Algorithms Final Project
Danielle Nash, Ryan Reede, Drew Hoo
Algorithms, Fall 2016 - Prof. Bento

I. The Problem

While getting more familiar with different aspects of this project, we debated many different solutions, before landing on our final implementation. The first of which included comparing every query image to every database image, and maintaining a list of each comparison along with the score associated with that comparison. We planned to do a number of operations to the database image, such as rotations and horizontal and vertical flips, in each comparison to account for potential changes to the image data.

After abandoning this idea because it would require a lot of passes on the data and would not scale well for multiple query images, we moved more towards a computer vision approach, which relied on extracting important features from the image. After deciding on feature extraction, we debated how to plot each image in the feature space and look for nearby neighbors. We looked at methods from Machine Learning to best understand how to do this, and a naive implementation of the k-Nearest Neighbors algorithm immediately come to mind. *k*-NN works very well in lower dimensional spaces and when time and scare are not the most immediate concern. Unfortunately for us, time was of the essence and we needed to find an algorithm that could scale and perform quicker than $O(n*d)$. Even though we were getting good results from just the first few simple features we implemented with *k*-NN, we decided to search for a faster alternative that would scale efficiently.

Through research, we found two solutions to our problems, a *k*-d tree and a locality-sensitive hash (LSH). We began to work with both and found that although neither would be retrieving results as accurately as the naive *k*-NN, we could still perform quite well and much quicker. The ultimate decision then came down to a matter of speed, and since the performance of the *k*-d tree and LSH seemed about the same (empirically) on as few as three features, but we went with the LSH method – capable of indexing and querying in higher dimensions- because we were not sure how large our feature vector would become.

II. Our Final Solution

A. Preprocessing:

We took a number steps to preprocess the images before generating a feature vector, so that the features extracted would be more relevant and normalized. First, we de-noised the image to remove small groups of pixels that were surrounded by the opposite value: For each pixel *p* in the image, we find a 4x4 matrix of pixels from the original image data such that this 4x4's upper left-hand pixel is *p.* When near an edge or corner, the data used to generate this mini-matrix would wrap around the edges of the original image. Once each 4x4 was extracted, we would check the outside 12 pixels,

verify that they were all the same color, and if so, force that the inner four pixels too became this color. Our algorithm also counted the number of instances where the outside 12 pixels were either all 0, or all 1 – and after going through the entire image, the most prominent of the two counts would tell us with high likelihood what the background color of the image was.

Following the denoise function we rotated the image around a major axis. The major axis was found using random lines through the image, and seeing which line hit the most points in the shape. After finding this line, we calculated the angle between this line and the X axis, using the inverse tangent. As long as there was a significant rotation that needed to be made (more than 5 degrees and less and 85 degrees) then the array was rotated using scipy.ndimage.interpolation. We had implemented a deskew function, but since this is only really important in MNIST images, we decided to just use the rotation function because it would work on all different types of shapes, and not just numbers or letters.

The final preprocessing step we took was the creation of the bounding box, which would start from the top-left and bottom-right corners of the image and work its way diagonally checking each column and row of image for anything other than the background color. If it finds this, then it stops searching in that direction and return the appropriate dimension from that side. Once the searchers return, the image is then resized to the new dimensions plus padding to make sure we have a little room to work with. The bounding box helped give a better idea of the dimensions of the shape, which was used later. This meant that the dimensions of a longer skinnier shape, like a '1' would be very different from a wider more square shape, like the image of a '5.'

B. Feature Extraction:

Now that we were working with clean and normalized images, we moved to feature extraction to generate the feature vector for each image. The first features to extract all dealt with pixel counts: foreground pixels, horizontal symmetry, vertical symmetry, number of shape pixels in the right half, and number of shape pixels in the left half of the image. All of these features were extracted on two sweeps which went through half of the pixels in the image, which equates to one sweep through the entire image. Each of these feature values were saved as percentages, in order to normalize the entire feature vector, but also to normalize the values across images that might be different sizes after the bounding box function. We decided on these features because they can tell a lot about the area that a shape encompasses.

Next we also used the FAST corner detection to find the number of corners and the position of the corners in the shape. We then used the position of the corners to find the centroid - since we could not approximate the number of corners each image had, we needed a generalized way to utilize corners and make sure our feature vector was the same standard length. We had implemented a Harris Corner Detection algorithm to find the number of corners, but decided to actually use the FAST implementation because it was much faster and more accurate than our own implementation.

C. Locality-Sensitive Hashing (LSH)

After feature extraction, we then used a pure python implementation of LSH to hash on the feature vector for each image. We tried many different combination of features, as well as different scoring techniques, but decided that normalizing the features to all be a percentage (less than 1) gave us the most accurate results. After hashing all of the database images, and getting the feature vector for the query image, we used LSH query([query image feature vector], *k*) method which returns an array of feature vectors the size *k* (if there are at least *k* neighbors).

While parsing the database images, we had saved both the images file name, and the images feature vector into two separate lists. After getting all of the feature vectors for the k nearest neighbors, we used this map to find the index of this feature vector, and therefore used this index to find the actual image file name.

In some extreme cases, the LSH query function returned less than k neighbors. In order to get around this and find exactly k neighbors, we then used whatever neighbors it did find, and found their closest neighbors. In the extremely rare case that a query image returned no neighbors, we decided to just print this query image k times. We decided this was the best course of action because there were no other neighbors to look at, and we really did not want to implement a slower algorithm which could give us the correct number of neighbors.

IV. Generalization

While we were working with the MNIST data set for some time, we really tried to focus our approach on a scalable implementation, that could work for other data sets as well. That is why we took so much time to rotate the image, and find the bounding box, and use percentages throughout the entire feature vector. It means that our data was much more normalized and could find similar images that were simple scalers of one another, or that were rotations of one another. That is also why we decided to use the LSH function, because we decided that it was more important to scale timewise, than it was to get the exact nearest neighbors.

VI. Empirical Runtime:

14.46s - Average Preprocessing time for 1000 db images
16.5676s - Average NN search time for 1000 db images and 1000 query images

Before finding the NN for each query image, we preprocesses the image then run neighbors. This means that, in effect, the NN search time also includes the time it takes to preprocess each of the query images. Going by the average preprocessing time, this means that query time only took approximately 2.1076s for each of the query images. This means that our LSH will efficiently scale for even larger data sets.

VII. Team Contributions:

Danielle: Implemented Horizontal and Vertical Symmetry. Implemented the rotation function, as well as Implemented a Harris Corner Detection, which was then scrapped for the FAST corners. Worked on the main file to parse each query and database image, and use the LSHash.

Ryan: Initial development on our python object class *Image*. Built out the functions for denoising and determining the background color, principal components analysis, finding surrounded pixels and brought code developed for deskew into our codebase.

Drew: Implemented bounding box and centroid algorithms. Wrote the performance evaluator executable, and empirical runtime research. Responsible for the Linux terminal, like compiling our executables, and overall navigating the server.

Work was divided evenly among the group in terms of time spent on the project.

VIII. Code Contributions

Citations:

- FAST Edge Detection: https://www.edwardrosten.com/work/fast.html
- LSH: https://github.com/kayzhu/LSHash
- Scipy Rotation: https://docs.scipy.org/doc/scipy-0.16.0/reference/generated/scipy.ndimage.interpolation.rotate.html