P5: Identify Fraud from Enron Email

Alex Bugrimenko

Section 1. Project Overview

Objective

Enron Corporation was an American energy, commodities, and services company. Before its bankruptcy on December 2, 2001, Enron employed approximately 20,000 staff and was one of the world's major companies, with claimed revenues of nearly $111 billion during 2000. At the end of 2001, it was revealed that it's reported financial condition was sustained by an institutionalized, systematic, and creatively planned accounting fraud, known since as the *Enron scandal*. Enron has since become a well-known example of willful corporate fraud and corruption.

There were 29 Enron executives and board members named as defendants in a federal lawsuit and proceeds totaling $1.1 billion that plaintiffs lawyers say each made by selling Enron stock between October 1998 and November 2001. The biggest names among them are:

- Kenneth Lay, Chairman and chief executive: Sold 1.8 million shares for $101 million
- Jeffrey Skilling, former chief executive: Sold 1.1 million shares for $66.9 million.
- Andrew Fastow, former chief financial officer: Sold 561,423 shares for $30.4 million.

As a result of FBI investigation and consequent federal lawsuit, a lot of otherwise restricted information, including intercompany communications in a form of emails became publicly available, which makes an interesting case for applying machine learning algorithms.

The goal of this project is to detect fraud by analyzing available information about Enron employee compensation and communications between them. In particular, we are interested to find patterns in available data and identify people responsible for the fraud. Some of them are well known persons of interest (POI) – those that were found guilty in a federal lawsuit – and this is important information that could be used for supervised learning algorithms and to validate results of our models.

It seems that every single action of Enron executives was not illegal by itself, they planned and executed this accounting fraud very carefully, and therefore it is really difficult to detect any wrong doing manually. It also makes this task a perfect candidate for machine learning, where algorithms can classify large amounts of data, find correlation between different data elements and discover patterns that could be used to automatically detect any similar patterns or deviation from them.

In this project my goal would be to build a model to identify a person of interest (POI). The result of my model is going to be discrete (POI flag is a yes/no flag) therefore I'm going to use supervised classification algorithms with discrete output, such as Naïve Bayes, Support Vector Machine and Decision Tree.

Dataset description

There are two major data sets available: finances and email corpus.

> The code used for data exploration is in the DataExploration.py file.

"Payments to Insiders" (Exhibit 3b.2)

Dataset "Finances" is based on the data from pdf document (**enron61702insiderpay.pdf**), however it is not limited to the data presented in the doc, and it was enriched from other sources, including some basic statistics from email corpus. It contains information about salary, bonus and other types of compensations for 146 Enron employees. For each name in the list we have the following information available:

Financial data (in USD):

- salary
- deferral_payments
- total_payments
- loan_advances
- bonus
- restricted_stock_deferred
- total_stock_value
- long_term_incentive
- exercised_stock_options
- other
- shared_receipt_with_poi
- deferred_income
- expenses
- restricted_stock
- director_fees

Email address and basic stats about exchanged emails:

- email_address
- to_messages
- from_messages
- from_poi_to_this_person
- from_this_person_to_poi

A flag indicating if this person is a known Person of Interest (POI):

- poi

There are 18 people marked as a POI: HANNON KEVIN P, COLWELL WESLEY, RIEKER PAULA H, KOPPER MICHAEL J, SHELBY REX, DELAINEY DAVID W, LAY KENNETH L, BOWEN JR RAYMOND M, BELDEN TIMOTHY N, FASTOW ANDREW S, CALGER CHRISTOPHER F, RICE KENNETH D, SKILLING JEFFREY K, YEAGER F SCOTT, HIRKO JOSEPH, KOENIG MARK E, CAUSEY RICHARD A, GLISAN JR BEN F.

