Go to the svm directory to find the starter code (svm/svm_author_id.py).

Import, create, train and make predictions with the sklearn SVC classifier. When creating the classifier, use a linear kernel (if you forget this step, you will be unpleasantly surprised by how long the classifier takes to train). What is the accuracy of the classifier?

In [ ]:

```
#!/usr/bin/python

"""
    This is the code to accompany the Lesson 2 (SVM) mini-project.

    Use a SVM to identify emails from the Enron corpus by their authors:
    Sara has label 0
    Chris has label 1
"""

import sys
from time import time
sys.path.append("../tools/")
from email_preprocess import preprocess


### features_train and features_test are the features for the training
### and testing datasets, respectively
### labels_train and labels_test are the corresponding item labels
features_train, features_test, labels_train, labels_test = preprocess()

#########################################################
### your code goes here ###
#features_train = features_train[:len(features_train)/100]
#labels_train = labels_train[:len(labels_train)/100]
from sklearn import svm
clas=svm.SVC(kernel="linear")
clas.fit(features_train,labels_train)
clas.score(features_test,labels_test)
#########################################################
```

```
no. of Chris training emails: 7936
no. of Sara training emails: 7884
```

Place timing code around the fit and predict functions, like you did in the Naive Bayes mini-project. How do the training and prediction times compare to Naive Bayes?

slower

In [ ]:

```
t0 = time()
clas.fit(features_train,labels_train)
clas.score(features_test,labels_test)
print "training time:",round(time()-t0,3),"s"
```

One way to speed up an algorithm is to train it on a smaller training dataset. The tradeoff is that the accuracy almost always goes down when you do this. Let’s explore this more concretely: add in the following two lines immediately before training your classifier.

features_train = features_train[:len(features_train)/100]

labels_train = labels_train[:len(labels_train)/100]

These lines effectively slice the training dataset down to 1% of its original size, tossing out 99% of the training data. You can leave all other code unchanged. What’s the accuracy now?

```
In [ ]: features_train = features_train[:len(features_train)/100]
        labels_train = labels_train[:len(labels_train)/100]
        clas=svm.SVC(kernel="linear")
        clas.fit(features_train,labels_train)
        clas.score(features_test,labels_test)
```

Keep the training set slice code from the last quiz, so that you are still training on only 1% of the full training set. Change the kernel of your SVM to “rbf”. What’s the accuracy now, with this more complex kernel?

```
In [ ]: clas=svm.SVC(kernel="rbf")
        clas.fit(features_train,labels_train)
        clas.score(features_test,labels_test)
```

Keep the training set size and rbf kernel from the last quiz, but try several values of C (say, 10.0, 100., 1000., and 10000.). Which one gives the best accuracy?

10000

```
In [ ]: clas=svm.SVC(C=10000,kernel="rbf")
        clas.fit(features_train,labels_train)
        clas.score(features_test,labels_test)
```

Once you've optimized the C value for your RBF kernel, what accuracy does it give? Does this C value correspond to a simpler or more complex decision boundary?

Complex

(If you're not sure about the complexity, go back a few videos to the "SVM C Parameter" part of the lesson. The result that you found there is also applicable here, even though it's now much harder or even impossible to draw the decision boundary in a simple scatterplot.)

```
In [ ]: clas.score(features_test,labels_test)
```

Now that you’ve optimized C for the RBF kernel, go back to using the full training set. In general, having a larger training set will improve the performance of your algorithm, so (by tuning C and training on a large dataset) we should get a fairly optimized result. What is the accuracy of the optimized SVM?

In [ ]:

```
features_train, features_test, labels_train, labels_test = preprocess()
clas=svm.SVC(C=10000,kernel="rbf")
clas.fit(features_train,labels_train)
clas.score(features_test,labels_test)
```

What class does your SVM (0 or 1, corresponding to Sara and Chris respectively) predict for element 10 of the test set? The 26th? The 50th? (Use the RBF kernel, C=10000, and 1% of the training set. Normally you'd get the best results using the full training set, but we found that using 1% sped up the computation considerably and did not change our results--so feel free to use that shortcut here.)

And just to be clear, the data point numbers that we give here (10, 26, 50) assume a zero-indexed list. So the correct answer for element #100 would be found using something like answer=predictions[100]

In [ ]:

```
features_train = features_train[:len(features_train)/100]
labels_train = labels_train[:len(labels_train)/100]
clas=svm.SVC(kernel="linear")
clas.fit(features_train,labels_train)

clas.predict(features_test(10))
clas.predict(features_test(26))
clas.predict(features_test(50))
```

There are over 1700 test events--how many are predicted to be in the “Chris” (1) class? (Use the RBF kernel, C=10000., and the full training set.)

In [ ]:

```
features_train, features_test, labels_train, labels_test = preprocess()
clas=svm.SVC(C=10000,kernel="rbf")
clas.fit(features_train,labels_train)
pred = clas.predict(features_test)
yote = 0
for x in pred:
    if x == 1:
        yote = yote +1
print yote
```

In [ ]: