Identify Fraud from Enron Email

February 27, 2017

```
In [1]: import pandas as pd
        import numpy as np
        import regex as reg
        import seaborn as sns
        import matplotlib as mpl
        import matplotlib.pyplot as plt
        import pickle
        import re
        import os
        %matplotlib inline
```

There are 2 kind of data available for this analysis: - The structured data: a dictionary containing numerous features - The unstructured: a lots of emails.

I will attempt to use both of them to identify person of interest. First with the structured, then the unstructured.

1 STRUCTURED DATA SET

1. THE DATA SET

```
In [2]: from feature_format import featureFormat

        data_dict = pickle.load(open("final_project_dataset.pkl", "r") )

        features_list = ["bonus", 'deferral_payments', 'deferred_income',
                         'director_fees', 'exercised_stock_options',
                         "expenses", 'from_messages', 'loan_advances',
                         'long_term_incentive', 'other', 'restricted_stock',
                         'restricted_stock_deferred', 'salary', 'to_messages',
                         'total_payments', 'total_stock_value']

        data = featureFormat(data_dict, features_list)

In [3]: print 'Total number of data points: {}'.format(len(data_dict))
        POI = 0
        for person in data_dict:
            if data_dict[person]['poi'] == True:
                POI += 1
        non_POI = len(data_dict) - POI

        print ('Alocation across class (POI/non-POI): {}'.
               format(POI/float(non_POI)))
        print 'Number of features used: {}'.format(len(features_list))

Total number of data points: 146
Alocation across class (POI/non-POI): 0.140625
Number of features used: 16
```

```
In [4]: data_avail = {}
        for feature in features_list:
            data = featureFormat(data_dict, [feature])
            data_avail[feature] = len(data)

        data_avail

Out[4]: {'bonus': 82,
         'deferral_payments': 39,
         'deferred_income': 49,
         'director_fees': 17,
         'exercised_stock_options': 102,
         'expenses': 95,
         'from_messages': 86,
         'loan_advances': 4,
         'long_term_incentive': 66,
         'other': 93,
         'restricted_stock': 110,
         'restricted_stock_deferred': 18,
         'salary': 95,
         'to_messages': 86,
         'total_payments': 125,
         'total_stock_value': 126}
```

The median of allocation across data points (non-missing/ missing) are 0.57.

Therefore, I defined features with many missing values as any features that have less than 0.57 available data points.

```
In [5]: features_with_many_nan = 0
        total_miss = 0
        for feature in data_avail:
            total_miss += data_avail[feature]
            if (data_avail[feature] / float(len(data_dict))) < 0.57:
                features_with_many_nan += 1
        values = np.array(list(data_avail.values()))
        print ('Median of data points available: {}'
               .format(np.median(values)/len(data_dict)))
        print ('Features with many missing values: {}'
                .format(features_with_many_nan))

Median of data points available: 0.58904109589
Features with many missing values: 7
```

