Machine Learning on Enron Data

Project Overview

In 2000, Enron was one of the largest companies in the United States. By 2002, it had collapsed into bankruptcy due to widespread corporate fraud. In the resulting Federal investigation, a significant amount of typically confidential information entered into the public record, including tens of thousands of emails and detailed financial data for top executives.

In this project, I will use these email and financial data to identify persons of interests in the Enron fraud case. Persons of interests (POIs) are defined as individuals who were indicted, reached a settlement of plea deal with the government, or testified in exchange for prosecution immunity. The goal of this project is to build a POI identifier using machine learning skills. Machine learning is a powerful data mining technique, allowing us to understand potential patterns of a big dataset. This report will cover the following sessions:

1. Enron dataset
2. Feature selection
3. Algorithm selection and tuning
4. Validation and evaluation

Enron Data

There are a total of 146 persons and 21 features in this dataset. Among the persons, there are 18 POIs. It is noted that there are some missing values, marked as NaN. Excluding these missing values will be the first step in the following analyses.

```
### Load the dictionary containing the dataset
data_dict = pickle.load(open("final_project_dataset.pkl", "r") )

print "Total number of data points (people):", len(data_dict.keys())
print data_dict.keys()
```

```
Total number of data points (people):  146
```

```
['METTS MARK', 'BAXTER JOHN C', 'ELLIOTT STEVEN', 'CORDES WILLIAM R',
'HANNON KEVIN P', 'MORDAUNT KRISTINA M', 'MEYER ROCKFORD G', 'MCMAHON
JEFFREY', 'HORTON STANLEY C', 'PIPER GREGORY F', 'HUMPHREY GENE E',
'UMANOFF ADAM S', 'BLACHMAN JEREMY M', 'SUNDE MARTIN' …
```

```
print "Number of features:", (len(data_dict['METTS MARK']))
print data_dict['METTS MARK']
```

```
Number of features:  21
```

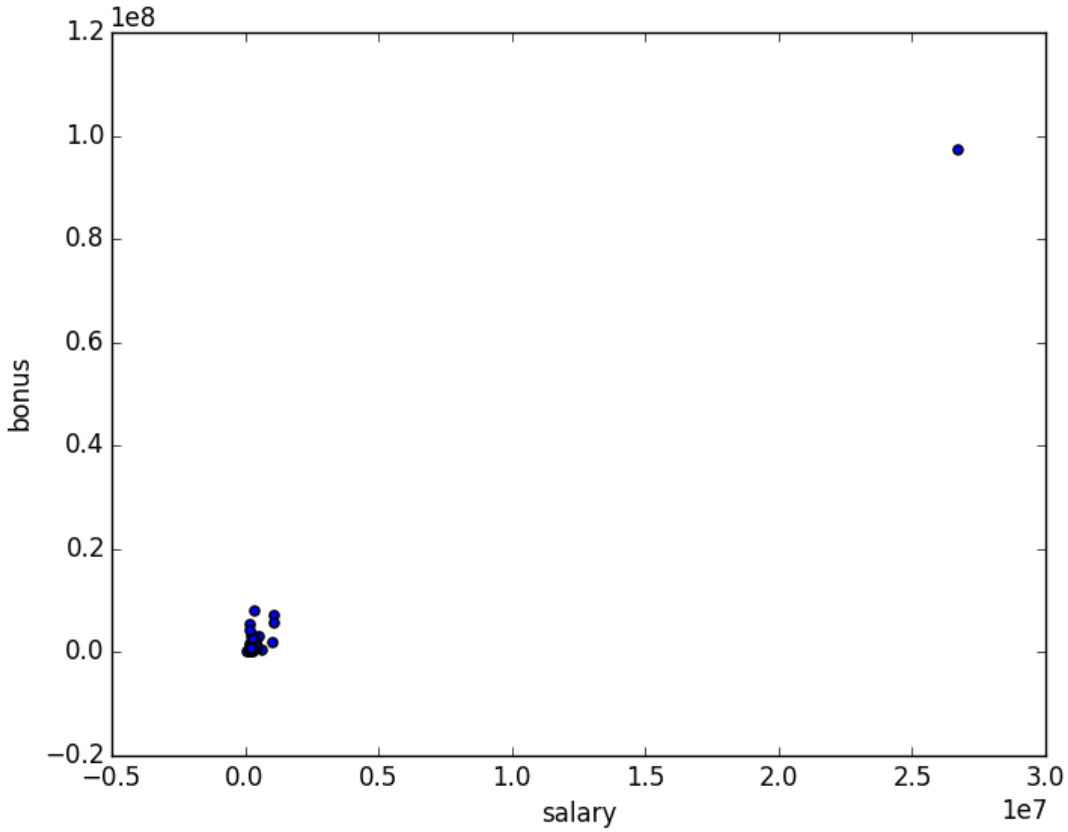
```
{'salary': 365788, 'to_messages': 807, 'deferral_payments': 'NaN',
'total_payments': 1061827, 'exercised_stock_options': 'NaN', 'bonus':
600000, 'restricted_stock': 585062, 'shared_receipt_with_poi': 702,
'restricted_stock_deferred': 'NaN', 'total_stock_value': 585062,
'expenses': 94299, 'loan_advances': 'NaN', 'from_messages': 29,
'other': 1740, 'from_this_person_to_poi': 1, 'poi': False,
'director_fees': 'NaN', 'deferred_income': 'NaN',
'long_term_incentive': 'NaN', 'email_address': 'mark.metts@enron.com',
'from_poi_to_this_person': 38}
```

```
# count POIs
i = 0
for key in data_dict:
    if data_dict[key]['poi'] == True:
        i = i + 1
        # print key
print "Number of POIs:", i
```

```
Number of POIs:  18
```

