Data Analyst Project 5 - Detect Fraud from the Enron Scandal with Machine Learning

1. PROJECT OVERVIEW

1.1. Goal and background

The goal of this machine learning project is to build a classifier to identify "Persons of Interest" (POI) in the Enron scandal.
The Enron scandal was the largest case of corporate fraud in the history of the United States and several persons were indicted or found guilty of misappropriation of large sums of money.

We have two types of features available in the dataset: financial and email-related.
The financial ones include salary, bonus, expenses, stock options and other forms of financial compensations.
The email-related ones indicate the volume of emails sent and received by each person, and also in relation with the identified POI's.

1.2. Interest of machine learning

As not one single feature -like "salary"- can identify a "POI" on its own, machine learning can be used to reveal patterns among persons and features.
We will investigate different selections of features with different classifiers in the process.

1.3. Outliers and other preparations

Outliers: a quick manual investigation of the dataset in CSV form on Excel showed two immediate outliers that could be removed.
- a person named "TOTAL": it contained the sum of all persons
- a person named "THE TRAVEL AGENCY IN THE PARK": most likely a transcript error.

These outliers were removed with the command 'data_dict.pop()'.

Without outliers, the data set contained 144 observations and of those, 18 (12.5%) were classified as POI and 126 (87.5%) as non-POI.

Other preparations:

- Index of libraries import
- Computation and ranking of missing values ("NaNs") per feature

2. FEATURES SELECTION

2.1. Manual pre-selection

Based on the computation of NaNs per feature, we excluded from the selection three features with large NaNs rates as above 0.75:

- loan_advances	0.972603
- director_fees	0.883562
- restricted_stock_deferred	0.876712

as well as the 'email adress' which was unique to each person.

All remaining features, beside the original "poi" as per instructor's notes, were separated in financial and email lists:

- financial_features=['salary', 'deferral_payments', 'total_payments', 'exercised_stock_options', 'bonus', 'restricted_stock', 'total_stock_value', 'expenses', 'other', 'deferred_income', 'long_term_incentive']

- email_features=['to_messages', 'shared_receipt_with_poi', 'from_this_person_to_poi', 'from_poi_to_this_person']

2.2. Creation of new features

- The email_features are absolute values of email sent or received by a person, it doesn't convey the intensity of communication with a POI as would a "percentage of emails sent to a POI out of all emails sent" for example.

To get a better indication of its relation with a POI, we created three new ratio-based features :
- 'from_poi_to_this_person_ratio'
- 'from_this_person_to_poi_ratio'
- 'shared_receipt_with_poi_ratio'

- Among the financial features, after further exploration of the Excel file, we created a sub-total of all direct payments and stocks received by a person as 'pay_n_stock':
  - pay_n_stock_features = ['salary', 'bonus', 'other', 'total_stock_value']

Note after reviewer's comment: we chose to add those four features into a single one as they gave an overall view of the total compensation package assigned to a person.

Compensation's plans are build and validated by upper management (ie. the C class or CEO-CFO-CTO etc.) and could be used discretionarily to "buy silence or complicity" in a criminal scheme.
But they could also differ in their balance of components, although best practices discourage that: some persons may get an low salary but a huge bonus, some get little salary + bonus but a huge stock value and so on.
Thus adding a total 'pay_n_stock' feature may reveal some global picture missed otherwise with single feature focus.

All four new features were added to the dataset.

2.3. Automated pre-selection

We use SelectKBest as an automated feature selection with 'score_func = f_classif' for classification task.
We did not modify the default "k value = 10" as the main goal was to obtain the ranking.
Below are the results sorted in decreasing order:

```
[(True, 'pay_n_stock', 26.275511769875575),
 (True, 'exercised_stock_options', 24.815079733218194),
 (True, 'total_stock_value', 24.182898678566879),
 (True, 'bonus', 20.792252047181535),
 (True, 'salary', 18.289684043404513),
 (True, 'from_this_person_to_poi_ratio', 16.409712548035792),
 (True, 'deferred_income', 11.458476579280369),
 (True, 'long_term_incentive', 9.9221860131898225),
 (True, 'restricted_stock', 9.2128106219771002),
 (True, 'shared_receipt_with_poi_ratio', 9.1012687391935039),
 (False, 'total_payments', 8.7727777300916756),
 (False, 'shared_receipt_with_poi', 8.589420731682381),
 (False, 'expenses', 6.0941733106389453),
 (False, 'from_poi_to_this_person', 5.2434497133749582),
 (False, 'other', 4.1874775069953749),
 (False, 'from_poi_to_this_person_ratio', 3.1280917481567192),
 (False, 'from_this_person_to_poi', 2.3826121082276739),
 (False, 'to_messages', 1.6463411294420076),
 (False, 'deferral_payments', 0.22461127473600989)]
```

