Jervis Muindi Monday, December 12, 2011
Biometrics Final Project
Digit Recognition

Handwritten Digit Recognition:

Introduction

In the relatively young field of computer pattern recognition, one of the challenging tasks is doing accurate automated recognition of human handwriting. Indeed, this is a challenging problem precisely because there is a considerable variety in handwriting from person to person. Although, this variance does not cause any problems to us humans, it does make it more difficult to teach computers to do general handwriting recognition.

Problem Description

However, the prime focus of this project is not complete human handwriting recognition but rather a more limited form of the problem in handwritten digit recognition of numbers 0 - 9. Handwritten digit recognition has numerous applications. A good example is in facilitating the processing of cheques. The goal in such a scenario is for the computer to read in the account numbers and complete the financial transaction, such as a money transfer, automatically.

Another good example were the problem of digit recognition appears again is in the automated routing of mail by the Postal Service using zip code numbers. Obviously, in both scenarios the cost of a misclassification can be rather costly, so it is imperative that any classification that is to be used in such environments be as conservative as possible.

Thus, the goal here is to create a classifier that will be able to perform reasonably well in classifying handwritten digits and for the purposes of this project, the minimum target accuracy rate that we are targeting is 80%.

Data set

MNIST

Figure 1 - A sample of the MNIST training set

The primary dataset that is used in training the classifier is the MNIST dataset[1] published by Yann LeCun of Courant Institute at New York University. The dataset contains a 60,000 labeled training set and a 10,000 labeled test set.

The handwritten data samples come from approximately 250 different writers and completely different writers were sampled for the test set. That is, there is no intersection between the writers of the test set and training set.

The MNIST dataset is available as binary files stored in an IDX file format and a visualization of what the numbers look like can be seen in the figure above. A much bigger visualization of the data set is also available in the appendix.

The pre-processing done was to read in the data into an image matrix and perform a basic normalization procedure for each image where by each pixel value was divided by the maximum pixel value for that sample image.

Independent Dataset

In addition to the MNIST dataset, I also collected an independent dataset of handwritten digits. The data was obtained from 20 randomly selected students in the Biometrics class. Each person whose handwriting was sampled, was asked to write the digits 0 through 9 in their normal handwriting. A sample of the data collected from a single person can be seen at the appendix at the end of this report.

The preprocessing done here involved scanning the handwritten digits using a scanner and then manually extracting and labeling the 200 samples. Next, the extracted image samples are converted to grayscale, resized to 28 by 28 to have the same image dimension as the images in the MNIST training set before finally being converted to a binary image.

[1] (LeCun, 1998)

My Work

This project is a collaborative work with Varun Ravishankar and for my part of the project I am going to be working directly with the independent dataset to see how good a classifier can be built for it using the MNIST training set as the training data.

Feature Selection

The feature that I decided to use in building the classifier is the normalized **pixel values**. The pixels that make up an image are actually good features for doing digit recognition in this context because the pixel patterns will largely stay consistent for each class. That is, although each person has a unique writing style, the numbers that they write are for the most part very similar. For example, regardless of how one writes the digit one, it will likely have the element of a straight vertical line and this helps distinguish it from the other digits.

Classifier

The classifier that I am going to be building is the **nearest neighbor classifier**. I chose this classifier because not only is it very simple to implement but also because the fact that we have a huge training dataset means that such a classifier is able to do the classification task very well. The idea here is that the training set will contain many different variations of writing the digits 0 through 9 and so given any new test image of a digit, there is a fairly good chance that it will be very similar to one of the labeled images in the training set, and thus allowing the classifier to correctly classify it.

Testing Approach

For validating and testing performance of my various approaches, I took a quarter of the independently gathered dataset – 50 samples – and used them to benchmark and gauge how well a certain approach was working.

Approach 1 : Basic Preprocessing

My first approach was to just take the features that I get from the feature extraction phase and apply the nearest neighbor classifier on that to get a sense of baseline performance for the classifier. As you recall, in the feature extraction the only thing that I did was to read in the data and normalize the pixel values.

When I ran the nearest neighbor classifier[2] on this, I got an **accuracy of 72%.** While this was in line with my initial expectations, I wanted to see what could be done to reduce the error rate and improve accuracy.

Approach 2: Use a Bounding Box

In order to better understand why my current approach was not doing so well, I took a look at some the digits that it incorrectly classified. A few of these mismatched digits are shown in the figure on the right. The first number on the left is the test-digit and the number on the right is the classified digit. In the top sample, the 5 is misclassified as a 9 and in the bottom sample, the 9 is misclassified as a 4 (yes that's a 4).

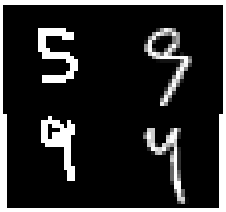


Figure 2 - Inscorrect classification

Studying these figures, and numerous other examples, I realized that one of things that was contributing to the poor performance was the fact that the test images were not correctly aligned and centered in the same way as the images in the training MNIST test. This misalignment causes trouble because the nearest neighbor classifier uses the pixel values as features and so if the digits are not centered in the same place, there is a higher chance for an incorrect classification.

To resolve, this issue, I took the approach of using a bounding box. What I did is to find the bounding box for the digits in both the MNIST training set and the independently gathered dataset and scale it to 28 by 28 pixels. With this approach[3], I was able to **improve the accuracy to 90%**

[2] This is implemented in approach1.m matlab file
[3] This is implemented in approach2.m matlab file

Approach 3: Use a Bounding Box and 3-Nearest Neighbor

Having achieved pretty a decent accuracy rate, I next tried to change the value of K in the nearest neighbor classifier from 1 to 3 and see what impact, if any, it will have on the performance. When I tried running this[4] I got a value of 88%.

Applying PCA

Although the previous approaches worked really with 90% accuracy in one case, I did ponder if it was able to do even better. To that end, I decided to try PCA and see if the performance can improve. To my surprise, I found that doing PCA and projecting the image vector from 784 dimensions to 20 dimensions and then running nearest neighbor[5] on this projected data set negatively impacted performance. This approach had a shocking 9% accuracy.

Results

So running the approaches I discussed above on full test set of 200 samples, I got the following results summarized in the table below:

Method	Accuracy Rate on Full Test Set
Approach 1 : Basic Preprocessing	69.5%
Approach 2: Use a Bounding Box	92%
Apply PCA to reduce to 20 dimensions	9%
Nearest Neighbor with K=3	90.5%

[4] This is implemented in approach3.m matlab file
[5] You can see my implementation in the dopca.m matlab file

The poor performance of PCA here suggests that for the feature that I chose to use for my classifier – pixel values – projecting down to a lower dimension resulted in the loss of separability of the original data that was previously present in the higher dimensions.

The other thing that I also tried was setting K=3 for the nearest neighbor classifier. With this approach, the accuracy decreased a little bit. This can be attributed to the fact that for some digits the Euclidean distance between different digits can be fairly close

Conclusion

From the results we can see that the approach with the best accuracy was bounding the digits in a bounding box and then applying nearest neighbor. This technique resolves the issue with prior approach where the digits were not centered and aligned in the same way.

Applying PCA to reduce the dimensionality of the image vector resulted in poor performance due to the loss of separability in the lower dimensions.

Overall though we see that the classifier did fairly well and the maximum accuracy that I was able to achieve is 92% which is a considerable margin above the 80% minimum accuracy rate that I was hoping for.

