Lab 6: SVMs on an Extended MNIST

In addition to the concepts in the MNIST demo (mnist_svm.ipynb), you will learn:

- Use the `skimage` module for some basic pre-processing of images in machine learning
- Run and test an SVM classifier on a dataset you have created
- Perform error handling in python

In the MNIST demo (mnist_svm.ipynb), 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. In image processing, this is called a 'detection' as opposed to 'classification' problem. Detection is vital in OCR and related problems since the non useful characters must be rejected. For this lab we will create a very simple version of this problem.

Loading the MNIST data

We first import the standard modules

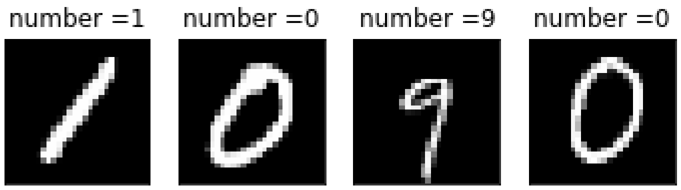
In [6]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn import linear_model, preprocessing
```

Next, fetch the digits with `fetch_mldata` command as shown in the demo. Save the digits data matrix and labels to variables `Xdig` and `ydig`. Also, recall that the pixel values in `Xdig` are between 0 and 255. Create a scaled version of `Xdig` called `Xdigs` where the components are between -1 and 1.

In [7]:
```
# TODO
from sklearn.datasets import fetch_mldata
mnist = fetch_mldata("MNIST original")

# Load MNIST data
Xdig = mnist.data
ydig = mnist.target

Xdigs = Xdig/255 *2 -1  # Rescale MNIST data
```

Create a function `plt_digit` that plots the digits. You can use the code from the demo. Test the function by plotting four random digits. Use the `plt.title` command to print the numeric label in `ydig` above each digit.

In [8]:
```
# TODO: Select four random digits and plot them using the subplot command
def plt_digit(x):
    nrow = 28
    ncol = 28
    xsq = x.reshape((nrow,ncol))
    plt.imshow(xsq,  cmap='Greys_r')
    plt.xticks([])
    plt.yticks([])

# Select random digits
nplt = 4
nsamp = Xdigs.shape[0]
Iperm = np.random.permutation(nsamp)

# Plot the images using the subplot command
for i in range(nplt):
    ind = Iperm[i]
    plt.subplot(1,nplt,i+1)
    plt_digit(Xdigs[ind,:])
    title = 'number ={0:d}'.format(ydig[ind].astype(int))
    plt.title(title)
```



Exception Handling

In the routines we will develop below, we will need to handle error conditions, called exceptions. A very nice description of how to perform exception handling in python is given in

https://docs.python.org/3/tutorial/errors.html (https://docs.python.org/3/tutorial/errors.html)

As described there, errors are described by a class that derives from a base class Exception. When the error occurs, the program raises the exception with the raise command. The calling function can catch the exception with the try ... except control flow. We will define our exception as follows which has an optional string argument.

In [9]:
```
class ImgException(Exception):
    def __init__(self, msg='No msg'):
        self.msg = msg
```

Exceptions are used as follows: First, when there is an error in some function, you `raise` the exception as follows:

```
foo():
    ...
    if (error):
       raise ImgException("File not found")

    # Code that will not execute if the error condition occured
```

The function that calls `foo()` can catch the error using the following syntax:

```
try:
    foo()

    # Continue processing in case when there was no exception
    ....

except ImgException as e:
    print("foo() didn't work")
    print("Error msg = %s" % e.msg)
```

Get Non-Digit Characters

We will now build a set of non-digit characters. As a simple source, we will get hand-written lowercase letters 'a' to 'z' and process them with the `skimage` package. The `skimage` module is a very powerful package that has a similar interface as OpenCV. We first import the relevant modules.

In [10]:
```
import matplotlib.image as mpimg
import skimage.io
from skimage.filters import threshold_otsu
from skimage.segmentation import clear_border
from skimage.measure import label, regionprops
from skimage.morphology import closing, square
from skimage.color import label2rgb
from skimage.transform import resize
import matplotlib.patches as mpatches
from skimage import data
import skimage
```

We can get a set of character images from a very nice website

http://www.ee.surrey.ac.uk/CVSSP/demos/chars74k/
(http://www.ee.surrey.ac.uk/CVSSP/demos/chars74k/)

Go to this website, and download the file `EnglishHnd.tgz`. After you untar this file, there are a large number of `.png` files in the directory:

```
EnglishHnd\English\Hnd\Img
```

Each directory has about 55 samples of hand-written letters and numbers. After you have downloaded this file, complete the function `load_img` to load an image from a character and sample index.

Alternatively, the files are available on Google Drive:

https://drive.google.com/file/d/0BxOz-SM9a1h4UksxSXBjQ0dabUk/view?usp=sharing (https://drive.google.com/file/d/0BxOz-SM9a1h4UksxSXBjQ0dabUk/view?usp=sharing)

You can download and unzip the file.

The code at the end will test the function to see if it working correctly. For one sample, it should print the image and a second it should say the file was not found.

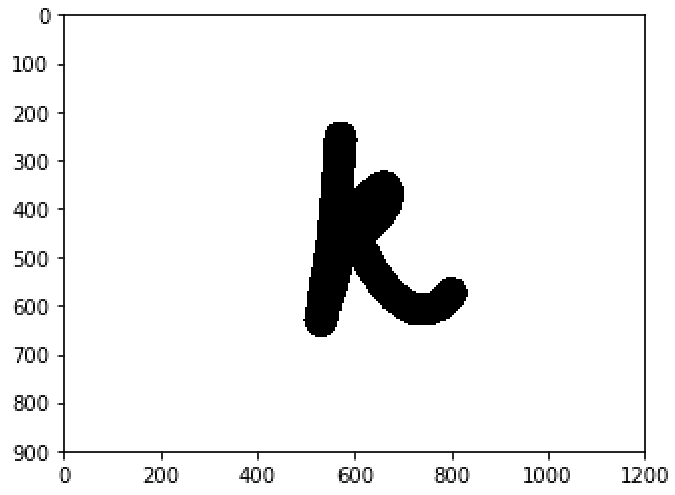
In [11]:
```
import os.path

def zeroFiller(x):
    #Input x must be a string
    if len(x)==1: x= "00%s" % (x)
    elif len(x)==2: x= "0%s" % (x)
    return x

def load_img(char_ind, samp_ind):
    """
    Returns the image from the dataset given a character and sample index.

    If the file doesn't exist, it raises an Exception with the filename.
    """
    # TODO:  Set the file name based on char_ind and samp_ind
    char_ind, samp_ind = str(char_ind), str(samp_ind)
    char_ind, samp_ind = zeroFiller(char_ind), zeroFiller(samp_ind)
    fname = "Img\Sample%s\img%s-%s.png" % (char_ind, char_ind, samp_ind )

    # TODO:  Use the os.path.isfile command to check if the file exists.
    if os.path.isfile(fname):

        # TODO:  Use the skimage.io.imread() command to read the png file and r
        img = skimage.io.imread(fname)
        return img

    else:
        error_img_msg = "%s not found" % (fname)
        raise ImgException(msg = error_img_msg)
```

Test the `load_img` function. This should:

- Plot the image in `Sample047\img047-006.png`
- Say that the `Sample047\img047-070.png` is not found.

In [12]:
```
char_ind = 47
samp_inds = [6,70]
for samp_ind in samp_inds:
    try:
        img = load_img(char_ind=char_ind, samp_ind=samp_ind)
        print("Char = %d samp=%d" % (char_ind, samp_ind))
        plt.imshow(img)
    except ImgException as e:
        print(e.msg)
```

```
Char = 47 samp=6
Img\Sample047\img047-070.png not found
```



The images in the sample directory have very high resolution. Complete the following method to find the image and place it in a 28 x 28 box. You can look at this very nice demo of the `skimage` methods here:

http://scikit-image.org/docs/dev/auto_examples/segmentation/plot_label.html (http://scikit-image.org/docs/dev/auto_examples/segmentation/plot_label.html)

The code is somewhat complex, so I have provided some of the steps, esp. for the thresholding.

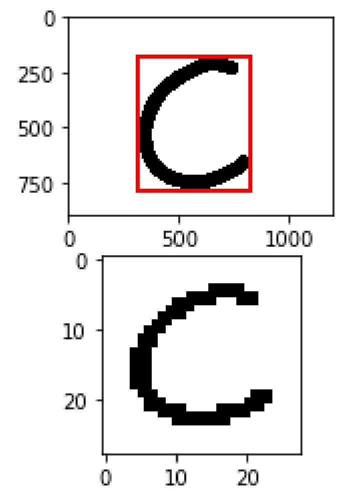
In [13]:
```
def mnist_resize(img):
    """
    Extracts a character from the image, and places in a 28x28 image to match t

    Returns:
    img1:  MNIST formatted 28 x 28 size image with the character from img
    box:   A bounding box indicating the locations where the character was foun
    """
    # Image sizes (fixed for now).  To match the MNIST data, the image
    # will be first resized to 20 x 20.  Then, the image will be placed in cent
    # offet by 4 on each side.
    nx_img = 20
    ny_img = 20
    nx_box = 28
    ny_box = 28
    offx = 4
    offy = 4

    # TODO:  Convert the image to gray scale using the skimage.color.rgb2gray m
    bw = skimage.color.rgb2gray(img)

    # Threshold the image using OTSU threshold
    thresh = threshold_otsu(bw)
    bw = closing(bw < thresh, square(3)).astype(int)

    # Get the regions in the image.
    # This creates a list of regions in the image where the digit possibly is.
    regions = regionprops(bw)

    # TODO:  Find region with the largest area.  You can get the region area fr
    area_max = 0
    for region in regions:
        if region.area > area_max:
            region_max = region
            area_max = region.area

    # Raise an ImgException if no region with area >= 100 was found
    if (area_max < 100):
        raise ImgException("No image found")

    # Get the bounding box of the character from region_max.bbox
    minr, minc, maxr, maxc = region_max.bbox
    box = [minr,minc,maxr,maxc]

    # TODO:  Crop the image in bw to the bounding box
    bw_crop = bw[minr:maxr , minc:maxc]

    # TODO:  Resize the cropped image to a 20x20 using the resize command.
    # You will need to use the mode = 'constant' option
    bw_resize = resize(bw_crop, (ny_img, nx_img), mode='constant')

    # TODO:  Threshold back to a 0-1 image by comparing the pixels to their mec
    mean = np.mean(bw_resize)
    bw_resize[bw_resize >= mean] = 1           #NOTE THAT CHAR IS WHITE AND BAC
    bw_resize[bw_resize < mean] = 0
    # TODO:  Place extracted 20 x 20 image in larger image 28 x 28
    img1 = np.zeros((ny_box,nx_box))
    img1[offy:-offy, offx:-offx] = bw_resize
    img1 = (img1 + 1)%2                        #CHAR IS BLACK AND BACKGROUND WH

    return img1, box
```

Now test the `mnist_resize` program by completing the following code. Create two subplots:

- subplot(1,2,1): The original image with the bounding box for the character that was found in the image.
- subplot(1,2,2): The MNIST resized image.

In [14]:

```
# Load an image
img = load_img(13,9)

try:
    # Resize the image
    img1, box = mnist_resize(img)

    # TODO:  Plot the original image, img, along with a red box around the capt
    # Use the mpatches.Rectangle and ax.add_patch methods to construct the rect
    minr, minc, maxr, maxc = box
    rect = mpatches.Rectangle((minc, minr), maxc - minc, maxr - minr, fill=Fals

    fig, (ax1, ax2) = plt.subplots(2, sharex=False, sharey=False)
    ax1.add_patch(rect)
    ax1.imshow(img)

    # TODO:  Plot the resized 28 x 28 image, img1.  You can use the plt_digit(i
    ax2.imshow(img1,cmap='Greys_r')

except ImgException as e:
    print(e.msg)
```



Now, run the command `nlet=1000` times to get 1000 letter images. In each iteration, select a random image from a lowercase letter and add it to a matrix `Xlet`.

In [39]:
```
# Dimensions
nlet = 1000
nrow = 28
ncol = 28
npix = nrow*ncol
Xlet = np.zeros((nlet, npix))

i = 0
while i < nlet:
    # TODO:  Generate a random character and sample
    while char_ind is not 45 and char_ind is not 51:
        char_ind = np.random.randint(37,63)
    samp_ind = np.random.randint(50)


    try:
        # TODO:  Load the image with load_img function
        img = load_img(char_ind, samp_ind)

        # TODO:  Reize the image with mnist_resize function
        img1, box = mnist_resize(img)

        # TODO:  Store the image in a row of Xlet[i,:] and increment i
        Xlet[i,:] = np.ravel(img1)
        i += 1

        # Print progress
        if (i % 50 == 0):
            print ('images captured = {0:d}'.format(i))
    except ImgException:
        # Skip if image loading or resizing failed
        pass
```

```
images captured = 50
images captured = 100
images captured = 150
images captured = 200
images captured = 250
images captured = 300
images captured = 350
images captured = 400
images captured = 450
images captured = 500
images captured = 550
images captured = 600
images captured = 650
images captured = 700
images captured = 750
images captured = 800
images captured = 850
images captured = 900
images captured = 950
images captured = 1000
```

Since this takes a long time to generate, save the matrix `Xlet` to a file `Xlet.p` using the `pickle.dump` command.

In [20]:
```
import pickle

# TODO
with open( "Xlet.p", "wb" ) as fp:
    pickle.dump( Xlet, fp)
```

Reload the data `Xlet` from the file `Xlet.p`

In [37]:
```
# TODO
with open( "Xlet.p", "rb" ) as fp:
    Xlet = pickle.load(fp)
```

Create Extended Training Data

Now, create an extended data set by combining `ndig=5000` randomly selected digit samples and `nlet=1000` letters.

- Select `ndig=5000` random samples from `Xdigs` and their labels in `ydig`.
- Rescale the letters `Xlet` to a new matrix `Xlets = 2*Xlet-1` to make the pixel values go from -1 to 1.
- Use the `np.vstack` command to create a 6000 element alpha-numeric data set `X`
- Create a corresponding label vector y where all the non-digit characters are labeled with a non-digit label, `letter_lbl=10`.

In [65]:
```
# TODO
# X = Array with 6000 characters (5000 digits + 1000 letters)
# y = Array with 6000 labels (0-9 for the digits, 10 = non-digit)

nlet = 1000
ndig = 5000
letter_lbl = 10

Xlets = 2*Xlet - 1
ylet = np.array([10] * nlet)

ind = np.random.choice(Xdigs.shape[0], ndig )

X = np.vstack([Xdigs[ind], Xlets])
y = np.hstack([ydig[ind], ylet])

print(X.shape)
```

```
(6000, 784)
```

Run the SVM classifier

First create the SVM classifer. Use an "rbf" classifier with `C=2.8` and `gamma=.0073`. Not sure if these are the best parameters, you could try to search for better ones.

In [66]:
```
from sklearn import svm

# TODO:  Create a classifier: a support vector classifier
svc = svm.SVC(probability=False,  kernel="rbf", C=2.8, gamma=.0073,verbose=10)
```

Get 5000 training samples `Xtr,ytr` and 1000 test samples `Xts,yts`. Remember to randomly select them.

In [67]:
```
# TODO

Iperm = np.random.permutation(6000)

ntr = 5000
nts = 10000
Xtr = X[Iperm[:ntr],:]
ytr = y[Iperm[:ntr]]
Xts = X[Iperm[ntr:ntr+nts],:]
yts = y[Iperm[ntr:ntr+nts]]
```

Use the `svc.fit` command to fit on the training data. This may take a few minutes

In [68]:
```
# TODO
svc.fit(Xtr,ytr)
```

```
[LibSVM]
```

Out[68]:
```
SVC(C=2.8, 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=10)
```

Measure the accuracy on the test samples. You should get about 96% accuracy. You can get better by using more training samples, but it will just take longer to run.

In [69]:
```
# TODO
yhat_ts = svc.predict(Xts)
acc = np.mean(yhat_ts == yts)
print('Accuaracy = {0:f}'.format(acc))
```

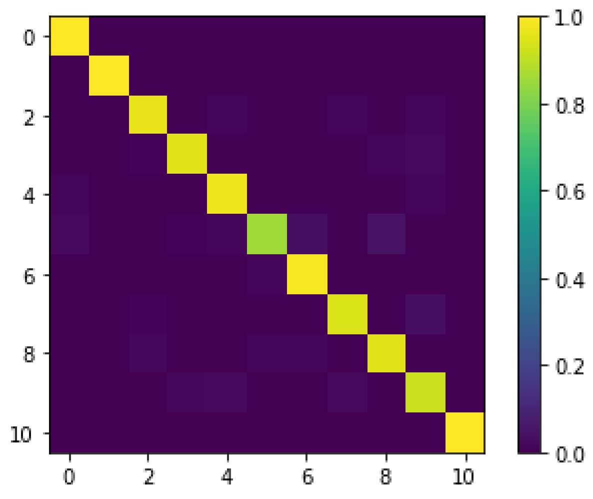
```
Accuaracy = 0.965000
```

Print the normalized confusion matrix

In [70]:
```
# TODO
from sklearn.metrics import confusion_matrix
C = confusion_matrix(yts,yhat_ts)

