[Analysis of Enron Financial Data and Enron Email Corpus]

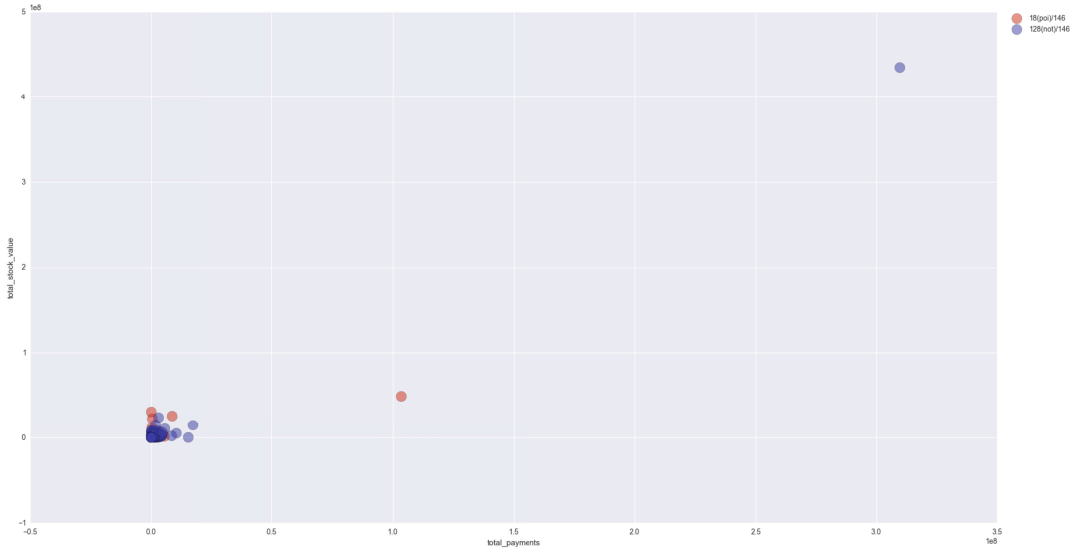
1. Project goal, machine learning, and the background, outliers and first instinct of the data sets.

The goal of this project is to detect Person of Interest (POI) among Enron employees who may have committed fraud, based on the public Enron financial (referred as “financial data”) and email (referred as “Enron corpus”) data sets captured around its bankruptcy on December 2, 2001. Machine learning algorithms are providing the power in measuring information correlation, building up contextual models and classifying large data set with defined features, especially for high-dimensional features (thousands plus) later seen in Enron corpus.

Besides financial data and Enron corpus, a list of POI emails is provided in poi_email_addresses.py which returns a list of 91 POI email addresses of 29 individuals (referred as “POI list”). Since email addresses are the keys that string up all 3 data sets, financial data and Enron corpus are compared with POI list, in the table below. As expected, the binary data sets are both skewed – more non-POIs than POIs.

	Financial data (fin data)	**Email corpus (corpus)**
No. of data points	145	2331 (each for from/to)
No. of features	21	Numerous (free text)
No. of POIs	18 (by feature “poi”)	18 (cross-check POI list)
No. of non-POIs	127	2313
No. of POI emails	18	18
No. of unique emails	111	2331
No. of POI emails in corpus	14	-
No. of POI emails in fin data	-	14
POI emails in this set only	andrew.fastow@enron.com joe.hirko@enron.com michael.kopper@enron.com scott.yeager@enron.com	jeff.richter@enron.com larry.lawyer@enron.com m..forney@enron.com tim.despain@enron.com
No. of non-POI emails in corpus	72	-
No. of non-POI emails in fin data	-	72

Do notice that the total of 145 data points in financial data is the result after removing 1 data point that represents TOTAL. Plot below shows TOTAL (blue dot) at the upper right corner and it doesn't represent real data. The POI (red dot) that is far away from others is LAY KENNETH L (Ken Lay).