First, I would like to take a look at the financial data, to see whether there are any outliers. I started by plotting the salary and bonus features:

```
### plot features

features = ["salary", "bonus"]
data = featureFormat(data_dict, features)

for point in data:
    salary = point[0]
    bonus = point[1]
    plt.scatter(salary, bonus)

plt.xlabel("salary")
plt.ylabel("bonus")
plt.show()
```



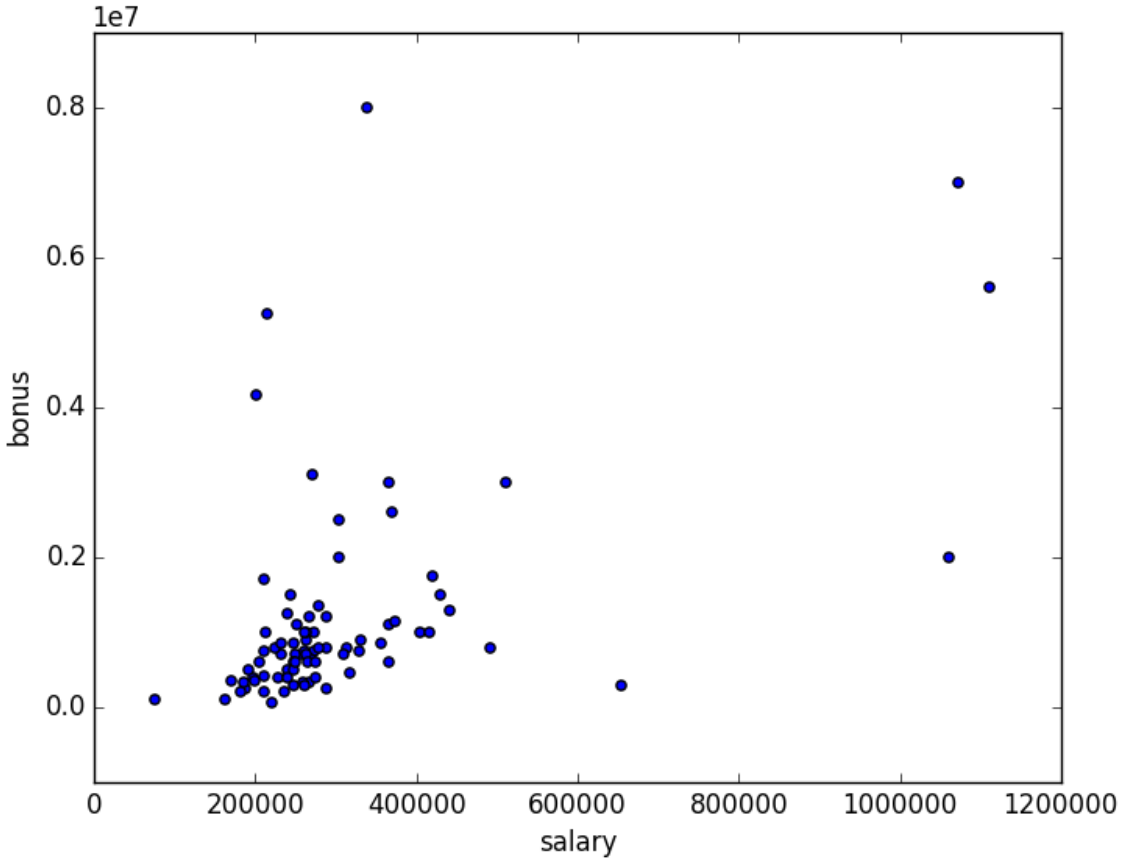
From the plot, one point far away looks like an outlier. To check the outliers, I printed out the top 10 salary data points:

```
### check outliers

top10 = []
for key in data_dict:
    if data_dict[key]['salary'] != 'NaN':
        top10.append((key,(data_dict[key]['salary'])))

print "Top 10 salary:", (sorted (top10, key = lambda x:x[1], reverse = True)[:10])
```

```
Top 10 salary: [('TOTAL', 26704229), ('SKILLING JEFFREY K', 1111258),
('LAY KENNETH L', 1072321), ('FREVERT MARK A', 1060932), ('PICKERING
MARK R', 655037), ('WHALLEY LAWRENCE G', 510364), ('DERRICK JR. JAMES
V', 492375), ('FASTOW ANDREW S', 440698), ('SHERRIFF JOHN R', 428780),
('RICE KENNETH D', 420636)]
```

The highest salary point is 'TOTAL', which should be removed. Even though the salary of SKILLING JEFFREY K, LAY KENNETH L and FREVERT MARK A are also very high,

they are definitely the POIs I would like to keep in the dataset. After excluding the 'TOTAL' outlier, the plot looks better:



Using the similar strategy, I also checked the rest of financial features, including 'total_payments', 'total_stock_value' and so on. LAY KENNETH L and SKILLING JEFFREY K look like two outliers in some of the plots. However, they should stay in the dataset, since they are two obvious POIs.

Feature Selection

I created two new features: 'from_poi_ratio' and 'to_poi_ratio' to reduce the number of features in email data.

```
for key in data_dict:
    if data_dict[key]['from_poi_to_this_person'] != 'NaN' and data_dict[key]['from_messages'] != 'NaN':
        from_ratio = float (data_dict[key]['from_poi_to_this_person'])/float(data_dict[key]['from_messages'])
    else:
        from_ratio = float(0.)
    data_dict[key]['from_poi_ratio'] = from_ratio


for key in data_dict:
    if data_dict[key]['from_this_person_to_poi'] != 'NaN' and data_dict[key]['to_messages'] != 'NaN':
        to_ratio = float(data_dict[key]['from_this_person_to_poi'])/float(data_dict[key]['to_messages'])
    else:
        to_ratio = float(0.)
    data_dict[key]['to_poi_ratio'] = to_ratio
```

