Identifying persons involved in Enron fraud

9th May 2020

1 Introduction

Our goal was to use machine learning methods to identify persons actively involved in Enron scandal – the huge accounting fraud which led Enron, the American energy company, to bankruptcy in 2002. We used a dataset combining financial data and emails of top executives of the company. These data were made public during the federal investigation.

The original dataset contains 146 observations, out of which 18 is identified as persons of interest (POI). After checking basic descriptive statistics, we found out that there is an obvious outlier caused by inserting "TOTAL" value from financial table. After removal of this observation, our dataset still contains outliers with very high salaries or bonuses – data points representing Jeffrey Skilling, Kenneth Lay, or John Lavorato. We decided to keep these in the sample (robustness check was performed by rerunning our model without these observations), because these outliers are most likely not caused by error. Moreover, in our analysis, we don't run regressions, which could be very sensitive to leverage points.

The dataset suffers from missing values. Financial data are often not available. After selecting features we wanted to use in our model, we removed all observations with missing values and we were left with 61 observations, out of which 14 were identified as POI. Later, we reran the model with full dataset and missing values replaced by mean values; however, we obtained very poor results.

2 Selection of features

To keep the project simple, we decided to select only features that we suspect to possibly play a role in identifying persons of interest. We created three new features related to emails: share of emails sent to POI (number of all sent emails used as a weight), share of emails received from POI (number of all received emails used as a weight), share of emails received, where a POI was among recipients (number of all received emails used as a weight). From financial features, we decided to work with salary, bonus, and also newly created feature bonus/salary, since we suspect POIs to get unusually high bonuses.

In total, we work with 6 features. We didn't scale our features, because scaling doesn't influence results of Random Forest Classifier, which is classifier of our choice. The most important features for our best Random Forest Classifier were *share of emails sent to POIs* with weight 0.22, and *share of received emails shared with POI* with importance 0.19. On the other hand, *salary* had the lowest importance of 0.13.

We played with some other features, but their inclusion didn't improve our results. Alternatively, it could have been better to include all features and automatically select the best ones using function SelectKBest. One could also use principal components to keep most of the variation in the data. It is not a good idea to keep all available features in the model, because we would end up having very low number of degrees of freedom, because our sample is quite small.

3 Algorithm

We used python package *sklearn* to run classifiers. At first, we ran simple Gaussian Naive Bayes classifier, without any priors specified. The results were very poor. Therefore, we employed Decision Tree Classifier and Random Forest Classifier, with the latter being the best in terms of score, precision and recall. Later we also used grid search to play a bit with different parameters of the Random Forest Classifier. To train our classifier, we decided to use 90% of our dataset, with 10% remained to testing. Samples were created randomly with keeping the share of POIs same in both groups. We believe Random Forrest Classifier is very useful classifier in our scenario, when the final sample has only 61 observations. Bootstrapping involved in this classifier can lead us to more robust results.

4 Tuning of parameters

It is important to tune parameters of classifiers to achieve best results. With our Random Forest Classifier, we set the number of trees in the forest (parameter *n_estimators*) to 1000, since our sample is relatively small and the code runs very fast. We used entropy criterion. A grid search was used to find the best values for parameters *min_samples_split*, which sets minimum number of samples required to split an internal node, and *max_features*, which controls how many features are looked at when trying to find the best split. The best performing combination is *min_samples_split* set to 2, and *max_features* set to 1.

5 Validation

We should always validate our classifier carefully to make sure that the good performance is not limited to one specific subsample chosen when splitting the data. Therefore, we created 30 randomly split samples, each with 30 % of data points dedicated to the testing. Our classifier was then tested on these 30 samples, with both *precision* and *recall* ranging between 0.57 and 0.68.

6 Evaluation metrics

To compare our classifiers, as well as to validate the chosen best one, we used *precision* and *recall* metrics. *Precission* tells us what share of observations that our algorithm tag as persons of interest are actual persons of interest. *Recall* tells us what share of actual persons of interest our algorithm identifies correctly. These metrics represent often a trade-off, because with an algorithm that never misses a person of interest, it is hard to achieve that it doesn't flag innocent people. With both our metrics being roughly 0.6 (precision typically a bit higher than recall), we have quite a balanced algorithm.