Identifying Fraud from Enron Email

Laurent de Vito

November 6, 2015

Contents

1 The data set

1.1 Project objective

We would like to build a person of interest identifier (or classifier) based on financial and email data made public as a result of the Enron scandal.

1.2 Summary of the data set

The Enron email and financial data were combined into a dictionary, where each key-value pair in the dictionary corresponds to one person. The dictionary key is the person's name, and the value is another dictionary, which contains the names of all the features and their values for that person. The features in the data fall into three major types, namely financial features, email features and POI labels.
financial features: ['salary', 'deferral_payments', 'total_payments', 'loan_advances', 'bonus', 'restricted_stock_deferred', 'deferred_income', 'total_stock_value', 'expenses', 'exercised_stock_options', 'other', 'long_term_incentive', 'restricted_stock', 'director_fees'] (all units are in US dollars)
email features: ['to_messages', 'email_address', 'from_poi_to_this_person', 'from_messages', 'from_this_person_to_poi', 'poi', 'shared_receipt_with_poi'] (units are generally number of emails messages; notable exception is email_address, which is a text string)
POI label: [poi] (boolean, represented as integer)

We will have to learn from imbalanced data. Indeed, the data set is made up of only 145 samples[1] and the class for the persons of interest (label 1) is underrepresented with as few as 18 samples, whereas 127 samples are attributed to non-persons of interest (label 0). There are few data, but as we will see in the next section, there are even fewer data than meets the eye.

1.3 Consistency checks

According to the document *Enron Statement of Financial Affairs (In re: Enron Corp.) 01-16034*, the feature *total_payments* is the sum of all payments, that is, *salary*, *bonus*, *long_term_incentive*, *deferred_income*, *deferral_payments*, *loan_advances*, *other*, *expenses* and *director_fees*. This can be checked with:

```
features_list = ['poi'] + vars_full_list
keys, data = featureFormat(data_dict,
                           features_list,
                           remove_NaN=True,
                           sort_keys = True)
consistency_df = pandas.DataFrame(data, columns=features_list, index=keys)

consistency_df["diff"] = -consistency_df["total_payments"]
for var in payments_vars:
  consistency_df["diff"] = consistency_df["diff"] + consistency_df[var]
print "inconsistency detected for:"
print consistency_df[ ["diff","poi"] ][ np.abs(consistency_df["diff"]) != 0 ]
```

Notice that the function **featureFormat** returns also the keys of the data set such that the panda data frame can inherit those keys as indices.

This piece of code gives:

```
inconsistency detected for:
                        diff  poi
BELFER ROBERT       -201715    0
BHATNAGAR SANJAY -15180562    0
```

[1] We did not count the last row named "TOTAL".

Let's print the values of the attributes of Robert Belfer using

```
print consistency_df.ix["BELFER ROBERT"]
```

We get

```
poi                             0
salary                          0
bonus                           0
long_term_incentive             0
deferred_income                 0
deferral_payments          -102500
loan_advances                   0
other                           0
expenses                        0
director_fees                3285
total_payments             102500
exercised_stock_options      3285
restricted_stock                0
restricted_stock_deferred   44093
total_stock_value          -44093
Name: BELFER ROBERT, dtype: float64
```

This is not in agreement with what can be read off from the pdf document about Robert Belfer. We trusted the pdf document (at least because the total payments are correct) and manually typed in the data pertaining to Robert Belfer.

What about Sanjay Bhatnagar ?

```
poi                              0
salary                           0
bonus                            0
long_term_incentive              0
deferred_income                  0
deferral_payments                0
loan_advances                    0
other                       137864
expenses                         0
director_fees               137864
total_payments             15456290
exercised_stock_options     2604490
restricted_stock           -2604490
restricted_stock_deferred  15456290
total_stock_value                0
Name: BHATNAGAR SANJAY, dtype: float64
```

We changed those numbers for the ones we found in the pdf document.

The feature *total_stock_value* is the sum of *exercised_stock_options*, *restricted_stock* and *restricted_stock_deferred*. It turned out that Robert Belfer and Sanjay Bhatnagar were also the only persons whose total stock value was inconsistent. As we typed in the values of all their attributes based on the pdf document, this issue was also fixed.

1.4 Outliers

1.4.1 Dealing with NaNs

Right at the start, we extracted the features from the data dictionary with the command

```
data = featureFormat(my_dataset, features_list, remove_NaN=True, sort_keys = True)
```

and carried out the analysis, until we get concerned about the many zero values we observed in plots.

Consider for instance Figure 1. Most of the values of the feature *restricted_stock_deferred*

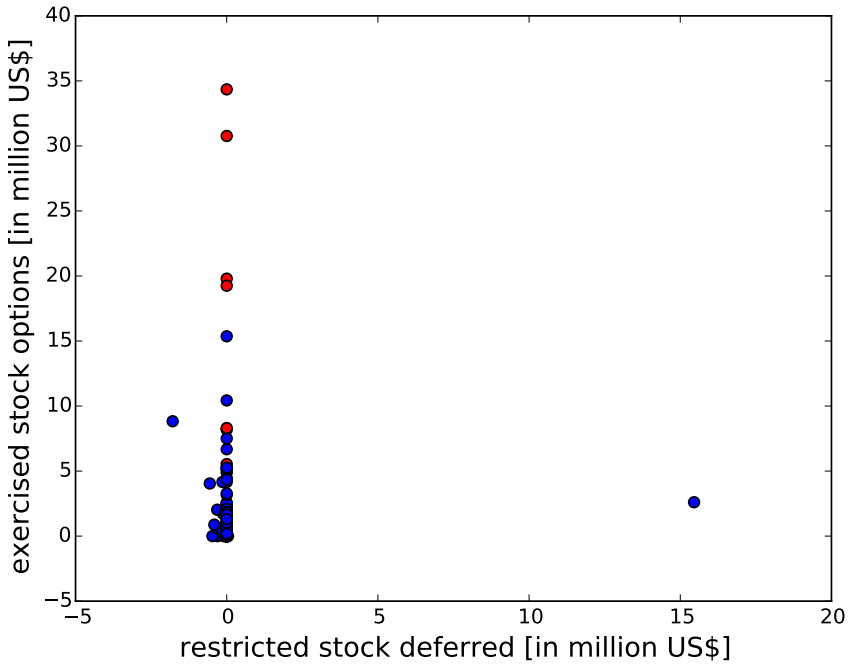

Figure 1: *restricted_stock_deferred* vs. *exercised_stock_options*. Blue is for non-POI whereas red is for POI.

are clearly clamped to exactly 0. Furthermore, if the feature value is not 0, then the sample it originates from is related to a non-POI. An algorithm could exploit this finding: In a decision tree for instance, an unseen input for which the value of *restricted_stock_deferred* is non-zero would be flagged to 0 (non-POI). An issue immediately arises: How reliable is such an outcome ?

We have to keep in mind that all NaN were set to 0. Suppose we hadn't plotted any samples for which no information was available, a sensible course of action after all. Then only samples of the class for non-POI would show up. A decision tree could therefore not rip any benefit from such a plot. An algorithm such as Gaussian Naive Bayes would have to fit an univariate Gaussian distribution for *restricted_stock_deferred* and independently for *exercised_stock_options* for both classes. How could it be trained since there are no members of a class ?

Using

```
keys, data = featureFormat(my_dataset,
                           features_list,
                           remove_NaN = False,
                           sort_keys = True)
pandas_df = pandas.DataFrame(data, columns=features_list, index=keys)
print pandas_df.isnull().sum()
```

we got the number of NaNs for each feature:

```
poi                          0
salary                      51
bonus                       64
long_term_incentive         80
deferred_income             96
deferral_payments          108
loan_advances              142
other                       53
expenses                    50
director_fees              130
total_payments              21
exercised_stock_options     44
restricted_stock            35
restricted_stock_deferred  128
total_stock_value           20
```

Notice that the function **featureFormat** in tools/feature_format.py experienced two useful modifications[2]: 1/ it returns also the keys of the data set such that the panda data frame can inherit those keys as indices and 2/if "NaN" must not be removed (and replaced by 0), they are set to None, so that pandas will actually interpret them as NaN. The default, setting "NaN" to 0, is a particular choice of imputation (certainly not the one we would go for), and imputation is preferably done at a later stage of the analysis.

We issued the command

```
pandas_df.dropna(axis=0, how="all", inplace=True)
```

to drop rows if all values are NaN, and we did not see any changes.

Let's consider all entries of the panda data frame that are not NaN. How many POI / non-POI do we have for each feature ? The following

```
features_list = ['poi'] + vars_full_list # vars_full_list is the list of financial features
keys, data = featureFormat(my_dataset, features_list, remove_NaN=False, sort_keys = True)
pandas_df = pandas.DataFrame(data, columns=features_list, index=keys)
```

[2]Because of this API change, the function **test_classifier** in final_project/tester.py had to be adjusted: Instead of

```
data = featureFormat(...)
```

at line 25, we should have:

```
_,data = featureFormat(...)
```

```
pandas_df["poi"] = pandas_df["poi"].astype(bool)
no_of_nans = []
no_of_pois = []
no_of_nonpois = []
total_no_rows = len(pandas_df)
for var in vars_full_list:
  nans = pandas_df[var].isnull().sum()
  no_of_nans.append(nans)
  pois = len(pandas_df[ pandas_df.poi & pandas_df[var].notnull() ])
  no_of_pois.append(pois)
  no_of_nonpois.append(total_no_rows - nans - pois)
df = pandas.DataFrame( { "#NaN": no_of_nans,
                         "#poi": no_of_pois,
                         "#non-poi": no_of_nonpois },
                       index = vars_full_list)
print df
```

gives the answer:

```
                           #NaN  #non-poi  #poi
salary                       51        77    17
bonus                        64        65    16
long_term_incentive          80        53    12
deferred_income              96        38    11
deferral_payments           108        32     5
loan_advances               142         2     1
other                        53        74    18
expenses                     50        77    18
director_fees               130        15     0
total_payments               21       106    18
exercised_stock_options      44        89    12
restricted_stock             35        93    17
restricted_stock_deferred   128        17     0
total_stock_value            20       107    18
```

We readily see that we must exclude the features *director_fees* and *restricted_stock_deferred*. *loan_advances* is useless and was discarded too. We kept *deferred_income* though the proportion of POI is by no means representative. Care must be exercised when dealing with this feature.

The original dataset was trimmed down to 11 features.

1.4.2 Detection of outliers

The samples that do not pertain to any real person were deleted from the dataset, see Section 1.4.2.1. We detected also some samples that should be considered outliers, see Sections 1.4.2.2 and 1.4.2.3. They must be removed from the training dataset only. Indeed, if we were to remove all samples that are not part of any well-defined cluster from both the training and test datasets, any classifier would perform excellently, but the dataset would no longer be representative. For the final evaluation of a classifier performance, the procedure consists in generating multiple training and test datasets from the original dataset and to assess the

mean performance of the classifier (well, technically, it is not exactly the mean performance). We modified the routine **test_classifier** in final_project/tester.py so that the outliers are removed from the training datasets only.

1.4.2.1 Samples not pertaining to a real person

The last row named "TOTAL" of the dataset must obviously be removed. Since the dataset is fairly small, we can scrutinize the names of all entries. We found a row named "THE TRAVEL AGENCY IN THE PARK" that was also deleted.

1.4.2.2 First pass

To start off, we are facing a huge number of features. In *Data Analysis with R*, we learned that a scatter-plot matrix is an appropriate tool to rapidly scrutinize many variables for patterns (if general, for linear trends since the correlation coefficients are usually printed) and outliers. We would have liked to use the python module **seaborn** for that purpose because it is versatile and produce beautiful plots, but the method **seaborn.pairplot** cannot deal with NaNs. Our fallback option was pandas. The observations were colored by the labels (a categorical or binary variable). We considered all financial features that are listed in Section 1.2.

```
import pickle
import sys
sys.path.append("../tools/")
import matplotlib
matplotlib.use('Qt4Agg')
import pandas
from pandas.tools.plotting import scatter_matrix

vars_list = [
     'salary',
     'deferral_payments',
     'total_payments',
     'bonus',
     'deferred_income',
     'total_stock_value',
     'expenses',
     'exercised_stock_options',
     'other',
     'long_term_incentive',
     'restricted_stock']
features_list = ['poi'] + vars_list
# Load the dictionary containing the dataset
data_dict = pickle.load(open("final_project_dataset.pkl", "r") )
# remove obvious outlier found in Lesson 7
data_dict.pop("TOTAL",0)
# Store to my_dataset
my_dataset = data_dict
# Extract features and labels from dataset for local testing
keys, data = featureFormat(my_dataset, features_list, remove_NaN=False, sort_keys = True)
```

```
# Create a pandas data frame
pandas_df = pandas.DataFrame(data, columns=features_list, index=keys)
# The feature "poi" must be of type bool
pandas_df["poi"] = pandas_df["poi"].astype(bool) # from float64 to bool
# Change data of Robert Belfer and Sanjay Bhatnagar
...
# Express all features in million US$
for var in vars_list:
  pandas_df[var] = pandas_df[var]/1e6
# Produce scatterplot matrix
colors = []
two_colors = ["blue","red"]
for k in range(0,len(pandas_df)):
  colors.append( two_colors[pandas_df.ix[k]["poi"]] )
pandas_df.columns = ["poi","A","B","C","D","E","F","G","H","I","J","K"]
matrix_ax = scatter_matrix(pandas_df.drop("poi",axis=1,inplace=False),
                           c=colors,
                           marker="o"
                           s=30,
                           alpha=1)
for ax in matrix_ax[:,0]:
    ax.grid('off', axis='both')
    ax.set_yticks([])
for ax in matrix_ax[-1,:]:
    ax.grid('off', axis='both')
    ax.set_xticks([])
plt.savefig('foo.pdf', bbox_inches='tight')
plt.show()
```

Notice that the features were rescaled so that their unit is now million US$.

The result[3] is depicted in Figure 2. What immediately draws the attention, is the presence of very few outliers that extremely distort the bivariate plots. It is surprising that those outliers are not necessarily labelled as POI. This is evident if we zoom in on specific scatter plots, Figures 3-6. Most, but not all, outliers are colored in red, that is, they correspond to POI. Blue points are reserved for non-POI. This convention will hold throughout. The bivariate plots were created using the following function:

```
def create_scatter(pandas_df, var1, var2, colors):
  plt.scatter(pandas_df[var1], pandas_df[var2],
      c=colors,
      marker="o",
      s=40,
      alpha=1)
  plt.xlabel(var1 + " [in million US$]", fontsize=16)
  plt.ylabel(var2 + " [in million US$]", fontsize=16)
```

We observe that the values of most features extend over several orders of magnitude. Furthermore most values cluster around zero. This is indicative that the

[3]The colors are wrong in the scatterplot matrices. This is a bug in pandas.

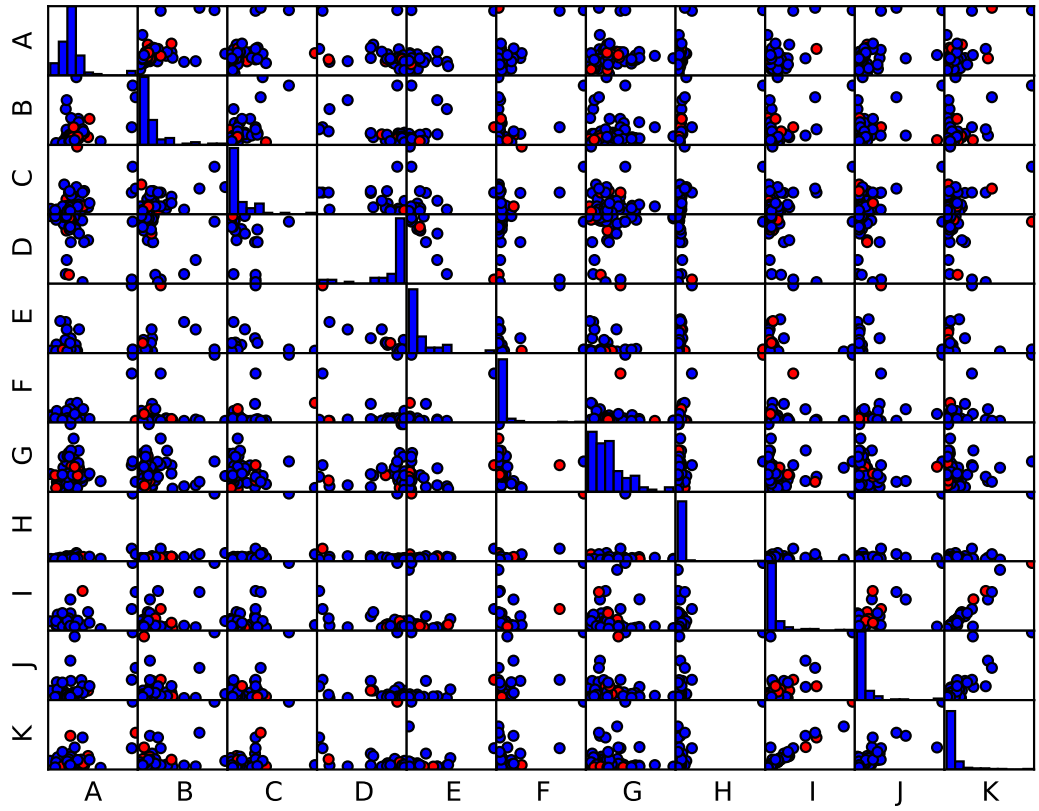

Figure 2: Scatterplot matrix of the following features:

- A: *salary*
- B: *bonus*
- C: *long_term_incentive*
- D: *deferred_income*
- E: *deferral_payments*
- F: *other*
- G: *expenses*
- H: *total_payments*
- I: *exercised_stock_options*
- J: *restricted_stock*
- K: *total_stock_value*

Labels 0 (non-POI) are colored blue whereas labels 1 (POI) are colored red.

plots are fairly deceptive and that a log transform might be appropriate. If we redraw the scatter plots in log space (or equivalently log transform all features),

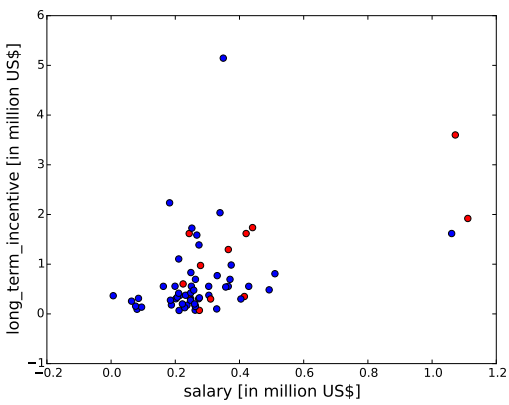

Figure 3

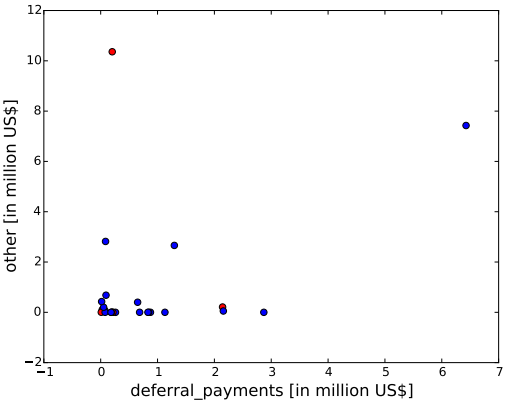

Figure 4

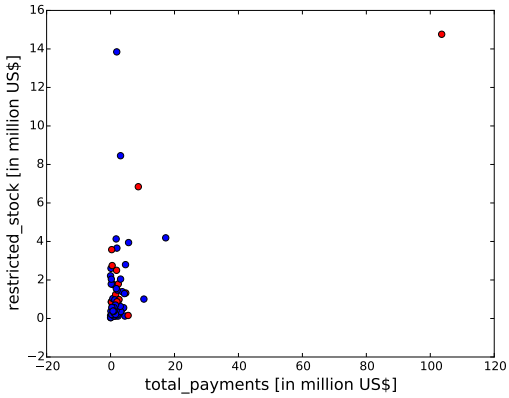

Figure 5

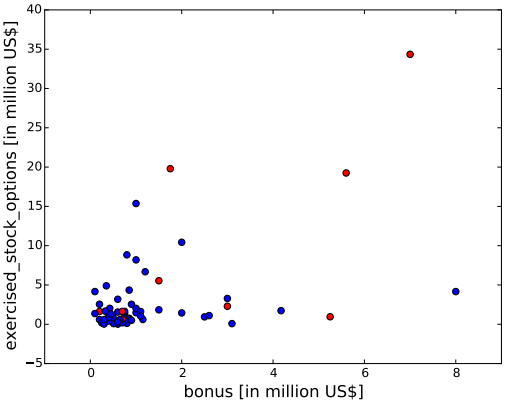

Figure 6

Figures 7-10, we get a totally different perspective.

All features benefited from this transformation except perhaps *salary*.

	salary	poi	long_term_incentive
BANNANTINE JAMES M	0.000477	False	NaN
GRAY RODNEY	0.006615	False	0.365625
WESTFAHL RICHARD K	0.063744	False	0.256191
REYNOLDS LAWRENCE	0.076399	False	0.156250
BAZELIDES PHILIP J	0.080818	False	0.093750
IZZO LAWRENCE L	0.085274	False	0.312500
OVERDYKE JR JERE C	0.094941	False	0.135836

That the salaries are spread over many orders of magnitude is true only for the first two employees with the lowest salaries so that an global transform might not be of tremendous help. Let's replot *salary* vs. *long_term_incentive* without taking the log for *salary*, Figure 11. With the help of the following function

```
def spot_outlier(df, main_feature_name, second_feature_name, no_print=4, asc=False):
  df.sort_index(by=main_feature_name, ascending=asc, inplace=True)
  print df[[main_feature_name,"poi",second_feature_name]].head(no_print)
```

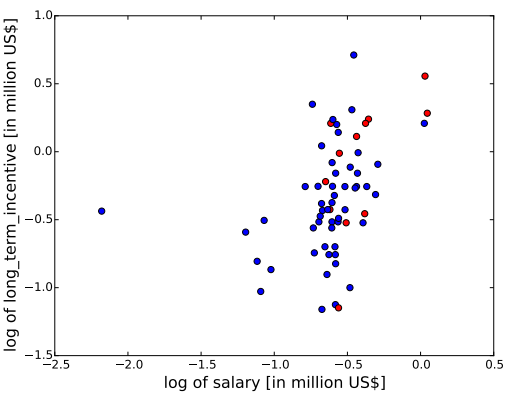

Figure 7

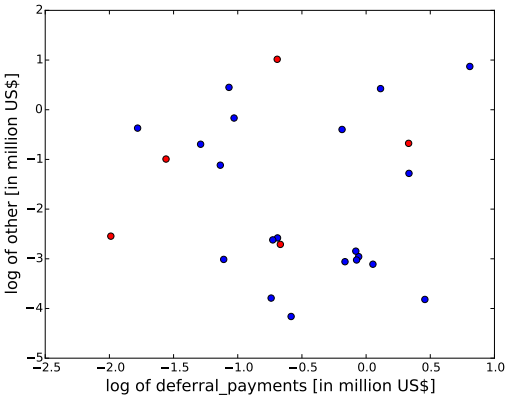

Figure 8

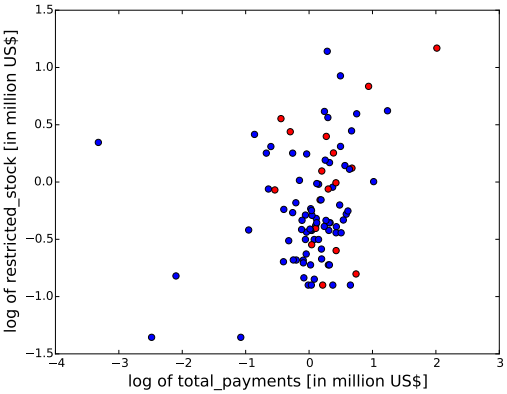

Figure 9

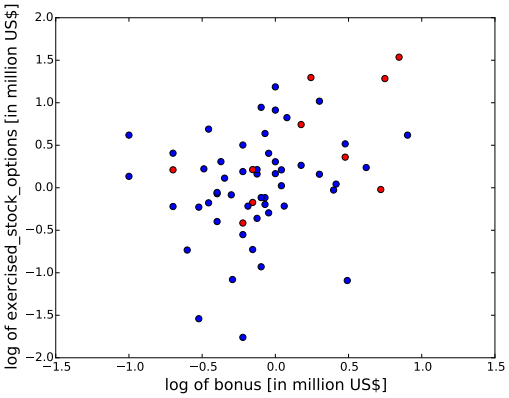

Figure 10

we easily caught the outliers that pop out in Figure 11:

	salary	poi	long_term_incentive
SKILLING JEFFREY K	1.111258	True	1.920000
LAY KENNETH L	1.072321	True	3.600000
FREVERT MARK A	1.060932	False	1.617011
PICKERING MARK R	0.655037	False	NaN
WHALLEY LAWRENCE G	0.510364	False	0.808346
DERRICK JR. JAMES V	0.492375	False	0.484000

It would be safe to remove Mark Frevert from the training datasets since it is not a POI and there is an overwhelming number of non-POI. To our great surprise, tests indicated a gain in recall but a loss in precision if he was considered an outlier (see Section 2.2.2 for an interpretation of the performance metrics). Because the net effect in terms of the F1 metric was neutral, we kept his sample.

Kenneth Lay and Jeffrey Skilling played a leading role in the corruption scandal that led to the downfall of Enron, so at first glance it seems awkward to remove them from the training datasets, all the more since there are very few samples from POI, so we would like to be conservative about what to remove. Nevertheless, the

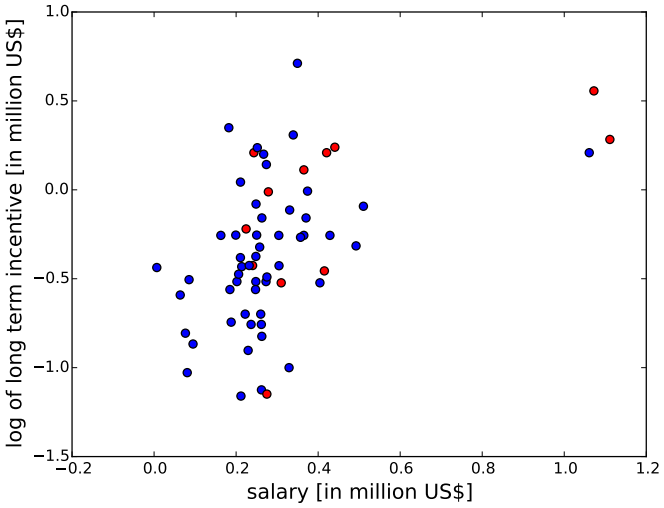

Figure 11

samples for Kenneth Lay and Jeffrey Skilling in the dataset detrimentally distort all plots and are by no means representative. Put it differently, they wouldn't be part of any cluster. So, their samples could be removed from the training datasets.

This approach, though sound, is nevertheless questionable. We expect the POIs to populate the upper right quadrant of Figure 11. So Kenneth Lay and Jeffrey Skilling are at the right place. If a POI actually stands out, then it is Ben Glisan Jr. because it is the only one far from this quadrant (the sample with the second lowest long term incentive). We must be careful since the long term incentive of most samples is not available. As a result, the plot might be misleading. Furthermore, Amanda Martin, a non-POI, had the largest long-term incentive, her salary was comparable to POIs, and thus is considered an outlier.

We ran tests with a Gaussian Naive Bayes classifier because it is lightning fast (no parameters need tuning): The latter approach yields far better results in terms of F1 and recall metrics. So Ben Glisan Jr. and Amanda Martin were removed from the training datasets whereas Kenneth Lay and Jeffrey Skilling were retained.

The limit of this approach can be appreciated with the plot of *salary* vs. *bonus*, Figure 12. As previously, we could argue that POIs are expected to fill the upper right quadrant of the plot and so Rex Shelby, the POI whose sample lies at the bottom left, is an outlier. If we remove this sample, the performance of our classifiers deteriorates a bit.

As we can see, the selection of outliers is an iterative process, but for the sake of clarity and concision, the details of the iterations are omitted. Moreover, in Section 2.2 on the performance evaluation of various algorithms, we will only use the final set of outliers we report in this section.

The power of the log transform is perhaps best illustrated through the use of histograms. We made a log transform of *other* for instance and plotted its histogram using

```
np.log10(pandas_df["other"]).hist(bins=20)
```

Results are shown in Figures 13 and 14. The log transform reveals the bimodality of the data distribution. An outlier (sample with a marginally small value for *other*)

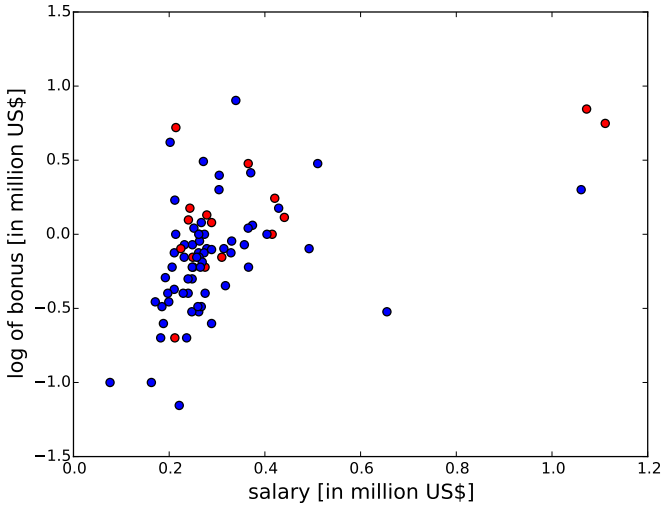

Figure 12

can also be easily identified: It is Robert Walls Jr., a non-POI.

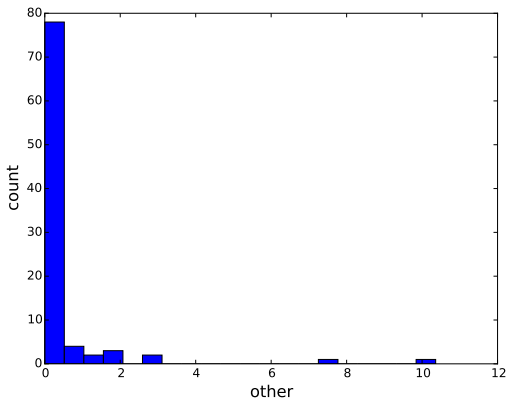

Figure 13

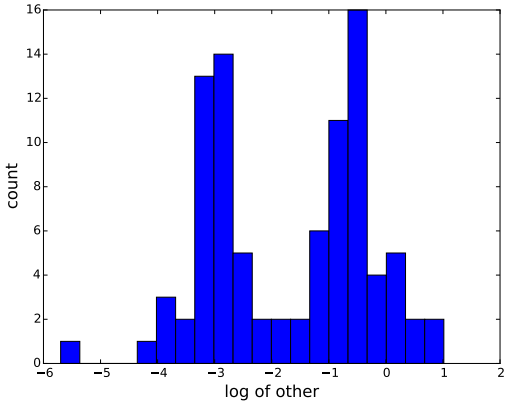

Figure 14

The log transform can only be carried out for positive values. Could it be that some features take on negative values ? With the following code

```
mi, ma, cnt = [], [], []
for var in vars_list:
  mi.append( np.min(pandas_df[var]) )
  ma.append( np.max(pandas_df[var]) )
  cnt.append( pandas_df[var][ pandas_df[var] <= 0. ].count() )
print pandas.DataFrame( { "min" : mi, "max": ma, "#neg.": cnt }, index = vars_list )
```

we got:

```
                #neg.          max        min
```

salary	0	1.111258	0.000477
bonus	0	8.000000	0.070000
long_term_incentive	0	5.145434	0.069223
deferred_income	49	-0.000833	-3.504386
deferral_payments	0	6.426990	0.007961
other	0	10.359729	0.000002
expenses	0	0.228763	0.000148
total_payments	0	103.559793	0.000148
exercised_stock_options	0	34.348384	0.009803
restricted_stock	0	14.761694	0.032460
total_stock_value	0	49.110078	0.028798

There were two outliers for *deferral_payments*, *total_stock_value* and *restricted_stock*, namely Robert Belfer and Sanjay Bhatnagar but we already caught them in Section 1.3 and fixed the values of their attributes.

We can negate *deferred_income* and take its log.

To recap: We found Ben Glisan Jr. on one side as POI and Amanda Martin and Robert Walls Jr. on the other side as non-POI. Furthermore we established that all features except *salary* are better expressed in log space. This transformation is easily done using

```
if "deferred_income" in  list(pandas_df.columns.values):
  pandas_df["deferred_income"] = -pandas_df["deferred_income"]
for var in vars_list:
  if var != "salary":
    pandas_df[var] = np.log10(pandas_df[var])
```

1.4.2.3 Second pass

Though an outlier is specific to a particular feature (or a group of feature more generally), let's nevertheless remove all of them from the scatter-plot matrix. What we have after the first pass, is depicted in Figure 15: That's far much better and this improvement is reflected in the histograms on the diagonal: They are either more bell-shaped (sometimes multimodal) or more uniform. However, we can spot additional outliers. Let's zoom on particular bivariate plots, Figures 16-19.

Evidently, as we progress, it is getting harder to tell if a sample is an outlier. What makes this exercise even more difficult, is that the plots might be deceptive since too few information is available.

- For *exercised_stock_options* vs. *salary*, Figure 16:

	salary	poi	exercised_stock_options
SKILLING JEFFREY K	1.111258	True	1.284431
LAY KENNETH L	1.072321	True	1.535906
FREVERT MARK A	1.060932	False	1.018431
PICKERING MARK R	0.655037	False	-1.540638
WHALLEY LAWRENCE G	0.510364	False	0.516266
DERRICK JR. JAMES V	0.492375	False	0.946055

  Mark Pickering is away from the bulk of non-POI samples. However, tests indicated that it is better to keep his sample in the training datasets.

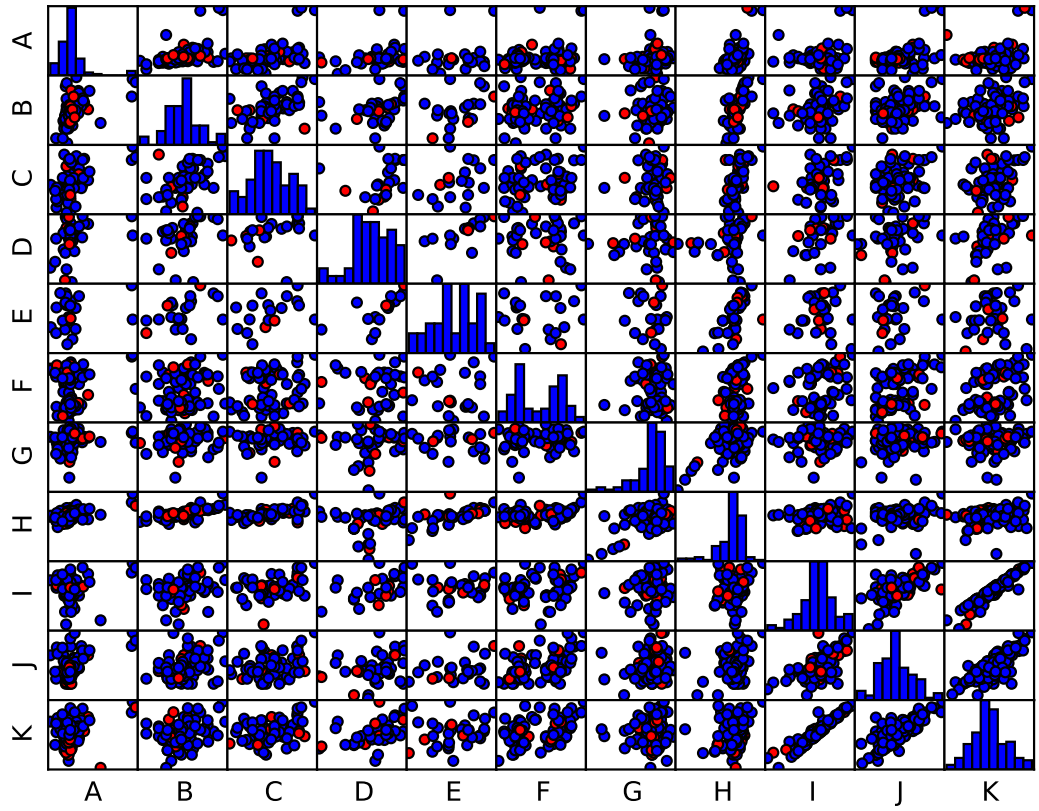

Figure 15: Scatterplot matrix of the following features:

- A: *salary*
- B: *bonus*
- C: *long_term_incentive*
- D: *deferred_income*
- E: *deferral_payments*
- F: *other*
- G: *expenses*
- H: *total_payments*
- I: *exercised_stock_options*
- J: *restricted_stock*
- K: *total_stock_value*

All features are in log space, except *salary*. Outliers found in the first pass were removed. Labels 0 (non-POI) are in blue whereas labels 1 (POI) are in red.

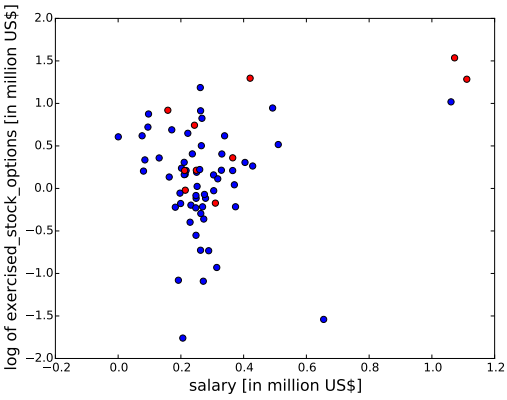

Figure 16

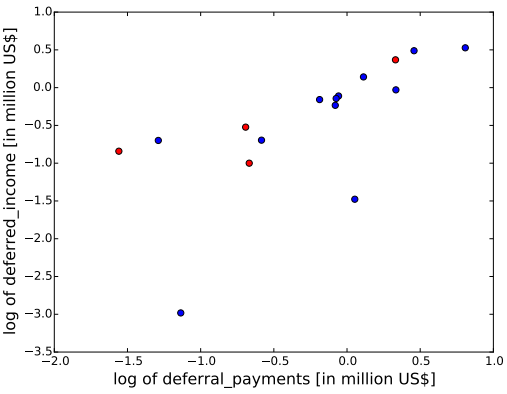

Figure 17

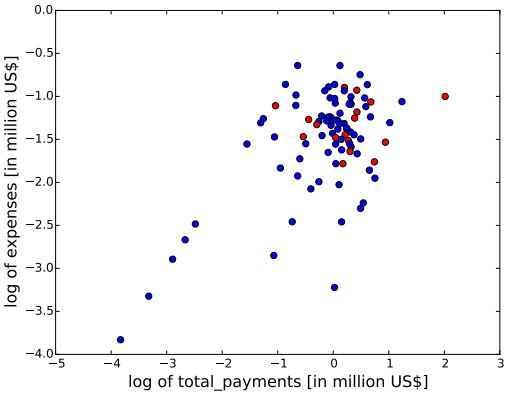

Figure 18

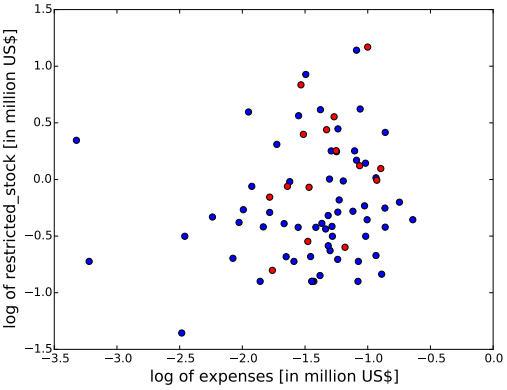

Figure 19

- For *deferred_income* vs. *deferral_payments*, Figure 17:

	deferred_income	poi	deferral_payments
BOWEN JR RAYMOND M	-3.079355	True	NaN
GAHN ROBERT S	-2.982132	False	-1.135952
SHELBY REX	-2.380176	True	NaN
BANNANTINE JAMES M	-2.292089	False	NaN
WESTFAHL RICHARD K	-1.966576	False	NaN
POWERS WILLIAM	-1.756962	False	NaN
LEMAISTRE CHARLES	-1.602060	False	NaN
JAEDICKE ROBERT	-1.602060	False	NaN
DUNCAN JOHN H	-1.602060	False	NaN
WINOKUR JR. HERBERT S	-1.602060	False	NaN
PIPER GREGORY F	-1.477126	False	0.053092
URQUHART JOHN A	-1.435736	False	NaN

  The issue that a plots are deceptive is now exacerbated since there are too few samples. Nevertheless, we removed Robert Gahn from the training datasets.

To recap: On top of the outliers spotted in the first pass (Ben Glisan Jr.,

Amanda Martin and Robert Walls Jr.), we found Robert Gahn, a non-POI.

We used pandas extensively, but we must pass the dictionary my_dataset to the testing function. This is accomplished with:

```
my_dataset = {}
for k in range(0,len(pandas_df)):
  rowname = pandas_df.index[k]
  row = {}
  for var in features_list:
    row[var] = pandas_df.ix[k][var]
  my_dataset[rowname] = row
```

1.5 Engineered features

We engineered two new features we called *ef1* and *ef2* as the ratio of *total_payments* over *total_stock_value* and the ratio of *bonus* over *long_term_incentive* respectively:

```
pandas_df["ef1"] = pandas_df["total_payments"] / pandas_df["total_stock_value"]
pandas_df["ef2"] = pandas_df["bonus"] / pandas_df["long_term_incentive"]
vars_list.append("ef1")
vars_list.append("ef2")
features_list = ['poi'] + vars_list
```

The rationale behind *ef2* is that we suppose that POIs had a tendency to prefer quick gains over long-term income perspectives. *ef1* is the balance between payments and stock values.

1.6 Imputation

The dictionary will contain NaN. Some learning algorithm cannot accept NaN and so imputation of the missing values must be done. In pandas, this step is easily carried out:

```
for var in vars_list:
  m = pandas_df[var].mean()
  pandas_df[var].fillna(m, inplace=True)
```

The NaN for a given feature is replaced that the mean of that feature. This is a sensible approach.

1.7 Feature scaling

In log space, the range of all features is roughly the same. Consequently, feature scaling should not be necessary.

However, according to the sk-learn documentation, many elements used in the objective function of a learning algorithm (such as some kernels of Support Vector Machines) assume that all features are centered around 0 and have variance in the same order. If a feature has a variance that is orders of magnitude larger that others, it might dominate the objective function and make the estimator unable to learn from other features correctly as expected.

Consequently, to be on the safe side, we standardized all features:

```
for var in vars_list:
  pandas_df[var] = StandardScaler().fit_transform(pandas_df[var])
```

With this scaling, we got far better results in terms of the precision metric for the Support Vector Classifier. It brought about only slight differences for the recall metric.

1.8 Feature selection

Why would we want to reduce the set of features ? Usually, feature selection is a mean to reduce the dimensionality of a problem to make the training of a classifier less computationally intensive (sometimes even tractable since our computational resources are limited) at the expense of a (hopefully) small loss in performance. Since our dataset is small, this is definitely not of a concern.

Feature selection is also a mean to pinpoint the features that play a major role when it comes to classification, and those that do not. It thus provides a better understanding of the dataset. Though interesting, our primary focus is on classification performance.

Finally, feature selection can improve the performance of classifiers. Some features might be unrelated or vaguely related to the labels. They act as noise and are misleading. Consequently, they have a detrimental effect on the performance of classifiers. Dropping them out will improve the accuracy of classifiers. Some algorithms are equipped with powerful devices to automatically prune out or ignore such features. As a result, the feature selection step can be bypassed. The classification algorithms we will try do not incorporate such devices, so feature selection is necessary.

Notice that regularization (done by Support Vector Classifiers) partially allows to mitigate this issue. However, regularization is usually applied to all features and no only to those that ought to be dropped out.

For feature selection, the entire data set was used but the outliers were removed.

1.8.1 Unsupervised feature selection

What is the true (sometimes called intrinsic) dimension of our data set ? We use PCA to investigate this issue:

```
n_components = features.shape[1]
pca = RandomizedPCA(whiten=True,random_state=40).fit(features)
index = [ str(x) for x in range(1,n_components+1) ]
pca_df = pandas.DataFrame( { "explained variance" : pca.explained_variance_ratio_,
                             "components": index } )
print ggplot( aes(x="components", y="explained variance"),data=pca_df) +
      geom_bar(stat="identity") +
      theme_matplotlib()
```

The variance explained by each component is shown in Figure 20. The dimension of the dataset might be 10: The last component does not explain much of the variance and might be noise.

If we add the engineered features to the feature list, we get the plot in Figure 21, which indicates that the intrinsic dimension might be 11 though we have 13 features.

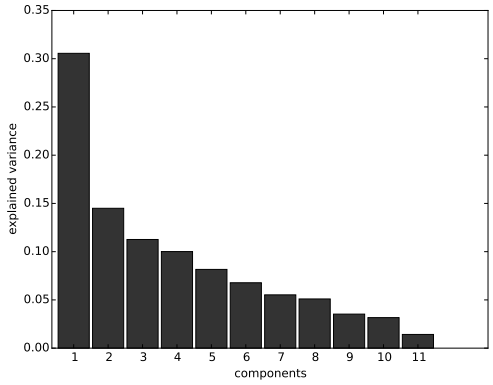

Figure 20

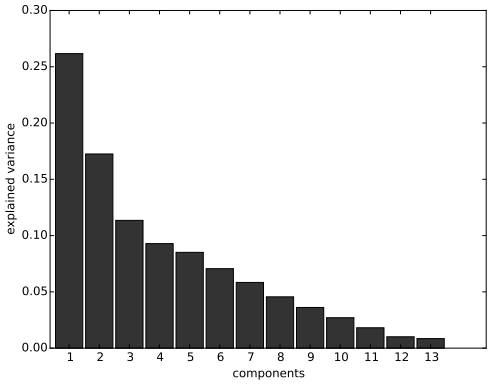

Figure 21

We focused on dimensionality reduction, but there is another advantage in using PCA: Some supervised feature selection methods like SelectKBest do not look at correlations between predictors. But collinearities might be numerically harmful. With PCA, the resultant components are orthogonal to each other, and so collinearities are no longer an issue.

After PCA, the data dictionary and the feature list must be re-built:

```
pca_dataset = {}
names = list(pandas_df.index)
vars_list = [ "comp" + str(x) for x in range(1,n_components+1) ]
for k in range(0,features.shape[0]):
  rowname = names[k]
  row = {}
  for comp in range(0,n_components):
    row[vars_list[comp]] = features[k][comp]
    row["poi"] = labels[k]
  pca_dataset[rowname] = row
features_list = ["poi"] + vars_list
my_dataset = pca_dataset
```

1.8.2 Supervised feature selection

SelectKBest is a univariate feature selection from sk-learn. The following

```
selection = SelectKBest(k="all")
features = selection.fit_transform(features,labels)
selectk_df = pandas.DataFrame( { "weight" : selection.scores_,
                                 "features": ["A","B","C","D","E","F","G","H","I","J","K"] } )
print ggplot( aes(x="components", y="weight"), data=selectk_df) +
      geom_bar(stat="identity") +
      ylab("selection scores") +
      theme_matplotlib()
```

gives the plot in Figure 22. We readily see that all features have weights, there is no

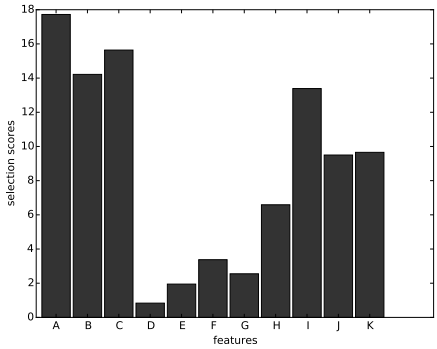

Figure 22: Selection scores of

- A: *salary*
- B: *bonus*
- C: *long_term_incentive*
- D: *deferred_income*
- E: *deferral_payments*
- F: *other*
- G: *expenses*
- H: *total_payments*
- I: *exercised_stock_options*
- J: *restricted_stock*
- K: *total_stock_value*

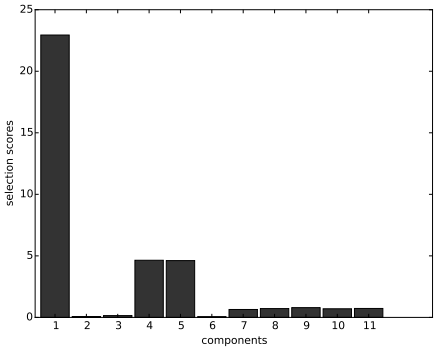

Figure 23: Selection scores of PCA components

evident score cut-off. Furthermore, it is quite hard intuitively to tell which features could be dropped out, if any. As a result, to find the optimal number of features, we adopted a brute-force-approach: We let the number of features to select k sweep the full range of available features. See Section 2.2.

If we add the engineered features to the feature list, we get the plot in Figure 24. Apparently, the two features we came up with have no (or marginal) effect on the classification. Results with the engineered features are presented in Section 2.2.4.

After SelectKBest, the data dictionary and the feature list must be re-built:

```
selectkbest_dataset = {}
names = list(pandas_df.index)
vars_list = [ pandas_df.columns[i+1] for i in range(0,len(vars_list))
              if selection.get_support()[i] ] # +1 since 0 is "poi"
for k in range(0,features.shape[0]):
  rowname = names[k]
  row = {}
  for feature in range(0,K):
    row[vars_list[feature]] = features[k][feature]
  row["poi"] = labels[k]
```

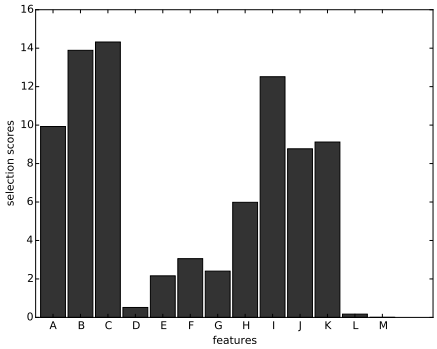

Figure 24: Selection scores of

- A: *salary*
- B: *bonus*
- C: *long_term_incentive*
- D: *deferred_income*
- E: *deferral_payments*
- F: *other*
- G: *expenses*
- H: *total_payments*
- I: *exercised_stock_options*
- J: *restricted_stock*
- K: *total_stock_value*
- L: *ef1*
- M: *ef2*

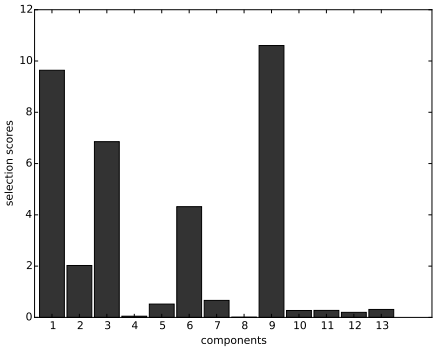

Figure 25: Selection scores of PCA components

```
  selectkbest_dataset[rowname] = row
features_list = ['poi'] + vars_list
my_dataset = selectkbest_dataset
```

1.8.3 Pipelining supervised and unsupervised feature selections

By pipelining PCA and SelectKBest before running the classification algorithm, we could rip the benefit of both feature selection methods. We were expecting that SelectKBest would indicate that the last component (the component explaining the least variance) in Figure 20 would have a tiny weight compared to the other components. What we got is depicted in Figure 23.

This is not in agreement with Figure 20, except that the main component, the component explaining most variance, has the highest score. How can we interpret such an astonishing result ? It cannot be ruled out that the component that explains

the least variance of the dataset nevertheless plays a major role in classification and vice-versa. This points out a major weakness of unsupervised feature selection for classification.

If we add the engineered features to the feature list, we get the plot in Figure 25. Components 4 and 8 have no effect on classification.

2 Classification

2.1 Analysis validation procedure

2.1.1 Validation

The performance assessment is based on the splitting of the dataset into training and test datasets. Classifiers are trained on the training dataset and their performance is evaluated on the test dataset. Without this splitting, a classifier that is trained on the full dataset would very likely overfit and thus generalize poorly on unseen input. It is precisely this generalization ability of the classifiers that is assessed when part of the available data is hold out as a test dataset.

It would not be fair to evaluate the performance of classifiers based on a single split of the dataset in training and test datasets because there are many ways the dataset can be split. To overcome this issue, we resort to k-fold cross-validation: The dataset is split in k smaller sets and the following procedure is used for each of the k folds:

- The classifier is trained using k-1 folds.
- The performance is evaluated on the remaining part of the data.

The final performance of the classifier could be the mean of a given metric over all runs for instance.

Since the dataset is imbalanced, it is very likely that some folds will have too much non-POI or too much POI, and rarely the same proportion of classes as in the original dataset. This might have detrimental effects on the performance of classifiers, predictions being either trivial or too hard. To avoid this issue, stratified shuffle cross-validation is used: The folds are made by preserving the percentage of samples for each class.

2.1.2 Parameter tuning

Most algorithms come with a set of parameters that determine their behavior (e.g. smooth decision boundary for large values of the regularization parameter) and that can be tuned for a given problem. Parameter tuning can greatly improve the performance of a classifier. Finding the best combination of parameters is treated as a search problem and is done for each training dataset using once again cross-validation.

This cross-validation must be adapted to the main characteristics of the dataset, namely that it is imbalanced. As a result, even for parameter tuning, stratified shuffle cross-validation must be used. In final_project/tester.py, we then have:

```
clf.cv = StratifiedShuffleSplit(labels_train, 10, random_state = 42)
clf.fit(features_train, labels_train)
```

where clf is a GridSearchCV object. The fit method carries out the cross-validation and the best parameters are automatically set for predictions later on.

2.1.3 Performance metrics

We must defined how the performance of a classifier is assessed.

Since our dataset is imbalanced, it may be better to avoid the accuracy metric in favor of other metrics such as precision and recall. Indeed, a classifier that would always predict non-POI whatever the input would already do a good job in terms of accuracy because of the overwhelming proportion of non-POI. The validation strategy for imbalance datasets has two main components: Stratified random splits, meaning that the folds are made by preserving the percentage of samples for each class, and an appropriate scoring function (recall, precision or F1).

What metric should we favor, for cross-validation and ultimately in order to rank the classifiers for the problem at hand ?

- If the identifier doesn't have great precision, but it does have good recall, then nearly every time a POI shows up in the test set, we are able to identify him or her. The cost of this is that we sometimes get some false positives, where non-POIs get flagged.
- If the identifier doesn't have great recall, but it does have good precision, then every time a POI gets flagged in the test set, we know with a lot of confidence that it is very likely not to be a false alarm. On the other hand, the price we pay for this is that we sometimes miss real POIs since we are effectively reluctant to pull the trigger on edge cases.
- If the identifier has a really great F1 score, then we have the best of both worlds: Both the false positive and false negative rates are low, which means that we can identify POI's reliably and accurately. If the identifier finds a POI then the person is almost certainly a POI, and if the identifier does not flag someone, then she or he is almost certainly not a POI.

We will preferably optimize the f1-score or the recall, and our preference for a given classifier will be based on the f1-score, the precision and the recall. We did not optimize with respect to the precision since the precision is usually higher than the recall when the dataset is imbalanced: This reflects the tendency of algorithms to always predict label 0 whatever the input.

2.2 Algorithms

Our strategy to find the best classifier was as follows: We relied on SelectKBest to extract the k best features where k swept the full range of available features. For each k, we recorded the evaluation metrics (recall, precision and f1). Except for parameter-free algorithms, the f1-score was optimized in a first sweep, whereas the recall was optimized in a second sweep. For completeness, we repeated this exercise with the engineered features, so that we could assess their impact on classification.

Since this exercise is time-consuming, we did not assess the impact of PCA on performance.

Notice that the f1-score might be slightly higher when the recall is optimized than when the f1-score itself is optimized. A possible explanation for this arcane behavior is that the f1 score is calculated directly from the true positives, false

positives and false negatives cumulated over all runs. Since it depends non-linearly upon the latter quantities, what is reported is most likely not to be the recall averaged over all runs.

2.2.1 Naive Bayes

The Gaussian Naive Bayes algorithm does not necessitate any parameter tuning and is amazingly fast. The results are given in Figure 26. Overall best results are obtained for $k = 11$ (all features): The precision is 0.396, the recall is 0.448 and the f1-score is 0.420.

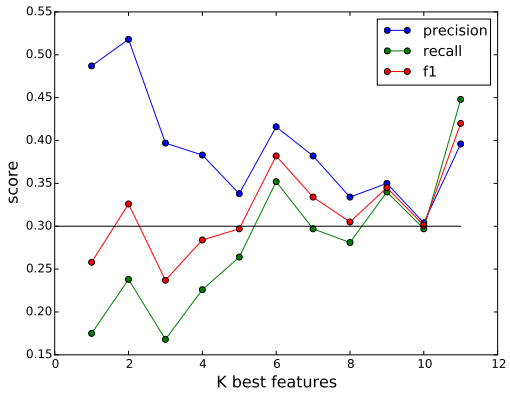

Figure 26: Results with the Naive Bayes Classifier.

	Naive Bayes	DTC	SVC
k	11	2	7
Precision	**0.396**	**0.400**	0.331
Recall	**0.448**	0.323	0.269
F1	**0.420**	0.358	0.297

	Naive Bayes	DTC	SVC
k	13	4	9
Precision	0.406	**0.516**	0.238
Recall	**0.492**	0.322	0.166
F1	**0.445**	0.397	0.196

Table 1: Performance of Naive Bayes, decision tree classifier (DTC) and Support Vector Classifier (SVC). Left: Without engineered features. Right: With both engineered features. k is the number of features selected by SelectKBest. Notice: The engineered features were not selected by SelectKBest.

2.2.2 Decision trees

The parameters min_samples_split and min_samples_leaf of the decision tree were optimized by cross-validation using exhaustive grid search over the range [2:9]x[1:7]:

```
parameters = {'min_samples_split': range(2,10),
              "min_samples_leaf": range(1,8) }
dtree = tree.DecisionTreeClassifier(random_state=40)
clf = GridSearchCV(dtree, parameters)
```

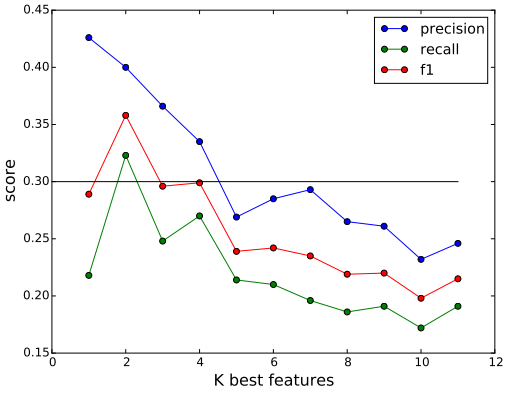

Figure 27: Results with the Decision Tree Classifier, the F1-score is optimized.

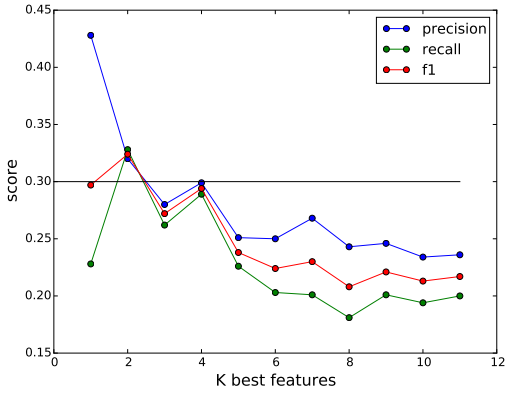

Figure 28: Results with the Decision Tree Classifier, the recall is optimized.

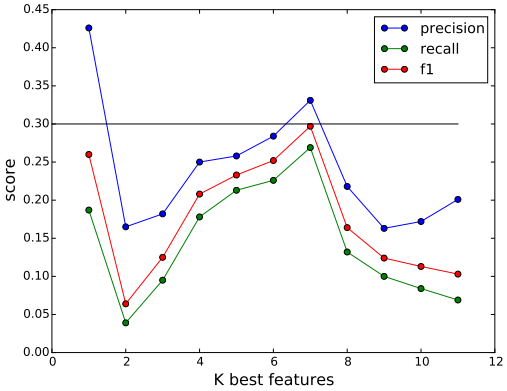

Figure 29: Results with the Support Vector Classifier, the F1-score is optimized.

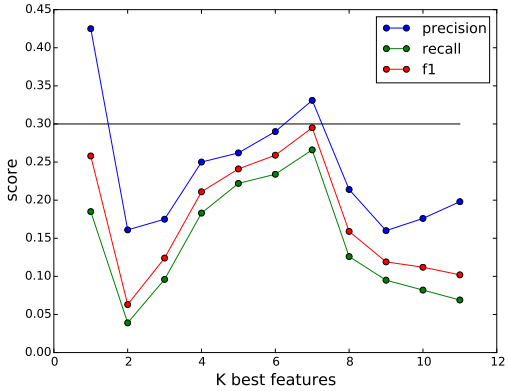

Figure 30: Results with the Support Vector Classifier, the recall is optimized.

It is important to set the parameter random_state of the decision tree classifier to get identical results across the runs.

Results are given in Figures 27 and 28. Overall best results are obtained when the f1-score is optimized for $k = 2$ (*bonus* and *exercised_stock_options*): The precision is 0.400, the recall is 0.323 and the f1-score is 0.358.

2.2.3 SVC

The kernel and the smoothing parameter of the SVC were optimized by cross-validation:

```
parameters = {'kernel':('linear', 'rbf'),
              'C':[0.01, 0.03, 0.1, 0.3, 1,3,10,30,100] }
svc = SVC()
clf = GridSearchCV(svc, parameters, scoring="accuracy")
```

Results are given in Figures 29 and 30. The differences between both plots are vanishingly small and so cannot be visually perceived. We see that a good performance under the various metrics is obtained for $k = 7$ (*salary*, *bonus*, *long_term_incentive*, *total_payments*, *exercised_stock_options*, *restricted_stock* and *total_stock_value*) : The precision is 0.331, the recall is 0.269 and the f1-score is 0.297.

2.2.4 Effect of engineered features on performance

If we add the engineered features to the feature list, whatever the type of classifier we chose, the engineered features got not selected for $k \leq 11$. This is consistent with what we found out in Section 1.8.2: The scores of the engineered features are almost zero.

Results with and without the engineered features are given in Table 1. All performance metrics of the Gaussian Naive Bayes Classifier and the Decision Tree Classifier are boosted when the engineered features are considered whereas the performance of the Support Vector Classifier deteriorates. We have no sound explanation for the poor results of the Support Vector Classifier all the more since the two variables *bonus* and *exercised_stock_options* that were retained for the Decision Tree Classifier are among the variables for the Support Vector Classifier.

The engineered features had no weight when it comes to the classification task, but most likely, the new features interact (possibly non-linearly) with the other features, and SelectKBest cannot assess this sort of interaction.

Acknowledgments

This is the fourth (...) draft of my work and I would like to thank the three reviewers for their suggestions and comments.