David Katz

Harrison Zhao

**Overview**

**Training set used**

The dataset we used can be found at <http://www.seas.upenn.edu/~taskar/ocr/>. We separated this dataset into a training set which contained 40,000 entries and a test set which contained the remaining ~12,000. Each line of the dataset represents a 16x8 image. The image was vectorized by summing the pixel values in 2x2 blocks. These feature vectors were generated for each member of the training set and test set. To perform K nearest neighbors (KNN) learning on the test set, the Euclidean distance was calculated between each test vector and each training vector. For a given test vector, we approximated p(Letter|Image) to be the percentage of times that letter appeared in the top K (which we selected as 50). These probabilities were pre-processed and stored into a text file.

As previously mentioned, we used distributed system to perform the KNN learning. Training vectors were distributed across multiple computers, and the process of calculating Euclidean distances was multithreaded. Our server, which we will demo in class, only uses one machine to perform KNN but still multithreads the checking of feature vectors.

The second component of our system was maximum a posteriori (MAP) word estimation. This was performed by attempting to maximize P(Word|Images). This is equivalent to maximizing:

P(Letteri|Imagei) was approximated from KNN, P(Letteri) was approximated from the training set and P(Word) was approximated from the Brown Corpus. This expression means that for every word with the same length of the set of images, we must loop through every letter in order to compute the most likely word. Therefore words of larger length tended to take significantly longer to process both because the sequence of images is longer and because the Brown Corpus contains more longer words. Figures we generated displaying a combination of an English dictionary and the Brown Corpus are shown below. Only words in the Brown corpus which were included in the dictionary were used (with 1 added to their frequencies), and all words present in the dictionary only were given a count of 1.

In order to test our system, we generated 1000 words of each length 1-15. Once a word was drawn from the smoothed Brown Corpus, a sequence of images with known estimated category-likelihoods was drawn from our test set to represent the word. We then attempted to predict what the word was from this sequence of characters and computed the percent accuracy and tested the percent accuracy with 1, 2 or 3 guesses. Our system also has the ability to generate words from an input file, and to display the images and the top predictions to an html file. Interestingly, including P(Letteri) resulted in slightly lower accuracy so we approximated it to be constant and thus did not include it in our MAP estimate. The results for the MAP and maximum likelihood (which does not include P(Word)) are shown below.

Thus, using MAP vs ML has a significant impact on accuracy, especially for words of length 2-5. Similarly allowing more guesses produces a large improvement in this range as well. We believe that our system tends to perform better at larger word lengths because there are fewer words which are distinguished by only one character. As such each image has a lower impact on the resulting word. Unfortunately, when drawing words of arbitrary length weighted from the Brown Corpus, our overall accuracy was just 93.1%. This is because although we achieve >99% accuracy for words of length 9 and up, smaller words occur far more frequently in English.

We also experimented with a spellchecking layer. The idea behind this was that a very poorly drawn character can completely destroy our systems ability to recognize a word. To attempt to correct this, we replaced the character with the lowest likelihood below a certain threshold with that threshold. While this did slightly increase performance for words generated with one random error, it greatly reduced performance on words with no errors, and ultimately was not used in the server we set up. The results are shown below.

There is a node.js webserver that frontends the handwriting recognition server. The node.js webserver serves the webpage where clients can write their own words and asynchronously handles requests from clients. The node.js server then sends to the handwriting recognition server a pixel map of a word to be processed via sockets. After the OCR completes, a word is sent to the node.js server. The node.js server then alerts the client with the top three results of the OCR.