In order to select the most important features, first of all, I tried the Decision Tree classifier with all 14 financial features and 3 email features (i.e. 'from_poi_ratio', 'to_poi_ratio', 'share_receipt_with_poi').

```
### split data into training and testing sets
from sklearn import cross_validation
features_train, features_test, labels_train, labels_test = cross_validation.train_test_split(
    features, labels, test_size = 0.1, random_state=42)


### Decision Tree Classifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn.metrics import recall_score
from sklearn.metrics import precision_score

def decision_tree(features_train, features_test, labels_train, labels_test):
    t0 = time()
    clf = DecisionTreeClassifier(random_state = 11)
    clf.fit(features_train, labels_train)
    pred = clf.predict(features_test)
    acc = accuracy_score(labels_test, pred)
    precision = precision_score(labels_test, pred)
    recall = recall_score(labels_test, pred)

    print "Time for decision tree:", round(time()-t0, 3), "s"
    print "Accuracy for decision tree::", acc
    print "Precision for decision tree:", precision
    print "Recall for decision tree:", recall

    feature_scores = clf.feature_importances_
    indices = np.argsort(feature_scores)[::-1]
    print "Feature ranking for decision tree:"
    for i in range (len(features_list)-1):
        print (i+1, features_list[i+1], feature_scores[indices[i]])
    print feature_scores

decision_tree(features_train, features_test, labels_train, labels_test)
```

```
Time for decision tree: 0.005 s
Accuracy for decision tree:: 0.733333333333
Precision for decision tree: 0.0
Recall for decision tree: 0.0
Feature ranking for decision tree:
(1, 'salary', 0.34407646791956059)
(2, 'deferral_payments', 0.13341004583651642)
(3, 'expenses', 0.13206282513005194)
(4, 'deferred_income', 0.12614677522385079)
(5, 'long_term_incentive', 0.11101372367128669)
(6, 'restricted_stock_deferred', 0.093066072582879245)
```

```
(7, 'loan_advances', 0.060224089635854329)
(8, 'director_fees', 0.0)
(9, 'bonus', 0.0)
(10, 'other', 0.0)
(11, 'total_stock_value', 0.0)
(12, 'restricted_stock', 0.0)
(13, 'total_payments', 0.0)
(14, 'exercised_stock_options', 0.0)
(15, 'shared_receipt_with_poi', 0.0)
(16, 'to_poi_ratio', 0.0)
(17, 'from_poi_ratio', 0.0)
```

The feature scores indicated 7 important features. However, since the precision and recall were both 0, I would need to consider another approach to select features. Therefore, SelectKBest was used to loop over the feature list to search the best k.

```
### SelectKBest - looping over best K
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import f_classif

for j in range (2, len(features_list)):
    select = SelectKBest(f_classif, k = j)
    select.fit(features_train, labels_train)
    new_features_train = select.transform(features_train)
    new_features_test = select.transform(features_test)
    new_features_list = select.get_support(indices = False)
    print "New_features_train dimensions:", new_features_train.shape
    print "Selected", j, "features:"
    print new_features_list

    decision_tree(new_features_train, new_features_test, labels_train, labels_test)
```

```
New_features_train dimensions: (129, 3)
Selected 3 features:
[ True False False False False False False False  True False  True
False False False False False False]
Time for decision tree: 0.025 s
Accuracy for decision tree:: 0.866666666667
Precision for decision tree: 0.333333333333
Recall for decision tree: 1.0
Feature ranking for decision tree:
[ 0.18518908  0.406656    0.40815492]
```

```
New_features_train dimensions: (129, 4)
Selected 4 features:
[ True False False False False False False False  True False  True
False False  True False False False]
Time for decision tree: 0.026 s
Accuracy for decision tree:: 0.933333333333
```

```
Precision for decision tree: 0.5
Recall for decision tree: 1.0
Feature ranking for decision tree:
[ 0.07904412  0.49234248  0.15401504  0.27459835]
```

```
New_features_train dimensions: (129, 5)
Selected 5 features:
[ True False False False  True False False False  True False  True
False False  True False False False]
Time for decision tree: 0.014 s
Accuracy for decision tree:: 0.866666666667
Precision for decision tree: 0.333333333333
Recall for decision tree: 1.0
Feature ranking for decision tree:
[ 0.07057511  0.08554558  0.50132593  0.27073708  0.07181631]
```

After looping over the 17 features, the result indicated that the best k was 4. Selected features included **'salary', 'bonus', 'total_stock_value'** and **'exercised_stock_options'**.

In this dataset, the allocation of labels (i.e. POI/non-POI) is not balanced. According to documentations and discussions on machine learning classifiers, 2-dimensional classifiers such as SVM and K-means may generate tricky results for the imbalanced dataset. Since Decision Tree classifier was deployed here, feature scaling was not applicable for this method.

Algorithm

With the 4 chosen features, besides Decision Tree classifier I used for feature selection, I also tried AdaBoost and GaussianNB classifiers.

```
features_list = ['poi','salary', 'bonus', 'total_stock_value', 'exercised_stock_options']
```

```
### GaussianNB
from sklearn.naive_bayes import GaussianNB
NBclf = GaussianNB()
NBclf. fit(features_train, labels_train)
```

```
### Adaboost Classifier
from sklearn.ensemble import AdaBoostClassifier
t0 = time()
ABclf = AdaBoostClassifier(n_estimators = 100, random_state = 11)
ABclf.fit(features_train, labels_train)
pred = ABclf.predict(features_test)
acc = accuracy_score(labels_test, pred)
precision = precision_score(labels_test, pred)
recall = recall_score(labels_test, pred)

print "Time for Adaboost:", round(time()-t0, 3), "s"
print "Accuracy for Adaboost::", acc
print "Precision for Adaboost:", precision
print "Recall for Adaboost:", recall
```

In order to figure out the best parameters for the algorithms, GridSearchCV was used to tune parameters in Decision Tree and AdaBoost.

```
from sklearn.grid_search import GridSearchCV

### Tune Decision Tree Classifier
parameters = {"min_samples_split": [2, 3, 4, 5, 6, 7, 8],
              "criterion": ["gini", "entropy"],
              "splitter": ["best", "random"],
              }

tree = DecisionTreeClassifier(random_state = 11, max_features = "auto", class_weight = "auto", max_depth = None)
DTclf = GridSearchCV(tree, param_grid = parameters)
DTclf.fit(features_train, labels_train)
print "The best parameters for decision tree:"
print (DTclf.best_params_)

### Tune Adaboost Classifier
parameters = {"n_estimators": [1, 10, 50, 100, 200],
              "base_estimator__min_samples_split": [2, 3, 4, 5, 6, 7, 8],
              "base_estimator__criterion": ["gini", "entropy"],
              "base_estimator__splitter": ["best", "random"],
              }

tree = DecisionTreeClassifier(random_state = 11, max_features = "auto", class_weight = "auto", max_depth = None)
Adaboost_tuned = AdaBoostClassifier(base_estimator = tree, random_state = 11)
ABclf = GridSearchCV(Adaboost_tuned, param_grid = parameters)
ABclf.fit(features_train, labels_train)

print "The best parameters for Adaboost:"
print (ABclf.best_params_)
```

```
The best parameters for decision tree:
{'min_samples_split': 2, 'splitter': 'random', 'criterion': ‘entropy'}

The best parameters for Adaboost:
{'n_estimators': 10, 'base_estimator__criterion': 'gini',
'base_estimator__min_samples_split': 5, 'base_estimator__splitter':
'best'}
```

However, the test.py returned the values of precision and recall lower than 0.3.

```
AdaBoost:
Accuracy: 0.84446 Precision: 0.49094 Recall: 0.29800   F1: 0.37088

DecisionTree:
Accuracy: 0.77323 Precision: 0.26347 Recall: 0.26400   F1: 0.26374
```