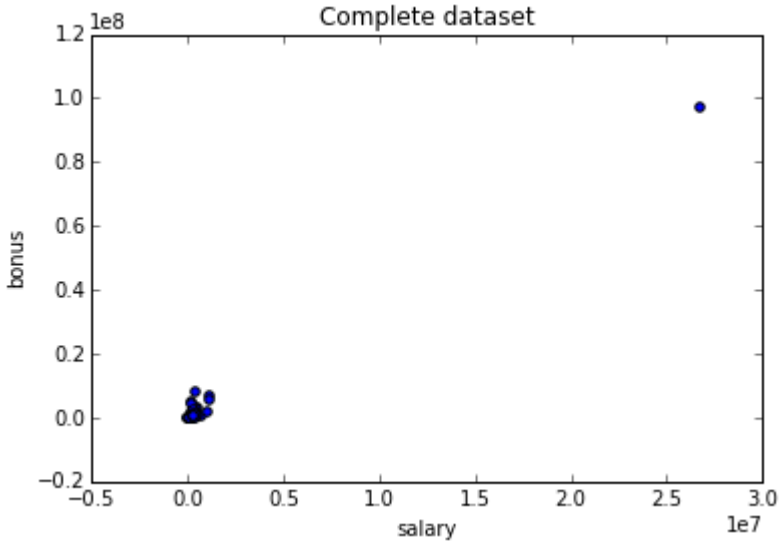
Among them, the CEO, Chairman and CFO - those, who took the largest amount of money (attribute "total_payments"):

- LAY KENNETH - $ 103,559,793
- SKILLING JEFFREY - $ 8,682,716
- FASTOW ANDREW - $ 2,424,083

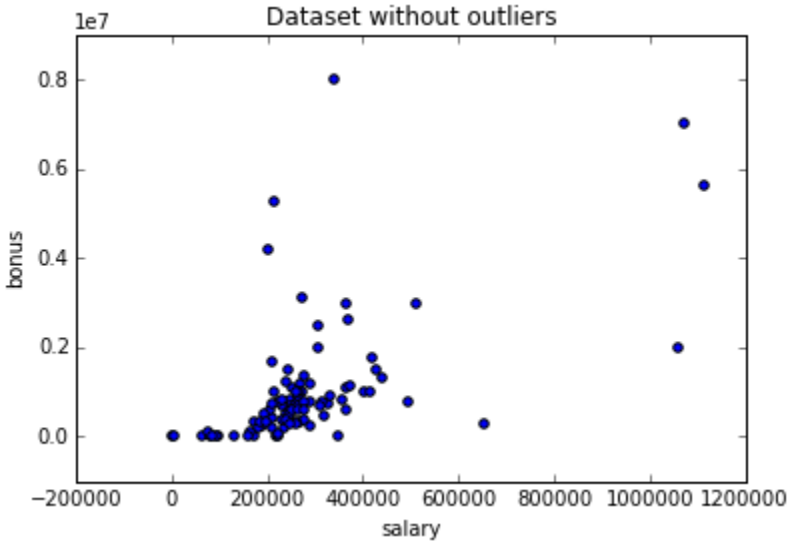
Not all listed employees have all attributes populated. For instance, there are about 14.38% of rows where total_payments is not populated. Missing data elements will be replaced with zeros, which makes sense for this particular application – if there is no data for total_payment for a particular person, it is likely because this person did not receive any. There are 16.67% of people reported with missing information about total payments, salary, bonus and stock value – almost no data is available for them.

Records with all zeros or empty values will be completely eliminated from the analysis (filtering is done in the *feature_format* function).

Other issue that has been found is import related. The "total" line from the original data was imported as one of the data elements. It clearly shown as an outlier in the following plot:



The picture is completely different as soon as we eliminated this erroneous data element (function *dataset_outlier_cleaner* in the DataExploration.py):



Enron Email Corpus

Enron email corpus consists of individual mailboxes of Enron employees: inbox, sent and deleted items. Each email is stored in individual file and all of them are located in the "maildir" folder, grouped by recipient. Each file in the corpus is an email with regular email characteristics, such as:

- From-name, from-email – senders name and email address
- To-name, to-email – recipients name and email address
- Subject – free form text
- Email text – free form text

Enron email corpus used mostly as a secondary source of data. It was preprocessed, analyzed and included into "Finances" dataset in a form of total number of emails from/to each person to/from a POI (features listed above in the section "Email address and basic stats about exchanged emails").

Section 2. Features Exploration

Data Validation

The basic statistics for each available feature provided in a table below (function *features_describe* in the DataExploration.py).

Feature	min	mean	max	% na	% zeros
bonus	70,000.00	671,335.30	8,000,000.00	44%	0%
deferral_payments	-102,500.0	220,557.9	6,426,990.0	73%	0%
deferred_income	-3,504,386.00	-192,347.52	-833.00	66%	0%
director_fees	3,285.00	9,911.49	137,864.00	88%	0%
exercised_stock_options	3,285.00	2,061,486.10	34,348,384.00	30%	0%
expenses	148.00	35,131.37	228,763.00	35%	0%
email_address					
from_messages	12	361.08	14,368	40%	0%
from_this_person_to_poi	0	24.46	609	40%	13%
from_poi_to_this_person	0	38.49	528	40%	8%
loan_advances	400,000.00	578,793.10	81,525,000.00	97%	0%
long_term_incentive	69,223.00	334,633.99	5,145,434.00	55%	0%
other	2.00	295,210.02	10,359,729.00	36%	0%
poi	0	0.12	1	0%	87%
restricted_stock	-2,604,490.00	862,546.39	14,761,694.00	24%	0%
restricted_stock_deferred	-1,787,380.0	72,911.57	15,456,290.00	88%	0%
salary	477.00	184,16.10	1,111,258.00	35%	0%
shared_receipt_with_poi	2.00	697.77	5,521.00	40%	0%
to_messages	57	1,230.1	15,149	40%	0%
total_payments	148.00	2,243,477.42	103,559,793.00	14%	0%
total_stock_value	-44,093.00	2,889,718.12	49,110,078.00	13%	0%

For this research I'm going to replace with zeros all empty values (na values). I think this is proper way to handle missing data for this particular task. For example, "loan advances" has highest rate of not populated rows – 97% of rows do not have this value populated. And I think this is reasonable to say

that if "loan advances" is empty, it means that this particular person did not make any loan requests, which would be essentially the same as 0 value.

Nevertheless, in a search for the best candidates for my features, I'm trying to use the values that have low "% na" ratio. In other words, I want to focus on features that can help me differentiate objects from one another.

Machine Learning Algorithms

To build a POI classifier I can use supervised learning algorithms which produce discrete output. I'm going to start with Naïve Bayes, Decision Tree, AdaBoost and Random Forest algorithms. The last two algorithms are "ensemble methods" and I expect them to give me the best results.

I was also trying to use SVM (Support Vector Machines) algorithm, however, the size of the data set is too small for this algorithm and I kept getting exception "*Precision or recall may be undefined due to a lack of true positive predictions.*" Therefore I excluded SVM from my list of algorithms.

Results Validation

Running several different algorithms with different sets of data (features) does not do any good unless we know how to evaluate the results. There are several different metrics I'm going to use to find the best model:

- Accuracy – shows the ratio of correct predictions.
- Precision – measures how accurate positive predictions are.
- Recall – measures what fraction of the positives a model identified.
- F1 Score – combination of the precision and recall ratios; it is a harmonic mean of precision and recall.

The goal is to maximize all of them. And when the model is ready to use, it should perform well on a new data, data that it has never been trained with.

A common danger in training a model is *overfitting* – producing a model that performs well on the training data but generalizes poorly on new data. The other side of this is *underfitting* – producing a model that does not perform well even on the training data. So we have to find a perfect balance using limited set of data we have.

The most fundamental approach involves using different data to train the model and to test the model. And the simplest way to do this is to split the data set we have. Usually, 2/3 of the data used for training and remaining 1/3 of the data used for testing the model.

