**3. Data Extraction**

The input to the system is a scanned image of the digit with a suitable format like JPEG. In this proposed system, MNIST dataset is used which was constructed from a number of scanned images from the National Institute of Standards and Technology. Handwriting samples are taken from 250 people for training and 250 other people for testing. The data set consists of 70,000 samples of scanned handwritten images. Out of this, 60,000 are used to train the network and the remaining 10,000 samples are used for testing. The set of images given for training are completely different from the images given for testing so as to know how better the system recognizes the digits.

The steps that are needed to detect handwritten digits -

1. Create a database of handwritten digits.
2. For each handwritten digit in the database, extract HOG features and train a Linear SVM.
3. Use the classifier trained in step 2 to predict digits.
4. **Creation of a Database:**

The dataset was constructed from a number of scanned documents dataset available from the National Institute of Standards and Technology (NIST). This is where the name the dataset comes from as the Modified NIST or MNIST dataset.

The first step is to create a database of handwritten digits. We are not going to create a new database but we will use the popular **MNIST database of handwritten digits.** The MNIST database is a set of 70000 samples of handwritten digits where each sample consists of a grayscale image of size 28×28. There are a total of 70,000 samples. We will use sklearn datasets package to download the MNIST database from matlab.org. This package makes it convenient to work with toy databases.

The size of of MNIST database is about 55.4 MB. Once the database is downloaded, it will be cached locally in your hard drive.

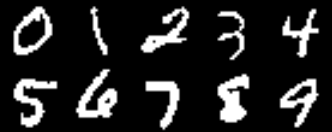
[](http://hanzratech.in/figures/mnist-dataset.png)

Figure 3.1: One sample for each handwritten digit in MNSIT database

There are approximate 7000 samples for each digit. The actual number of samples for each digit can be calculated by using collections. Counter class is used for calculating the total number of samples of all the digits.The actual samples for each digit was -

| **Digits** | **Number of samples** |
| --- | --- |
| 0 | 6903 |
| 1 | 7877 |
| 2 | 6990 |
| 3 | 7141 |
| 4 | 6824 |
| 5 | 6313 |
| 6 | 6876 |
| 7 | 7293 |
| 8 | 6825 |
| 9 | 6958 |

We will write 2 python scripts – one for training the classifier and the second for test the classifier. The first classifier is used to calculate the HOG features of all the samples in the MNIST database.

1. **Training a Classifier**

Training the classifier is the first step used to calculate the hog features of each sample and train a linear SVM with the HOG features of each sample along with the corresponding label. Training the classifier involves the following 3 steps

1. Calculate the HOG features for each sample in the database.
2. Train a multi-class linear SVM with the HOG features of each sample along with the corresponding label.
3. Save the classifier in a file

The first step is to import the required modules which are to be used in the project. The following 5 packages are to imported before being used in the project:

* 1. from sklearn.externals import joblib
  2. from sklearn import datasets
  3. from skimage.feature import hog
  4. from sklearn.svm import LinearSVC
  5. import numpy as np

We will use the sklearn.externals.joblib package to save the classifier in a file so that we can use the classifier again without performing training each time. Calculating HOG features for 70000 images is a costly operation, so we will save the classifier in a file and load it whenever we want to use it.

As discussed above sklearn.datasets package will be used to download the MNIST database for handwritten digits.

We will use skimage.feature.hog class to calculate the HOG features of each digit in the MNIST database.

We will use sklearn.svm.LinearSVCclass to perform prediction after training the classifier. We will store our HOG features and labels in numpy arrays. The next step is to download the dataset using the sklearn datasets.fetch\_mldata function. For the first time, it will take some time as 55.4 MB will be downloaded.

dataset = datasets.fetch\_mldata(“MNIST Original”)

Once, the dataset is downloaded we will save the images of the digits in a numpy array features and the corresponding labels i.e. the digit in another numpy array labels as shown below in the following two lines.

features = np.array(dataset.data, ‘int16’)

labels = np.array(dataset.target, ‘int’)

Next, we calculate the HOG features for each image in the database and save them in another numpy array named hog\_feature. Calculating HOG features for 70000 images is a costly operation, so we will save the classifier in a file and load it whenever we want to use it.

list\_hog\_fd = []

for feature in features:

fd = hog(feature.reshape((28, 28)), orientations = 9, pixels\_per\_cell = (14, 14), cells\_per\_block = (1, 1), visualize = False

list\_hog\_fd.append(fd)

hog\_features = np.array(list\_hog\_fd,’float64’)

First, we initialize an empty list list\_hog\_fd, where we append the HOG features for each sample in the database. So, in the for loop in the given above program, we calculate the HOG features and append them to the list list\_hog\_fd. Finally, we create a numpy array hog\_features containing the HOG features which will be used to train the classifier. This step will take some time, so be patient while this piece of code finishes.

To calculate the HOG features, we set the number of cells in each block equal to one and each individual cell is of size 14×14. Since our image is of size 28×28, we will have four blocks or cells of size 14×14 each. Also, we set the size of orientation vector equal to 9. So, our HOG feature vector for each sample will be of size 4×9 = 36. We are not interested in visualizing the HOG feature image, so we will set the visualise parameter to false.