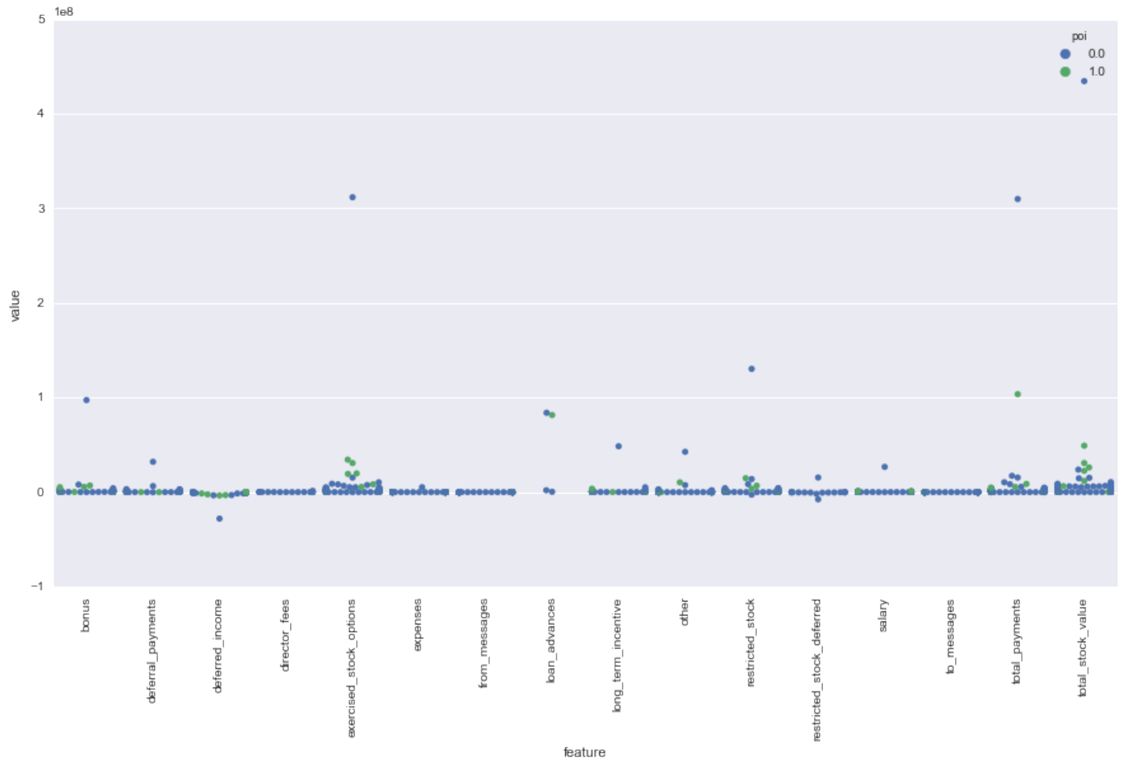
2. THE OUTLIERS: Before doing anything, it is imperitive to dive into the data, and try to understand it as deeply as you could.

First, I will attempt to visualize all the data points of all the features available. I want to see if there was some kind of special shape, and to inspect on the state of outliers as well as the noise (the noise are the data from people who are not person of interest)

So I'm going to plot the data across the x-axis, seperated by different features. I don't care about the y-axis, because I only want to look at the shape of the data.

```
In [6]: def prepare_plotting_data(input_data, features_list):
            '''
            Given the input_data, which is a dict of dict, extract feature
            one by one from the list of given features_list.
            Reformat it (remove NaN and 0)
            Return a pandas dataframe of the long format, containing
            these fields: (feature, poi, value)
            '''

            features_data = pd.DataFrame({'value': [], 'poi': [], 'feature': []})
            for feature in features_list:
                feature_data = featureFormat(input_data, ['poi', feature])
                extracted_feature_data = feature_data[:, 1]
                data = pd.DataFrame(feature_data, columns=('poi', 'value'))
                data['feature'] = feature
                features_data = pd.concat([features_data, data])
            return features_data
        features_data = prepare_plotting_data(data_dict, features_list)
        print features_data.head()

        def create_swarmplot(data):
            'Plot a swarmplot from the data created by prepare_plotting_data'

            fig = plt.figure(1, figsize=(15, 8))
            ax = fig.add_subplot(111)
            sns.swarmplot(x="feature", y="value", hue="poi", data=data);
            ax.set_xticklabels(features_list, rotation='vertical')
        create_swarmplot(features_data)

  feature  poi      value
0   bonus  0.0   600000.0
1   bonus  0.0  1200000.0
2   bonus  0.0   350000.0
```

```
3   bonus  1.0  1500000.0
4   bonus  0.0   325000.0
```



So there are quite a few outliers in this dataset.

The one in green represent those that we care about, person of interest, while the one in blue represent the noise, I will remove the blue outliers.

One widely used definition of outlier is that any data point more than 1.5 interquartile ranges (IQRs) below the first quartile or above the third quartile.