Also notice some missing values for these features (keep in mind to see if it will affect model performance later):

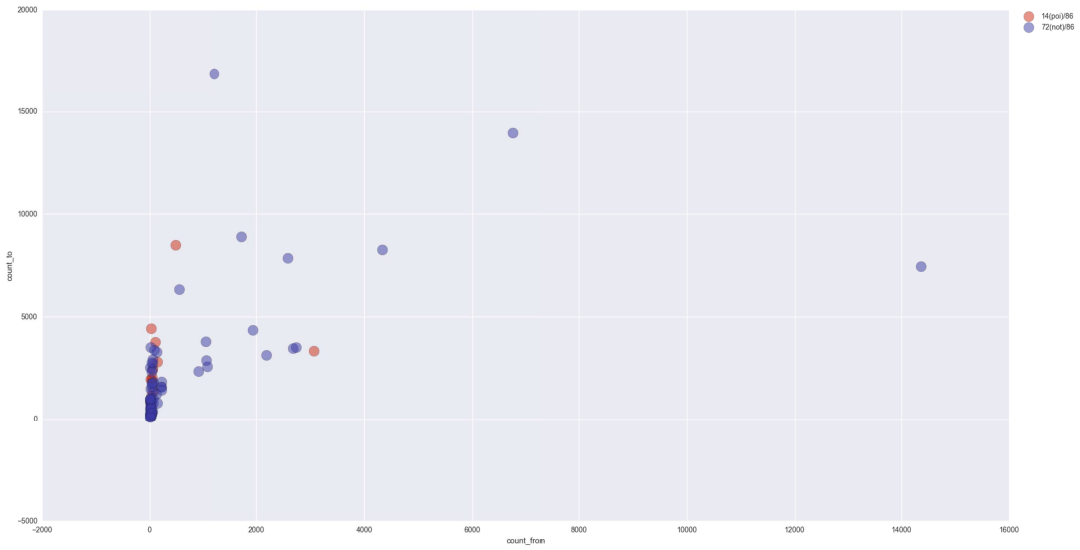
- 51 values missing for salary
- 64 values missing for bonus
- 97 values missing for deferred_income
- 44 values missing for exercised_stock_options

As for Enron corpus data set. Along with the numbers in the table below, a few things to notice while cleaning and vectoring the data:

- People received more emails then what they sent as many emails had multiple recipients, so I'll focus on emails sent from the addresses
- There are emails missing for the reference paths, but only account for negligible portion
- Reference paths of the same address can lead to duplicated emails in other folders, and it should be cleaned up before training
- The dimension of features can be problematic. I'll first narrow the scope to 14 POIs and 72 non-POIs (total of 86) who also present in financial data.

	POIs	Non-POIs	Total
Email addresses	18	2,313	2,331
Emails-from	4,711	397,031	401,742
Emails-from missing	0	14	14
Emails-to	41,028	1,874,621	1,915,649
Emails-to missing	0	13	13
Emails total	45,739	2,271,679	2,317,418

Plot below show the raw counts of emails to/from the 86 addresses (14 POIs and 72 non-POIs who also present in financial data). POI (red dot) who sent out highest number of emails is David Delainey, ex-CEO of Enron's trading unit, Enron North America. And POI who received the highest number of emails is Tim Belden, former head of trading in Enron Energy Services.



Do notice that Enron corpus contains emails dated from 1979-12-31 to 2002-06-03 before the forming of Enron in 1985 from the merger of Houston Natural Gas and InterNorth. But time-stamp vectors can reflect chronicle events. So I'll keep emails of all dates for training.

Financial data can indicate the motive for cheating and will be easier to process and train. High-dimension Corpus can represent a social network, but requires more data transformation and can overpower financial data. So my approach will be to try with all major features separately for financial data and Enron corpus, and see if there is a way to combine and engineer the features for better performance.

2. Feature selection, selection process, scaling and engineering

This section will be broken into 2 processes for financial data and Enron corpus as the first one contains organized numeric data and second one contains raw text.

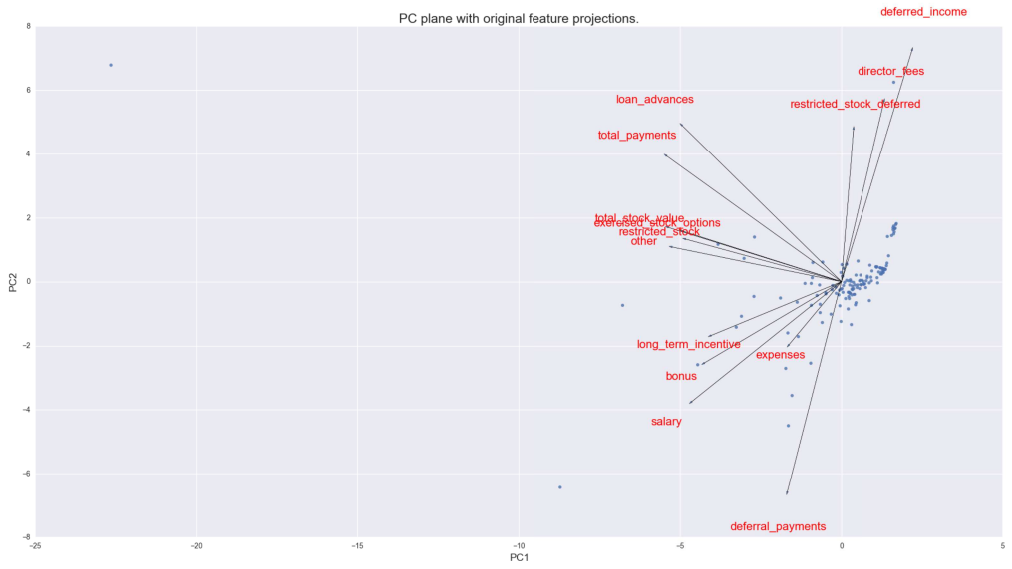
Financial Data

First notice that 3 features: “shared_receipt_with_poi,” “from_this_person_to_poi” and “from_poi_to_this_person” contain POI information within. I won't use any of these and will create new features from Enron corpus instead later.

Initially I tried each feature individually and found some top scorers (please refer to my classifiers_testing.py module for the code to run multiple algorithms):

- bonus – Precision: 0.55409 Recall: 0.37900 Nearest Neighbors
- deferred_income – Precision: 0.59534 Recall: 0.28100 Naive Bayes
- exercised_stock_options – Precision: 0.46055 Recall: 0.32100 Naive Bayes
- total_stock_value – Precision: 0.61467 Recall: 0.26400 Naive Bayes

Another systematic way is to find PCAs and explaining features (as discussed in Udacity's Google+ broadcast). For the 2 PCAs of highest variances [0.41500217, 0.12139004], the weights look like:



Since “bonus” and “exercised_stock_options” have pretty good performance by itself and they have discriminating power on PCA plot, I first reduced feature list to ['bonus', 'exercised_stock_options'] and I got pretty good performance (please refer to my classifiers_testing.py module for the code to run multiple algorithms):

	Naive Bayes	RBF SVM	Decision Tree	Nearest Neighbors	Random Forest	AdaBoost
Accuracy	0.84354	Divide by zero error	0.82038	0.87415	0.85569	0.81615
Precision	0.48537		0.40531	0.65243	0.54403	0.39572
Recall	0.28200		0.35850	0.38950	0.38300	0.37000
F1	0.35674		0.38047	0.48779	0.44953	0.38243
F2	0.30779		0.36698	0.42365	0.40710	0.37487

Although there are many missing values for “deferred_income”, in PCA plot it's dragging towards a different direction on X-axis, so I tried again with ['bonus', 'exercised_stock_options', 'deferred_income']. The performance is still good but not obviously improved:

	Naive Bayes	RBF SVM	Decision Tree	Nearest Neighbors	Random Forest	AdaBoost
Accuracy	0.85971	Divide by zero error	0.82121	0.87421	0.86114	0.80764
Precision	0.51368		0.33267	0.59256	0.52960	0.28111
Recall	0.33800		0.25000	0.46490	0.25050	0.22250
F1	0.40772		0.28547	0.41169	0.34012	0.24840
F2	0.36282		0.26307	0.42365	0.28001	0.23218

So I dropped “deferred_income” and plotted to see the distribution of these 2 features ['bonus', 'exercised_stock_options'].

There seems to be a right-skewed tendency. So I did a log transformation on ['bonus', 'exercised_stock_options'] and train again. The performance is good especially that RBF SVM is now predicting. But overall this didn't obviously improve from raw values.

	Naive Bayes	RBF SVM	Decision Tree	Nearest Neighbors	Random Forest	AdaBoost
Accuracy	0.77654	0.86362	0.81546	0.86685	0.85300	0.81223
Precision	0.27364	0.65919	0.39405	0.60632	0.53017	0.38710
Recall	0.27350	0.23500	0.37100	0.38350	0.39100	0.37800
F1	0.27357	0.34648	0.38218	0.46983	0.45007	0.38249
F2	0.27353	0.26971	0.37539	0.41392	0.41266	0.37978

And I tried scaling data with mean = 0 and stdev = 1. Performance is great but doesn't obviously improve.

	Naive Bayes	RBF SVM	Decision Tree	Nearest Neighbors	Random Forest	AdaBoost
Accuracy	0.86107	Divide by zero error	0.85133	0.87760	0.88127	0.84647
Precision	0.46739		0.43329	0.56788	0.58535	0. 41397
Recall	0.30100		0.37350	0.34300	0.37550	0.36450
F1	0.36618		0.40118	0.42768	0.45751	0. 38766
F2	0.32407		0.38410	0.37250	0.40450	0.37342

And I tried first log transforming and then scaling data with mean = 0 and stdev = 1. Performance came back to similar to using raw values (except for Naive Bayes). If later I need to combine Enron corpus and thus need to scale data, I'd first log transform financial values to maintain the same performance level.

	Naive Bayes	RBF SVM	Decision Tree	Nearest Neighbors	Random Forest	AdaBoost
Accuracy	Divide by zero error	Divide by zero error	0.84847	0.88853	0.88327	0.84353
Precision			0.42404	0.63667	0.59108	0.40566
Recall			0.38100	0.38200	0.40400	0.37300
F1			0.40137	0.47750	0.47995	0.38864
F2			0.38889	0.41522	0.43130	0.37910

For financial data, my final feature selection is ['bonus', 'exercised_stock_options']. According to some news, Enron paid huge incentives in exchange for the motive for cheating – approved to work.

P.S. I tried other feature combinations and transformation (majorly by PCA explaining power) but none came out as good as the above ones. So I didn't show other results here as it has exceeded the required 1-2 paragraphs for each question.

Enron Corpus

All the emails are in raw text format for each individual communication. Please refer to process_corpus.py for the code to extract, parse, clean up, transform and serialize the information. I saved “from”, “to”, “cc”, “bcc”, “date”, “subject”, “body”, “link_to”, “link_cc”, “link_bcc” and other statistics sections into individual pickle files for training efficiency. Special mention that “link_to”, “link_cc” and “link_bcc” are engineered features represent the communication between a dyad. The feature that represent this dyad is non-directional, however the count of initiation into this dyad is directional. For example, when A emails B, the count of (A, B) is incremented for A but not B. If B replies to A then the count of (A, B) is incremented for B but not A. And A's “link_to” feature set can look like:

{“Set(A, B)”: 14, → non-directional dyad, directional email count
“Set(A, C)”: 53,
“Set(A, N)”: 7}

And B's “link_to” feature set can look like:

{“Set(A, B)”: 10, → non-directional dyad, directional email count
“Set(B, C)”: 22,
“Set(B, X)”: 1}

Since all sections of Enron corpus have high feature dimension, I dropped 2 ensemble methods (Random Forest and AdaBoost) from the algorithm list and reduced k-folds from 1000 to 500 to save testing time. For the 86 addresses (14 POIs and 72 non-POIs who also present in financial data) selected, there are:

- 52356 total emails sent
- 20665 unique emails sent (there are duplicated emails in different folders)

I ran each set of features individually. As expected one generated highest performance is “link_to” (saved in corpus_links_to.pkl) which contains the counts for dyad communication links mentioned just above. Among these 86 addresses, 10635 dyad links (features) were established.

	Naive Bayes	RBF SVM	Decision Tree	Nearest Neighbors
Accuracy	0.80240	Divide by zero error	Divide by zero error	Divide by zero error
Precision	0.50813			
Recall	0.37500			
F1	0.43153			
F2	0.39574			

P.S. Again I tried other feature sets but none came out as good as ”links_to”. So I didn't display other results here. Comparing “to” and “links_to” sets of features, counting emails sent to a dyad (instead of a recipient) did provide better performance.

Below are some of the top “links_to” features scored by scikit-learn SelectPercentile. They do reflect the links between real POIs (I checked more but only showing a few here) although only Naive Bayes successfully finished.

Top 10 scores (f_classif for ANOVA F-value)

```
[ (  3169, Set(['jeff.skilling@enron.com', 'rex.shelby@enron.com']), 11.720930232558139),
  (  4562, Set(['kevin.hannon@enron.com', 'jeff.skilling@enron.com']), 11.720930232558139),
  (  8098, Set(['ben.glisan@enron.com', 'wes.colwell@enron.com']), 11.720930232558139),
  (  6613, Set(['kenneth.lay@enron.com', 'david.delainey@enron.com']), 11.69000429526906),
  (  3759, Set(['kenneth.lay@enron.com', 'mark.koenig@enron.com']), 11.626732170987641),
  (  2952, Set(['jeff.skilling@enron.com', 'mark.koenig@enron.com']), 11.448350459707951),
  (  3120, Set(['paula.rieker@enron.com', 'jeff.skilling@enron.com']), 11.448350459707951),
  (  4490, Set(['kenneth.lay@enron.com', 'kevin.hannon@enron.com']), 10.64646046267984),
  (  3936, Set(['tim.belden@enron.com', 'christopher.calger@enron.com']), 10.37590545177),
  (  7986, Set(['richard.causey@enron.com', 'david.delainey@enron.com']), 10.37590545177)]
```

However, this feature set is limited to existing addresses only, I need to further engineer it for newly added addresses. And I came up with the corpus score which is the sum of score of each email communication to the existing dyad. Basically when C emails to A, this email's score will be the sum of ANOVA F-value score of (A, B), (A, C), (A, N) and so on. This way also solves the problem that when combining individual features and high dimensional feature set, high-dimension set can shadow other features. For example if A's “link_to” feature set can look like:

{“Set(A, B)”: 14, → non-directional dyad, directional email count
“Set(A, C)”: 53,
“Set(A, N)”: 7}

And the scores of these features are {“Set(A, B)”: 10.65, “Set(A, C)”: 0.17, “Set(A, N)”: 3.28}

A's corpus score will be 14*10.65 + 53*0.17 + 7*3.28 = 135.65

For the code to generate “corpus_score,” see the block starts at line 102 in trial_corpus.py.

Here are real sample “corpus_score” data points from Enron corpus (Refer to trial_corpus.py for code to generate corpus_score.):

[('steven.kean@enron.com', {'corpus_score': 362644.16755031369, 'poi': False}),
('louise.kitchen@enron.com', {'corpus_score': 334612.30509264645, 'poi': False}),
('john.lavorato@enron.com', {'corpus_score': 314304.49524639547, 'poi': False}),
('david.delainey@enron.com', {'corpus_score': 270333.36290661368, 'poi': True}),
('paula.rieker@enron.com', {'corpus_score': 243415.30526072075, 'poi': True}),
('sally.beck@enron.com', {'corpus_score': 241187.91731730322, 'poi': False}),
('kenneth.lay@enron.com', {'corpus_score': 205053.63241625292, 'poi': True}),
('mike.mcconnell@enron.com', {'corpus_score': 159922.86115578224, 'poi': False}),
('richard.shapiro@enron.com', {'corpus_score': 137388.20200602108, 'poi': False}),
('john.sherriff@enron.com', {'corpus_score': 100741.00386136328, 'poi': False})]

So for Enron corpus, the feature selected will be the only “corpus_score” which is a result of the engineered social relationship.

Quick summary, along with the best feature selection from financial data – [“bonus”, “exercised_stock_options”], I'll try with this additional “corpus_score” to see if any improvement can be made. And because “bonus” and “exercised_stock_options” are in US Dollar and “corpus_score” is score total, I'll first log transform “bonus” and “exercised_stock_options,” append “corpus_score,” and scale all together to use all these 3 features for training.

3. Algorithm selection.

After combining “bonus,” “exercised_stock_options” and “corpus_score,” I did a quick check to see how many didn't have “corpus_score” and how many has 0 “corpus_score.” I think it's ok as the new feature also reflects the skewness of the POIs.

	No Corpus_Score	Corpus_Score	Total
No. of POIs	4	14	18
No. of non-POIs	55	72	127
Total	59	33	145

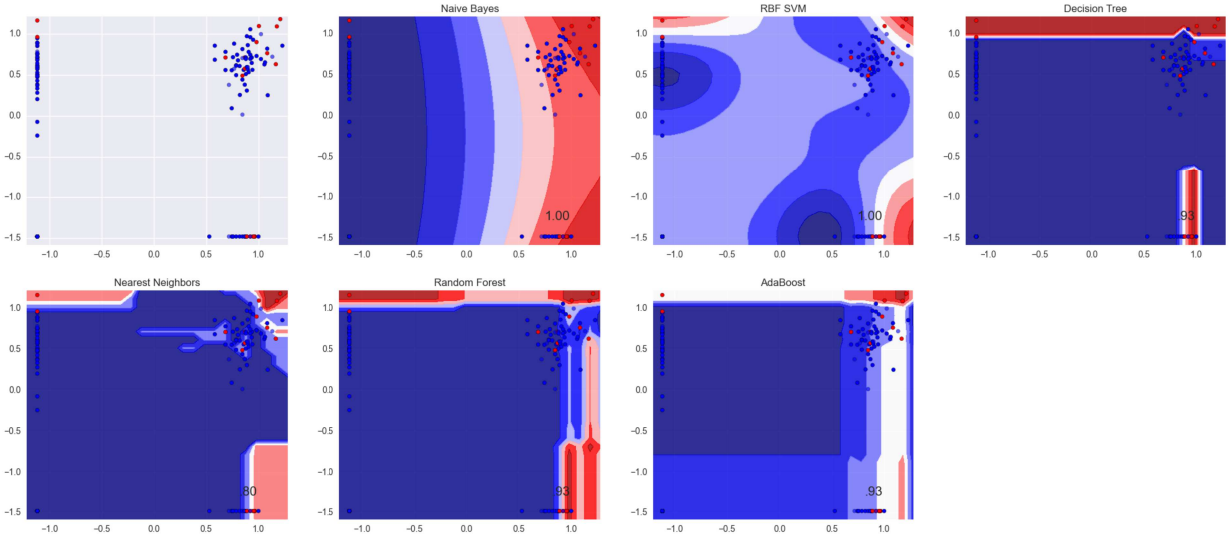
And the performance of these transformed and scaled feature set: [“bonus”, “exercised_stock_options”, “corpus_score”] (10% test size 1000 folds).

	Naive Bayes	RBF SVM	Decision Tree	Nearest Neighbors	Random Forest	AdaBoost
Accuracy	0.85260	Divide by zero error	0.85027	0.87407	0.86680	0.85767
Precision	0.38106		0.43351	0.53990	0.50095	0.45779
Recall	0.16900		0.40100	0.37550	0.26500	0.36600
F1	0.23415		0.41662	0.44294	0.34663	0.40678
F2	0.19017		0.40711	0.39985	0.29256	0.38129

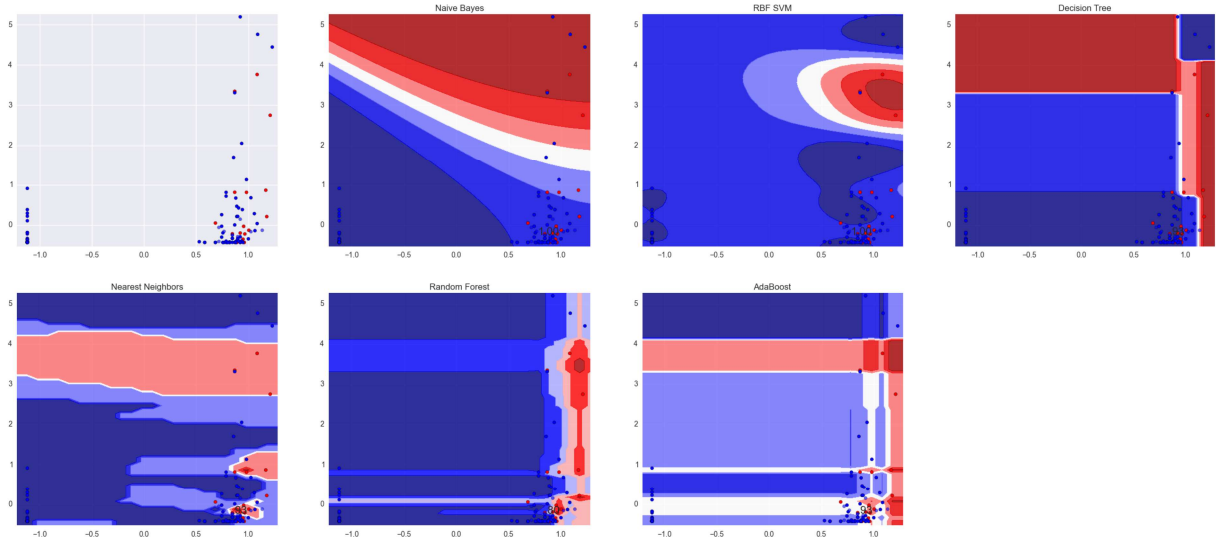
P.S. Even by just using raw [“bonus”, “exercised_stock_options”] I could already reach precision = 0.65243 and recall = 0.38950, I'm continuing using the 3 transformed/scaled features [“bonus”, “exercised_stock_options”, “corpus_score”] for more readable visualizations, comparisons, consistency and project completeness. Either way the performance will pass required level, but I want to have more learning experience.

I plotted the decision boundaries for these transformed and scaled feature set: [“bonus”, “exercised_stock_options”, “corpus_score”] to see the differences.

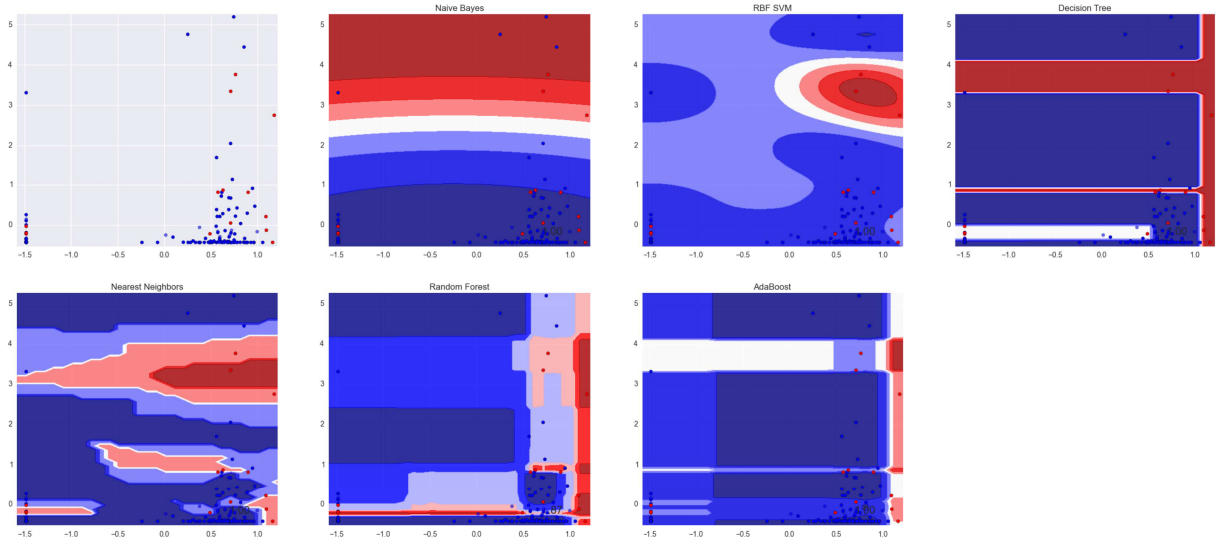
X: bonus, Y: exercised_stock_options



X: bonus, Y: corpus_score



X: exercised_stock_options, Y: corpus_score



During feature selection process, I tested with 6 algorithms: Naive Bayes, RBF SVM, Decision Tree, Nearest Neighbors, Random Forest and AdaBoost. I used same default configuration for all the tests. In general, AdaBoost was slow and didn't provide exceptional performance. RBF SVM more often failed to predict and threw divide-by-zero errors. Among others, Nearest Neighbors has been more reliable and tended to give higher precision. And Nearest Neighbors drew better boundaries for this scattering data pattern without requiring too much resource. So I'll keep going with Nearest Neighbors.

4. Parameter tuning.

While testing with scikit-learn KneighborsClassifier, I only made custom change to n_neighbors = 3, and that had given me pretty good performance. But if I use n_neighbors = 5 (default) the performance will go down. This is a good example for not properly tuning for different data characteristics, like this data set that doesn't have too many data points but skewed distribution (more non-POIs and less POIs). So if the algorithm is not properly tuned for the data characteristics, the model may not get a good bias-variance balance.

I used scikit-learn GridSearchCV to try a range of parameters for KneighborsClassifier. But since the data set is skewed (more non-POIs and less POIs), I'll need to tune the parameters for GridSearchCV too, majorly the scoring function.

KneighborsClassifier parameter range I used (keep it low as there are only 145 data points):

```
parameters = {"n_neighbors": [2, 5],
              "weights": ["uniform", "distance"],
              "p":[1, 2]}
```

GridSearchCV scoring methods I tried:

- scoring=None (default)
  best_params_={'n_neighbors': 2, 'weights': 'uniform', 'p': 1}
- scoring="average_precision" (scoring="precision" threw errors)
  best_params_={'n_neighbors': 2, 'weights': 'distance', 'p': 1}
- scoring="recall"
  best_params_={'n_neighbors': 2, 'weights': 'distance', 'p': 2}

I decided to keep these 2 configurations as they gave more meaningful performance:

- KNeighborsClassifier(n_neighbors=2, weights="distance", p=1)
- KneighborsClassifier(n_neighbors=3) → the randomly chosen value for the testing of classifiers earlier

5. Validation, classic mistake and strategy.

Validation is used to measure the quality of the model. By saying “how good” it actually means how good the desired results it can predict. For example, if you want to find as many POIs as possible for further inspection, you will want higher recall. If you want to find the most likely POIs to reduce further inspection, you will want higher precision.

One classic mistakes can be testing on training set for validation, this makes it hard to find over-fitting scenario as the mode is always optimized for training set. A testing set is needed to ensure the model is also generalized. Only using accuracy for skewed data can be problematic too as even pure guessing can very possibly grant high accuracy. So precision and recall are here to clarify real performance. And if training/test set are coming from a not well shuffled data, the model can be optimized for skewed data which will work really bad if the test set is balanced. Stratified sampling is necessary to make sure the balance of data splitting.

In addition to k-folds validation (500 or 1000 folds) with scikit-learn StratifiedShuffleSplit and training – testing set splitting (10% test size, both already in tester.py), I augmented it to calculate more performance measurements. All the measurements I used are:

- Accuracy – a reference, not really useful for the skewed data set (best value at 1 and worst at 0)
- Precision – fraction of all predicted positives to be real positives (best value at 1 and worst at 0)
- Recall – fraction identified of all real positives (best value at 1 and worst at 0)
- F1 – harmonic mean of precision and recall (best value at 1 and worst at 0)
- F2 – harmonic mean of precision and recall, weights recall higher than precision (best value at 1 and worst at 0)
- Sensitivity – same as recall
- Specificity – fraction of all predicted negatives to be real negatives (best value at 1 and worst at 0)
- Positive likelihood – greater than 1 indicates higher association, close to 1 have little practical significance, less than 1 indicates opposite association
- Negative likelihood – same as positive likelihood
- Matthews Correlation Coefficient – correlation between the actual and predicted, a balanced measure which can be used even if the classes are of very different sizes (best value at 1 and worst at -1, 0 no significance)
- Discrimination power – normalized likelihood index (<= 1 poor, >=3 good, fair otherwise)
- Confusion matrix – counts of true_positives, false_positives, true_negatives, false_negatives

6. Evaluation metrics.

Table below has the result from running KneighborsClassifier with tuned parameters on 145 POIs with the transformed/scaled features [“bonus”, “exercised_stock_options”, “corpus_score”].

So you can see from the default case, on skewed data, it's really easy to get high accuracy while most other measures have bad scores. Even random guess (Matthews Cor ~ 0) can get high accuracy.

Comparing the first 2 columns, there is a trade-off between precision and recall. When I tuned the parameter n_neighbors=3, I got really high precision with less false positives. But the recall rate also dropped as there are less true positives. And when I tuned the parameters to n_neighbors=2, weights='distance', p=1, I got a more balanced performance results with more false positives.

Depending on if you want to catch more suspects for further inspection (higher recall) or to catch the more accurate suspects to reduce inspection work (higher precision), either configuration will provide good enough performance.

*Default parameters: (n_neighbors=5, weights='uniform', algorithm='auto', leaf_size=30, p=2, metric='minkowski', metric_params=None)

Tuned parameters	n_neighbors=2, weights='distance', p=1	n_neighbors=3	None (default)
Accuracy	0.86247	0.87407	0.82553
Precision	0.48302	0.53990	0.19785
Recall	0.44800	0.37550	0.10100
F1	0.46485	0.44294	0.13373
F2	0.45459	0.39985	0.11196
Sensitivity	0.44800	0.37550	0.10100
Specificity	0.92623	0.95077	0.93700
Positive likelihood	6.07299	7.62734	1.60317
Negative likelihood	1.67795	1.52245	1.04227
Matthews Cor	0.38643	0.38237	0.05129
Discrimination power	1.27987	1.35189	0.28304
Total folds	1000	1000	1000
Total predictions	15000	15000	15000
True positives	896	751	202
False positives	959	640	819
False negatives	1104	1249	1798
True negatives	12041	12360	12181