Specification Change Response

Specification 1

Univariate or recursive feature selection is deployed, or features are selected by hand (different combinations of features are attempted, and the performance is documented for each one). Features that are selected are reported and the number of features selected is justified. For an algorithm that supports getting the feature importances (e.g. decision tree) or feature scores (e.g. SelectKBest), those are documented as well. **Required** Nice work in mentioning that GridSearchCV was used to tune for k from SelectKBest. However, I noticed that 3 features were finally chosen and the explanations provided weren't that clear. Try to provide more detail on why the features were reduced 13 to 3. I see an explanation about the correlation between a few features, so provide a lot of detail on this step. Also, you could compare the performance achieved with 13 versus 3 features and choose the one that gives the better results

I've concluded that a 5-feature algorithm seems to be the way to go. After rerunning all my analyses with the ideal 5 features, my conclusion on the best algorithm has changed. See this section of my notebook report:

```
13 grid_search.fit(features, labels)
14 print "Number of Ideal Features:", grid_search.best_params_
```

```
Number of Ideal Features: {'kbest__k': 13}
```

Feature Selction: Selecting Best Features for Analyses

Features with 13 top scores were (highest score is listed first): *'exercised_stock_options', 'total_stock_value', 'bonus', 'salary', 'deferred_income', 'long_term_incentive', 'restricted_stock', 'total_payments', 'from_messages', 'loan_advances', 'expenses', 'other', and 'from_poi_to_this_person'*

We decided to exclude 'total_stock_value' and 'total_payments' because we expected them to be highly correlated with their compoenents. For instance, we expected 'exercised_stock_options' and 'total_stock_value' to be correlated because the former is a component of the latter.

So, we got down to selecting features with top 11 SelectKBest: *'exercised_stock_options', 'bonus', 'salary', 'deferred_income', 'long_term_incentive', 'restricted_stock', 'from_messages', 'loan_advances', 'expenses', 'other', and 'from_poi_to_this_person'*

The 11-feature model did not perform any better than the 3-feature model with a Naive Bayes algorithm. So, I ran top-n analyses (n= 3, 4, 5, ..., 11). Their performance metrics were noted below. Models with 4 and 5 variables seems to perform the best. The former and latter performed better in Precision nd Recall, respectively. I went with the 5-feature model due to higher f1 performance metrics.

```
pd.DataFrame([[0.84277, 0.48281, 0.30900, 0.37683],
              [0.86307, 0.53081, 0.35750, 0.42725],
              [0.86214, 0.52273, 0.40250, 0.45480],
              [0.84550, 0.45192, 0.38300, 0.41461],
              [0.85021, 0.47145, 0.40050, 0.43309],
              [0.76479, 0.24795, 0.31800, 0.27864],
              [0.77067, 0.23194, 0.31150, 0.26590],
              [0.77753, 0.23712, 0.30150, 0.26546],
              [0.77753, 0.23691, 0.30100, 0.26514]],
             columns = ['Accuracy','Precision', 'Recall', 'F1'],
             index = ['Top 3','Top 4','Top 5','Top 6','Top 7', \
                      'Top 8','Top 9','Top 10','Top 11'])
```

	Accuracy	Precision	Recall	F1
Top 3	0.84277	0.48281	0.3090	0.37683
Top 4	0.86307	0.53081	0.3575	0.42725
Top 5	0.86214	0.52273	0.4025	0.45480
Top 6	0.84550	0.45192	0.3830	0.41461
Top 7	0.85021	0.47145	0.4005	0.43309
Top 8	0.76479	0.24795	0.3180	0.27864
Top 9	0.77067	0.23194	0.3115	0.26590

Specification 2

Performance of the final algorithm selected is assessed by splitting the data into training and testing sets or through the use of cross validation, noting the specific type of validation performed. **Required** To pass this section, specify the name of the cross-validation method that's used to partition the data. Also, to achieve consistent results across both the poi_id.py and tester.py file, I recommend partitioning the data using the same cross-validation method in both files. In the poi_id.py file, I noticed that a train_test_split method was used. While this is acceptable, it isn't the best way to partition the data, especially due to the class imbalance problem that exists in the data.

Thank you for the comment on *train_test_split* method. I stuck to the *train_test_split* method because that seems to be asked method because it was noted as such in the *poi_id.py* template file. I'd definitely consider the *stratified shuffle split* method for future undertakings.

As for "specify the name of the cross-validation method", I've noted the *stratified shuffle split* cross validation method with some explanation in the free-response question sheet. i.e.,

5. What is validation, and what's a classic mistake you can make if you do it wrong? How did you validate your analysis?

Validation is the process of analyzing algorithm performance with the test data. The most common mistake is overfitting where the algorithm works great with the train data but poor with the remaining data. We have noticed that the precision and recall metrics based on confusion metrics changed as we altered the *test_size* percentage.

In the project, we run the tester.py algorithm to validate our findings. The validation method is named *stratified shuffle split* cross validation. The *test_classifier* function indicates that the performance metrics were calculated based on 1,000 runs with randomized train-test subsamples – all using the same Enron dataset. Based on tabulations of *true_negatives*, *false_negatives*, *false_positives*, and *true_positives* from these 1,000 runs, the tester.py program calculates accuracy, precision, f1, and f2 performance metrics. This method is preferred because it averages out performance metrics from a huge number of run. A train-test data sampling may be biased that we may end up with a train sample with zero POIs. This bias should be minimized with 1,000 runs with randomized train-test subsample data shuffling. A very important aspect of *stratified shuffle split* cross validation is that we stratify the sampling for POIs in both the train and test datasets. This is essential due to the small proportion of POIs – that some pure random sampling may result in the train or test subsamples getting zero POI observation.