Since most of possible parameters had already been entered into GridSearchCV, to achieve a better precision and recall outcome, I had to revisit the chosen feature list. After rerunning the SelectKBest on the current 4 features in the list, I noticed that the "salary" might be a feature I could possibly drop. Therefore, I rerun GridSearchCV on Decision Tree and AdaBoost using the rest 3 features **'bonus', 'total_stock_value'** and **'exercised_stock_options'.**

```
The best parameters for decision tree:
{'min_samples_split': 2, 'splitter': 'random', 'criterion': 'gini'}

The best parameters for Adaboost:
{'n_estimators': 1, 'base_estimator__criterion': 'gini',
'base_estimator__min_samples_split': 2, 'base_estimator__splitter':
'random'}
```

The test.py returned the values of precision and recall for AdaBoost higher than 0.3 this time.

```
AdaBoostClassifier(algorithm='SAMME.R',
          base_estimator=DecisionTreeClassifier(class_weight='auto',
          criterion='gini', max_depth=None,
          max_features='auto', max_leaf_nodes=None,min_samples_leaf=1,
          min_samples_split=2, min_weight_fraction_leaf=0.0,
          random_state=11, splitter='random'),
          learning_rate=1.0, n_estimators=1, random_state=11)

Accuracy: 0.81592
Precision: 0.38648
Recall: 0.33450
F1: 0.35862
F2: 0.34375
Total predictions: 13000
True positives:  669
False positives: 1062
False negatives: 1331
True negatives: 9938
```

In addition, when applying the GaussianNB to test.py, it also returned the values of precision and recall higher than 0.3.

```
GaussianNB()
Accuracy: 0.84300
Precision: 0.48581
Recall: 0.35100
F1: 0.40755
F2: 0.37163
```

```
Total predictions: 13000
True positives:  702
False positives:  743
False negatives: 1298
True negatives: 10257
```

Validation and Evaluation

Ten-fold cross-validation was used in splitting the data into training set and testing set. Training/testing split method was use, with holding up 10% data in the whole dataset as a testing set. Having a testing set is important to valid whether the classifier works in the real data. If all data points were used as training data, the algorithm would be too specific and hard to generalize to a new data point.

```
### split data into training and testing sets
from sklearn import cross_validation
features_train, features_test, labels_train, labels_test = cross_validation.train_test_split(
    features, labels, test_size = 0.1, random_state=42)

### KFold validation for split and validate the classifier
from sklearn.cross_validation import KFold

k = KFold(len(labels), 10)
for train_index, test_index in k:
    features_train = [features[ii] for ii in train_index]
    features_test= [features[ii] for ii in test_index]
    labels_train = [labels[ii] for ii in train_index]
    labels_test= [labels[ii] for ii in test_index]
```

Accuracy, precision and recall scores were used as evaluation metrics.

```
clf.fit(features_train, labels_train)
pred = clf.predict(features_test)
acc = accuracy_score(labels_test, pred)
precision = precision_score(labels_test, pred)
recall = recall_score(labels_test, pred)

print "Accuracy for Adaboost::", acc
print "Precision for Adaboost:", precision
print "Recall for Adaboost:", recall
```

```
Accuracy: 0.81592
Precision: 0.38648
Recall: 0.33450
```

Given that the number of non-POIs is much larger than the number of POIs in this dataset, accuracy is a sub-optimal evaluation metric because that it is easily to get all non-POIs correctly classified. In this case, precision and recall are better metrics to evaluate how the classifier performs.

Precision in this dataset is the number of true POIs (who is POI and also classified as a POI) divided by the total number of individuals classified as POIs. It can be understood as the likelihood (0.38648) of a person identified as a POI is actually a true POI.

Recall in this dataset is the number of true POIs (who is POI and also classified as a POI) divided by the total number of individuals who are POIs (i.e. there are 18 POIs in this case). It can be understood as how likely (0.33450) a person will be flagged as a POI.

Both numbers are around 30%, means there are still about 70% classifications were incorrect. One possible way to increase both numbers is to further dig into the features. From the process, I noticed that feature selection is the most important step in the whole analyses, especially when the allocation of labels is imbalanced.