CS4100 Final Project

Zach Ferland

December 17, 2014

1 Introduction

I started with the goal to write a simple optical character recognition program with reasonable accuracy and speed. To achieve this in the timeframe for this project, I simplified a number of aspects. Including; the characters are limited to the set of digits { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, images contain only a single line of written digits, and it simply reads of a string of digits, it is not capable of recognizing multiple multi-digit numbers. It also handles limited conditions by making the assumptions: numbers are sufficiently spaced, numbers are at least as large as average, image is portrait oriented, and the image is taken aligned and not at an angle. With further work this program can be generalized and extended to handle more of these cases. As well as recognize more symbols, such as basic math symbols {x, +, - , *}, and the alphabet.

OCR has many practical uses, including: Simplifying data entry of checks, passports, business cards, documents, receipts, etc. Recognizing license plates, creating searchable text from images, assistive technologies and many more . This project is not nearly accurate enough, fast enough, or general enough for any real world applications but it does show a glimpse of what it takes to solve these more difficult problems.

This project will accept an image as input, in the format described above, and return a string of the digits in the image. Also the classification algorithm can be tested and benchmarked on a large test set.

This project uses the MNIST dataset from *http://yann.lecun.com/exdb/mnist/* It contains 28x28 sized images of handwritten digits. There is a training set of 60,000 examples, and a test set of 10,000 examples.

2 Algorithms

OCR has multiple parts. The two high-level parts include character segmentation and character classification. Both are explained in more detail below.

Character Segmentation:

Given an image of numbers written on a white piece of paper, the goal was to accurately segment each digit into its own image, so that it could later be

classified. To do this I had to use a number of image processing algorithms, for which I made use of OpenCV.

The first step was to preprocess the image, so that segmentation would be more accurate. Given an image; it was converted from RGB to 8-bit gray-scale, noise is reduced using some blur, it is converted to a binary image using an adaptive threshold. Lastly the image is cleaned up using erosion and dilation, which reconnects segments which may have become disconnected in prior steps.

Now the image is ready to be segmented. Given the preprocessed image, I find the contours. Contours in OpenCV are found by grouping continuous pixels of the same color. I could have also segmented the image in this situation by taking the vertical and horizontal histogram of the image and then appropriately defining the segments. Once I have each contour, they get are sorted from left to right along the x-axis and each one is separated into its own image.

Lastly the format and the size of each number are normalized. The goal is get each number is the exact same size and format as the training examples in the MNIST dataset. This will allow for accurate classification. Each image is resized with anti-aliasing to fit inside a 20x20 size box. The images are already in gray scale, so the colors just get inverted. In the MNIST dataset, in the range 0 to 255, the value 0 represents white while the value 255 represents black. Lastly the images are centered by mass inside a new 28x28 image. The set of images are now in the same format as the MNIST training examples.

Character Classification:

Now that each character from the original image is segmented we can classify each one.

For character recognition I used K Nearest Neighbors. Its simplicity allowed me to code it myself in the timeframe of this project. I could have chosen one of the more complex algorithms, which have been shown to perform better, but training could have taken significant time, and I would probably have had to resort to using a library. Also the last assignment didn't cover KNN and I was curious how a more template based model like KNN would perform.

K Nearest Neighbors being a nonparametric model can become a bit slow, as it is memory based learning where it must keep all training samples on hand and when classifying a point will have to compare it to all training samples. The speed is not an issue in this simplified version though. I used Euclidean distance to judge 'closeness'. Each image has the dimension 28x28, so there exists 784 pixels. This can be represented in a single vector containing gray scale values 0-255 for all pixels.

I also implemented a version of K Nearest Neighbors using locality sensitive hashing to improve query times of nearest neighbors. Using LSH gives you full control over how much you want to trade accuracy for speed. This is often called approximate nearest neighbors, because it a probabilistic model where the neighbors returned have a high probability of being near a point, but it

won't necessarily find you the k closest points. I was able to gain a massive increase in speed with a reasonable decrease in accuracy as shown in the next section.

3 Results

This section inlcudes results of running both KNN and KNN with LSH on the MNSIT test set as well a section about running it on actual images. K is equal to 3 for all KNN without LSH. Cross validation would be useful to determine an optimal value, but computationally time consuming, k=3 was reportedly used most often on MNIST *(http://yann.lecun.com/exdb/mnist/)*. Mutiple k value results are reported for KNN with LSH. Note that KNN with LSH is approximate, so accuracy can very plus or minus a percent or two. Reported results are not an average, but a single reporting.

KNN, k = 3, Euclidean Distance

Training = 60000 Testing = 10000
Result: 9705 correct out of 10000 (97.05%)
Average Time per Classification: 1338 milliseconds

Training = 60000 Testing = 500
Result: 483 correct out of 500 (96.60%)
Average Time per Classification: 1336 milliseconds

Training = 30000 Testing = 500
Result: 477 correct out of 500 (95.40%)
Average Time per Classification: 670 milliseconds

Training = 15000 Testing = 500
Result: 468 correct out of 500 (93.60%)
Average Time per Classification: 330 milliseconds

Training = 5000 Testing = 500
Result: 459 correct out of 500 (91.80%)
Average Time per Classification: 117 milliseconds

KNN, k = 1, Euclidean Distance, with LSH

Training = 60000 Testing = 10000
Result: 9324 correct out of 10000 (93.24%)
Average Time per Classification: 22 milliseconds

KNN, k = 3, Euclidean Distance, with LSH

Training = 60000 Testing = 10000
Result: 9149 correct out of 10000 (91.49%)
Average Time per Classification: 18 milliseconds

Training = 30000 Testing = 10000
Result: 8759 correct out of 10000 (87.59%)
Average Time per Classification: 11 milliseconds

KNN, k = 5, Euclidean Distance, with LSH

Training = 60000 Testing = 10000
Result: 9203 correct out of 10000 (92.03%)
Average Time per Classification: 18 milliseconds

KNN, k = 7, Euclidean Distance, with LSH

Training = 60000 Testing = 10000
Result: 9062 correct out of 10000 (90.62%)
Average Time per Classification: 23 milliseconds

KNN, k = 9, Euclidean Distance, with LSH

Training = 60000 Testing = 10000
Result: 8910 correct out of 10000 (89.10%)
Average Time per Classification: 18 milliseconds

OCR on Images

I provided a small set of images that this program was able to accurately classify, except for a digit or two. The images include digits written by 3 different individuals. How to run the program on an image is included in the "README.txt". As stated before, this a very simplified version. All images have a single line of digits, are spaced sufficiently, are slightly larger than average and digits are aligned well when the image is taken. Overall in these limited condition it works sufficiently well. Further work would have to be done to handle more general conditions and a larger test set is needed to accurately determine effectiveness.