But it really is depends. In this case, I will just go with the default 1.5 first, and will come back to tune it later, if the situation calls for.

```
In [7]: def outliers_identifier(data, distance):
            '''
            Given a numpy array data, identify the outliers according to
            the input IQRs ranges, return a list containing the index of
            all the outliers
            '''
            # Calculate the q25, q75 and the IQR
            q75, q25 = np.percentile(data, [75 ,25])
            iqr = q75 - q25

            threshold_high = q75 + iqr*distance
            threshold_low = q25 - iqr*distance
```

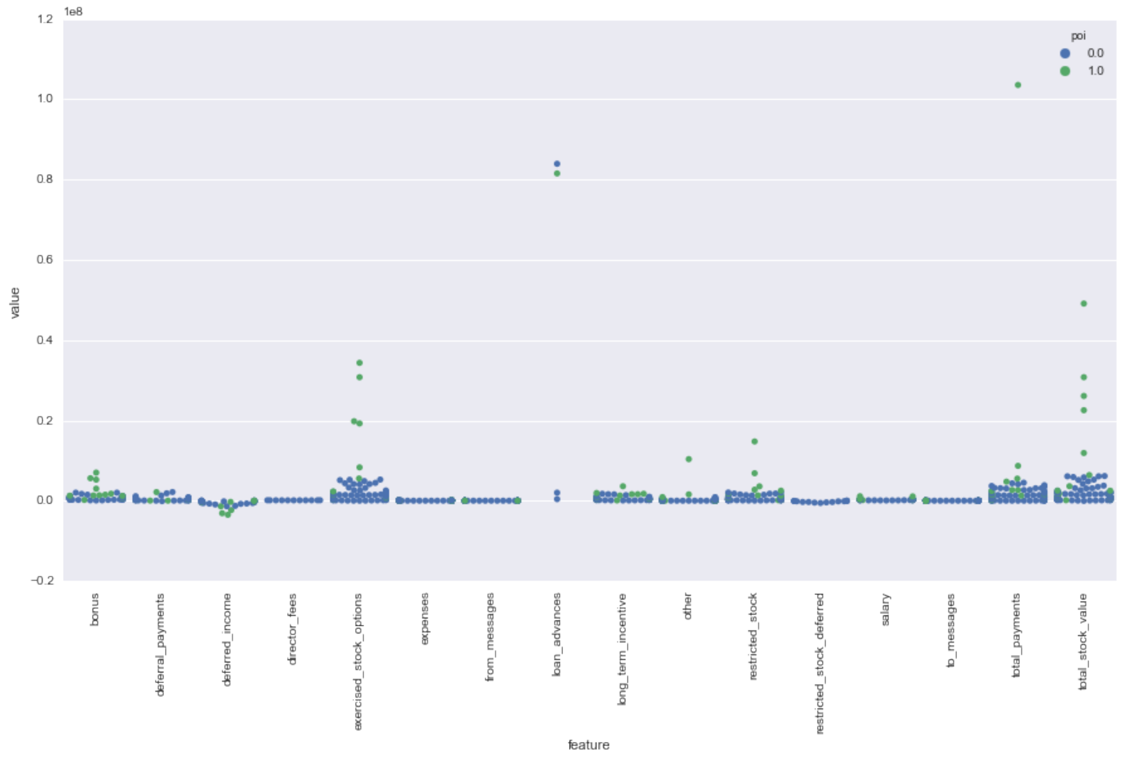
```
        # Subset the outliers, which are either lower than threshold_low,
        # or higher than threshold_high
        outliers = data[(data > threshold_high) + (data < threshold_low)]
        outliers_index = []

        for datapoint in outliers:
            if len(np.where(data == datapoint)[0]) == 1:
                outliers_index.append(np.where(data == datapoint)[0][0])
            else:
                for datapoint_in_list in np.where(data == datapoint)[0]:
                    outliers_index.append(datapoint_in_list)
        return outliers_index
```

I removed the outliers, and put all the information in a long pandas dataframe, and created a swarmplot to show the shape of the data.

I added another field 'id' to keep track of the origin of features and values.

```
In [8]: # Add id to the data_dict
        id = 1
        for key in data_dict:
            data_dict[key]['id'] = id
            id += 1

        def remove_outliers(input_data, features_list):
            '''
            Given the input_data, which is a dict of dict, extract feature
            one by one from the list of given features_list.
            Reformat it (remove NaN and 0)
            Find and remove the outliers
            Return a pandas dataframe of the long format, containing
            these fields: (id,feature, poi, value)

            '''
            cleaned_data = pd.DataFrame({'value': [], 'id': [],
                                         'poi': [], 'feature': []})
            total_deleted_points = 0
            for feature in features_list:
                # Extract feature information, stored them under long format
                # with corresponding 'poi' and 'id' data, remove 'NaN' and zeroes.
                data = featureFormat(input_data, ['poi', 'id', feature],
                                     remove_any_zeroes=True)

                # Take out the feature data
                extracted_feature_data = data[:, 2]

                # Use the function outliers_identifier to identify the outliers
                outliers_index = np.apply_along_axis(outliers_identifier, 0,
                                                     extracted_feature_data, 1.5)

                # Remove duplicate values
                outliers_index = np.unique(outliers_index)

                # Remove the datapoint according to the index stored in
                # outliers_index. Deleted_points will record the number
                # of deletions, and serve as adjustment point, since
                # the index will move up with each deletion.
                deleted_points = 0
                for datapoints in outliers_index:
                    datapoints = datapoints - deleted_points
                    if data[datapoints][0] == 1:
                        pass
                    else:
                        data = np.delete(data, datapoints, 0)
                        deleted_points += 1
                total_deleted_points += deleted_points

                # Store the data on a pandas dataframe outside of the loop
                # once done removing the outliers
                data = pd.DataFrame(data, columns=('poi', 'id', 'value'))
                data['feature'] = feature
                cleaned_data = pd.concat([cleaned_data, data])
            print ('removed {} outliers out of {} data points'
                   .format(total_deleted_points,
                           (len(cleaned_data) + total_deleted_points)))
            return cleaned_data
        outliers_free_data = remove_outliers(data_dict, features_list)
        print outliers_free_data.head()

        create_swarmplot(outliers_free_data)
```

```
removed 103 outliers out of 1193 data points
  feature   id  poi      value
0   bonus  1.0  0.0   600000.0
1   bonus  2.0  0.0  1200000.0
2   bonus  3.0  0.0   350000.0
3   bonus  5.0  1.0  1500000.0
4   bonus  6.0  0.0   325000.0
```



It looks so much better now, most of the noises had been removed.

3. MODELLING Sklearn doesn't really play well with pandas, so I will first reshape it to wide pandas, then convert it back to numpy.

```
In [9]: # Pivot to wide pandas using 'id', 'feature' and 'value'. 'poi' is
        # temporarily left behind, I will go back in pick it up later
        outliers_free_data_w = outliers_free_data.pivot(index='id',
                                                        columns='feature',
                                                        values= 'value')

        # Reset the index, but keep the id column
        outliers_free_data_w = outliers_free_data_w.reset_index(drop=False)

        # Pick up the 'poi' data left behind earlier
        outliers_free_data_w = pd.merge(outliers_free_data_w,
                                  (outliers_free_data[['id', 'poi']]
                                   .drop_duplicates()),
                                  how='left', on='id')

        # Store the name of the features and labels, these information
        # will be used later.
        feature_names = (list(outliers_free_data_w.columns.values)
                         [:len(outliers_free_data_w.columns)-1])
        label_names = np.array(['not poi', 'poi'])

        # Convert to numpy array
        outliers_f_data_w = outliers_free_data_w.as_matrix()
```

Split the targets and the features.

```
In [10]: def targetFeatureSplit(data):
             """
             Split the target and the features, here targetFeatureSplit assume
             that the target are at the end of the array
             """
             target = []
             features = []
             for item in data:
                 target.append(item[-1])
                 features.append(item[:(len(item) - 1)])

             return target, features

         cleaned_labels, cleaned_features = targetFeatureSplit(outliers_f_data_w)
```

I’ve previously cleared out all of the ‘NaN’ values when removing outliers.
But they reappeared, when I mold the dataframe back to the wide format.
So now, I will just have to clean it, once again.
This time, however, instead of 0.0, I will change the missing value to the mean of the feature.

```
In [11]: from sklearn.preprocessing import Imputer
         imp = Imputer(missing_values='NaN', strategy='mean', axis=0)
         imp.fit(cleaned_features)
         cleaned_features = imp.transform(cleaned_features)

         cleaned_features.shape

Out[11]: (145, 17)
```

The data set are small, increasing the chances of overfitting. In this kind of circumstances, a dedicated testing set will only make it worst: - Smaller training data, futher increasing the chances of overfitting. - Small testing data, not enough to cover all the possibilities.

Therefore, the traditional train test split or nested cross validation is a no go, since using them will also mean that I will have to seperate part of the data for future testing.

Choose the algorithm and tune the parameters:

```
- 30 folds cross validation, using all the data available.
- F1 score will be used as the metrics to determine the performance of models
and parameters.

Recall score might be a sounded one in the beginning, but it carries a lot of
bias with it. For an instance, the model with the highest recall score is the
one who predict everything as person of interest.
```

Estimate the performance of the chosen model:

```
- 1000 folds Stratified ShuffleSplit cross-validation
- The total precision and recall rate must be higher than 30%
- The precision and recall rate for class '1.0' must not be '0.0', since
class '1.0' are the interested class.
```

There are a lot of algorithms avaiable, so I will just try a lot.

- A pipeline are constructed and put in a function to help move things forward easier.
- SelectKBest are used to find the meaningful features.
- GridSearchCV are used to fine-tuning the parameters for the algorithms and to find the best combination of features.
- Print best score for assessment. Without any special circumstances, the algorithm with the highest score should be chosen.

```
In [20]: from sklearn.pipeline import Pipeline
         from sklearn.model_selection import GridSearchCV
         from sklearn.feature_selection import SelectKBest
         from sklearn.preprocessing import StandardScaler

         def modelling(method, parameters):
             """
             Given a method (algorithm) and a tuning parameters, train
             the model and print the results
             """
             pipeline = Pipeline([
                     ('scaler', StandardScaler()),
                     ('features_selection', SelectKBest()),
                     ('classifier', method)])
             clf = GridSearchCV(pipeline, param_grid=parameters, cv=30,
                                scoring='f1')
             clf.fit(cleaned_features, cleaned_labels)
             print clf.best_score_
             return clf.best_estimator_
```

```
In [21]: from sklearn.naive_bayes import GaussianNB
         method = GaussianNB()
         parameters = dict(features_selection__k=range(6,18))
         gnb_clf = modelling(method, parameters)
```

```
0.372413793103
```

```
In [22]: from sklearn.ensemble import ExtraTreesClassifier

         method = ExtraTreesClassifier()
         parameters = dict(features_selection__k=range(8,14),
                           classifier__criterion=['gini'],
                           classifier__n_estimators=[5, 10, 15],
                           classifier__min_samples_split=[4, 5],
                           classifier__min_samples_leaf=range(1, 4),
                           classifier__class_weight=['balanced'])

         ett_clf = modelling(method, parameters)
```

```
0.333333333333
```

```
In [23]: from sklearn.ensemble import RandomForestClassifier

         method = RandomForestClassifier()
         parameters = dict(features_selection__k=range(4, 10),
                            classifier__n_estimators=[5, 10, 20, 50],
                            classifier__min_samples_split=[2, 3, 4, 5],
                            classifier__min_samples_leaf=range(1, 7))
         rfc_clf = modelling(method, parameters)
```

```
0.28275862069
```

```
In [24]: from sklearn import svm
         method = svm.SVC()
         parameters = dict(features_selection__k=range(4,18),
                            classifier__C=[1, 10, 100, 1000],
                            classifier__kernel=['linear', 'poly','rbf'])

         svm_clf = modelling(method, parameters)
```

```
0.289655172414
```

```
In [25]: from sklearn.neighbors import KNeighborsClassifier
         method = KNeighborsClassifier()
         parameters = dict(features_selection__k=range(4,18),
                            classifier__algorithm=['ball_tree'],
                            classifier__n_neighbors=[1,2,3,4,5],
                            classifier__leaf_size=[1, 2, 5, 10, 30, 50],
                            classifier__weights=['uniform', 'distance'])

         knb_clf = modelling(method, parameters)
```

```
0.227586206897
```