The next step is to create a Linear SVM object. Since there are 10 digits, we need a multi-class classifier. The Linear SVM that comes with sklearn can perform multi-class classification.

clf = LinearSVC()

We preform the training using the fit member function of the clf object.

clf.fit(hog\_features, labels)

The fit function required 2 arguments –one an array of the HOG features of the handwritten digit that we calculated earlier and a corresponding array of labels. Each label value is from the set — [0, 1, 2, 3,…, 8, 9]. When the training finishes, we will save the classifier in a file named digits\_cls.pkl as shown in the code below –

joblib.dump(clf, “digits\_cls.pkl”, compress = 3)

The compress parameter in the joblib.dump function is used to set how much compression is done and I am quoting this from the documentation –

Compress is an integer from 0 to 9. Optional compression level for the data 0 is no compression. Higher means more compression, but also slower read and write times. Using a value of 3 is often a good compromise.

1. **Testing the Classifier**

Now, we will use another python script to test the classifier. The different code modules being used in testing the classifier is shown in detail in the following blocks.

* 1. import cv2
  2. from sklearn.externals import joblib
  3. from skimage.feature import hog
  4. import numpy as np

We will use the sklearn.externals.joblib package to save the classifier in a file so that we can use the classifier again without performing training each time. Calculating HOG features for 70000 images is a costly operation, so we will save the classifier in a file and load it whenever we want to use it.

We will use skimage.feature.hog class to calculate the HOG features of each digit in the MNIST database.

We will use sklearn.svm.LinearSVCclass to perform prediction after training the classifier. We will store our HOG features and labels in numpy arrays. The next step is to download the dataset using the sklearn datasets.fetch\_mldata function. For the first time, it will take some time as 55.4 MB will be downloaded.

clf **=** joblib**.**load("digits\_cls.pkl")

The above command is used to load the classifier from digits\_cls.pkl which is created in the previous script.

im **=** cv2**.**imread("/home/bikz05/Desktop/photo8.jpg")

This command is used to read the input image which is to be converted into computer formatted text.

im\_gray **=** cv2**.**cvtColor(im, cv2**.**COLOR\_BGR2GRAY)

im\_gray **=** cv2**.**GaussianBlur(im\_gray, (5, 5), 0)

The above lines of code is used to convert the color image which is given as input from the user to the grayscale image and perform the operations on the converted greyscale image. We then apply a Gaussian filter to the grayscale image to remove noisy pixels.

ret, im\_th **=** cv2**.**threshold(im\_gray, 90, 255, cv2**.**THRESH\_BINARY\_INV)

We convert the grayscale image into a binary image using a threshold value of 90. All the pixel locations with grayscale values greater than 90 are set to 0 in the binary image and all the pixel locations with grayscale values less than 90 are set to 255 in the binary image.

ctrs,hier **=** cv2**.**findContours(im\_th**.**copy(), cv2**.**RETR\_EXTERNAL, cv2**.**CHAIN\_APPROX\_SIMPLE)

We calculate the contours in the image and calculate the bounding box for each contour.

rects **=** [cv2**.**boundingRect(ctr) **for** ctr **in** ctrs]

**for** rect **in** rects:

cv2**.**rectangle(im, (rect[0], rect[1]), (rect[0] **+** rect[2], rect[1] **+** rect[3]), (0, 255, 0), 3)

leng **=** int(rect[3] **\*** 1.6)

pt1 **=** int(rect[1] **+** rect[3] **//** 2 **-** leng **//** 2)

pt2 **=** int(rect[0] **+** rect[2] **//** 2 **-** leng **//** 2)

roi **=** im\_th[pt1:pt1**+**leng, pt2:pt2**+**leng]

For each bounding box, we generate a bounding square around each contour. For each rectangular region, calculate HOG features and predict the digit using Linear SVM. Draw the rectangles. Make the rectangular region around the digit.

roi **=** cv2**.**resize(roi, (28, 28), interpolation**=**cv2**.**INTER\_AREA)

roi **=** cv2**.**dilate(roi, (3, 3))

Then resize each bounding square to a size of 28×28 and dilate it.

roi\_hog\_fd **=** hog(roi, orientations**=**9, pixels\_per\_cell**=**(14, 14), cells\_per\_block**=**(1, 1), visualise**=**False)

Calculate the HOG features for each bounding square. Remember here that the HOG feature vector for each bounding square should be of the same size for which the classifier was trained, else you will get an error.

nbr **=** clf**.**predict(np**.**array([roi\_hog\_fd], 'float64'))

cv2**.**putText(im, str(int(nbr[0])), (rect[0], rect[1]),cv2**.**FONT\_HERSHEY\_DUPLEX, 2, (0, 255, 255), 3)

Predict the digit using the classifier which was developed in the first step. Draw the bounding box and the predicted the digit on the input image.

cv2**.**imshow("Resulting Image with Rectangular ROIs", im)

Finally, the image is being displayed. This is code being used in our project to detect the digits in the given input image and display it on the output screen.

* Assumption during testing

There are a few assumptions, we have assumed in the testing images –

1. The digits should be sufficiently apart from each other. Otherwise if the digits are too close, they will interfere in the square region around each digit. In this case, we will need to create a new square image and then we need to copy the contour in that square image.
2. For the images which we used in testing, fixed thresholding worked pretty well. In most real world images, fixed thresholding does not produce good results. In this case, we need to use adaptive thresholding.
3. In the pre-processing step, we only did Gaussian blurring. In most situations, on the binary image we will need to open and close the image to remove small noise pixels and fill small holes.