Our Mission

In this lesson you gained some insight into a number of techniques used to understand how well our model is performing. This notebook is aimed at giving you some practice with the metrics specifically related to classification problems. With that in mind, we will again be looking at the spam dataset from the earlier lessons.

First, run the cell below to prepare the data and instantiate a number of different models.

In [1]:

```
# Import our libraries
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from sklearn.ensemble import BaggingClassifier, RandomForestClassifier, AdaBoostClassifier
from sklearn.svm import SVC
import tests as t

# Read in our dataset
df = pd.read_table('smsspamcollection/SMSSpamCollection',
                   sep='\t',
                   header=None,
                   names=['label', 'sms_message'])

# Fix our response value
df['label'] = df.label.map({'ham':0, 'spam':1})

# Split our dataset into training and testing data
X_train, X_test, y_train, y_test = train_test_split(df['sms_message'],
                                                    df['label'],
                                                    random_state=1)

# Instantiate the CountVectorizer method
count_vector = CountVectorizer()

# Fit the training data and then return the matrix
training_data = count_vector.fit_transform(X_train)

# Transform testing data and return the matrix. Note we are not fitting the testing data into the Count
testing_data = count_vector.transform(X_test)

# Instantiate a number of our models
naive_bayes = MultinomialNB()
bag_mod = BaggingClassifier(n_estimators=200)
rf_mod = RandomForestClassifier(n_estimators=200)
ada_mod = AdaBoostClassifier(n_estimators=300, learning_rate=0.2)
svm_mod = SVC()
```

Step 1: Now, fit each of the above models to the appropriate data. Answer the following question to assure that you fit the models correctly.

In [4]:
```
# Fit each of the 4 models
# This might take some time to run
naive_bayes.fit(training_data, y_train)
bag_mod.fit(training_data, y_train)
rf_mod.fit(training_data, y_train)
ada_mod.fit(training_data, y_train)
svm_mod.fit(training_data, y_train)
```

Out[4]:
```
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma='auto', kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)
```

In [6]:
```
# The models you fit above were fit on which data?

a = 'X_train'
b = 'X_test'
c = 'y_train'
d = 'y_test'
e = 'training_data'
f = 'testing_data'

# Change models_fit_on to only contain the correct string names
# of values that you oassed to the above models

models_fit_on = {c, e} # update this to only contain correct letters

# Checks your solution - don't change this
t.test_one(models_fit_on)
```

```
That's right!  You need to fit on both parts of the data pertaining to training data!
```

Step 2: Now make predictions for each of your models on the data that will allow you to understand how well our model will extend to new data. Then correctly add the strings to the set in the following cell.

In [8]:
```
# Make predictions using each of your models
naive_bayes_preds = naive_bayes.predict(testing_data)
bag_mod_preds = naive_bayes.predict(testing_data)
rf_mod_preds = naive_bayes.predict(testing_data)
ada_mod_preds = naive_bayes.predict(testing_data)
svm_mod_preds = naive_bayes.predict(testing_data)
```

In [9]:
```
# Which data was used in the predict method to see how well your
# model would work on new data?

a = 'X_train'
b = 'X_test'
c = 'y_train'
d = 'y_test'
e = 'training_data'
f = 'testing_data'

# Change models_predict_on to only contain the correct string names
# of values that you oassed to the above models

models_predict_on = {f} # update this to only contain correct letters

# Checks your solution - don't change this
t.test_two(models_predict_on)
```

```
That's right! To see how well our models perform in a new setting, you will want to predict on the tes
t set of data.
```

Now that you have set up all your predictions, let's get to topics addressed in this lesson - measuring how well each of your models performed. First, we will focus on how each metric was calculated for a single model, and then in the final part of this notebook, you will choose models that are best based on a particular metric.

You will be writing functions to calculate a number of metrics and then comparing the values to what you get from sklearn. This will help you build intuition for how each metric is calculated.

> **Step 3**: As an example of how this will work for the upcoming questions, run the cell below. Fill in the below function to calculate accuracy, and then compare your answer to the built in to assure you are correct.