# Normalize the confusion matrix
Csum = np.sum(C,1)
C = C / Csum[None,:]

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

```
[[ 1.     0.     0.     0.     0.     0.     0.     0.     0.     0.     0.   ]
 [ 0.     1.     0.     0.     0.     0.     0.     0.     0.     0.     0.   ]
 [ 0.     0.     0.967  0.     0.012  0.     0.     0.013  0.     0.014  0.   ]
 [ 0.     0.     0.011  0.957  0.     0.     0.     0.     0.012  0.027  0.   ]
 [ 0.012  0.     0.     0.     0.976  0.     0.     0.     0.     0.014  0.   ]
 [ 0.024  0.     0.     0.011  0.012  0.859  0.039  0.     0.049  0.     0.   ]
 [ 0.     0.     0.     0.     0.     0.013  0.987  0.     0.     0.     0.   ]
 [ 0.     0.     0.011  0.     0.     0.     0.     0.949  0.     0.041  0.   ]
 [ 0.     0.     0.022  0.     0.     0.013  0.013  0.     0.951  0.     0.   ]
 [ 0.     0.     0.     0.022  0.024  0.     0.     0.026  0.     0.919  0.   ]
 [ 0.     0.     0.     0.     0.     0.     0.     0.     0.     0.     1.   ]]
```

Out[70]:
```
<matplotlib.colorbar.Colorbar at 0x16a00470908>
```



Plotting some error samples

We now plot some errors. Plot up to four images where yhat == 10 but yts != 10. That is, the true image was a digit, but the classifier classified it as a non-digit. Note there may be less than four such errors (when I ran it I got only three such errors). In that case, just plot only the errors you got. If there are no errors, print "No such error found"

In [71]:
```
# TODO
errList = []
nplt = 4

for i in range(1000):
    if yhat_ts[i] == 10 and yts[i] != 10:
        errList.append(i)

indx = min(nplt, len(errList))
if len(errList)>0:
    for i in range(indx):
        plt.subplot(1,indx,i+1)
        plt_digit(Xts[errList[i]])
        title = 'true={} est={}'.format(yts[errList[i]], yhat_ts[errList[i]])
        plt.title(title)
else:
    print('No such error found')
```

```
No such error found
```

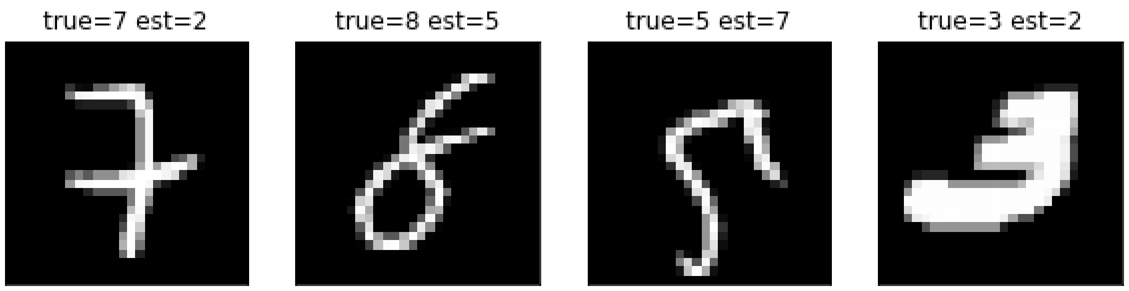
Now plot up to four images where yhat != 10, but yts == 10. That is, the image was a non-digit, but the classifier thought it was an image. I happened to get no such images. If you find no such examples, print "No such error found".

In [72]:
```
# TODO
errList = []
nplt = 4

for i in range(1000):
    if yhat_ts[i] != 10 and yts[i] == 10:
        errList.append(i)

indx = min(nplt, len(errList))
if len(errList)>0:
    for i in range(indx):
        plt.subplot(1,indx,i+1)
        plt_digit(Xts[errList[i]])
        title = 'true={} est={}'.format(yts[errList[i]], yhat_ts[errList[i]])
        plt.title(title)
else:
    print('No such error found') #VERIFY WITH CONFUSION MATRIX
```

```
No such error found
```

Finally, plot up to four images where yts != yhat and both yts < 10 and yhat < 10.

In [64]:
```
# TODO
nplt = 4
Ierr = np.where(yts != yhat_ts)[0] # No need for less than ten since No error i

plt.figure(figsize=(10, 4))
for i in range(nplt):
    plt.subplot(1,nplt,i+1)
    ind = Ierr[i]
    plt_digit(Xts[ind,:])
    title = 'true={0:d} est={1:d}'.format(yts[ind].astype(int), yhat_ts[ind].as
    plt.title(title)
```



In [ ]: