Report

Introduction to Machine Learning Project

Identify Fraud from Enron Email

Prepared by: Jeff Daniels
November 30, 2017

Introduction:

This project intends to use machine learning tools to identify persons of interest (poi) from a sample of Enron employees. Widespread fraud occured at Enron resulting in a federal investigation resuting in several poi who were indicted, reached a plea deal, or testified in exchange for immunity. The federal investigation also yielded a large public dataset containing detailed financial information and emails of employees.

A hand generated list of poi will be used to train a classifier that will identify poi based on their financial data. Machine learning is useful becuase we have a quantified dataset of financial information that is hopefully related to the likelihood of an employee commiting fraud. If a satisfactory algorithm is created it can be used to find other poi from the thousands of employees at Enron and not just in the approximately 145 samples in this dataset. A really good algorithm might also be useful in detecting fraud at other companies.

In [1]:
```
#!/usr/bin/python2.7

"""
Setup Environment
"""

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

from tester import dump_classifier_and_data, test_classifier #
import tester

from helpers import construct_dataframe, create_new_features, dump_preperation, good_corr_features, \
                    good_corr_features_scores, k_best_features, load_dataset, outlier_table, prep_df, print_cv_scores, \
                    print_val_scores, process_nan, remove_all_zeros, remove_outliers, train_test_split

from sklearn.model_selection import GridSearchCV

from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LogisticRegression

from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

from sklearn.decomposition import PCA

from sklearn.feature_selection import SelectKBest
```

```
/usr/local/lib/python2.7/dist-packages/sklearn/cross_validation.py:41: DeprecationWarning: This module was deprecated in version 0.18 in favor of the model_selection module into which all the refactored classes and functions are moved. Also note that the interface of the new CV iterators are different from that of this module. This module will be removed in 0.20.
  "This module will be removed in 0.20.", DeprecationWarning)
```

Exploratory Data Analysis

The dataset is loaded and converted into a dataframe for easier analysis and manipulation.

In [2]:
```
data_dict = load_dataset()
enron = construct_dataframe(data_dict)
enron.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 146 entries, ALLEN PHILLIP K to YEAP SOON
Data columns (total 20 columns):
poi                          146 non-null int64
salary                       95 non-null float64
to_messages                  86 non-null float64
deferral_payments            39 non-null float64
total_payments               125 non-null float64
exercised_stock_options      102 non-null float64
bonus                        82 non-null float64
restricted_stock             110 non-null float64
shared_receipt_with_poi      86 non-null float64
restricted_stock_deferred    18 non-null float64
total_stock_value            126 non-null float64
expenses                     95 non-null float64
loan_advances                4 non-null float64
from_messages                86 non-null float64
other                        93 non-null float64
from_this_person_to_poi      86 non-null float64
director_fees                17 non-null float64
deferred_income              49 non-null float64
long_term_incentive          66 non-null float64
from_poi_to_this_person      86 non-null float64
dtypes: float64(19), int64(1)
memory usage: 24.0+ KB
```

In [3]:
```
"""
Percentage of Dataset that is null
"""
print "Percentage of Dataset that is null"
enron.isnull().sum().sum()/float(enron.shape[0]*enron.shape[1])
```

```
Percentage of Dataset that is null
```

Out[3]:
```
0.45308219178082193
```

In [4]:
```
"""
Percentage of dataset who are poi
"""
print "Percentage of dataset who are poi"
enron['poi'].mean()
```

```
Percentage of dataset who are poi
```

Out[4]:
```
0.12328767123287671
```

The dataset contains 146 people and 20 columns of features. There could potentially be 2920 data points, but 45% of the dataset contains null values. All but the poi feature contains null values. Because none of the classifiers accept null values, these data points will have to be addressed.

About 12% of the dataset are poi. This represents 18 people.

Split the Dataset into Training and Test Sets

It is generally good practice to seperate the dataset into training and test sets. It is the first step in validating a classifier using a training and test data split. Other validation techniques are available, but working with a training set will reduce the "data snooping" bias. A stratified shuffle split will be used so that the same proportion of poi will occur in the training and test sets.

In [5]:
```
train, test = train_test_split(enron)
df = train.copy()
```

Remove Outliers

Let us start by identifying the outliers in the dataset. I have created a function that returns a table of employees whose features are more than 3 standard deviations from the mean of that feature.

In [6]:
```
outlier_table(df)
```

Out[6]:

	Outlier Feature	Value	poi
TOTAL	salary	26704229	0
KEAN STEVEN J	to_messages	12754	0
TOTAL	deferral_payments	32083396	0
TOTAL	total_payments	309886585	0
TOTAL	exercised_stock_options	311764000	0
TOTAL	bonus	97343619	0
TOTAL	restricted_stock	130322299	0
WHALLEY LAWRENCE G	shared_receipt_with_poi	3920	0
BHATNAGAR SANJAY	restricted_stock_deferred	15456290	0
TOTAL	total_stock_value	434509511	0
TOTAL	expenses	5235198	0
KAMINSKI WINCENTY J	from_messages	14368	0
TOTAL	other	42667589	0
KEAN STEVEN J	from_this_person_to_poi	387	0
DELAINEY DAVID W	from_this_person_to_poi	609	1
TOTAL	deferred_income	-27992891	0
TOTAL	long_term_incentive	48521928	0
DIETRICH JANET R	from_poi_to_this_person	305	0

This man/woman named 'TOTAL' is an outlier in 11 categories. We know from the course that this is a was probably a data entry error. Likely 'TOTAL' was calculated in a spreadsheet and the exported .csv file contained it. The best thing to do is simply discard it.

In [7]:
```
if 'TOTAL' in df.index:
    df = df.drop(['TOTAL'])
outlier_table(df)['poi'].mean()
outlier_table(df)
```

Out[7]:

	Outlier Feature	Value	poi
SKILLING JEFFREY K	salary	1111258	1
LAY KENNETH L	salary	1072321	1
KEAN STEVEN J	to_messages	12754	0
LAY KENNETH L	total_payments	103559793	1
HIRKO JOSEPH	exercised_stock_options	30766064	1
LAY KENNETH L	exercised_stock_options	34348384	1
SKILLING JEFFREY K	bonus	5600000	1
LAY KENNETH L	bonus	7000000	1
PAI LOU L	restricted_stock	8453763	0
LAY KENNETH L	restricted_stock	14761694	1
WHALLEY LAWRENCE G	shared_receipt_with_poi	3920	0
BHATNAGAR SANJAY	restricted_stock_deferred	15456290	0
HIRKO JOSEPH	total_stock_value	30766064	1
LAY KENNETH L	total_stock_value	49110078	1
URQUHART JOHN A	expenses	228656	0
KAMINSKI WINCENTY J	from_messages	14368	0
LAY KENNETH L	other	10359729	1
KEAN STEVEN J	from_this_person_to_poi	387	0
DELAINEY DAVID W	from_this_person_to_poi	609	1
RICE KENNETH D	deferred_income	-3504386	1
MARTIN AMANDA K	long_term_incentive	5145434	0
LAY KENNETH L	long_term_incentive	3600000	1
DIETRICH JANET R	from_poi_to_this_person	305	0

Removing TOTAL seems to give us a better set of outliers. In fact, this group of outliers is 61% poi. This is probably a case when outliers represent something exceptional, such as fraud, that we are trying to identify. Therefore, I will keep these samples in the dataset.

So for now, the only outlier we ignore is TOTAL. The problem with this removing this outlier is supposing what would happen if TOTAL was in the test set. Then this outlier wouldn't have been removed and a bad classification may have occured. Are there other outliers that are in the test set that won't be considered?

In [8]:
```
df = remove_outliers(df)
```

Create new features

New features were created that illustrated the ratio of emails from, sent to, or shared receipt with a poi to the total number of emails received or sent. Hopefully this would represent some sort of measure of collusion between parties to commit fraud.

In [9]:
```
df = create_new_features(df)
```

Process the NaN Values

Something must be done about the null values in the dataset. There already exist zero values in the dataset so null values are somehow different. I will assume that these values are null because of lack of information. The options are to set all of them to zero, or to set them to a median value for that feature. I decided to go with set them all to zero.

In [10]:
```
df = process_nan(df)
```

Remove all Zeros

The tester code has an option that is exercised to remove any rows where all the values are zero. It seems like a good idea to carry out this practice in our EDA.

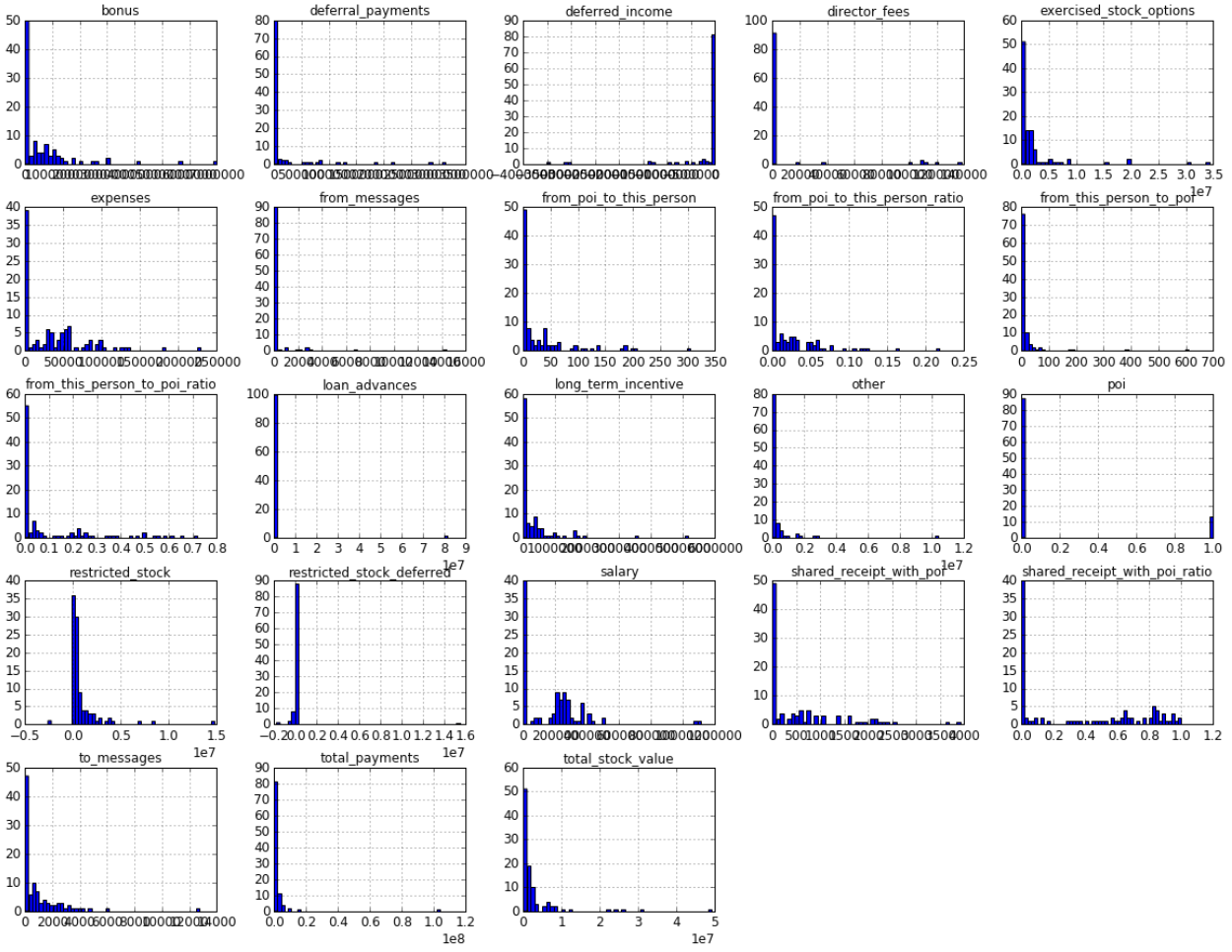
In [11]:
```
df = remove_all_zeros(df)
```

Look at the clean data

Histogram plots show that most of the features are right skewed. This is shown to be the case even with the x-axis displayed with a log scale. Most values for a feature are zero or close to it. Some sort of scaling might be useful to address the skew, either a log scale or a standardization.

In [35]:

```
"""
Histogram Plots of the data
"""

plt.rcParams['axes.labelsize'] = 14
plt.rcParams['xtick.labelsize'] = 12
plt.rcParams['ytick.labelsize'] = 12

df.hist(bins=50, figsize=(20,15), log = False)
plt.show()
```



Validate three classifiers

Validation measures the ability of a classifier to generalize data. Typically, the classifier is trained on one set of data and then the performance of that classifier is evaluated on a seperate test set of data. Validation should prevent a classifier from overfitting the data. A model can be created from the training data that accurately classifies all the samples. If this model is then applied to another dataset and poorly classifies that data, then overfitting has probably occurred. The purpose of validation is to help a classifier generalize not just training data well, but all data. The goal is for the performance of a classifier on the training set to be similar to the performance on the test set.

A Gaussian Naive Bayes, Decision Tree, and logistic regression classifiers were considered. All were trained on the training set and validated by running the classifier with the test set data. The two performance metrics I have chosen to evaluate the classifiers are precision and recall. Precision is the probability that a person predicted to be a poi is actually a poi. It is the ratio of true positives to the sum of true positives and false positives:

$$precision = \frac{tp}{tp + fp}$$

Recall is the ratio of true positives to the sum of true positives and false negatives. It is the fraction of poi in our dataset whom the classifier predicted were poi:

$$recall = \frac{tp}{tp + fn}$$

Because poi made up about 12% of the dataset, it would be easy to create a classifier that identified all samples as non-poi and achieve an 88% accuracy. Then the algorithm would find no poi but have high precision. If you want to identify all poi, then a high recall rate is desireable but a lazily constructed algorithm would just classify all samples as poi. Practically, it would be expensive to investigate everyone. Finding a balance between precision and recall is a classic classifier conundrum. An easy way to consider both precision and recall in validating a classifier is to use the harmonic mean of the two scores which is known as the f1 score:

$$f1 = \frac{2tp}{2tp + fp + fn}$$

In [13]:
```
"""
Train three classifiers on the training set
Test them on the Test set
Display a dataframe with evaluation scores
"""

"""
Prepare the training and test sets for classifiers
"""
features_train, labels_train = prep_df(train)
features_test, labels_test = prep_df(test)


"""
Summarize Classifiers before tuning
"""

val_before_tuning = \
pd.DataFrame(columns = ["accuracy", "precision", "recall", "f1"])

# GaussianNB classifier
clf = GaussianNB()
val_before_tuning = \
val_before_tuning.append(print_val_scores(clf, features_train, features_test,
                                          labels_train, labels_test,
                                          print_out = False))

# Decision Tree Classifier
clf = DecisionTreeClassifier(random_state = 42)
val_before_tuning = \
val_before_tuning.append(print_val_scores(clf, features_train, features_test,
                                          labels_train, labels_test,
                                          print_out = False))

# Logistic Regression Classifier
clf = LogisticRegression()
val_before_tuning = \
val_before_tuning.append(print_val_scores(clf, features_train, features_test,
                                          labels_train, labels_test,
                                          print_out = False))

val_before_tuning.index = ['Gaussian Naive Bayes', 'Decision Tree', 'Logistic Regression']

print "Validation Scores before Tuning and Feature Selection"
val_before_tuning
```

```
Validation Scores before Tuning and Feature Selection
```

Out[13]:

	accuracy	precision	recall	f1
Gaussian Naive Bayes	0.772727	0.000000	0.0	0.000000
Decision Tree	0.795455	0.166667	0.2	0.181818
Logistic Regression	0.727273	0.000000	0.0	0.000000

Validation using a training and test data split have given us some poor results. The Naive Bayes and Logistic Regression classifiers both had precision and recall equal to zero. This is partly because the classifiers may be inappropriate or could use some tuning. It may also be because the validation method isn't appropriate. Ultimately the classifier I choose will be validated using the test_classifier function in tester. test_classifier uses a cross validation technique where the entire dataset is divided into training and test sets for multiple iterations and returns a mean of the validation scores for all iterations. test_classifier uses StratifiedShuffleSplit to split the dataset into a training set of 90% of the samples and a test set of 10%. There are 1000 iterations of this split with a different training and test sets each time. This cross validation strategy is useful for smaller datasets like ours since it reuses the data several times.

For comparison, let us see how cross validation scores from test_classifier compare to our simple validation.

In [14]:
```
"""
test_classifier performance before tuning and feature selection
"""

my_dataset, features_list = dump_preperation(enron, list(features_train.columns))

test_classifier(GaussianNB(), my_dataset, features_list)
test_classifier(DecisionTreeClassifier(random_state=42), my_dataset, features_list)
test_classifier(LogisticRegression(), my_dataset, features_list)
```

```
GaussianNB(priors=None)
        Accuracy: 0.74713       Precision: 0.23578      Recall: 0.40000 F1: 0.29668     F2: 0.351
09
        Total predictions: 15000        True positives:  800    False positives: 2593   False neg
atives: 1200    True negatives: 10407

DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, presort=False, random_state=42,
            splitter='best')
        Accuracy: 0.81413       Precision: 0.29020      Recall: 0.27250 F1: 0.28107     F2: 0.275
87
        Total predictions: 15000        True positives:  545    False positives: 1333   False neg
atives: 1455    True negatives: 11667

LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
          verbose=0, warm_start=False)
        Accuracy: 0.78233       Precision: 0.17812      Recall: 0.17500 F1: 0.17654     F2: 0.175
61
        Total predictions: 15000        True positives:  350    False positives: 1615   False neg
atives: 1650    True negatives: 11385
```

In [15]:
```
test_classifier_before_tuning = \
pd.DataFrame([[0.747, 0.236, 0.400, 0.297, 0.351],
            [0.814, 0.290, 0.273, 0.281, 0.276],
            [0.782, 0.178, 0.175, 0.177, 0.177]],
             columns = ["accuracy", "precision", "recall", "f1", "f2"],
            index = ['Gaussian Naive Bayes', 'Decision Tree', 'Logistic Regression'])

print "test_classifier Scores before Tuning and Feature Selection"
test_classifier_before_tuning
```

```
test_classifier Scores before Tuning and Feature Selection
```

Out[15]:

	accuracy	precision	recall	f1	f2
Gaussian Naive Bayes	0.747	0.236	0.400	0.297	0.351
Decision Tree	0.814	0.290	0.273	0.281	0.276
Logistic Regression	0.782	0.178	0.175	0.177	0.177

Indeed, the cross validation is a little more forgiving and probably more accurate than validating with just one train/test split. Still, none of the classifiers meet the rubric requirement of precision and accuracy greater than 0.3. The next section will try to improve classifier performance through feature selection and hyperparameter tuning.

Tuning Hyperparameters and Performance Metric

Tuning the classifiers means adjusting the hyperparameters to optimize classifier performance. Hyperparameters are parameters that are set before training the classifier on the data. The data can change, the classifier will change its coefficients, but the hyperparameters will stay the same. The classifier performance will be judged on f1 score. By maximizing the f1 score, the precision and recall scores will both hopefully be indidually maximized as well. Also tuning with more than one scoring criteria is inconvenient.

As explained before, cross validation is a good way of reusing data to get what is hopefully a more accurate performance score. For tuning the classifiers, cross validation will be used to evaluate the performance of the classifier hyperparameters. Generally a 3 fold cross validation will be used for tuning but only on the training data so that the classifier can be validated again on the test data.

Tune the Gaussian Classifier

There are not many tuning parameters in the Gaussian Naive Bayes classifier. I'll do my best to pick out good features. good_corr_features ranks the features that highly correlate with the 'poi' label then creates subsets of the features, returning the subsets features that performed best. The first subset contains only the highest correlating feature, the second has the two highest correlating features, and so on until the number of subsets equals the number of features.

In [16]:
```
print 'Correlation of Features with poi'
df.corr()['poi'].sort_values(ascending = False)
```

```
Correlation of Features with poi
```

Out[16]:
```
poi                          1.000000
exercised_stock_options      0.494995
total_stock_value            0.493908
salary                       0.447438
bonus                        0.437867
long_term_incentive          0.374413
restricted_stock             0.370327
from_this_person_to_poi_ratio    0.342256
total_payments               0.282558
loan_advances                0.259998
shared_receipt_with_poi      0.252883
other                        0.249747
shared_receipt_with_poi_ratio    0.230686
from_poi_to_this_person      0.199290
expenses                     0.197379
from_this_person_to_poi      0.188880
to_messages                  0.125637
from_poi_to_this_person_ratio    0.105680
from_messages               -0.027681
restricted_stock_deferred   -0.027782
director_fees               -0.112199
deferral_payments           -0.117645
deferred_income             -0.312149
Name: poi, dtype: float64
```

In [17]:
```
"""
Run good_corr_features and display the list of features selected
"""
list(good_corr_features(features_train, labels_train, clf = GaussianNB()).columns)
```

Out[17]:
```
['exercised_stock_options']
```

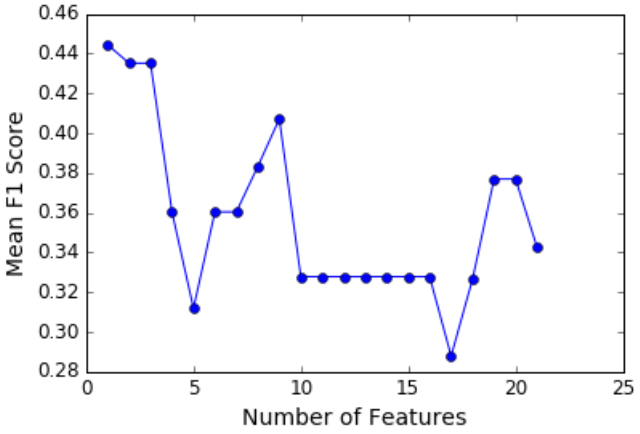
So it appears that my feature selection function good_corr_features returned a list containing only one feature. It will be interesting to see if this was an effective strategy.

In [18]:
```
"""
Automated Feature Selection for Gaussian NB
"""

print "Before Feature Selection"
print_cv_scores(GaussianNB(), features_train, labels_train)

"""
List the features that correspond to the highest f1 score
"""
good_gnb_features_train = good_corr_features(features_train, labels_train, clf = GaussianNB())

print ""
print 'Good Correlated Features'
print list(good_gnb_features_train.columns)
print ""
print "After Feature Selection"
gnb_fs_score = print_cv_scores(GaussianNB(), good_gnb_features_train, labels_train)
gnb_fs_score.index = ['Gaussian Naive Bayes w/ Feature Selection']

print ""
print "Validation on Test Set"
good_gnb_features_test = features_test[list(good_gnb_features_train.columns)]
_ = print_val_scores(GaussianNB(), good_gnb_features_train, good_gnb_features_test, labels_train,
 labels_test)
```

```
Before Feature Selection
Mean Cross Validation Scores
accuracy 0.699049316696
precision 0.272222222222
recall 0.466666666667
f1 0.299783549784

Good Correlated Features
['exercised_stock_options']

After Feature Selection
Mean Cross Validation Scores
accuracy 0.879976232917
precision 0.666666666667
recall 0.4
f1 0.444444444444

Validation on Test Set
Validation Scores
accuracy 0.840909090909
precision 0.0
recall 0.0
f1 0.0
```

The validation scores haven't changed. The f1 score is 0.0. This may be an example of overfitting. The set of features chosen based on the training set didn't work well on the test set. Feature selection can be an art of finding the happy medium between too few features and too many features which can both cause overfitting. Let us examine the mean f1 scores found from the cross validation versus the number of features.

In [19]:
```
"""
Plot Mean F1 Score from cross validation vs. Number of features
"""

results_df = good_corr_features_scores(features_train, labels_train, GaussianNB())
fig = plt.figure()
plt.plot(results_df, marker = 'o')
plt.xlabel('Number of Features')
plt.ylabel('Mean F1 Score')
plt.show()
```



A feature list with only one feature does appear to have the best performance but I would like to ignore feature lists that have too little and too many features. There is a local maxima in performace where the number of features is 9 which looks promising. Lets retrain and validate using this new set of features.

In [20]:
```
"""
Manual Feature Selection for GaussianNB
Select the top 9 correlated features
"""
num_gnb_features = 9


print "Before Feature Selection"
print_cv_scores(GaussianNB(), features_train, labels_train)

"""
List the features selected
"""
top_corr = list(abs(features_train.corrwith(labels_train)).sort_values(
            ascending = False).index)
good_gnb_features_train = features_train[top_corr[0:num_gnb_features]]
good_gnb_features_test = features_test[list(good_gnb_features_train.columns)]

print ""
print 'Features selected with good correlations to POI'
print list(good_gnb_features_train.columns)
print ""
print "After Feature Selection"
gnb_fs_score = print_cv_scores(GaussianNB(), good_gnb_features_train, labels_train)
gnb_fs_score.index = ['Gaussian Naive Bayes w/ Feature Selection']

print ""
print "Validation on Test Set"
_ = print_val_scores(GaussianNB(), good_gnb_features_train, good_gnb_features_test, labels_train,
 labels_test)
```

```
Before Feature Selection
Mean Cross Validation Scores
accuracy 0.699049316696
precision 0.272222222222
recall 0.466666666667
f1 0.299783549784

Features selected with good correlations to POI
['exercised_stock_options', 'total_stock_value', 'salary', 'bonus', 'long_term_incentive', 'restr
icted_stock', 'from_this_person_to_poi_ratio', 'deferred_income', 'total_payments']

After Feature Selection
Mean Cross Validation Scores
accuracy 0.849376114082
precision 0.527777777778
recall 0.383333333333
f1 0.407142857143

Validation on Test Set
Validation Scores
accuracy 0.840909090909
precision 0.25
recall 0.2
f1 0.222222222222
```

Manually selecting the top 9 features with the highest correlation to poi decreased the cross validation scores but increased the validation scores on the test set. This new set of features includes one new feature 'from_this_person_to_poi_ratio' which compares the number of emails from this person to a poi to total emails sent by this person. The new feature ranks 7/9 for correlation with poi among the features selected. The features selected and their correlations with poi are below:

In [21]:
```
abs(features_train.corrwith(labels_train)).sort_values(
            ascending = False)[0:num_gnb_features]
```

Out[21]:
```
exercised_stock_options          0.494995
total_stock_value                0.493908
salary                           0.447438
bonus                            0.437867
long_term_incentive              0.374413
restricted_stock                 0.370327
from_this_person_to_poi_ratio    0.342256
deferred_income                  0.312149
total_payments                   0.282558
dtype: float64
```

Now that I have chosen some features, let us try implementing the Naive Bayes classifier with or without principle component analysis. This will be done by implementing a Pipeline along with GridSearchCV which will test all the combinations of hyperparameters that I provide. Along with determining whether PCA improves performance, the PCA hyperparameter n_components will be tuned so that hopefully the best version of PCA will be compared to not using PCA.

In [22]:

```
"""
Tune the Naive Bayes Classifier
"""
cv_folds = 3

pipe = Pipeline([
    ('reduce_dim', PCA()),
    ('classify', GaussianNB())
])

N_FEATURES_OPTIONS = [2, 3, 4,5, 6, 7, 8]

param_grid = [{'reduce_dim': [PCA(random_state=42)],
               'reduce_dim__n_components': N_FEATURES_OPTIONS},
              {'reduce_dim': [None]}
             ]

estimator = GridSearchCV(pipe, param_grid = param_grid, cv = cv_folds, scoring = 'f1')
estimator.fit(good_gnb_features_train, labels_train)

clf_gnb = estimator.best_estimator_

print "Before Tuning"
print_cv_scores(GaussianNB(), good_gnb_features_train, labels_train, cv = cv_folds)
print ""
print "After Tuning"
dt_tune_score = print_cv_scores(clf_gnb, good_gnb_features_train, labels_train, cv = cv_folds)
dt_tune_score.index = ['Gaussian Naive Bayes w/ Tuning']
print""
_ = print_val_scores(clf_gnb, good_gnb_features_train, good_gnb_features_test,
                                                    labels_train, labels_test)

print ""
clf_gnb.named_steps
print ""

my_dataset, features_list = dump_preperation(enron, list(good_gnb_features_train.columns))
test_classifier(clf_gnb, my_dataset, features_list)
```

```
Before Tuning
Mean Cross Validation Scores
accuracy 0.849376114082
precision 0.527777777778
recall 0.383333333333
f1 0.407142857143

After Tuning
Mean Cross Validation Scores
accuracy 0.82917409388
precision 0.527777777778
recall 0.466666666667
f1 0.434920634921

Validation Scores
accuracy 0.840909090909
precision 0.25
recall 0.2
f1 0.222222222222


Pipeline(memory=None,
     steps=[('reduce_dim', PCA(copy=True, iterated_power='auto', n_components=6, random_state=42,
  svd_solver='auto', tol=0.0, whiten=False)), ('classify', GaussianNB(priors=None))])
        Accuracy: 0.85653       Precision: 0.45202      Recall: 0.35800 F1: 0.39955     F2: 0.373
54
        Total predictions: 15000        True positives:  716    False positives:  868   False neg
atives: 1284    True negatives: 12132
```

GridSearchCV has shown us that using PCA with n_components = 6 improves recall. Running this new classifier with the 9 features selected also yields better test_classifier performance. Manual feature selection and PCA improved the performance of the Gaussian Naive Bayes classifier to where it now meets the requirements for submission.

Tune the Decision Tree Classifier

The Decision Tree Classifier has the attribute feature*importances* which can be used in conjunction with SelectKBest to select the features. SelectKBest is feature selection method that selects the best features based on univariate statistical tests. Using GridSearchCV, different numbers of features can be tested in the SelectKBest preprocessing step. The decision tree classifier also has many more hyperparameters than the Gaussian Naive Bayes classifier so it is well suited to GridSearchCV using parameter grids with larger dimensions. The classifier will be evaluated with or without a scaler and using either SelectKBest or PCA to reduce dimensionality. Hyperparameters will be tuned as well.

In [32]:

```
"""
Tune the Decision Tree Classifier
"""
cv_folds = 3

pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('reduce_dim', SelectKBest()),
    ('classify', DecisionTreeClassifier(random_state = 42))
])

N_FEATURES_OPTIONS = [2, 4, 8, 16, 20]
MAX_DEPTH_OPTIONS = [2, 4, 6, 8]
MIN_SAMPLES_OPTIONS = [1,2, 4, 8]
CRITERION_OPTIONS = ['entropy', 'gini']

param_grid = [{'reduce_dim': [PCA(random_state=42)],
               'reduce_dim__n_components': N_FEATURES_OPTIONS,
               'classify__criterion': CRITERION_OPTIONS,
               'classify__max_depth': MAX_DEPTH_OPTIONS,
              'classify__min_samples_leaf': MIN_SAMPLES_OPTIONS},
              {'scaler': [None],
               'reduce_dim': [PCA(random_state=42)],
               'reduce_dim__n_components': N_FEATURES_OPTIONS,
               'classify__criterion': CRITERION_OPTIONS,
               'classify__max_depth': MAX_DEPTH_OPTIONS,
              'classify__min_samples_leaf': MIN_SAMPLES_OPTIONS},
              {'reduce_dim': [SelectKBest()],
               'reduce_dim__k': N_FEATURES_OPTIONS,
               'classify__criterion': CRITERION_OPTIONS,
               'classify__max_depth': MAX_DEPTH_OPTIONS,
              'classify__min_samples_leaf': MIN_SAMPLES_OPTIONS},
              {'scaler': [None],
                  'reduce_dim': [SelectKBest()],
               'reduce_dim__k': N_FEATURES_OPTIONS,
               'classify__criterion': CRITERION_OPTIONS,
               'classify__max_depth': MAX_DEPTH_OPTIONS,
              'classify__min_samples_leaf': MIN_SAMPLES_OPTIONS}
             ]

estimator = GridSearchCV(pipe, param_grid = param_grid, cv = cv_folds, scoring = 'f1')
estimator.fit(features_train, labels_train)

clf_dt = estimator.best_estimator_

print "Before Tuning"
print_cv_scores(DecisionTreeClassifier(random_state=42), features_train, labels_train, cv = cv_fo
lds)
print ""
print "After Tuning"
dt_tune_score = print_cv_scores(clf_dt, features_train, labels_train, cv = cv_folds)
dt_tune_score.index = ['Decision Tree w/ Tuning']
print""
_ = print_val_scores(clf_dt, features_train, features_test,
                                                    labels_train, labels_test)

clf_dt.named_steps
```

```
Before Tuning
Mean Cross Validation Scores
accuracy 0.839572192513
precision 0.533333333333
recall 0.316666666667
f1 0.333333333333

After Tuning
Mean Cross Validation Scores
accuracy 0.889483065954
precision 0.633333333333
recall 0.533333333333
f1 0.564814814815

Validation Scores
accuracy 0.840909090909
precision 0.25
recall 0.2
f1 0.222222222222
```

Out[32]:
```
{'classify': DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=6,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, presort=False, random_state=42,
            splitter='best'),
 'reduce_dim': SelectKBest(k=8, score_func=<function f_classif at 0x7e8b1b2432a8>),
 'scaler': StandardScaler(copy=True, with_mean=True, with_std=True)}
```

It seems that decision tree classifier works best with StandardScaler and SelectKBest with 8 features. The validation scores improved slightly so it does not seem like overfiting has occured. The new feature 'from_this_person_to_poi_ratio' is included among the features chosen. The new feature has less than half the importance of some of the features chosen. The new feature ranks 7/8 in importance for the features selected. The features selected are listed below with their scores.

In [24]:
```
k_best_features(clf_dt, features_train)
```

Out[24]:
```
[(31.804845880114033, 'exercised_stock_options'),
 (31.620178218715953, 'total_stock_value'),
 (24.530714329629465, 'salary'),
 (23.246288774158632, 'bonus'),
 (15.978062688236019, 'long_term_incentive'),
 (15.576085413176642, 'restricted_stock'),
 (13.002739226503381, 'from_this_person_to_poi_ratio'),
 (10.579685358065349, 'deferred_income')]
```

In [25]:
```
"""
test_classifier on the tuned Decision Tree Classifier
kbest features selected
"""

my_dataset, features_list = dump_preperation(enron, list(features_train.columns))
test_classifier(clf_dt, my_dataset, features_list)
```

```
Pipeline(memory=None,
     steps=[('scaler', StandardScaler(copy=True, with_mean=True, with_std=True)), ('reduce_dim',
SelectKBest(k=8, score_func=<function f_classif at 0x7e8b1b2432a8>)), ('classify', DecisionTreeCl
assifier(class_weight=None, criterion='gini', max_depth=6,
            max_features=None, max_leaf_nodes=None,
...         min_weight_fraction_leaf=0.0, presort=False, random_state=42,
            splitter='best'))])
        Accuracy: 0.80860       Precision: 0.27139      Recall: 0.25850 F1: 0.26479     F2: 0.260
98
        Total predictions: 15000        True positives:  517    False positives: 1388   False neg
atives: 1483    True negatives: 11612
```

Running our tuned Decision Tree Classifier through the test_classifier function yields performance that does not meet the rubric requirements. How could this be? Why are cross validation scores from the training set so different from the validation scores on the test set and cross validation scores on the entire dataset? One reason is that my cross validation uses only 3 folds whereas test classifier uses 1000 folds. Three folds seems a reasonable number to use when tuning the classifier because it takes less time. However, using a 70/30 training/test set for validation may not be the best method for this particular dataset.

Tune the Logistic Regression Classifier

Feature selection for the Logistic Regression Classifier will be done with either PCA or SelectKBest. Tuning will also be done with or without StandardScaler, though it is good practice to always use it with logistic regression so that training runs faster.

In [33]:

```
"""
Tune the Logistic Regression
"""
cv_folds = 3

pipe = Pipeline([('scaler', StandardScaler()),
                 ('reduce_dim', SelectKBest()),
                 ('classify', LogisticRegression(random_state=42))])

N_FEATURES_OPTIONS = [2, 4, 8, 12]
C_OPTIONS = [0.0001, 0.001, 0.01, 0.1, 0.5, 1]
TOL_OPTIONS = [1e-06, 1e-5, 1e-4]
PENALTY_OPTIONS = ['l1', 'l2']

param_grid = [{'scaler': [None],
                 'reduce_dim': [PCA(random_state=42)],
                 'reduce_dim__n_components': N_FEATURES_OPTIONS,
                 'classify__tol': TOL_OPTIONS,
               'classify__C': C_OPTIONS,
               'classify__penalty': PENALTY_OPTIONS},
                 {'reduce_dim': [PCA(random_state=42)],
                 'reduce_dim__n_components': N_FEATURES_OPTIONS,
                 'classify__tol': TOL_OPTIONS,
               'classify__C': C_OPTIONS,
               'classify__penalty': PENALTY_OPTIONS},
               {'scaler': [None],
                 'reduce_dim': [SelectKBest()],
                 'reduce_dim__k': N_FEATURES_OPTIONS,
                 'classify__tol': TOL_OPTIONS,
               'classify__C': C_OPTIONS,
               'classify__penalty': PENALTY_OPTIONS},
               {'reduce_dim': [SelectKBest()],
                 'reduce_dim__k': N_FEATURES_OPTIONS,
                 'classify__tol': TOL_OPTIONS,
               'classify__C': C_OPTIONS,
               'classify__penalty': PENALTY_OPTIONS}
              ]

estimator = GridSearchCV(pipe, param_grid = param_grid, cv = cv_folds, scoring = 'f1')
estimator.fit(features_train, labels_train)

clf_lr = estimator.best_estimator_

print "Before Tuning"
print_cv_scores(LogisticRegression(), features_train, labels_train, cv = cv_folds)
print ""
print "After Tuning"
dt_tune_score = print_cv_scores(clf_lr, features_train, labels_train, cv = cv_folds)
dt_tune_score.index = ['Logistic Regression w/ Tuning']
print""
_ = print_val_scores(clf_lr, features_train, features_test,
                                                    labels_train, labels_test)

clf_lr.named_steps
```

```
Before Tuning
Mean Cross Validation Scores
accuracy 0.740344622698
precision 0.15873015873
recall 0.233333333333
f1 0.188888888889

After Tuning
Mean Cross Validation Scores
accuracy 0.839572192513
precision 0.46455026455
recall 0.683333333333
f1 0.547008547009

Validation Scores
accuracy 0.772727272727
precision 0.142857142857
recall 0.2
f1 0.166666666667
```

Out[33]:
```
{'classify': LogisticRegression(C=0.001, class_weight=None, dual=False, fit_intercept=True,
           intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
           penalty='l2', random_state=42, solver='liblinear', tol=1e-06,
           verbose=0, warm_start=False),
 'reduce_dim': SelectKBest(k=12, score_func=<function f_classif at 0x7e8b1b2432a8>),
 'scaler': StandardScaler(copy=True, with_mean=True, with_std=True)}
```

GridSearchCV chose SelectKBest with k=12 and to use StandardScaler. The other hyperparameters also seem reasonable. Like the other two classifiers, 'from_this_person_to_poi_ratio' was among the features selected. It has a nearly median feature score, ranking 7/12 in feature importance. The features selected are shown below with their scores.

In [27]:
```
k_best_features(clf_lr, features_train)
```

Out[27]:
```
[(31.804845880114033, 'exercised_stock_options'),
 (31.620178218715953, 'total_stock_value'),
 (24.530714329629465, 'salary'),
 (23.246288774158632, 'bonus'),
 (15.978062688236019, 'long_term_incentive'),
 (15.576085413176642, 'restricted_stock'),
 (13.002739226503381, 'from_this_person_to_poi_ratio'),
 (10.579685358065349, 'deferred_income'),
 (8.5030949081436145, 'total_payments'),
 (7.1049999999999969, 'loan_advances'),
 (6.6952546282121688, 'shared_receipt_with_poi'),
 (6.5192142392300685, 'other')]
```

In [28]:
```
my_dataset, features_list = dump_preperation(enron, list(features_train.columns))
test_classifier(clf_lr, my_dataset, features_list)
```

```
Pipeline(memory=None,
     steps=[('scaler', StandardScaler(copy=True, with_mean=True, with_std=True)), ('reduce_dim',
SelectKBest(k=12, score_func=<function f_classif at 0x7e8b1b2432a8>)), ('classify', LogisticRegre
ssion(C=0.001, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', random_state=42, solver='liblinear', tol=1e-06,
          verbose=0, warm_start=False))])
        Accuracy: 0.81273       Precision: 0.34081      Recall: 0.43300 F1: 0.38141     F2: 0.410
78
        Total predictions: 15000        True positives:  866    False positives: 1675   False neg
atives: 1134    True negatives: 11325
```

Running the Logistic Regression classifier through the test_classifier function yielded satisfactory performance. Although its f1 score is slightly lower than the Naive Bayes classifier, its recall score is higher. To decide between the two means deciding whether you value precision or recall more.

In [29]:
```
"""
Create a Table Summarizing Algorithms
"""


scores = np.array([[.77273, .00000, .00000, .00000, .74713, .23578, .40000, .29668, .35109],
          [.84091, .25000, .20000, .22222, .84667, .40446, .31750, .35574, .33177],
          [.79546, .16777, .20000, .18182, .81413, .29020, .27250, .28107, .27587],
          [.84091, .25000, .20000, .22222, .80860, .27139, .25850, .26479, .26098],
          [.72727, .00000, .00000, .00000, .78233, .17812, .17500, .17654, .17561],
          [.77272, .14286, .20000, .16667, .81273, .34081, .43300, .38141, .41078]])

summary = pd.DataFrame(scores,
                index = [['Gaussian Naive Bayes', 'Gaussian Naive Bayes','Decision Tree','Decisi
on Tree',
                          'Logistic Regression','Logistic Regression'],
                         ['Before Tuning','After Tuning','Before Tuning','After Tuning',
                          'Before Tuning','After Tuning']],
                columns = [['Train/Test', 'Train/Test', 'Train/Test', 'Train/Test',
                            'Cross Validation','Cross Validation', 'Cross Validation', 'Cross Va
lidation',
                            'Cross Validation'],
                           ['accuracy', 'precision', 'recall', 'f1',
                            'accuracy', 'precision', 'recall', 'f1', 'f2']])

summary
```

Out[29]:

		Train/Test				Cross Validation				
		accuracy	**precision**	**recall**	**f1**	**accuracy**	**precision**	**recall**	**f1**	**f2**
Gaussian Naive Bayes	**Before Tuning**	0.77273	0.00000	0.0	0.00000	0.74713	0.23578	0.4000	0.29668	0.35109
	After Tuning	0.84091	0.25000	0.2	0.22222	0.84667	0.40446	0.3175	0.35574	0.33177
Decision Tree	**Before Tuning**	0.79546	0.16777	0.2	0.18182	0.81413	0.29020	0.2725	0.28107	0.27587
	After Tuning	0.84091	0.25000	0.2	0.22222	0.80860	0.27139	0.2585	0.26479	0.26098
Logistic Regression	**Before Tuning**	0.72727	0.00000	0.0	0.00000	0.78233	0.17812	0.1750	0.17654	0.17561
	After Tuning	0.77272	0.14286	0.2	0.16667	0.81273	0.34081	0.4330	0.38141	0.41078

Discusion of Validation Methods

Two validation methods were used, splitting the data into training and test sets and cross validation using test_classifier. They yielded very different results. I suppose that cross validation was probably a more accurate way of validating the classifier, its 1000 folds reduces the variability associated with the training and test splits but it is also slower. It would be nice to see how well a train/test split validates a larger dataset, but cross validation seems to work well with this small dataset.

Validate with tester.main()

This section will prepare the data for tester.py, the evaluation script for the algorithm that has been provided for the project.

I settled on using the Logistic Regression Classifier because it has a higher recall. This means more actual poi were identified than with the Naive Bayes Classifier which had lower recall but higher precision. I felt that a higher recall is more important so that fewer guilty people get away. However, this also means you are wasting time investigating more innocent people. Another attribute of Logistic Regression is the probability of a certain person being a poi which seems like interesting information.

Let us review what features were selected for this classifier. Recall the SelectKBest was use and found that 12 features shown below and sorted by importance. One new feature that I created, 'from_this_person_to_poi_ratio', will be used by the classifier.

In [34]:
```
k_best_features(clf_lr, features_train)
```

Out[34]:
```
[(31.804845880114033, 'exercised_stock_options'),
 (31.620178218715953, 'total_stock_value'),
 (24.530714329629465, 'salary'),
 (23.246288774158632, 'bonus'),
 (15.978062688236019, 'long_term_incentive'),
 (15.576085413176642, 'restricted_stock'),
 (13.002739226503381, 'from_this_person_to_poi_ratio'),
 (10.579685358065349, 'deferred_income'),
 (8.5030949081436145, 'total_payments'),
 (7.1049999999999969, 'loan_advances'),
 (6.6952546282121688, 'shared_receipt_with_poi'),
 (6.5192142392300685, 'other')]
```

The script tester.py was provided to validate whether the classifier chosen is adequate for submission. We will now dump the dataset, features, and labels in the required format for tester.main().

In [30]:
```
"""
Dump the classifier, my_dataset, and features_list
Run tester.main()
"""

my_dataset, features_list = dump_preperation(enron, list(features_train.columns))
dump_classifier_and_data(clf_lr, my_dataset, features_list)
tester.main()
```

```
Pipeline(memory=None,
     steps=[('scaler', StandardScaler(copy=True, with_mean=True, with_std=True)), ('reduce_dim',
SelectKBest(k=12, score_func=<function f_classif at 0x7e8b1b2432a8>)), ('classify', LogisticRegre
ssion(C=0.001, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', random_state=42, solver='liblinear', tol=1e-06,
          verbose=0, warm_start=False))])
        Accuracy: 0.81273       Precision: 0.34081      Recall: 0.43300 F1: 0.38141     F2: 0.410
78
        Total predictions: 15000        True positives:  866    False positives: 1675   False neg
atives: 1134    True negatives: 11325
```

References

"Hands-On Machine Learning with Scikit-Learn and TensorFlow by Aurelien Geron (O'Reilly). Copyright 2017 Aurelien Geron, 978-1-491-96229-9"

https://scikit-learn.org/stable/user_guide.html (https://scikit-learn.org/stable/user_guide.html)

Obviously a lot of time was spent looking at the sklearn documentation to complete this project. I won't bother to list all the pages I visited.

"Hands-On Machine Learning..." proved to be an excellent text for the course. Most of this report was written alongside the "Chapter-2: End-to-End Machine Learning Project" which would explain some of the idiosyncracies in how I created dataframes to analyze the data and implement machine learning rather than rely on tester.main().