In [11]:
```
# accuracy is the total correct divided by the total to predict
def accuracy(actual, preds):
    '''
    INPUT
    preds - predictions as a numpy array or pandas series
    actual - actual values as a numpy array or pandas series

    OUTPUT:
    returns the accuracy as a float
    '''
    return np.sum(preds == actual)/len(actual)


print(accuracy(y_test, naive_bayes_preds))
print(accuracy_score(y_test, naive_bayes_preds))
print("Since these match, we correctly calculated our metric!")
```

```
0.988513998564
0.988513998564
Since these match, we correctly calculated our metric!
```

> **Step 4**: Fill in the below function to calculate precision, and then compare your answer to the built in to assure you are correct.

In [12]:
```
# precision is the true positives over the predicted positive values
def precision(actual, preds):
    '''
    INPUT
    (assumes positive = 1 and negative = 0)
    preds - predictions as a numpy array or pandas series
    actual - actual values as a numpy array or pandas series

    OUTPUT:
    returns the precision as a float
    '''
    tp = len(np.intersect1d(np.where(preds==1), np.where(actual==1)))
    pred_pos = (preds==1).sum()
    return tp/(pred_pos)


print(precision(y_test, naive_bayes_preds))
print(precision_score(y_test, naive_bayes_preds))
print("If the above match, you got it!")
```

```
0.972067039106
0.972067039106
If the above match, you got it!
```

> **Step 5**: Fill in the below function to calculate recall, and then compare your answer to the built in to assure you are correct.

In [13]:
```
# recall is true positives over all actual positive values
def recall(actual, preds):
    '''
    INPUT
    preds - predictions as a numpy array or pandas series
    actual - actual values as a numpy array or pandas series

    OUTPUT:
    returns the recall as a float
    '''
    tp = len(np.intersect1d(np.where(preds==1), np.where(actual==1)))
    act_pos = (actual==1).sum()
    return tp/act_pos


print(recall(y_test, naive_bayes_preds))
print(recall_score(y_test, naive_bayes_preds))
print("If the above match, you got it!")
```

```
0.940540540541
0.940540540541
If the above match, you got it!
```

> **Step 6**: Fill in the below function to calculate f1-score, and then compare your answer to the built in to assure you are correct.

In [14]:
```
# f1_score is 2*(precision*recall)/(precision+recall))
def f1(preds, actual):
    '''
    INPUT
    preds - predictions as a numpy array or pandas series
    actual - actual values as a numpy array or pandas series

    OUTPUT:
    returns the f1score as a float
    '''
    tp = len(np.intersect1d(np.where(preds==1), np.where(actual==1)))
    pred_pos = (preds==1).sum()
    prec = tp/(pred_pos)
    act_pos = (actual==1).sum()
    recall = tp/act_pos
    return 2*(prec*recall)/(prec+recall)


print(f1(y_test, naive_bayes_preds))
print(f1_score(y_test, naive_bayes_preds))
print("If the above match, you got it!")
```

```
0.956043956044
0.956043956044
If the above match, you got it!
```

Step 7: Now that you have calculated a number of different metrics, let's tie that to when we might use one versus another. Use the dictionary below to match a metric to each statement that identifies when you would want to use that metric.

In [17]:
```
# add the letter of the most appropriate metric to each statement
# in the dictionary
a = "recall"
b = "precision"
c = "accuracy"
d = 'f1-score'


seven_sol = {
'We have imbalanced classes, which metric do we definitely not want to use?': c, # letter here,
'We really want to make sure the positive cases are all caught even if that means we identify some nega
'When we identify something as positive, we want to be sure it is truly positive': b, # letter here,
'We care equally about identifying positive and negative cases': d # letter here
}

t.sol_seven(seven_sol)
```

```
That's right!  It isn't really necessary to memorize these in practice, but it is important to know th
ey exist and know why might use one metric over another for a particular situation.
```

> **Step 8:** Given what you know about the metrics now, use this information to correctly match the appropriate model to when it would be best to use each in the dictionary below.

In [18]:
```
# use the answers you found to the previous questiona, then match the model that did best for each metr
a = "naive-bayes"
b = "bagging"
c = "random-forest"
d = 'ada-boost'
e = "svm"


eight_sol = {
'We have imbalanced classes, which metric do we definitely not want to use?': a,
'We really want to make sure the positive cases are all caught even if that means we identify some nega
'When we identify something as positive, we want to be sure it is truly positive': c,
'We care equally about identifying positive and negative cases': a
}

t.sol_eight(eight_sol)
```

```
That's right!  Naive Bayes was the best model for all of our metrics except precision!
```

In [ ]:
```
# cells for work
```

In [19]:
```
def print_metrics(y_true, preds, model_name=None):
    '''
    INPUT:
    y_true - the y values that are actually true in the dataset (numpy array or pandas series)
    preds - the predictions for those values from some model (numpy array or pandas series)
    model_name - (str - optional) a name associated with the model if you would like to add it to the pi

    OUTPUT:
    None - prints the accuracy, precision, recall, and F1 score
    '''
    if model_name == None:
        print('Accuracy score: ', format(accuracy_score(y_true, preds)))
        print('Precision score: ', format(precision_score(y_true, preds)))
        print('Recall score: ', format(recall_score(y_true, preds)))
        print('F1 score: ', format(f1_score(y_true, preds)))
        print('\n\n')

    else:
        print('Accuracy score for ' + model_name + ' :' , format(accuracy_score(y_true, preds)))
        print('Precision score ' + model_name + ' :', format(precision_score(y_true, preds)))
        print('Recall score ' + model_name + ' :', format(recall_score(y_true, preds)))
        print('F1 score ' + model_name + ' :', format(f1_score(y_true, preds)))
        print('\n\n')
```

In [21]:

```
# Print Bagging scores
print_metrics(y_test, bag_mod_preds, 'bagging')

# Print Random Forest scores
print_metrics(y_test, rf_mod_preds, 'random forest')

# Print AdaBoost scores
print_metrics(y_test, ada_mod_preds, 'adaboost')

# Naive Bayes Classifier scores
print_metrics(y_test, naive_bayes_preds, 'naive bayes')

# SVM Classifier scores
print_metrics(y_test, svm_mod_preds, 'svm')
```

```
Accuracy score for bagging : 0.9885139985642498
Precision score bagging : 0.9720670391061452
Recall score bagging : 0.9405405405405406
F1 score bagging : 0.9560439560439562



Accuracy score for random forest : 0.9885139985642498
Precision score random forest : 0.9720670391061452
Recall score random forest : 0.9405405405405406
F1 score random forest : 0.9560439560439562



Accuracy score for adaboost : 0.9885139985642498
Precision score adaboost : 0.9720670391061452
Recall score adaboost : 0.9405405405405406
F1 score adaboost : 0.9560439560439562



Accuracy score for naive bayes : 0.9885139985642498
Precision score naive bayes : 0.9720670391061452
Recall score naive bayes : 0.9405405405405406
F1 score naive bayes : 0.9560439560439562
```

```
Accuracy score for svm : 0.9885139985642498
Precision score svm : 0.9720670391061452
Recall score svm : 0.9405405405405406
F1 score svm : 0.9560439560439562
```

As a final step in this workbook, let's take a look at the last three metrics you saw, f-beta scores, ROC curves, and AUC.

For f-beta scores: If you decide that you care more about precision, you should move beta closer to 0. If you decide you care more about recall, you should move beta towards infinity.

> **Step 9:** Using the fbeta_score works similar to most of the other metrics in sklearn, but you also need to set beta as your weighting between precision and recall. Use the space below to show that you can use [fbeta in sklearn (http://scikit-learn.org/stable/modules/generated/sklearn.metrics.fbeta_score.html)](http://scikit-learn.org/stable/modules/generated/sklearn.metrics.fbeta_score.html) to replicate your f1-score from above. If in the future you want to use a different weighting, [this article (http://mlwiki.org/index.php/Precision_and_Recall)](http://mlwiki.org/index.php/Precision_and_Recall) does an amazing job of explaining how you might adjust beta for different situations.

In [24]:

```python
#import fbeta score
from sklearn.metrics import fbeta_score

#show that the results are the same for fbeta and f1_score
print(fbeta_score(y_test, bag_mod_preds, beta=.8))
print(fbeta_score(y_test, bag_mod_preds, beta=1))
print(f1_score(y_test, bag_mod_preds))
```

