SVMs on Extended MNIST

Siddharth Choudhary - sc7530

1 Lab: SVMs on Extended MNIST

In the MNIST demo, we saw how SVMs can be used for the classic MNIST problem of digit recognition. In this lab, we are going to extend the MNIST dataset by adding a number of non-digit letters and see if the classifier can distinguish the digits from the non-digits. All non-digits will be lumped as a single 11-th class. This is a highly simplified version of 'detection' problem (as opposed to 'classification' problem). Detection is vital in OCR and related problems since the non useful characters must be rejected.

In addition to the concepts in the demo, you will learn: * Combine multiple datasets * Select the SVM parameters (`C` and `gamma`) via cross-validation. * Use the `GridSearchCV` method to search for parameters with cross-validation.

Note: An earlier version of this lab made you manually create the combined letter and digit data. In this lab, we will download the data from NIST website. But, the old lab is still useful to look at if you want to see how to use `skimage` package for a number of image pre-processing tasks.

As usual, we download the standard packages

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn import linear_model, preprocessing
```

1.1 Downloading the EMNIST Dataset

After creating the highly popular MNIST dataset, NIST created an extended version of the dataset to include letters and digits. The extended datase (called EMNIST) also has many more examples per class.

To download the data, first go to the EMNIST webpage. Near the bottom, you will see a link for `MATLAB format dataset`. If you click on this link, you will download a `zip` file with several datasets in it. The total file is 726M, so it may take some time and diskspace to download. Extract two files: * `emnist-digits.mat`: This is a file of digits `0` to `9`, but with more examples per class. * `emnist-letters.mat`: This is a file of letters `a/A` to `z/Z`. The lower and upper case letters are grouped into the same class.

Once you get these two files, you can save yourself the diskspace and remove all the other files.

You can download the files manually, or you can run the following commands which will download the files automatically.

[2]:
```
from tqdm import tqdm
import requests
import os
import zipfile

def download_file(src_url, dst_fn):

    if os.path.exists(dst_fn):
        print('File %s already exists' % dst_fn)
        return

    print('Downloading %s' % dst_fn)

    # Streaming, so we can iterate over the response.
    r = requests.get(src_url, stream=True)

    # Total size in MB.
    total_size = int(r.headers.get('content-length', 0));
    block_size = 1024
    wrote = 0
    with open(dst_fn, 'wb') as f:
        with tqdm(total=total_size//block_size, unit='kB',
                          unit_scale=True, unit_divisor=1024) as pbar:
            for data in r.iter_content(block_size):
                wrote = wrote + len(data)
                pbar.update(1)
                f.write(data)
    if total_size != 0 and wrote != total_size:
        print("ERROR, something went wrong")

# Get file names
matlab_dir = 'matlab'
digits_fn =  os.path.join(matlab_dir,'emnist-digits.mat')
letters_fn = os.path.join(matlab_dir,'emnist-letters.mat')

# Check if files exists
if os.path.exists(matlab_dir):
    if os.path.exists(digits_fn) and os.path.exists(letters_fn):
        print('Files already downloaded')
        files_exists = True
else:
    files_exists = False

if not files_exists:
    # First download the zip file if needed
    src_url = "http://www.itl.nist.gov/iaui/vip/cs_links/EMNIST/matlab.zip"
    dst_fn = 'matlab.zip'
```

```
    download_file(src_url, dst_fn)

    # Then, unzip the file
    print('Unzipping %s...' % dst_fn)
    zip_ref = zipfile.ZipFile(dst_fn, 'r')
    zip_ref.extractall('.')
    zip_ref.close()
    print('Unzip completed')
```

```
Downloading matlab.zip

709kkB [01:44, 6.92kkB/s]

Unzipping matlab.zip...
Unzip completed
```

Since MATLAB files are still widely-used, Python has excellent routines for loading MATLAB files. The function below uses the `scipy.io` package to extract the relevant fields from the MATLAB file. Specifically, the function extracts the training and test data from MATLAB file.

[3]:
```
import scipy.io
def load_emnist(file_path='emnist-digits.mat'):
    """
    Loads training and test data with ntr and nts training and test samples
    The `file_path` is the location of the `eminst-balanced.mat`.
    """

    # Load the MATLAB file
    mat = scipy.io.loadmat(file_path)

    # Get the training data
    Xtr = mat['dataset'][0][0][0][0][0][0][:]
    ntr = Xtr.shape[0]
    ytr = mat['dataset'][0][0][0][0][0][1][:].reshape(ntr).astype(int)

    # Get the test data
    Xts = mat['dataset'][0][0][1][0][0][0][:]
    nts = Xts.shape[0]
    yts = mat['dataset'][0][0][1][0][0][1][:].reshape(nts).astype(int)

    print("%d training samples, %d test samples loaded" % (ntr, nts))

    return [Xtr, Xts, ytr, yts]
```

Use the function above to get all the digit images from the `emnist-digits.mat` file.

[4]:
```
# TODO:  Load the digit data from emnist-digits.mat
Xtr_dig, Xts_dig, ytr_dig, yts_dig = load_emnist('matlab/emnist-digits.mat')
```

```
240000 training samples, 40000 test samples loaded
```

Next, use the function above to get all the letter characters from the `emnist-letters.mat` file.

[5]:
```
# TODO:  Load the digit data from emnist-letters.mat
Xtr_let, Xts_let, ytr_let, yts_let = load_emnist('matlab/emnist-letters.mat')
```

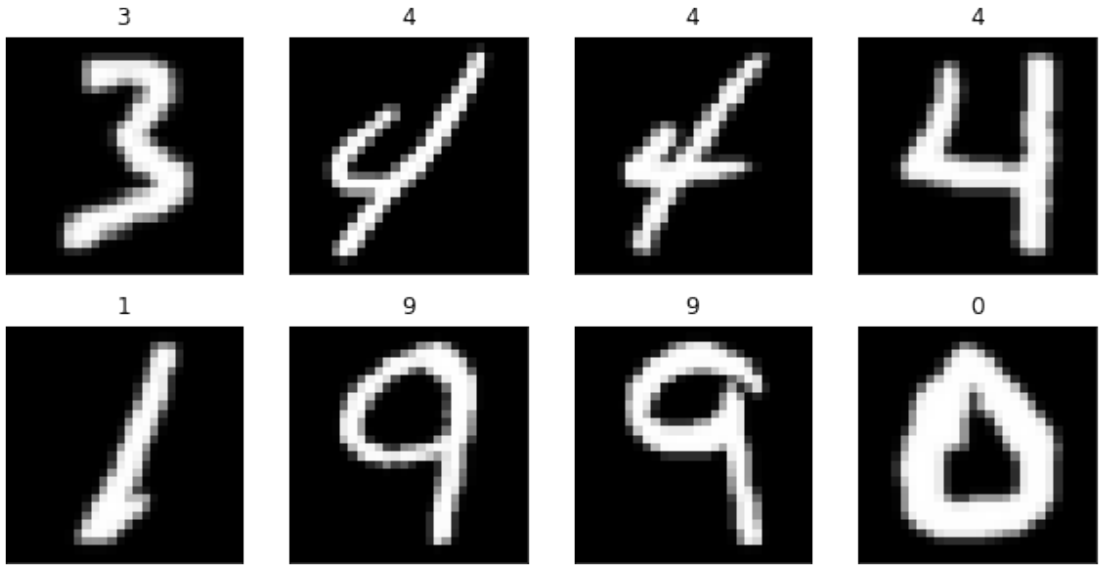
```
124800 training samples, 20800 test samples loaded
```

We will use the function from the demo to plot the digits.

[6]:
```
def plt_digit(x,y=None):
    nrow = 28
    ncol = 28
    xsq = x.reshape((nrow,ncol))
    plt.imshow(xsq.T,  cmap='Greys_r')
    plt.xticks([])
    plt.yticks([])
    if y != None:
        plt.title('%d' % y)
```

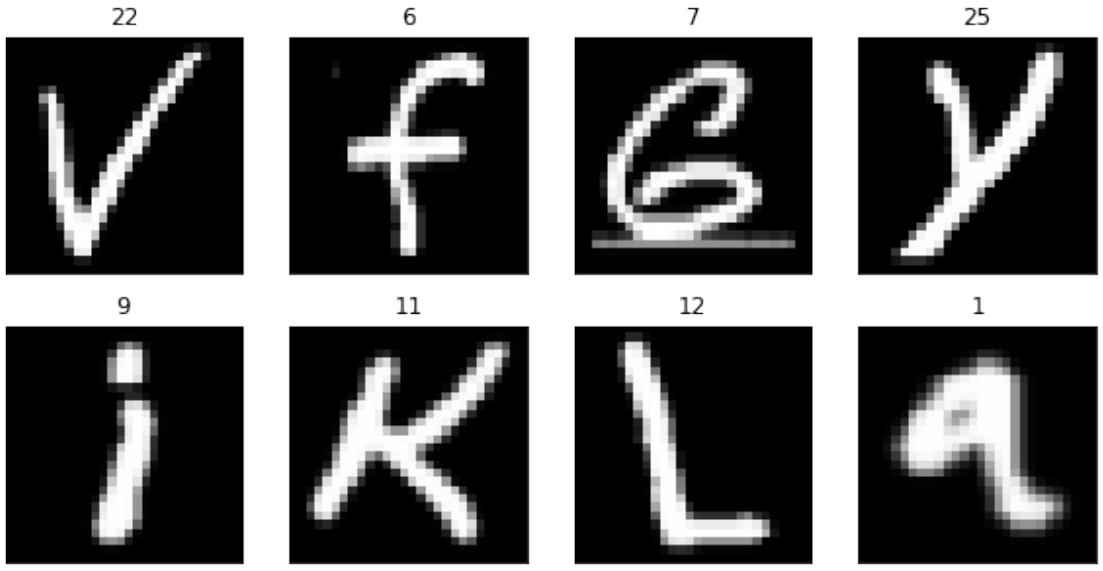
Plot 8 random samples from the digit training data. You can use the `plt_digit` function above with `subplot` to create a nice display. You may want to size your plot with the `plt.figure(figsize=(10,20))` command.

[7]:
```
# TODO: Plot 8 random samples from the training data of the digits
nplt = 8
ntr_dig = Xtr_dig.shape[0]
ind = np.random.choice(ntr_dig, nplt)

plt.figure(figsize=(10, 5))
for i, x in enumerate(ind):
    plt.subplot(2, 4, i+1)
    plt_digit(Xtr_dig[x], ytr_dig[x])
plt.show()
```



Next, plot 8 samples from the letters training data. You should see that the labels go from 0 to 25 corresponding to a to z. Upper and lower case letters belong to the same class.

[8]:
```
# TODO:  Plot 8 random samples from the training data of the letters
ntr_let = Xtr_let.shape[0]
ind = np.random.choice(ntr_let, nplt)

plt.figure(figsize=(10, 5))
for i, x in enumerate(ind):
    plt.subplot(2, 4, i+1)
    plt_digit(Xtr_let[x], ytr_let[x])
plt.show()
```



1.2 Creating a Non-Digit Class

SVM classifiers are VERY SLOW to train. The training is particularly slow when there are a large number of classes, since the one classifier must be trained for each pair of labels. To make the problem easier, we are going to lump all of the letters in one class and add that class to the digits.

Before we begin, we first need to remove all the letters corresponding to `i/I`, `l/L` and `o/O`. The reason is that these letters would get confused with the digits `0` and `1`. Create arrays `Xtr_let_rem` and `ytr_let_rem` from the data `Xtr_let` and `ytr_let`, where the samples `i` with `ytr_let[i] == 9, 12` or `15` are removed. Create `Xts_let_rem` and `yts_let_rem` similarly.

If you are clever, you can do this without a for-loop via python broadcasting and `np.all(..., axis=1)` command. But, you will receive full marks if you use a `for-loop`.

[9]:
```
remove_list = np.array([9,12,15])

# TODO:  Create arrays with labels 9, 12 and 15 removed
idxtr = np.all((ytr_let != remove_list[0], ytr_let != remove_list[1], ytr_let !=␣
 ↪remove_list[2]))
idxts = np.all((yts_let != remove_list[0], yts_let != remove_list[1], yts_let !=␣
 ↪remove_list[2]))

Xtr_let_rem, ytr_let_rem = Xtr_let[idxtr], ytr_let[idxtr]
Xts_let_rem, yts_let_rem = Xts_let[idxts], yts_let[idxts]
```

Since training and testing an SVM is VERY SLOW, we will use only a small subset of the training and test data. Of course, you will not get great results with this small dataset. But, we can at least illustrate the basic concepts.

Create arrays `Xtr1_dig` and `ytr1_dig` by selecting 5000 random training digit samples from `Xtr_dig` and `ytr_dig`. Create arrays `Xtr1_let` and `ytr1_let` by selecting 1000 random training letter samples from `Xtr_let_rem` and `ytr_let_rem`. Similarly, create test arrays `Xts1_dig,Xts1_let,yts1_dig,yts1_let` with 5000 digits and 1000 letters.

[10]:
```
# Number of training and test digits and letters
ntr_dig = 5000
ntr_let = 1000
nts_dig = 5000
nts_let = 1000

Iperm_tr_dig = np.random.permutation(Xtr_dig.shape[0])
Iperm_ts_dig = np.random.permutation(Xts_dig.shape[0])
Iperm_tr_let = np.random.permutation(Xtr_let.shape[0])
Iperm_ts_let = np.random.permutation(Xts_let.shape[0])

# TODO Create sub-sampled training and test data
Xtr1_dig, ytr1_dig = Xtr_dig[Iperm_tr_dig[:ntr_dig], :], ytr_dig[Iperm_tr_dig[:
 ↪ntr_dig]]
Xts1_dig, yts1_dig = Xts_dig[Iperm_ts_dig[:nts_dig], :], yts_dig[Iperm_ts_dig[:
 ↪nts_dig]]
Xtr1_let, ytr1_let = Xtr_let[Iperm_tr_let[:ntr_let], :], ytr_let[Iperm_tr_let[:
 ↪ntr_let]]
Xts1_let, yts1_let = Xts_let[Iperm_ts_let[:nts_let], :], yts_let[Iperm_ts_let[:
 ↪nts_let]]
```

Next, we create data by combining the digit and letter arrays. * Create an array `Xtr` by stacking `Xtr1_dig`, `Xtr1_let`. This should result in 6000 total samples. * Create a new label vector `ytr` where `ytr[i] = ytr1_dig[i]` for any digit sample and `ytr[i]=10` for any letter sample. Thus, all the letters are lumped into a single class with label 11.

Create test arrays `Xts` and `yts` similarly.

You may wish to use the `np.hstack` and `np.vstack` methods.

[11]:
```
# TODO:  Create combined letter and digit training and test data
Xtr, ytr = np.vstack((Xtr1_dig, Xtr1_let)), np.hstack((ytr1_dig, np.repeat(10,␣
 ↪ytr1_let.shape[0])))
Xts, yts = np.vstack((Xts1_dig, Xts1_let)), np.hstack((yts1_dig, np.repeat(10,␣
 ↪yts1_let.shape[0])))
```

The training data above takes values from 0 to 255. Rescale the data from -1 to 1. This will get slightly better performance on the SVM. Save the scaled data into arrays `Xtr1` and `Xts1`.

[12]:
```
# TODO:  Rescale the data from -1 to 1
Xtr1 = 2 *(Xtr / 255 - 0.5)
Xts1 = 2 *(Xts / 255 - 0.5)
```

1.3 Run the SVM classifier

First create the SVM classifer. Use an `rbf` classifier with `C=2.8` and `gamma=.0073`. We will look at how to select these parameters laters.

[13]:
```
from sklearn import svm

# TODO:  Create a classifier: a support vector classifier
svc = svm.SVC(C = 2.8, gamma = .0073)
```

Fit the classifier using the scaled training data. SVMs are insanely slow to train. But, in this lab, we have kept the training size very small. So, the fitting should take about a minute or two.

[14]:
```
# TODO:  Fit the classifier on the training data.
svc.fit(Xtr1, ytr)
```

[14]:
```
SVC(C=2.8, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma=0.0073, kernel='rbf',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
```

Measure the accuracy on the test data. This too will take another huge amount of time. Print the accuracy. If you did everything right, you should get an accuracy of around 89%.

[15]:
```
# TODO:  Measure error on the test data
yhat = svc.predict(Xts1)
acc = np.mean(yhat == yts)
print('Accuracy : {}'.format(acc))
```

```
Accuracy : 0.9005
```

The error rate is quite a bit higher than what we got in the digits only case. Actually, had we done a classifier using all 36 labels instead of collapsing the letters to a single class, the SVM classifier would have done much better. The reason is that the "letters" class is now extremely complex.

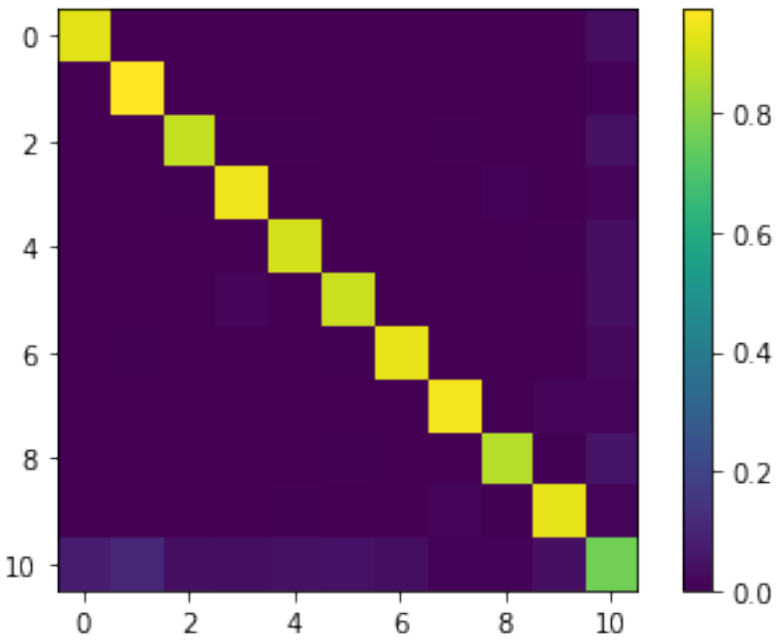
Print a confusion matrix. You should see that the error rate on the "letters" class is much higher.

[16]:
```
# TODO:  Print a confusion matrix
from sklearn.metrics import confusion_matrix

C = confusion_matrix(yts, yhat)
print(C)
Csum = np.sum(C,1)
C = C / Csum[None,:]

print(np.array_str(C, precision=3, suppress_small=True))
plt.imshow(C, interpolation='none')
plt.colorbar()
```

```
[[492   0   0   0   0   0   0   0   1   0  35]
```

```
 [  0 509   0   1   1   0   0   0   1   1   9]
 [  0   1 431   2   2   0   1   2   1   0  45]
 [  0   0   2 469   0   1   0   1   4   0  14]
 [  0   0   0   0 444   0   0   0   0   3  41]
 [  0   0   0   9   0 449   1   1   0   0  41]
 [  0   2   0   0   1   2 461   0   0   0  23]
 [  0   1   0   0   0   0   0 490   1   7  13]
 [  1   1   1   1   0   2   0   0 427   2  57]
 [  0   1   0   1   3   1   0   7   2 462  15]
 [ 41  59  20  18  22  24  17   5   5  20 769]]
[[0.932 0.    0.    0.    0.    0.    0.    0.    0.002 0.    0.035]
 [0.    0.975 0.    0.002 0.002 0.    0.    0.    0.002 0.002 0.009]
 [0.    0.002 0.889 0.004 0.004 0.    0.002 0.004 0.002 0.    0.045]
 [0.    0.    0.004 0.955 0.    0.002 0.    0.002 0.008 0.    0.014]
 [0.    0.    0.    0.    0.91  0.    0.    0.    0.    0.006 0.041]
 [0.    0.    0.    0.018 0.    0.896 0.002 0.002 0.    0.    0.041]
 [0.    0.004 0.    0.    0.002 0.004 0.943 0.    0.    0.    0.023]
 [0.    0.002 0.    0.    0.    0.    0.    0.957 0.002 0.014 0.013]
 [0.002 0.002 0.002 0.002 0.    0.004 0.    0.    0.868 0.004 0.057]
 [0.    0.002 0.    0.002 0.006 0.002 0.    0.014 0.004 0.939 0.015]
 [0.078 0.113 0.041 0.037 0.045 0.048 0.035 0.01  0.01  0.041 0.769]]
```

[16]: `<matplotlib.colorbar.Colorbar at 0x7f465fb41b70>`



Print: * What fraction of digits are mislabeled as letters?
* What fraction of letters are mislabeled as digits?

[17]:
```
# TODO:  Print above two error rates
print('Fraction of digits mislabeled as letters : {}'.format(np.sum(C[:9, -1])))
print('Fraction of digits mislabeled as digits : {}'.format(np.sum(C[-1, :9])))
```

```
Fraction of digits mislabeled as letters : 0.278
Fraction of digits mislabeled as digits : 0.4162545379091226
```

1.4 Selecting gamma and C via Cross-Validation (Using For-Loops)

In the above example, and in the demo, we used a given `gamma` and `C` value. The selection of the parameters depend on the problem and decent performance of the SVM requires that you select these parameters carefully. The best way to select the parameters is via cross validation. Specifically, generally, one tries different values of `gamma` and `C` and selects the pair of values the lowest test error rate.

In the code below, we will try to use 3 values for `C` and `gamma` as specified in the arrays `C_test` and `gam_test`. For each `C` and `gamma` in these arrays, fit a model on the training data and measure the accuracy on the test data. Then, print the `C` and `gamma` that result in the best accuracy.

Normally, you would try a large number of values for each of the parameters, but an SVM is very slow to train – even with this small data set. So, we will just do 3 values of each. Even then, this could take 30 minutes or so to complete.

In this lab, you may do the parameter search over `C` and `gamma` in one of two ways: * This section: Use for loops and manually search over the parameters. This is more direct and you will see and control exactly what is happening. * Next section: Use the `GridSearchCV` method in the `sklearn` package. This takes a little reading, but once you learn this method, you can more easily use this for complex parameter searches.

You only need to submit the solutions to one of the two sections. Pick whichever one you want.

[18]:
```
C_test = [0.1,1,10]
gam_test = [0.001,0.01,0.1]

nC = len(C_test)
ngam = len(gam_test)
acc = np.zeros((nC,ngam))

# TODO:  Measure and print the accuracy for each C and gamma value.  Store the␣
 ↪results in acc
for indc, c in enumerate(C_test):
    for indg, g in enumerate(gam_test):

        print('C = {},     G = {}'.format(c, g))

        svc = svm.SVC(C = c, gamma = g)
        svc.fit(Xtr1, ytr)
        yhat = svc.predict(Xts1)
        acc[indc, indg] = np.mean(yhat == yts)
```

```
C = 0.1,     G = 0.001
C = 0.1,     G = 0.01
C = 0.1,     G = 0.1
C = 1,     G = 0.001
C = 1,     G = 0.01
C = 1,     G = 0.1
C = 10,     G = 0.001
C = 10,     G = 0.01
C = 10,     G = 0.1
```

[19]:
```
# TODO:  Print the accuracy matrix
acc
```

[19]:
```
array([[0.7685    , 0.58266667, 0.16666667],
       [0.85733333, 0.88533333, 0.21383333],
       [0.88483333, 0.884     , 0.21766667]])
```

[23]:
```
# TODO:  Print the maximum accuracy and the corresponding best C and gamma
acc_max = np.max(acc)
imax = np.argmax(acc)
c_max = C_test[imax // nC]
g_max = gam_test[imax % nC]
print('Best Accuracy : {}'.format(acc_max))
print('Best C : {}'.format(c_max))
print('Best Gamma : {}'.format(g_max))
```

```
Best Accuracy : 0.8853333333333333
Best C : 1
Best Gamma : 0.01
```

1.5 Using `GridSearchCV` (Optional Section)

In the previous section, you would have likely used `for-loops` to search over the different `C` and `gamma` values. Since this type of parameter search is so commonly used, `sklearn` has an excellent method `GridSearchCV` that can perform all the operations for you. In this lab, `GridSearchCV` is not that useful. But, once you get to more complex parameter searches, the `GridSearchCV` method can save you writing a lot of code. Importantly, `GridSearchCV` supports parallelization so that fits with different parameters can be fit at the same time. In this optional section, we will show how to use this method.

You do not have to do this section, if you did the previous section.

The `GridSearchCV` method does the train-test split in addition to the parameter search. In this case, you have already a fixed train-test split. So, you first need to combine the train and test data back into a single dataset.

Create arrays `X` and `y` from `Xtr1`, `Xts1`, `ytr` and `yts`. Use `np.vstack` and `np.hstack`.

[ ]:
```
# TODO:  Create combined trained and test data X and y.
# X = ...
# y = ...
```

Normally, `GridSearchCV` will do *K*-fold validation and automatically split the data into training and test in each fold. But, in this case, we want it to perform only one fold with a specific train-test split. To do this, we need to do the following: * Create a vector `test_fold` where `test_fold[i] = -1` for the samples `i` in the training data (this indicates that they should not be used as test data in any fold) and `test_fold[i] = 0` for the samples `i` in the test data (this indicates that they should be as test data in fold 0). * Call the method `ps = sklearn.model_selection.PredefinedSplit(test_fold)` to create a predefined test split object.

[ ]:
```
# TODO:  Create a pre-defined test split object
# import sklearn.model_selection
# test_fold = ...
# ps = sklearn.model_selection.PredefinedSplit(test_fold)
```

Next, read about the `GridSearchCV` method to set up a classifier that includes searching over the parameter grid.
* For the `param_grid` parameter, you will want to create a dictionary to search over `C` and `gamma`. You will also need to select the `kernel` parameter. * Set `cv = ps` to use the fixed train-test split. * Set `verbose=10` to monitor the progress

[ ]:
```
# TODO:  Create a GridSearchCV classifier
# clf = ...
```

Fit the classifier using the `fit` method. The fit method will now search over all the parameters. This will take about 30 minutes.

[ ]:
```
# TODO: Fit the classifier
```

Print the `best_score_` and `best_params_` attributes of the classifier to find the best score and parameters

[ ]:
```
# TODO:  Print the best parameter and score of the classifier
```

Finally, you can print the test and train score from the `cv_results_['mean_test_score']` and `cv_results_['mean_train_score']`.

[ ]:
```
# TODO:  Print the mean test score for each parameter value.
```

[ ]:
```

```