a positive or a negative. For example, logistic regression is particularly useful in the medical field where tumor benign/malignancy prediction is based off of measurements and medical test results, and I sought to mirror this behavior in predicting whether or not someone was a person of interest based off of their previous financial and email behavior. Consequently, it provided us with some excellent results.

I chose to tune my algorithms both manually and automatically using **GridSearchCV**. This is where the project became particularly fascinating as the results for each algorithm changed drastically even when one parameter differed even slightly. Manual tuning included determining which parameters to add to each algorithm and the process of tediously adding and removing features for optimal performance.

GridSearchCV assisted us in this process by providing us with a convenient way to perform linear combinations for all of the different parameters and report the best results (which we can get using both clf.best_estimator_ and clf.best_params_).

Parameter tuning is crucial in machine learning because it **optimizes** our algorithm's performance on our given data set. To measure our algorithm's performance, we need to validate and evaluate our data for each different combination of our selected parameters. Unfortunately, algorithms tend to be overwhelmingly general in nature and not specifically tuned to any one particular data set. As a result, it becomes imperative to iteratively (and sometimes painstakingly) tune our algorithms until we eventually obtain a result that we are finally content with.

The process used to manually tune each algorithm is shown below. Each metric was the result of **50 iterations** and the **mean** was used to obtain the **final values.**

- All original features used **(no new features)**

Metric	**Logistic Regression**	**Decision Tree**
Recall	0.6	0.4
Precision	0.429	0.33
Accuracy	0.86	0.837

- All original features used **(all new features)**

Metric	**Logistic Regression**	**Decision Tree**
Recall	0.6	0.4
Precision	0.75	0.33
Accuracy	0.93	0.837

- Top 11 features selected from SelectKBest **(without new features)**

Metric	**Logistic Regression**	**Decision Tree**
Recall	0.4	0.44
Precision	0.25	0.356
Accuracy	0.791	0.849

- Top 11 features selected from SelectKBest **(new features added)**

Metric	**Logistic Regression**	**Decision Tree**
Recall	0.8	0.56
Precision	0.44	0.547
Accuracy	0.86	0.893

- Top 11 features selected from SelectKBest **(new features added except "stock_and_payments" → *Our Final Algorithm*)**

Metric	**Logistic Regression**	**Decision Tree**
Recall	0.8	0.52
Precision	0.571	0.493
Accuracy	0.91	0.879

Logistic regression was chosen as our final classifier algorithm with the following parameters obtained using GridSearchCV

- {tol : 0.1, C = 10^9, class_weight: balanced}

The final parameters for our **decision tree classifier** included

- {criterion: entropy, min_samples_split: 20}

One of the most interesting observations I discovered while manually tuning the algorithms was that when e-mail features were excluded from the final features, the recall rate, precision rate, and accuracy scores dropped **significantly** (0.4, 0.25, and 0.79 respectively for logistic regression). SelectKBest had no e-mail features included in the top features, yet their inclusion was necessary in order to achieve better results. The **recall rate** actually **doubled** to **0.8** when the ratio of POI e-mails sent/received to total emails sent/received was included in the data (concluding that POIs are more likely to interact with each other than non-POIs.)

Validation

Validation is crucial in machine learning because it allows us to find our optimal model complexity between the infamous bias and variance tradeoff via algorithmic **regularization**. During this painstaking process, high bias relates to underfitting the data while high variance relates to overfitting it. For an idea model, we want the best of both worlds, thus the importance of splitting our data into two independent training and testing sets. A common mistake in validation is over-fitting your data, where the trained model performs particularly well on the training set yet markedly worse on the cross-validation and test datasets.

I validated my analysis in an evaluate function located in 'selection_and_tuning.py'. Over 50 randomized trials, I used sklearn's **train_test_split** function to split 70% of the data into a training set and 30% into a testing set. I prematurely reported my recall, precision, and accuracy without performing any actual **cross-validation**. Interestingly, the best parameters were the same when using only train_test_split and when using train_test_split followed by a stratified shuffle split. The only difference was that the latter reported lower results, which makes sense. For the

sake of consistency and to see why cross-validation is so crucial, I decided to keep my original results above.

Using the 'best_estimator_' attribute from GridSearchCV, I cross-validated the data using 1000 folds StratifiedShuffleSplit ('test_classifier' attribute from 'tester.py'). **StratifiedShuffleSplit** returns *stratified* splits, meaning that the same (or very similar) percentage of POIs is included for each test set created as in the entire overall set. As a result, performing stratified splits on our data helped improve our validation significantly.

Evaluation

The three evaluation metrics I used were **recall, precision, and accuracy.** Recall that the results from the train_test_split for our final algorithm were as follows:

- Recall: 0.8
- Precision: 0.571
- Accuracy: 0.91

Using the 'tester.py' file, which uses a StratifiedShuffleSplit, the results were as follows:

- Recall: 0.676
- Precision: 0.436
- Accuracy: 0.829

The results from StratifiedShuffleSplit were lower than using just the train_test_split. The reasoning behind this is that using stratified splits is more accurate than just splitting the data set, giving us an unsurprising reality check on our inflated results.

While 'tester.py' reported worse results than my own evaluation, the most important metric, recall, is still relatively high. Precision captures the ratio of true positives to the records that are actually POIs, essentially describing how often or not

false alarms are raised. Recall captures the ratio of true positives to the records flagged as POIs, which describes sensitivity. Due to the unbalanced nature of the dataset (too few POIs - 18/143 in total), accuracy is NOT a good metric (i.e. if non POIs had been predicted for all records, an accuracy score of 87.41% would have been achieved - but our recall rate & precision rate would be 0).

Undoubtedly, when attempting to find people that are guilty of fraud, having a good recall and a low precision is ideal. This means that we are able to identify a POI a large amount of the time but only sometimes flag non-POIs as guilty. If we were to have a high precision and a bad recall, then we would only flag people that are most certainly guilty and avoid flagging innocent people at all costs.

Conclusion

While fascinating and engaging, I found this project to be quite time consuming and challenging at points (the sparse nature of the dataset alone - only 18 POIs total - didn't help). If time weren't a factor, I would love to measure the performance of other algorithms against my dataset in order to see if I could score higher. Overall, I learned quite a bit from this exercise and would without hesitation recommend it to anyone trying to break into the field of machine learning algorithms.