GaussianNB delivered a score of 0.372413793103, which is the highest one.

```
In [28]: from sklearn.cross_validation import StratifiedShuffleSplit
         from sklearn.preprocessing import Imputer
```

```python
PERF_FORMAT_STRING = '''Accuracy: {:>0.{display_precision}f}
Precision: {:>0.{display_precision}f}
Recall: {:>0.{display_precision}f}
F1: {:>0.{display_precision}f}
F2: {:>0.{display_precision}f}'''
RESULTS_FORMAT_STRING = '''Total predictions: {:4d}
True positives: {:4d}
False positives: {:4d}
False negatives: {:4d}
True negatives: {:4d}'''

#cleaned_features, cleaned_labels
def test_classifier(clf, features, labels, folds = 1000):
    cv = StratifiedShuffleSplit(labels, folds, random_state = 42)
    true_negatives = 0
    false_negatives = 0
    true_positives = 0
    false_positives = 0
    for train_idx, test_idx in cv:
        features_train = []
        features_test  = []
        labels_train   = []
        labels_test    = []
        for ii in train_idx:
            features_train.append( features[ii] )
            labels_train.append( labels[ii] )
        for jj in test_idx:
            features_test.append( features[jj] )
            labels_test.append( labels[jj] )
        ### fit the classifier using training set, and test on test set
        clf.fit(features_train, labels_train)
        predictions = clf.predict(features_test)
        for prediction, truth in zip(predictions, labels_test):
            if prediction == 0 and truth == 0:
                true_negatives += 1
            elif prediction == 0 and truth == 1:
                false_negatives += 1
            elif prediction == 1 and truth == 0:
                false_positives += 1
            elif prediction == 1 and truth == 1:
                true_positives += 1
            else:
                print "Warning: Found a predicted label not == 0 or 1."
                print "All predictions should take value 0 or 1."
                print "Evaluating performance for processed predictions:"
                break
    try:
        total_predictions = (true_negatives + false_negatives
                             + false_positives + true_positives)
        accuracy = (1.0*(true_positives + true_negatives)
                    /total_predictions)
        precision = 1.0*true_positives/(true_positives+false_positives)
        recall = 1.0*true_positives/(true_positives+false_negatives)
        f1 = (2.0 * true_positives
              /(2*true_positives + false_positives+false_negatives))
        f2 = (1+2.0*2.0) * precision*recall/(4*precision + recall)
        print clf
        print (PERF_FORMAT_STRING.format(accuracy,
                                         precision,
                                         recall,
                                         f1, f2,
                                         display_precision = 5))
        print (RESULTS_FORMAT_STRING.format(total_predictions,
                                            true_positives,
                                            false_positives,
                                            false_negatives,
                                            true_negatives))
        print ""
    except:
        print "Got a divide by zero when trying out:", clf
        print "Precision or recall may be undefined due to a lack of true
        print true_positives
        print true_negatives
        print false_negatives
        print false_positives
```

```
In [29]: test_classifier(gnb_clf, cleaned_features, cleaned_labels, folds = 1000)
```

```
Pipeline(steps=[('scaler', StandardScaler(copy=True, with_mean=True, with_std=True)
Accuracy: 0.93560
Precision: 0.94188
Recall: 0.55100
F1: 0.69527
F2: 0.60087
Total predictions: 15000
True positives: 1102
False positives:   68
False negatives:  898
True negatives: 12932
```

2 UNSTRUCTURED DATA SET

1. DATA SET Another approach is text learning. In this approach, I will just forget about all the previously defined features, and instead, focus on figuring out if word frequencies will be better at finding the person of interest.

There are a total of 150 email directories with over 300,000 emails available. The emails stored in these directories are categorized into all kind of subdirectories: inbox, sent, archived, starwars… The emails also came with all shapes and flavours, so it might be a mess to parse them all accordingly to the main categories.

The first challenge is to find a way to parse these emails together:

- Use the parent directory where the emails belong to as the key is one approach, however, since the inrested emails are the one sent by the person only, while the directory hold not just emails sent by that person, but inbox from others as well. So not a very good approach for this one.
- My approach to solve this is to disregard the parent directory, and to just read every single email, and identify the person that sent the email (an email might be received by several, but can only be sent by one, making from_email_adress a unikey key).

```
In [7]: def extract_sender(f):
            '''
            Email identifier, will read an email and pick up the
            sender's email address
            '''
            try:
                email_pattern = re.compile(r'From: .+@.+')
                sender = email_pattern.search(f.read()).group()[6:]
            except AttributeError:
                sender = 'NaN'
            return sender

        def extract_email_and_path():
            """
            Walk through the directory that store the data, open email,
            one by one, use the function 'extract_sender' defined above
            to extract the sender's email.

            Register the sender's email as a new key to the text_dict if
            it does not exist, otherwise, store the path to the email.

            Return the text_dict once done
            """
            text_dict = {}
            sender_email = []
            contents = []
            processed = 0
            # Walk the directory
            (for dirname, dirnames, filenames in
             os.walk('.\enron_mail_20150507\maildir', topdown=False)):
                for filename in filenames:
```

```
                # When see a file, create a path that lead it
                path = os.path.join(dirname, filename) + '.'
                # Window masterrace
                path = path.replace('\\', '/')

                # Open the file and extract the from_email_adress
                with open(path, 'r') as f:
                    sender = extract_sender(f)
                # Store the from_email_adress and the path to the
                # email to a dict
                if text_dict.get(sender):
                    text_dict[sender]['email_path'].append(path)
                else:
                    text_dict[sender] = {'email_path' : [path]}
                processed += 1
        print 'processed: {}'.format(processed)
        return text_dict
text_dict = extract_email_and_path()
print 'Created dictionary containing: {}'.format(len(text_dict))
```

Add additional information to the text_dict:

```
- Poi status: If the person is identified as poi.
- Identifier: Assigned another unique id to the key, this id will be used
as the name for storage file later on.
The email can't be used for this, for most often than not, it will contain
special chars, easier to just have another set of unique id.
```

```
In [10]: # From the data dict available, extract and store the email and poi data
         data_dict_email = []
         data_dict_poi = []
         for person in data_dict:
             data_dict_email.append(data_dict[person]['email_address'])
             data_dict_poi.append(data_dict[person]['poi'])
         poi = []


         # Loop through all the email in the text_dict, if the email match the
         # data_dict's, get the poi status from it and conver to float,
         # otherwise, set it as 0.0. Assign id to the text_dict, running from 0
         # to len(text_dict), convert it to string and add '.txt' to the end.
         id = 0
         for email in text_dict:
             try:
                 list_index = data_dict_email.index(email)
                 if data_dict_poi[list_index] == True:
                     poi = 1.0
                 else:
```

```
                poi = 0.0
        except ValueError:
            poi = 0.0
        text_dict[email]['poi'] = poi
        text_dict[email]['content_file_id'] = str(id) + '.txt'
        id += 1
```

Parse the contents of all the email a person sent together and write it out as a text file using the unique file name previously given.

Only the people from enron are processed, because the power of my machine are limited.

```
In [16]: def parse_contents(list_of_files, identifier):
             '''
             Given a list of file directories, open them, one at a time,
             split the email to 2 part using the key 'X-FileName:', store
             the second part

             Once done reading all files, write the string to a new text
             file, under the given name.
             '''
             all_contents = ''
             # Parse contents from the provided list of files, one by one
             for data_file in list_of_files:
                 with open(data_file, 'r') as rf:
                     all_text = rf.read()
                 content = all_text.split("X-FileName:")
                 if len(content) > 1:
                     all_contents += content[1] + ' '

             # Write to a new text file
             with open(str(identifier), 'wb') as wf:
                 wf.write(all_contents)
             return 'contents parsed'

         contents = []
         poi = []

         # Check and parse contents for enron people only
         for email in text_dict:
             if email in data_dict_email:
                 parse_contents(text_dict[email]['email_path'],
                                text_dict[email]['content_file_id'])
                 contents.append(text_dict[email]['content_file_id'])
                 poi.append(text_dict[email]['poi'])

In [60]: contents = []
         poi = []
```

```
          import csv
          with open('poi.csv') as readfile:
              for row in csv.reader(readfile):
                  poi.append(float(row[0]))
              #poi.append(csv.reader(readfile))
          with open('contents.csv') as readfile:
              for row in csv.reader(readfile):
                  contents.append(row[0])
```

2. MODELLING This steemer will make it easier for the classifier to do its job.

It run nicely on small data sets, but when deploy on the main data set, it took eternity, in fact, I never see it finish.

So as much as I want to, I just can't incluse this steemer. And perhaps because of that, the result of the text mining were so terrible.

```
In [3]: from nltk.stem.snowball import SnowballStemmer

        import regex as reg
        def preprocess(all_text):
            words = ''
            ### remove punctuation

            text_string = reg.sub(ur"\p{P}+", "", all_text)
            ### split the text string into individual words, stem each word,
            ### and append the stemmed word to words (make sure there's a single
            ### space between each stemmed word)
            text_string = text_string.split()
            stemmer = SnowballStemmer("english")

            for word in text_string:
                words = words + stemmer.stem(word) + " "
            return words
```

The first step, just like with structured data, is to identify a reasonable classifier using cross-validation.

```
In [65]: from sklearn.pipeline import Pipeline
         from sklearn.metrics import classification_report
         from sklearn.grid_search import GridSearchCV
         from sklearn.feature_selection import SelectPercentile
         from sklearn.feature_extraction.text import CountVectorizer
         from sklearn.feature_extraction.text import TfidfTransformer
         from sklearn.decomposition import PCA
         def modelling_text(method, parameters):
             """
             Given a method (algorithm) and a tuning parameters, train
             the model and print the results
             """
```

```
            pipeline = Pipeline([
                    ('vectorizer', CountVectorizer(input = 'filename',
                                                  decode_error ='ignore')),
                    ('tfidf', TfidfTransformer()),
                    ('features_selection', SelectPercentile()),
                    ('classifier', method)
                ])
            cv = GridSearchCV(pipeline, cv=5, param_grid=parameters,
                              scoring='precision')
            cv.fit(contents, poi)
            print cv.best_score_
            return cv.best_estimator_
```

```
In [66]: from sklearn.naive_bayes import MultinomialNB
         method = MultinomialNB()
         parameters = dict(features_selection__percentile=[0.01])

         modelling_text(method, parameters)

0.0


Out[66]: Pipeline(steps=[('vectorizer', CountVectorizer(analyzer=u'word', binary=Fa
                 dtype=<type 'numpy.int64'>, encoding=u'utf-8', input='filename',
                 lowercase=True, max_df=1.0, max_features=None, min_df=1,
                 ngram_range=(1, 1), preprocessor=None, stop_words=None,
              ...lassif at 0x08E91BF0>)), ('classifier', MultinomialNB(alpha=1.0, c
```

```
In [67]: from sklearn.cluster import KMeans
         method = KMeans()
         parameters = dict(features_selection__percentile=[0.01],
                           classifier__n_clusters=[2],
                           classifier__max_iter=[500, 1000])

         modelling_text(method, parameters)

0.0


Out[67]: Pipeline(steps=[('vectorizer', CountVectorizer(analyzer=u'word', binary=Fa
                 dtype=<type 'numpy.int64'>, encoding=u'utf-8', input='filename',
                 lowercase=True, max_df=1.0, max_features=None, min_df=1,
                 ngram_range=(1, 1), preprocessor=None, stop_words=None,
              ...2, n_init=10, n_jobs=1, precompute_distances='auto',
             random_state=None, tol=0.0001, verbose=0))])
```

Once again, hardware constraint. I could only use fold=5, for it simply taking too much time. The score was 0.0, so not much to go on, for now.

3 CONCLUSION

My attempt to gather insights from text-ming was unsuccesful. The reason to this, is mostly due to limitation of hardware:

```
- Failure in steeming words.
- Limitation in parameters tuning.
```

With a stronger machine, things could be different.

As for structured data, the final model used GuassianNB as the main algorithm, performing at:

```
- Accuracy: 0.93560
- Precision: 0.94188
- Recall: 0.55100
- F1: 0.69527
- F2: 0.60087
```