Enron Analysis Using Machine Learning

Introduction

The goal of this project is to develop a machine learning model which can be trained to identify people who are potentially engaging in corporate fraud. At Enron, stock price fell from a $90 per share to $1 due to financial fraud and corruption. Given a dataset from Enron containing employee email metadata and individual earnings data, the goal is to train a machine learning model to look for patterns and identify features that can be used to identify persons of interest (POI) in other suspected instances of corporate fraud.

If a machine learning model can reliably identify corporate fraud it would be a valuable tool that could assist in criminal investigations. A machine learning model can use its computing power to scan millions of records for patterns of corruption much faster than a human could. It could also make objective and new categories and "follow the money" as it is laundered through various entities. Our process here will be to train one or more machine learning models to accurately identify and predict corporate fraud by training a subset of the Enron data and then testing the model with the rest of the Enron dataset.

Data

The data provided includes 146 records of people who worked at Enron. Eleven people are already identified as a Person of Interest (POI) meaning they are suspected of committing corporate fraud. The data includes financial fields pertaining to each person's income from salary, stock and other sources of income from the company. It also includes data about emails sent to/from each person and others.

Outliers

There were three records which appear to be outliers and which were removed from the dataset: Records containing "TOTAL" and "THE TRAVEL AGENCY IN THE PARK" as names did not appear to have any relevance to the data and were removed. A record for 'LOCKHART EUGENE E' contained 'NaN' for every data point and was also removed. The data was converted from a pkl file to csv so that it can be manually reviewed for accuracy and cleaned if needed. A copy of the cleaned data can be viewed in a spreadsheet by opening file: Data.csv.

Features Used and Selection Process

Computers do well with numbers so an emphasis was placed on the numeric fields. The financial properties of the data include:
"bonus", "deferral_payments", "deferred_income", "director_fees", "exercised_stock_options", "expenses", "loan_advances", "long_term_incentive", "restricted_stock", "restricted_stock_deferred", "salary", "total_payments", "total_stock_value"

Features Added

We are evaluating two things:
1 - The total amount of money received by certain people vs the norm. POI are likely to have received much more money. 2 - The amount of communication between POIs. The proportion of correspondence (number of emails from/to) POIs is likely to be higher than with non-POIs.

I added a total income to see who may have gotten much more money than the others. In addition, assuming that POIs are likely to have more contact with other POIs, a calculated field was added to get the ratio of communications between POIs. In order to get the ratio of emails on the same scale as the financial data, columns were added to the financial data which calculate the logarithm of the total payments, salary, bonus, total stock value and exercised stock options features. We used a total of 19 features, including:

The Number of NaN rows for each of the 36 features was: {‘bonus’: 62, ‘bonus_log’: 62, ‘deferral_payments’: 105, ‘deferral_payments_log’: 106, ‘deferred_income’: 95, ‘deferred_income_log’: 143, ‘director_fees’: 127, ‘director_fees_log’: 127, ‘email_address’: 32, ‘exercised_stock_options’: 42, ‘exercised_stock_options_log’: 42, ‘expenses’: 49, ‘expenses_log’: 49, ‘from_messages’: 57, ‘from_poi_to_this_person’: 57, ‘from_this_person_to_poi’: 57, ‘loan_advances’: 140, ‘loan_advances_log’: 140, ‘long_term_incentive’: 78, ‘long_term_incentive_log’: 78, ‘name’: 0, ‘other’: 52, ‘poi’: 0, ‘poi_ratio_messages’: 57, ‘restricted_stock’: 34, ‘restricted_stock_deferred’: 126, ‘restricted_stock_deferred_log’: 141, ‘restricted_stock_log’: 35, ‘salary’: 49, ‘salary_log’: 49, ‘shared_receipt_with_poi’: 57, ‘to_messages’: 57, ‘total_payments’: 20, ‘total_payments_log’: 20, ‘total_stock_value’: 18, ‘total_stock_value_log’: 19 } A total of 19 features were selected using KBbestFeature: {‘bonus’, ‘bonus_log’ ‘deferral_payments’ ‘deferred_income’ ‘director_fees, ’exercised_stock_options’ ‘exercised_stock_options_log’ ‘expenses’ ‘loan_advances’ ‘long_term_incentive’ ‘poi_ratio_messages’ ‘restricted_stock’ ‘restricted_stock_deferred’ ‘salary’ ‘salary_log’ ‘total_payments’ ‘total_payments_log’ ‘total_stock_value’ ‘total_stock_value_log’ } Finally, the selected features were scaled with MinMaxScaler.

Parameter Tuning

When evaluating each model, the initial results did not meet the criteria of at least 0.3 recall and precision. It was necessary to tune the parameters. I used SKLearn’s Pipeline and GridSearchCV classes to automate parameter selection.
Pipeline combines model transformers and an estimator into a step process. Next, GridSearchCV was given a range of values for several parameters in order to search for the best estimator over the parameter grid with cross-validation. The output was a list of the best parameters to apply to the test data.

Here are the best results after both automated and manual tuning:

Algorithm Test Results

Classifier

Accuracy.Train

Precision.Train

Recall.Train

Accuracy.Test

Precision.Test

Recall.Test

GaussianNB

0.53490

0.1739

0.800

0.73373

0.23914

0.457

with Pipeline

0.51160

0.1667

0.800

0.53533

0.20024

0.83

Decision Tree

0.86050

0.3333

0.200

0.79853

0.24951

0.2545

with Pipeline

0.86050

0.4000

0.400

0.805

0.23556

0.206

SVM

0.88370

0.0000

0.000

divide by 0

divide by 0

divide by 0

AdaBoost + DecisionTree

0.81390

0.2857

0.400

0.81447

0.28826

0.2665

Linear SVC

0.83720

0.3750

0.600

divide by 0

divide by 0

divide by 0

Random Forest

0.88370

0.0000

0.000

0.85807

0.41699

0.162

KNeighbors

0.90700

1.0000

0.200

0.85867

0.39547

0.1135

Logistic Regression

0.83720

0.3750

0.600

0.82773

0.31635

0.2515

with Pipeline

0.90700

1.0000

0.200

0.7746

0.16946

0.177

with best parameters:

0.75107

0.3101

0.708

0.75107

0.31012

0.708

Algorithm Selection

Several algorithms were tried, including: GaussianNB, DecisionTree, SVM, SVC, LinearSVC, AdaBoost, RandomForest, KNeighbors, and Logistic Regression. Each of these algorithms was run and tuned with PCA and GridSearchCV as well as manually to get the best combination of parameters. Results of the tuning were measured by the accuracy, precision and recall scores. LogisticRegression turned out to have the highest accuracy, precision and recall rates when tested and is the algorithm selected. While some algorithms scored very well on accuracy and precision, recall was lower.

Validation

To validate the performance of each algorithm, accuracy, recall and precision scores were calculated for each. Each category is required to be above 0.3 in order for the algorithm to be considered acceptable. To ensure we trained our model correctly, the data was split into two categories: training and testing. When training the data, we used a subset of the overall data, selected an algorithm and tuned the features until the results were acceptable. We then test that same model on the subset of the overall data which was reserved for testing. If the results do not pass acceptance criteria on the test dataset then it means we have made a mistake. One common mistake is overfitting the data by applying too many features or tuning parameters so that they work well on the training data only. Our model was tested with a Stratified Shuffle Split cross validation iterator in tester.py in order to create random training test sets of the data.

The best results are from the Logistic Regression algorithm which was tuned and evaluated as follows:

from sklearn import linear_model, decomposition, datasets from sklearn.pipeline import Pipeline from sklearn.model_selection import GridSearchCV

logistic = linear_model.LogisticRegression() params_lr2 = { + "logistic___tol":[10**-10, 10**-20], +"logistic___C":[0.05, 0.5, 1, 10, 10**2,10**5,10**10, 10**20], +"logistic___class_weight":['auto'], +"rbm___n_components":[2,3,4] *}*

logistic.fit(features_train,labels_train)

pred = logistic.predict(features_test) accuracy = accuracy_score(labels_test, pred) precision = precision_score(labels_test, pred) recall = recall_score(labels_test, pred)

clf_winner = Pipeline(steps=[("scaler", scaler), ("skb", SelectKBest(k=19)), ("clf_winner", LogisticRegression(tol=0.1, C = 1**19, class_weight='balanced'))])

Evaluation Metrics

Accuracy, precision and recall were used as the primary evaluation metrics. Since the overall dataset was very small, it is not likely to be a good metric to use on its own. It could be a starting point when applied to a much larger dataset. Precision is defined as (# of true positives)/(# of true positives + # of false positives). Recall is defined as: (# of true positives)/(# of true positive + # of false negatives).

These metrics are used to see if we are correctly identifying persons of interest. We want to minimize the number of false positives, i.e. the number of people who are incorrectly flagged as being involved in fraud.

References

Udacity - Intro to Machine Learning

SciKit Learn - Machine Learning in Python