2.4. Combined Selections

In preparation of multiple runs of different classifiers, we chose to create several combinations of feature selections with score > 10.00 as an arbitrary threshold (initial tests showed clear negative returns -ie performance drop- as we kept adding low-rated features)

Here are the conbinations we tested, the *_bis version including 'pay_n_stock' as we found it often slightly improved the results:
note: features_list = ['poi']

```
- features_1 = features_list + ['pay_n_stock']

- features_3 = features_list + ['exercised_stock_options',
                                 'total_stock_value',
                                'bonus']

- features_3_bis = features_list + ['exercised_stock_options',
                                'total_stock_value',
                                 'bonus',
                                'pay_n_stock']

- features_4 = features_list + ['exercised_stock_options',
```

'total_stock_value',
'bonus',
'salary']

- features_4_bis = features_list + ['exercised_stock_options',
  'total_stock_value',
  'bonus',
  'salary',
  'pay_n_stock']

- features_5 = features_list + ['exercised_stock_options',
  'total_stock_value',
  'bonus',
  'salary',
  'from_this_person_to_poi_ratio']

- features_5_bis = features_list + ['exercised_stock_options',
  'total_stock_value',
  'bonus',
  'salary',
  'from_this_person_to_poi_ratio',
  'pay_n_stock']

- features_6 = features_list + ['exercised_stock_options',
  'total_stock_value',
  'bonus',
  'salary',
  'from_poi_to_this_person_ratio',
  'deferred_income']

- features_6_bis = features_list + ['exercised_stock_options',
  'total_stock_value',
  'bonus',
  'salary',
  'from_poi_to_this_person_ratio',
  'deferred_income',
  'pay_n_stock']

3. CLASSIFIERS ORIGINAL SELECTION

3.1. Feature scaling:
(added after reviewer's comment)

As we are comparing measurements in the dataset that have different units (financial vs email_counts), we need to scale the features.

This is particularly important for some classifier such as K-Nearest Neighbors, Support Vector Machines and Logistic Regression.

Of course, not all classifiers require feature scaling (ex: decision-tree) but as Sebastian Raschka pointed in his article About Feature Scaling and Normalization, (http://sebastianraschka.com/Articles/2014_about_feature_scaling.html):
"When in doubt, just standardize the data, it shouldn't hurt".

3.2. Experimentation:

As for the feature selection, we chose to experiment with several classifiers in the initial round with the help of a simple function:

```
def clf(type):
    t0 = time()
    clf = type()
    clf.fit(features_train, labels_train)
    pred = clf.predict(features_test)
    print '+++ One fold score +++'
    print "Accuracy_score:", accuracy_score(pred, labels_test)
    print 'Precision:', precision_score(pred, labels_test)
    print 'Recall:', recall_score(pred, labels_test)
    print 'Time:',round(time()-t0,3) ,'s\n'
    print '+++ 1000 folds score +++'
    test_classifier(clf, my_dataset, my_features)
```

The following classifiers were tested against the six sets of features in #2.4. above:

- GaussianNB (Naive Bayes)
http://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html

- SVC (Support Vector Machines)
http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

- Decision Tree
http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

- Random Forest
http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

- AdaBoost

http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html

- Kneighbors
http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

- Logistic Regression
http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

3.3. Best performance before tuning

We found the following best performances' mix, in order of number of features and with the constant Top 3 classifiers (GaussianNB, DecisionTree & RandomForest).

	GaussianNB		Decision Tree		Random Forest	
Feature selection	Precision	Recall	Precision	Recall	Precision	Recall
Features_1	0.489	0.223	0.398	0.334	0.387	0.282
Features_3	0.486	0.351	0.368	0.391	0.585	0.298
Features_3_bis	0.467	0.324	0.366	0.411	0.534	0.321
Features_4	0.503	0.323	0.322	0.338	0.486	0.228
Features_4_bis	0.471	0.323	0.365	0.370	0.489	0.264
Features_5	0.495	0.327	0.291	0.337	0.436	0.200
Features_5_bis	0.462	0.316	0.315	0.362	0.444	0.212
Features_6	0.516	0.386	0.260	0.253	0.491	0.181
Features_6_bis	0.486	0.392	0.312	0.297	0.467	0.186
average	0.486	0.329	0.333	0.344	0.480	0.241

Green cells: top-3 scores per column
Red cell: worst score per column

Note: most combinations in the bracket Features_3 to Features_5_bis already exceed the project's requirement in "Task 5: achieve better than 0.3 Precision and Recall".

4. CLASSIFIERS ADVANCED TUNING

Tuning a classifier beyond its default settings can help improve its performance on a specific criteria, like Accuracy or Precision, F1_Score.

Note after reviewer's comment: importance of parameter tuning.

Classifiers, such as decision trees, random forests or support vector machines from sklearn.org, use parameters set to default in their "plug_n_play" configuration.
Experimenting and searching for optimal values of those parameters can take the algorithm from good performance to best performance possible.
But extreme-tuning can lead to a biased algorithm with overfit to the test harness, which may not perform as well in practice.

In this project, a double performance is expected for its predictions: both Precision and Recall must exceed 0.3.

4.1. Basic tuning with 'test_size = 0.3'

The starting point suggested by the project template showed either little improvement or actual decrease in performance for all combinations, compared to the default 'test_size' (=0.25) above.

4.2. Advanced tuning with GridSearchCV

One classifier, the Random Forest, showed significantly high Precision score (up to 0.6) with:
- features_3,
- features_3_bis
- features_4_bis

but barely passed the 0.3 in Recall.

So we investigated with a GridSearchCV to fine tune it via paramaters such as :

```
- tuned_parameters = {'n_estimators': [20,50,100],
                      'min_samples_split': [1,2,4],
                       'max_features': [1,2,3]}
```

5. FINAL RESULTS

The best final combination was:

```
- features_3_bis = features_list + ['exercised_stock_options',
                                    'total_stock_value',
                                    'bonus',
                                    'pay_n_stock']
```

with

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
        max_depth=None, max_features=3, max_leaf_nodes=None,
        min_samples_leaf=1, min_samples_split=3,
        min_weight_fraction_leaf=0.0, n_estimators=100, n_jobs=1,
        oob_score=False, random_state=None, verbose=0,
        warm_start=False)
```

```
Accuracy: 0.86486
Precision: 0.53217
Recall: 0.39500
F1: 0.45850
F2: 0.42185
```

Our two metrics explained:

- **Precision** = $\frac{True\ Positives}{True\ Positives + False\ Positives}$

This means that if our classifier identified a person as a POI, the probability that this person is indeed a real POI is 53.2 %.

- **Recall** = $\frac{True\ Positives}{True\ Positives + False\ Negatives}$

While the probability to correctly identify a POI out of POIs persons was 39.3 % (True Positive / (True Positive + False Negative) and reversely the probability to miss a POI (ie. let a "guilty person go away") was 60.7 %.

***Note on validation**:* the 'test_classifier()' function imported from 'tester.py' use a cross-validation tool named StratifiedShuffleSplit() with 1,000 folds.

"Provides train/test indices to split data in train test sets.
This cross-validation object is a merge of StratifiedKFold and ShuffleSplit, which returns stratified randomized folds. The folds are made by preserving the percentage of samples for each class."

Source:http://scikit-learn.org/0.17/modules/generated/sklearn.cross_validation.StratifiedShuffleSplit.htm

This method divides our dataset into train and test set multiple times (1,000 here) with each fold setting aside 10% of the data as test set.
This compensates for the small number of persons and low percentage of POI (around 13%) which could generate an error, like no POI in train set, with a single train_test_split fold.

Additional note on validation after reviewer's comment:
Learning the parameters of a prediction function and testing it on the same data is a methodological mistake and leads to "overfitting".

To prevent it, it is common practice to split the dataset between a training set and a test set, typically partitioning along 70/30 ratio, to validate the model.
The main drawback of this basic validation method is that there is often not enough data available to partition once without losing a significant part of the modelling capability or inversely its testing.

To circumvent this obstacle, we can use cross-validation techniques, also known as rotation estimation, which run multiple instances of partitions and report the average measures to estimate a more accurate prediction of the classifier's performance.