Though, ideally we want to use all available data for training and all available data for testing. Of course, we cannot do it in one run – a model tested on the same data it was trained on will perform exactly the same way. But on a small dataset we have so little data that every example is valuable for both – training and testing. Therefore I'm going to use *StratifiedShuffleSplit* method to validate my models. This method executes multiple runs (in my case, 1000) shifting training and test data on each run in such a way that it produces pretty much full coverage, where each data point most likely was used as a training point in one or more runs and it was also used as a test data in some other runs.

On the first stage, to validate how good my feature list is, I used listed above machine learning algorithms and ran each feature set through them, using default parameters for each algorithm. Results were validated using *StratifiedShuffleSplit* method with number of folds equal 1000.

> The code for all used algorithms is in the Classifiers.py file, to test my model I used method test_classifier in the tester.py.

Feature Scaling

Most features in the dataset are of the same nature and measured in the same units – US dollars. However, there are also basic statistics about number of emails. Therefore, I have mixed types of data in the dataset and in order to use all of them together, all features have to be scaled and presented in the same range of values – preferably in a 0 to 1 range.

Scaling features taking care of negative values as well (by moving them into 0-1 range), which allows me to use *SelectKBest* algorithm, described below.

For features scaling I used *MinMaxScaler* class from the sklearn.preprocess.

In order to make sure that scaling is done on each processing task, I'm using *Pipeline* class from the sklearn.pipeline. It allows me to define a sequence of tasks – scale features, fit classifier, test classifier – and make sure that all steps are always applied.

Automatically Discovered Features

There are 20 features available in the data set, however I think not all of them are equally valuable for the task. To define which features are more important, I'm going to use *SelectKBest* function. It can analyze all features supplied and grade each of them. Usually, features with relatively low grade can be omitted without losing accuracy of the classifier.

> The code is in the FeatureSelection.py file, function kbest_test().

I included in the analysis all of the features except of

- "email_address" – it has no value for classification
- "from_this_person_to_poi", "from_poi_to_this_person" – these two are very tricky to use, because they always have high values for POI and it is better to exclude them for now.

There are several functions can be used with *SelectKBest* algorithm. I need functions that focus on classification (not on regression), therefore I tried *chi2* and *f_classif*. Both of them gave me almost identical results.

The results of the analysis is a list of features, sorted by their corresponding score (importance):

- salary : 8.9038
- from_messages : 2.5183
- shared_receipt_with_poi : 1.7517
- deferred_income : 0.5491
- to_messages : 0.3496
- bonus : 0.2390
- director_fees : 0.2283
- deferral_payments : 0.2195
- total_payments : 0.1661

- expenses : 0.1588
- poi : 0.0779
- long_term_incentive : 0.0682
- other : 0.0313
- loan_advances : 0.0222
- exercised_stock_options : 0.0140
- restricted_stock : 0.0042
- restricted_stock_deferred : 0.000

This is a good starting point. The features scores are vary from 0 to 8.9. I think if I start with top 6 features, where the lowest feature score is only 0.239, that will give me good enough coverage.

> The code is in the Discovery.py file. To replicate the results set
> _IS_KBEST_FEATURES_ = True
> _IS_DEFAULT_PARAMS_ = True
> _IS_PARAMS_SEARCH_ = False

The results of using top 6 features in all my algorithms listed below. The features list:

'salary', 'from_messages', 'shared_receipt_with_poi', 'deferred_income', 'to_messages', 'bonus'

Algorithm	Total Predictions	Accuracy	Precision	Recall	F1 Score	Top 2 most Important Features
GaussianNB	13,000	0.8279	0.4158	0.2940	0.3445	
DecisionTree	13,000	0.7842	0.2586	0.2155	0.2351	bonus – 0.3362 salary – 0.2802
AdaBoost	13,000	0.7824	0.2502	0.2075	0.2268	bonus – 0.3362 salary – 0.2802
RandomForest	13,000	0.8293	0.3569	0.1365	0.1975	bonus – 0.3001 salary – 0.2527

The results are promising, but recall ratio is really low for all algorithms, which means that my models very likely will miss a lot of POIs.

Manually Selected Features

In order to improve my feature selection, I'm going to manually add and remove features one by one and retest the results. Choice of my actions on this stage is mostly based on intuition.

Based on my previous test, I can see that "bonus" feature was rank the highest among all algorithms, so it is probably wise to keep it.

The second highest rank was assigned to "salary", however when I excluded it I got slightly better results. The reasoning for my decision is that each employee should have a salary and most likely it is based on corporate rules, which makes me think that the amount of salary received by employee defined by her/his place in the company hierarchy and not necessarily can link this person to a fraud.

At the same time the size of "exercised stock option", for example, can be a sign of leverage management was creating to motivate this particular employee. And even more important, exercising that stock option in a right time, may show that this person likely had an "inside information" to make a decision to exercise options. So I added "exercised stock option" to the feature list.

After testing it for a while I ended up with only 3 features in my list:

'bonus', 'total_payments', 'exercised_stock_options'

The results of training on four algorithms using default parameters are listed below. Note, that "bonus" and "total_payments" features are relatively highly ranked by all three algorithms.

Algorithm	Total Predictions	Accuracy	Precision	Recall	F1 Score	Features Importance
GaussianNB	14,000	0.8499	0.4511	0.2330	0.3073	
DecisionTree	14,000	0.8007	0.3126	0.3295	0.3208	bonus – 0.4189 total_payments – 0.3903 stock_options – 0.1908
AdaBoost	14,000	0.8034	0.3210	0.3375	0.3290	bonus – 0.4046 total_payments – 0.3689 stock_options – 0.2265
RandomForest	14,000	0.8636	0.5454	0.2705	0.3616	bonus – 0.3211 total_payments – 0.3603 stock_options – 0.3186

AdaBoost seems to be the best, because it has highest Precision and Recall numbers and both of them are above 0.3. RandomForest has very good precision of 0.56, but relatively low recall of 0.298, which means that if it gives a POI flag to someone, that person is very likely to be a POI, however, there are probably a lot of POIs that will not be flagged.

> The code is in the Discovery.py file. To replicate the results set
> _IS_KBEST_FEATURES_ = False
> _IS_DEFAULT_PARAMS_ = True
> _IS_INCLUDE_ADDITIONAL_FEATURE_= False
> _IS_PARAMS_SEARCH_ = False

New Features

One of the reasons why I did not include many other attributes into my feature list is because we have relatively high rate of missing values for many of them. To overcome it, I'm going to sum up all payment related amounts into new feature "total_compensation". Below are the results of training using this new feature (I also included results from the previous test for easy comparison).

> New feature definition is in the DataExploration.py file, method features_add. It is defained as following:

```
my_dataset[k]["total_compensation"] = \
    getFloatOrZero(my_dataset[k]["bonus"]) + \
    getFloatOrZero(my_dataset[k]["director_fees"]) + \
    getFloatOrZero(my_dataset[k]["exercised_stock_options"]) + \
    getFloatOrZero(my_dataset[k]["expenses"]) + \
    getFloatOrZero(my_dataset[k]["loan_advances"]) + \
    getFloatOrZero(my_dataset[k]["long_term_incentive"]) + \
    getFloatOrZero(my_dataset[k]["other"]) + \
    getFloatOrZero(my_dataset[k]["restricted_stock"]) + \
```

```
    getFloatOrZero(my_dataset[k]["salary"]) + \
    getFloatOrZero(my_dataset[k]["total_payments"]) + \
    getFloatOrZero(my_dataset[k]["total_stock_value"]) + \
    getFloatOrZero(my_dataset[k]["deferral_payments"]) + \
    getFloatOrZero(my_dataset[k]["restricted_stock_deferred"])
```

Algorithm	Total Predictions	Accuracy	Precision	Recall	F1 Score	Features Importance
Features: ['poi', 'bonus', 'total_payments', 'exercised_stock_options']						
GaussianNB	14,000	0.8499	0.4511	0.2330	0.3073	
DecisionTree	14,000	0.8007	0.3126	0.3295	0.3208	bonus – 0.4189 total_payments – 0.3903 stock_options – 0.1908
AdaBoost	14,000	0.8034	0.3210	0.3375	0.3290	bonus – 0.4046 total_payments – 0.3689 stock_options – 0.2265
RandomForest	14,000	0.8636	0.5454	0.2705	0.3616	bonus – 0.3211 total_payments – 0.3603 stock_options – 0.3186
Features: ['poi', 'bonus', 'total_payments', 'exercised_stock_options', 'total_compensation']						
GaussianNB	15,000	0.8710	0.5309	0.2795	0.3662	
DecisionTree	15,000	0.8325	0.3730	0.3765	0.3747	bonus – 0.4732 total_payments – 0.1678 stock_options – 0.2049 compensation – 0.1542
AdaBoost	15,000	0.8310	0.3682	0.3735	0.3708	bonus – 0.4732 total_payments – 0.1678 stock_options – 0.2049 compensation – 0.1542
RandomForest	15,000	0.8617	0.4631	0.2355	0.3122	bonus – 0.2321 total_payments – 0.2265 stock_options – 0.2997 compensation – 0.2417

Interestingly enough the new feature set produced better results for almost all algorithms, except the RandomForest. The results for RandomForest were only slightly worse than in the first test, therefore I'm going to keep this new feature. Another interesting observation is RandomForest algorithm used all features more evenly, it assigned similar importance ratios to all of them, while other algorithms treated "bonus" as most important feature.

```
The code is in the Discovery.py file. To replicate the results set
_IS_KBEST_FEATURES_ = False
_IS_DEFAULT_PARAMS_ = True
_IS_INCLUDE_ADDITIONAL_FEATURE_= True
_IS_PARAMS_SEARCH_ = False
```

3. Machine Learning Algorithms Parameters Optimization

So far I've got reasonably good results, but perhaps I can improve it by optimizing parameters of machine learning algorithms.

Parameters optimization allows to reach the best possible balance between model's ability to generalize and successfully recognize the pattern. During this process I do not want to over-regulate the model, which could lead to overfitting. At the same time, I want to make sure that my model is working well not only on training data but on test data as well (not underfitted). I'm using cross validation *StratifiedShuffleSplit* to validate the results and after each parameter change I'm comparing models accuracy, precision, recall and F1 scores with the previous run to make sure I'm moving in a right direction. My final goal is to maximize all four scores by constantly cross validating results.

Some algorithms have very few parameters, while others have many different parameters that could be used. There are some basic rules can be applied to set parameters, however it usually takes significant amount of time and effort to manually test all possible combinations of parameters. Therefore, I'm going to use *GridSearch* library, which allows me to automatically discover the best set of parameters.

> The GridSearch code is in the Classifiers.py file and it could be initiated by setting _IS_PARAMS_SEARCH_ = True in the Discovery.py file.

There are no parameters could be optimized for Naïve Bayes algorithm, therefore there is not much I can do to improve the situation and the results are going to be exactly the same for this algorithm. For all other algorithms I'm going to use *GridSearch* results as a starting point for each classifier and after that adjust most important parameters manually to find the best value for my data set. For instance, for the Decision Tree algorithm I've got the following results from the *GridSearch*:

```
=== The best score: 0.8681  --- classifier is ===
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=4,  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')
-- search time: 0.147 s
```

The score is great! Though the one parameters that sticks out is "*max_depth*" – it limits the maximum growth of the decision tree and GridSearch recommended to use value 4. When I tried it I got validation score that is significantly lower than were received with default values (default value for "max_depth" is None, meaning that there is no limitation for tree growth). So I started to increase it by one and the best results I got was the result with max_depth = 7. After that all my evaluation metrics (accuracy, precision and recall) started to get lower again, therefore I left value 7 as the best value in the final version of my Decision Tree Classifier.

> The code is in the Discovery.py file. In order to replicate the results set
> _IS_KBEST_FEATURES_ = False
> _IS_DEFAULT_PARAMS_ = False
> _IS_INCLUDE_ADDITIONAL_FEATURE_= True
> _IS_PARAMS_SEARCH_ = False

The same approach I used for all other classifiers and the best results I got are listed in the table below. For better comparison I listed validation results used with default parameters and optimized parameters for each algorithm.

Algorithm	Total Predictions	Accuracy	Precision	Recall	F1 Score
Default parameters					
GaussianNB	15,000	0.8710	0.5309	0.2795	0.3662
DecisionTree	15,000	0.8325	0.3730	0.3765	0.3747
AdaBoost	15,000	0.8310	0.3682	0.3735	0.3708
RandomForest	15,000	0.8617	0.4631	0.2355	0.3122
Optimized parameters					
GaussianNB	15,000	0.8710	0.5309	0.2795	0.3662
DecisionTree	15,000	0.8465	0.4186	0.3895	0.4035
AdaBoost	15,000	0.8446	0.4121	0.3880	0.3997
RandomForest	15,000	0.8669	0.5014	0.3530	0.4143

Based on this table my best algorithms are Decision Tree and Random Forest. They both have good numbers, but Random Forest seems to have precision score higher roughly by .1 and the price paid for that is lower recall score. The recall score is lower only by .036, which reflects in the F1 score gain of .011 compare to the Decision Tree results. Therefore, I think Random Forest is the best algorithm I came up with.

The Best POI Classifier

The best POI classifier is Random Forest algorithm with the parameters described below.

> RandomForestClassifier(
> bootstrap=True, class_weight=None, criterion='gini', max_depth=9, max_features=None,
> max_leaf_nodes=None, min_samples_leaf=1, min_samples_split=2,
> min_weight_fraction_leaf=0.0, n_estimators=25, n_jobs=1, oob_score=False,
> random_state=42, verbose=0, warm_start=False)

The best results achieved using the features listed below. The number next to each feature shows feature importance for the Random Forest algorithm.

> 'bonus': 0.4906
> 'total_payments': 0.1664
> 'exercised_stock_options': 0.1945
> 'total_compensation': 0.1485

The scores for the Random Forest classifier:

> Accuracy: 0.86693
> Precision: 0.50142
> Recall: 0.35300
> F1: 0.41432
> F2: 0.37521
> Total predictions: 15000

> True positives: 706
> False positives: 702
> False negatives: 1294
> True negatives: 12298

The best POI classifier I came up with has relatively high F1 score of 0.41, which gives me pretty good confidence in POI identification. My POI identifier has precision score of .5 which is slightly higher than recall score of .35. That means that every time I receive a POI signal from classifier, it is likely that this person is a POI. However my confidence that I can identify all POIs is a bit lower, since my algorithm has recall score lower than precision score.

Conclusions

In this project I have developed a Person of Interest Identifier model. I analyzed the dataset, selected the best features suitable for the task, constructed new feature, used four supervised machine learning algorithms to classify the data, evaluated and optimized their performance and selected the best one.

The most important thing I've learned is that the results of selection and parameter optimization algorithms should not be used as a final choice; their results should be used only as a starting point for analysis and in order to get a good performance you have to spend extra time manually testing, tuning and adjusting algorithms and parameters.

References

- https://en.wikipedia.org/wiki/Enron
- "Data Science from Scratch" by Joel Grus
- http://scikit-learn.org/stable/modules/feature_selection.html
- http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html
- http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html
- http://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html
- http://scikit-learn.org/stable/modules/svm.html
- http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html
- http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html
- http://scikit-learn.org/stable/modules/generated/sklearn.grid_search.GridSearchCV.html#sklearn.grid_search.GridSearchCV