Enron Fraud Detection with Machine Learning

The Goal

What if there was a fraud-prevention system in place during Enron's run as one of America's largest companies? If there was, maybe the millions of dollars illegal obtained by their employees and ultimately their 2002 collapse into bankruptcy, could have been prevented. Essentially that is the goal of this project - to create and train a machine learning algorithm that can identify persons of interest (POI) that are involved in unethical business practices.

By using the financial information and email corpus in the dataset that was given, there's a good chance I can create a program to accurately identify POIs. Once this is accomplished, it is my hope that I can translate the same approach and apply it to modern day companies as a type of internal fraud-prevention system.

Exploring the Data

The dataset contains both financial and email information on Enron employees and persons involved with the company. Here is a summary of the dataset:

- # of People in the Dataset: 146
- # of Total Features: 21
  # of Financial Features: 14
  # of Email Features: 6
  # of Labels: 1
- # of Persons of Interest (a priori): 18
- # People with Blank "Total Payments" = 21
- # POIs with Blank "Total Payments" = 0

The most interesting observation I made had to do with the empty values of "Total Payments" for POIs vs Non-POIs. Because POIs have a 0% rate of representation out of all instances, having a value of zero for this feature could become a clue that a person is a POI.

Running Preliminary Tests Using the Default Information

Before making any changes to the data, I wanted to capture a few baseline measurements to gauge any improvements that occur after tuning my classifiers. The following are the baseline results of four different classifiers:

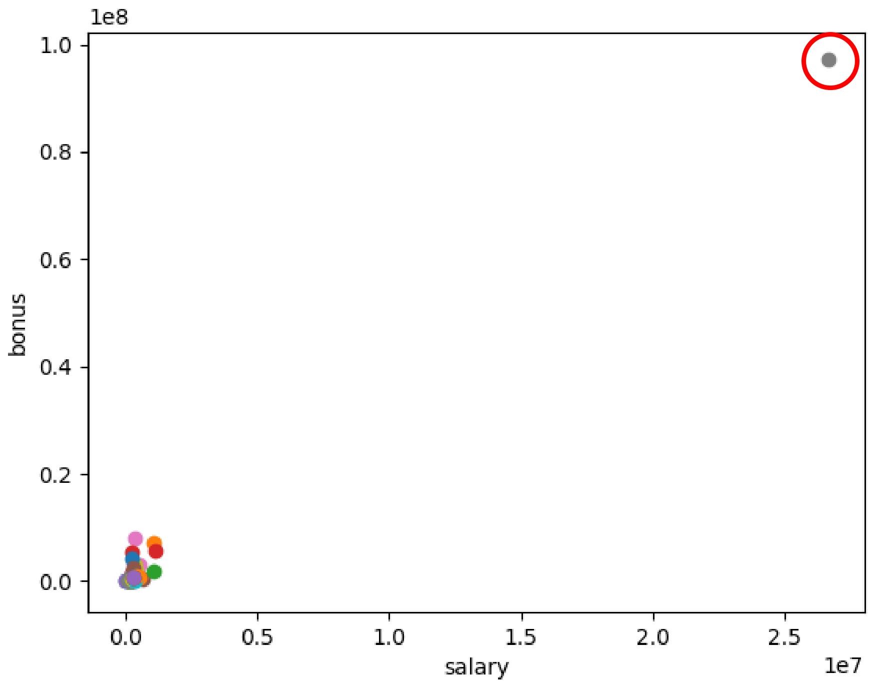
	Accuracy	**Precision**	**Recall**	**F1**
GaussianNB	0.376	0.156	0.834	0.263
DecisionTreeClassifier	0.790	0.204	0.200	0.202
RandomForestClassifier	0.850	0.311	0.103	0.155
AdaBoostClassifier	0.909	0.400	0.310	0.34

Dataset Cleaning

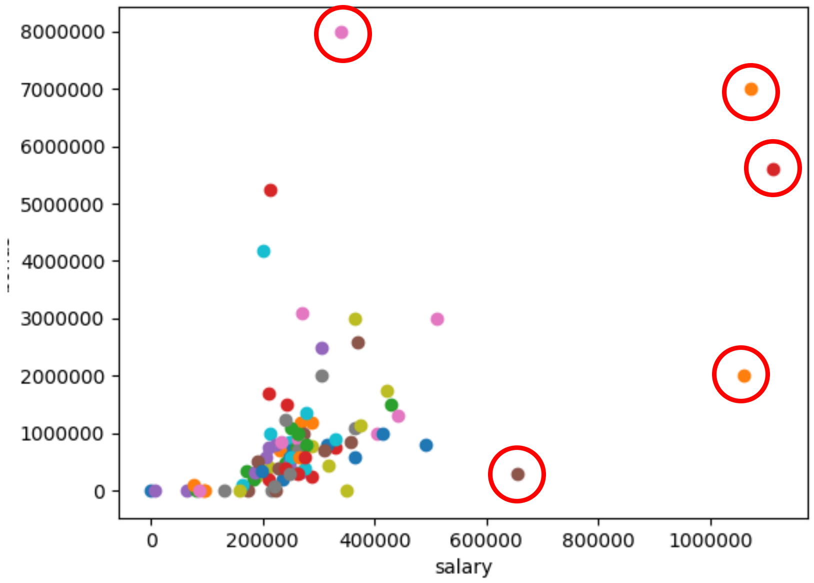
Cleaning your dataset is an important step in any machine learning project because it minimizes skewed calculations and inaccurate predictions. I decided to focus on identifying outliers and features that have blank values represented by 'NaN'.

Outliers

Using a scatterplot of the "salary" and "bonus" features, I was able to clearly spot an outlier that had a salary over $26 million and a bonus over $97 million.



After reviewing the financial information, the outlier from the plot above belonged to the row titled "TOTAL". This is a spreadsheet anomaly as the row's purpose is to sum all the values from each column. I decided to remove this outlier and after doing so, I ran the scatterplot again to see if any additional outliers would surface. Below are the results.



There still seemed to be four outliers remaining so I investigated the points and determined they belonged to: Kenneth Lay, Jeffrey Skilling, Mark Frevert, and John Lavorato, two of which are labeled as POIs, so for that reason I decided to leave these points in.

While reviewing the financial dataset, I also noticed two rows worth removing: “The Travel Agency in the Park” and “Eugene Lockhart”.

In summary, I removed a total of 3 outliers:

- “TOTAL”: The graph clearly showed this datapoint as an outlier and most likely came about as a spreadsheet anomaly.
- “THE TRAVEL AGENCY IN THE PARK”: This record is not actually a person.
- “LOCKHART EUGENE E”: This record contains all blank values, so it renders useless.

Replace ‘NaN’ Values With Zero

Now that we’ve eliminated a few outliers the next step is to replace all instances of ‘NaN’ with the numerical value of zero. This will help with future calculations. Here’s the summary of what I found:

# of ‘NaN’ Values Per Feature		
salary: 49	to_messages: 57	deferral_payments: 105
total_payments: 20	exercised_stock_options: 42	bonus: 62
restricted_stock: 34	shared_receipt_with_poi: 57	restricted_stock_deferred: 126
total_stock_value: 18	expenses: 49	loan_advances: 140
from_messages: 57	other: 52	from_this_person_to_poi: 57
poi: 0	director_fees: 127	deferred_income: 95
long_term_incentive: 78	email_address: 32	from_poi_to_this_person: 57

Creating New Features

When drawing conclusions about the relationship between features of a dataset, you'll sometimes get to a point where you wished additional features existed. These new features could help bolster a specific message you were trying to convey, or it could help you make a discovery. The great thing about machine learning is that these new features can be created simply by using what already exists. In my case, I decided to create two additional features: 'POI Interaction' and 'Total Money Earne d'.

POI Interaction

This new feature is simply the total number of messages 'to' and 'from' a person of interest divided by the total number of 'incoming' and 'outgoing' messages. Creating this new feature allows us to see what percentage of a person's email communication is with a POI, with the assumption being – the higher the percentage the more likely they are a POI.

Total Money Earned

My intuition was telling me to explore the idea of there being a high correlation between money earned and POIs, the only problem is that values for money earned were split amongst three different features: 'total_stock_value', 'exercised_stock_options', and 'salary'. Adding up these features provided me with a sum of each person's financial gain.

Selecting the Top Features

For optimal performance within my model and to avoid overfitting the data, I needed to select the minimum number of important features. It's important to remember that more features do not translate into more information. The objective is to pick the most valuable features that help us best understand what is going on within the data. Instead of randomly choosing features that I thought were good, I used SelectKBest to systematically choose my top features. What SelectKBest does is score each feature and ranks them on how much relevance or influence it has on the target variable. Here were my results:

'salary', 'total_payments', 'exercised_stock_options', 'bonus', 'total_stock_value', 'shared_receipt_with_poi', 'total_money_earned', 'deferred_income', 'restricted_stock', 'long_term_incentive'

To ensure that I made the correct decision on **which feature list to use** and **how many features** to include using SelectKBest, I needed to run some additional tests. My first test would determine which feature list I should use – the one including the features that I created or the original list. Using the test_classifier function to gauge the performance, here were my results:

	Accuracy	Precision	Recall	F1
Original Feature List	0.795	0.225	0.220	0.222
New Feature List	0.813	0.301	0.306	0.304

Because the results showed an improvement when I used the new feature list over the original feature list, I felt confident that I should use my engineered features in my final model.

My second test would determine the optimal number of features, k, that I should use for SelectKBest. Here were my results:

k value	Accuracy	Precision	Recall	F1
8	0.796	0.227	0.220	0.223
10	0.800	0.236	0.223	0.229
12	0.794	0.222	0.219	0.220

Based on the results, it looked like setting the k-value to 10 would be the optimal choice for our prediction model.

After all the testing, I ended up with a **top 10 features list** that was comprised of 10 features and 1 label. I will now go on to use this list within my preliminary testing and eventually in my prediction model.

Preliminary Testing

Now that I have cleaned and optimized data, I'm curious to see if there are any improvements in recall and precision compared to my original baseline tests. Using the same un-tuned classifiers as before, these were my results:

	Accuracy	Precision	Recall	F1
GaussianNB	0.844 ⬆	0.403 ⬆	0.360 ⬇	0.38 ⬆
DecisionTreeClassifier	0.813 ⬆	0.304 ⬆	0.309 ⬆	0.30 ⬆
RandomForestClassifier	0.860 ⬆	0.443 ⬆	0.185 ⬆	0.261 ⬆
AdaBoostClassifier	0.829 ⬇	0.308 ⬇	0.228 ⬇	0.262 ⬇

As you can, nearly all the metrics improved. This means I'm on the right path. The next step is to tune a classifier so that the precision and recall are both above a score of 0.30.

Choosing and Tuning a Classifier

In the end, I decided to use the Decision Tree classifier. Before I could complete my model, I needed to tune the parameters of my classifier for optimal results. Tuning is the process in which you adjust the different parameters of the class until it yields the best result. This tuning process is important because if done incorrectly, you could end up with false positives for your prediction model. Because our dataset contained such a low number of POIs compared to non-POIs, I used the StratifiedShuffleSplit function and GridSearchCV to help systematically determine what specific values to use for my parameters. The following steps were used for my Pipeline to determine optimal parameter values:

Scale Features Using MinMaxScaler ➡ Select Features Using SelectKBest ➡ Specific Classifier with Parameters

The parameters I chose to test for my Decision Tree classifier were:

'classify__min_samples_leaf': [1,2,3,4,5]

'classify__max_depth': [2,4,6,8,10]

'classify__criterion': ['gini', 'entropy']

'classify__min_samples_split': [2,4,6,8,10]

After running the function for my Decision Tree, I was given the following optimal parameters:

min_samples_leaf = 1

max_depth = 6

criterion='gini'

min_samples_split = 2

Validation

Validation is the process in which we test the performance of our model by running it against unseen data. We do this because a common mistake is overfitting our model and only testing it on the dataset it was trained on. When this happens, the model isn't truly learning to generalize but rather memorizing instead.

To test and validate my model, I will be using the Stratified Shuffle Split cross-validation function from tester.py. This method randomly shuffles the data into two sets – training and testing – and ensures that each split has a presentation of POI and Non-POI. This is important because our dataset is heavily imbalanced – 18 POI vs 128 Non-POI. If we don't use this method our model could end up making zero predictions and result in inconclusive scores.

Evaluation

Here were my average results:

	Accuracy	Precision	Recall	F1
DecisionTreeClassifier	0.819 ⬆	0.316 ⬆	0.307 ⬇	0.31 ⬆

The two main scores I focused on were precision and recall. Generally speaking, precision is the accuracy rate of true-positives while recall is how many items were correctly labeled out of all that existed.

My precision score is stating that whenever my model identifies someone as a person of interest, it accurately does so 31.6% of the time. This percentage seems low at first, but in the context of what we are trying to predict, having false-positives is better off than having false-negatives.

My recall score is representing my model's ability to correctly identify POIs out of the entire dataset. In other words, my model was able to identify 30.7% of all persons of interest. Not great, but not bad.

Resources Used Throughout:

1.) scikit-learn

2.) Udacity Forum

3.) Will Koehrsen github

4.) Arjan Hada github

5.) Jason Carter github

6.) Allan Reyes github