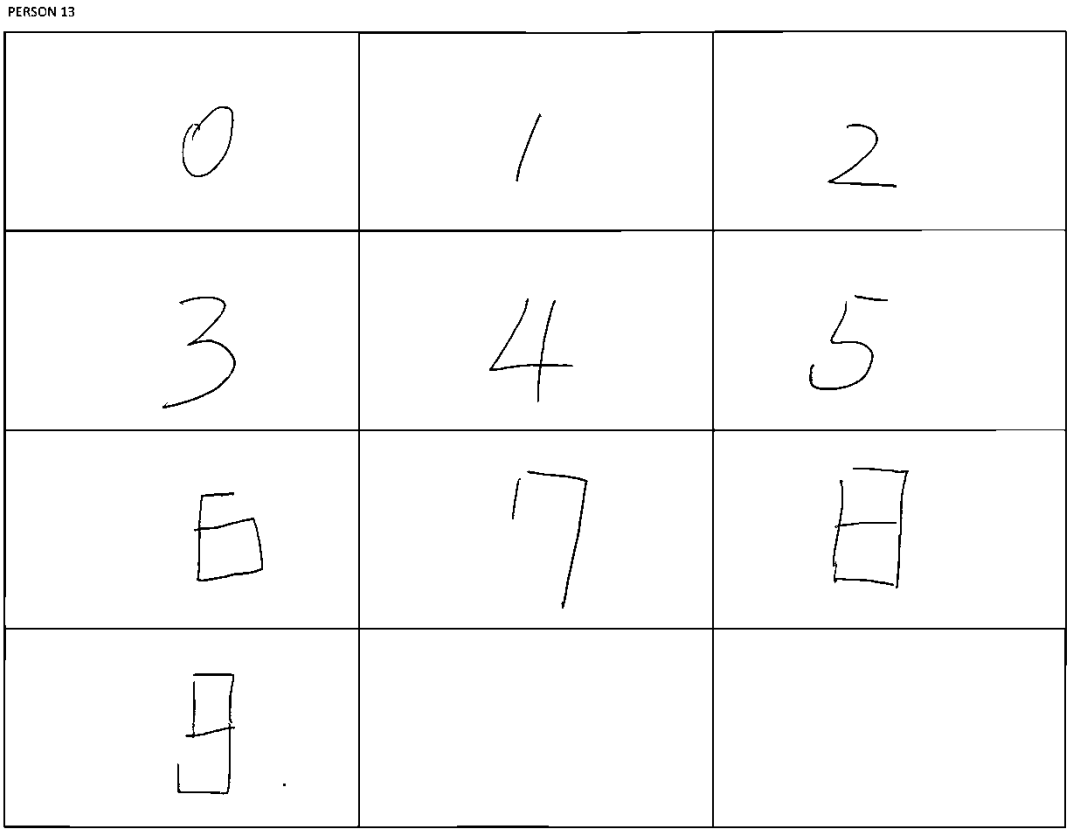
Future Work

This shows that is it possible to build a digit classification with a sufficiently high accuracy using only basic machine learning techniques. An interesting is to look at building a real-time classifier and a related application (mobile and/or desktop) that will take in user input and immediately do recognition and convert that to a digit.

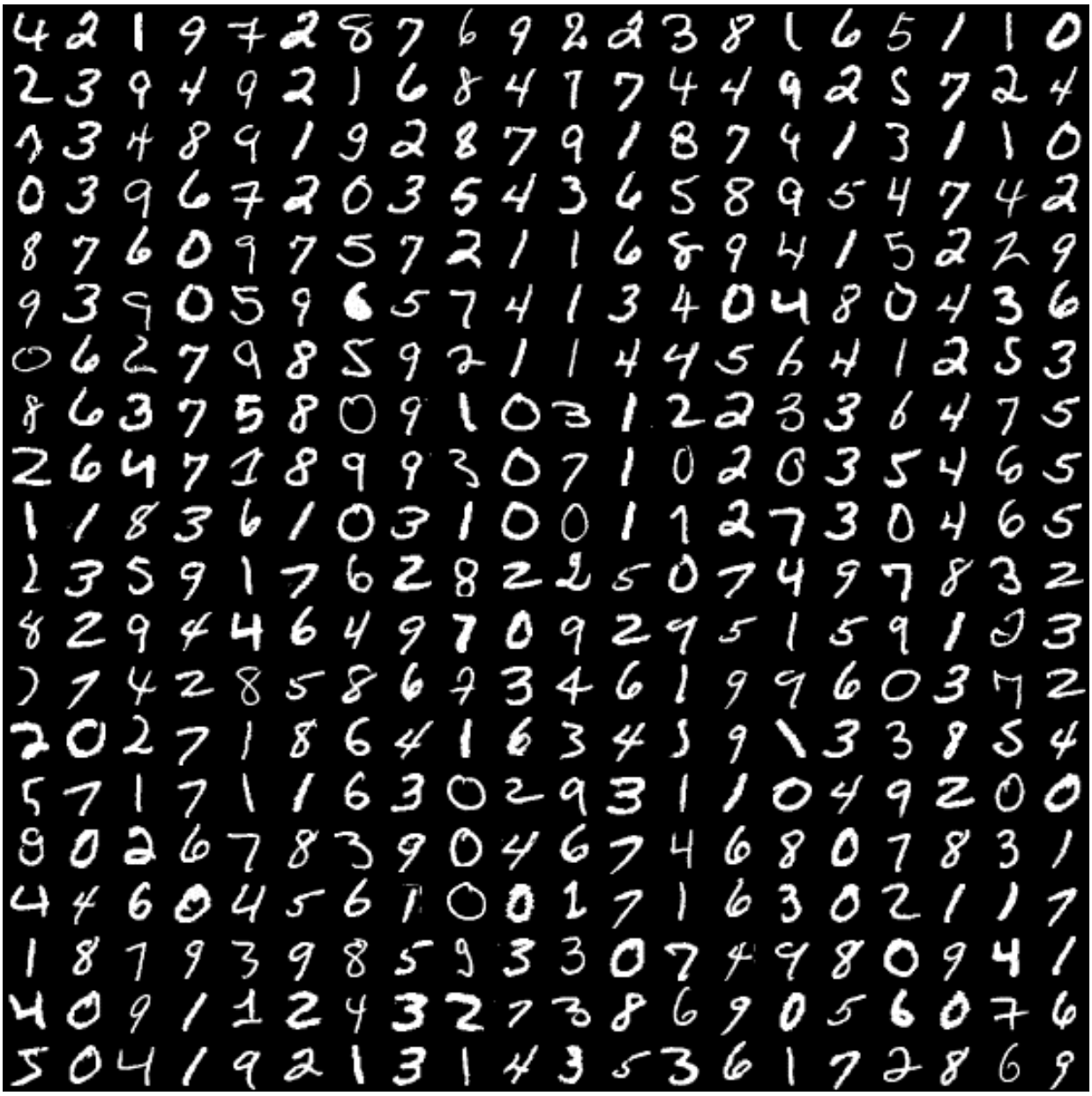
Bibliography

LeCun, Y. (1998). *MNIST handwritten digit database*. Retrieved November 2011, from New York University: http://yann.lecun.com/exdb/mnist/

Appendix



Item 1: A sample of the independently gathered test dataset



Item 2: Visualizing a sample of the training images from the MNIST database