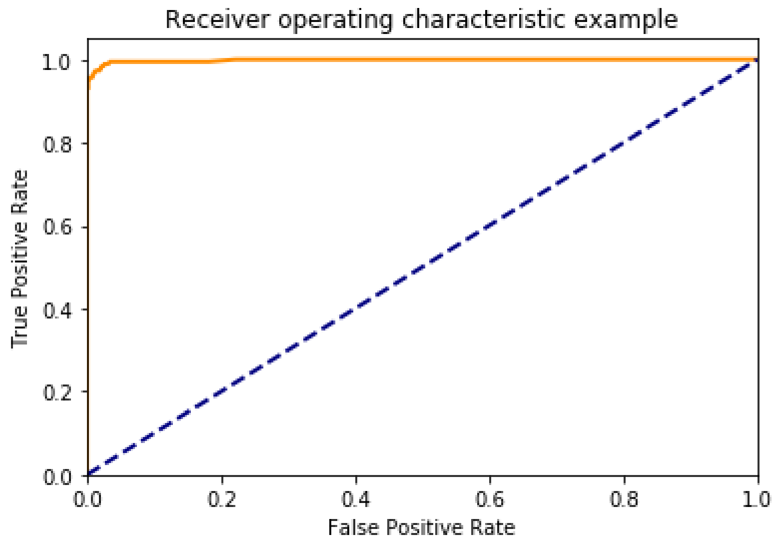
```
0.959515803631
0.956043956044
0.956043956044
```

> **Step 10:** Building ROC curves in python is a pretty involved process on your own. I wrote the function below to assist with the process and make it easier for you to do so in the future as well. Try it out using one of the other classifiers you created above to see how it compares to the random forest model below.

Run the cell below to build a ROC curve, and retrieve the AUC for the random forest model.

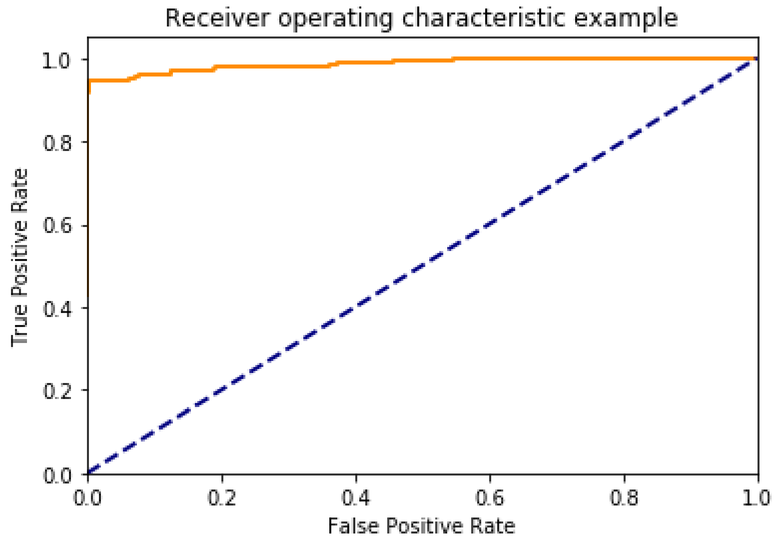
In [26]:
```
# Function for calculating auc and roc

def build_roc_auc(model, X_train, X_test, y_train, y_test):
    '''
    INPUT:
    model - an sklearn instantiated model
    X_train - the training data
    y_train - the training response values (must be categorical)
    X_test - the test data
    y_test - the test response values (must be categorical)
    OUTPUT:
    auc - returns auc as a float
    prints the roc curve
    '''
    import numpy as np
    import matplotlib.pyplot as plt
    from itertools import cycle
    from sklearn.metrics import roc_curve, auc, roc_auc_score
    from scipy import interp

    y_preds = model.fit(X_train, y_train).predict_proba(X_test)
    # Compute ROC curve and ROC area for each class
    fpr = dict()
    tpr = dict()
    roc_auc = dict()
    for i in range(len(y_test)):
        fpr[i], tpr[i], _ = roc_curve(y_test, y_preds[:, 1])
        roc_auc[i] = auc(fpr[i], tpr[i])

    # Compute micro-average ROC curve and ROC area
    fpr["micro"], tpr["micro"], _ = roc_curve(y_test.ravel(), y_preds[:, 1].ravel())
    roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])

    plt.plot(fpr[2], tpr[2], color='darkorange',
             lw=2, label='ROC curve (area = %0.2f)' % roc_auc[2])
    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver operating characteristic example')
    plt.show()
```

```
    return roc_auc_score(y_test, np.round(y_preds[:, 1]))


# Finding roc and auc for the random forest model
build_roc_auc(rf_mod, training_data, testing_data, y_train, y_test)
```



Out[26]: 0.9351351351351351

In [27]:
```
# Your turn here - choose another classifier to see how it compares

build_roc_auc(naive_bayes, training_data, testing_data, y_train, y_test)
```



Out[27]: 0.96820073384642935

In [ ]: