Fuzzy Digit Classifier

A small-scale application of fuzzy logic

By Douglas Martin

_____

CISC 871

Robin Dawes

1. Introduction

Handwritten digit recognition is an inherently fuzzy problem. For example, consider the digits in Figure 1. The digit on the left could be a 9 or a 4. The digit on the right could be a 7 and a 1. While we might not be able to classify these digits for certain, we would certainly say that they should have close to equal membership in both respective digit classes.

Figure 1. *Two handwritten digits. The digit on the left could be a 9 or a 4. The digit on the right could be a 7 or a 1.*

Classical digit classifiers put digits into crisp sets (i.e. it's one of 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9). A fuzzy digit classifier, at least the kind I wish to explore in this project, recognizes that digits may look like each other and assigns a score between 0 and 1 for each digit class (0 being not at all like the digit, and 1 being exactly like the digit). It might assign the digit on the left of Figure 1 a 0.9 for the class 9 and a 0.8 for the class 4. It's almost certainly not a 2, so it's membership in that set might be 0.1 or lower.

In [1], and later [2], Hanmandlu et al use the Box Approach for recognizing digits. After normalization and filtering – where the digits are corrected for slant, thinned, smoothed, and resized to a 42 x 32 pixel image – they are split into an imaginary grid of 24 boxes (6 rows, 4 columns). Then they calculate the average distance to the origin of every black pixel in each box (sqrt[$i^2 + j^2$] where i is the row of the pixel and j is the column). This gives them 24 features for each sample digit. They extract these features for a training set (the knowledge base) and calculate the mean and variance of each. Using these, they define a membership function for each digit as follows:

$$\mu_{\text{av}}(r) = \frac{1}{c}\sum_{j=1}^{c} \mathrm{e}^{-|x_j - m_j(r)|/\sigma_j^2(r)}$$

where r is the digit class (0..9), c is the number of features (24), x_j is the value of the jth feature (box) in sample x (the sample being considered), m_j is the mean of the jth feature of the training set, and σ^2_j is the variance of the jth feature of the training set.

They note that the variance in some of the fuzzy sets is large in some and small in others, so they use structural parameters, which they learn from gradient decent, to optimize the membership functions. These variances likely arise from different styles of digits. For example, 4's may have open or closed tops (4 vs 4).

For this project, I will attempt to replicate their classifier with some simplifications. First, I will construct handwritten digits in a precise grid of 42 x 32 pixel boxes using a tablet to avoid problems with segmentation. I will also use a thin 1pt brush to avoid thinning and smoothing. This avoids all the preprocessing steps that don't necessarily have anything to do with fuzzy logic and will take too much time to complete. Second, I will be using the membership function above without the structural parameters because implementation of gradient descent learning is beyond my current skill set. I am hoping that this will at least yield memberships that make sense (a sample digit might have a similar membership in 8 and 3) and have an accuracy of over 50%.

2. Previous Work

Just to be upfront, I have previously built a digit classifier that used the nearest neighbour algorithm to classify *typed* digits (i.e. not handwritten). However, very little of that project could be used in this one because of the very different approaches. The only reused code was a couple of methods to input an image and convert it to black and white array of pixels, and a Region class, which simply holds the array and provides a *toString* method to print the image to the console. Everything else had to be completely rewritten. Any method that was reused will be marked as such in the remainder of this report.

3. Implementation

This fuzzy digit classifier is split into 4 classes – *ImageUtils*, *FeatureExtractor*, *Classifier*, and *Region* – and a *Test* class for testing the accuracy. This section will briefly describe these classes and the methods they contain. Figure 2 shows the call graph for these classes and the code is included in Appendix A.

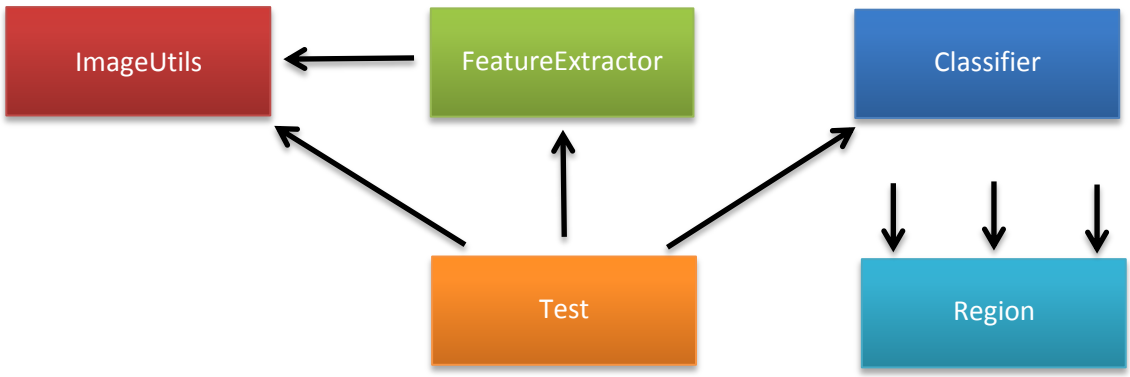

Figure 2. *The architecture of the fuzzy digit classifier. ImageUtils contains methods for extracting samples from images; FeatureExtractor contains methods to extract the features from a sample (AverageDistance of each box); Classifier contains the membership function of the digit it represents; and Test contains the main method to test the accuracy. Region is a container class used by most classes to store the sample digit array.*

ImageUtils.java

ImageUtils contains methods related to extracting multiple samples from an image file and turning them into 2 dimensional arrays. It is mainly used by the *FeatureExtractor* when extracting the features of the

training data to build the knowledge base, but it is also used by the *Test* class to extract the test samples for which to calculate memberships.

```
public static int[][][] loadImage(String filename)
```

The *loadImage* method takes the path of an image file (gif or jpg) and returns a 3D array of pixels (row, column, RGB value). This method was borrowed from my previous digit classifier.

```
public static int[][] thresholdImage2D (int[][][] RBGImage, int threshold)
```

The *thresholdImage2D* method takes a 3D image array, like the one produced by the *loadImage* method, and converts it to black and white (0 or 1) using a given threshold. So, if the average of the red, green and blue values is less than the threshold, the pixel is black, otherwise it is white. This eliminates the need for a 3D array because there is only one value for each pixel, so the returned array is 2D. This method was borrowed from my previous digit classifier.

```
public static Vector findRegions (int[][] img)
```

The *findRegions* method takes a 2D image array (i.e. a black and white image) and splits it into regions representing each sample digit in the image. In this simplified implementation, this method assumes that the image contains 100 digits in a 10 x 10 grid of 42 x 32 pixel boxes. Each digit is put into its own 42 x 32 array and stored in a Region object, which is added to a Vector. Once each digit has been extracted, the Vector is returned.

```
public static Vector findRegionsTest (int[][] img)
```

The *findRegionsTest* method is the same as the *findRegions* method except that it assumes that there are 30 digits in a 3 x 10 grid of 42 x 32 pixel boxes. This method is used for extracting the digits in the test images, which only contain 30 samples of a digit.

FeatureExtractor.java

The *FeatureExtractor* class is responsible for taking a sample set of the same digit, and calculating the features. In this case, I am using the Box Approach, so the features are the average distance to the origin of every black pixel in an imaginary grid of 24 boxes overlaid on top of the image. The feature vector (a vector containing the values of each feature for a sample) of each sample is stored in a CSV file to be read by the *Classifier* class. This saves time during testing because the features only need to be recalculated if something in the *FeatureExtractor* class changes.

```
public static void extractFeatures (String input, String output)
```

The *extractFeatures* method takes the path of an image containing samples of a digit class, extracts their features by calling the *getAverageDistance* method for each box in the imaginary grid, and saves them in a CSV file at the given output location.

```
public static double[] getFeatures(int[][] img)
```

Similar to *extractFeatures*, the *getFeatures* method takes a black and white image array of a single sample, extracts its features and returns the feature vector. This is used in the Test class for getting the feature vector of a test image.

```
private static double getAverageDistance (int[][] img, int row, int col)
```

The *getAverageDistance* method takes a black and white image array of a single sample, a row, and a column, and returns the average distance to the origin of all the black pixels in the 7 x 8 pixel box whose top left corner is located at (row, col).

Classifier.java

The *Classifier* class represents a single digit class (0-9) and contains methods to load the knowledge base, calculate the means and variances of all the features, and compute the membership of a given test sample in the digit class it represents.

```
public Classifier(int num, String samplesFile)
```

The *Classifier* constructor takes the path of a CSV file containing the extracted features of all the training samples of the *num* digit class. It then calls the 3 private methods described below to initialize the membership function.

```
private void loadSamples(String samplesFile)
```

The *loadSamples* method takes the path of a CSV file containing the extracted features of all the training samples, parses them, and puts them in a global Vector called *samples*.

```
private void calculateMeans()
```

The *calculateMeans* method uses the *samples* Vector to find the average of each feature, and puts them in the global array *means* (which is of size 24).

```
private void calculateVariances()
```

The *calulateVariances* method, similar to the *calculateMeans* method, uses the *samples* Vector to find the variance of each feature, and puts them in the global array *variances* (again, of size 24).

```
public double membership(double[] x)
```

Using the means and variances calculated by the previous methods upon initialization, the *membership* method computes the membership of the given feature vector *x*, in the digit class represented by this instance of Classifier.

4. Dataset

As mentioned earlier, the digits used to train and test this classifier were entered into a graphics program using a tablet. The image was split into a grid of 42 x 32 pixel boxes, where each box housed a single digit. This made it easier to extract them and didn't involve having to resize or filter the digits, which would have made this a much tougher project than it already is. I also used the same styles of letters (for example 4's with open tops instead of closed) to try to reduce the variations between digits of the same class. My goal is not to build the most robust and accurate digit classifier, I simply want to see if I can compute the membership functions and compare their similarities.

Training Data

I used 100 samples of each digit in a 10 x 10 grid, like the one shown in Figure 3. Each image is processed separately and used in its own classifier as the knowledge base.

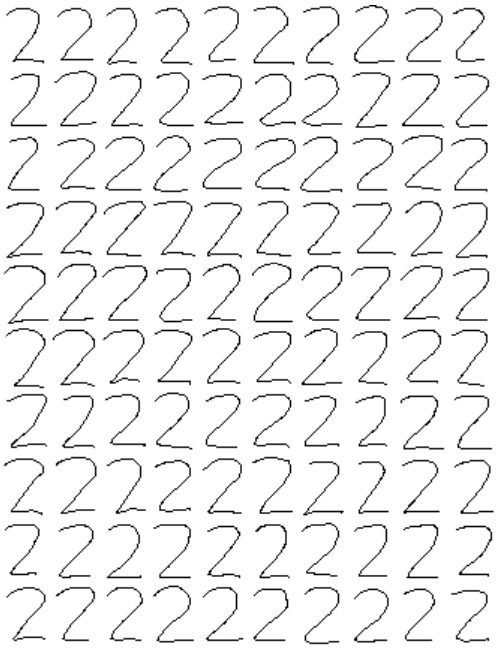

Figure 3. *An example training image containing 100 samples of the digit 2.*

Test Data

The test data was constructed similar to the training data, except there are only 30 in a 3 x 10 grid like the one shown in Figure 4.

Figure 4. *An example test image containing 30 samples of the digit 2.*

5. The Experiment

Setup

I tested my classifier in the *Test* class (see Appendix A). First, I extracted the features from the training set and initialized a *Classifier* for each digit. Then I did some isolated tests on the digits 0, 2 and 8 to gauge how well it was working. Finally I ran the classifiers on all the test data and computed the accuracy.

Results and Analysis

The output of 2 isolated tests on the digits 0 and 8 (seen in Figure 5), are shown in Figure 6. Both tests produce the correct results (i.e. the membership of the correct digit is the highest), but the memberships also make some sort of sense for the most part. For example, for the 8, the second highest memberships are in the class 3 and 0. This makes sense because 3's are only two small curves away from being an 8, and 0's are only a line segment in the middle away from being an 8.

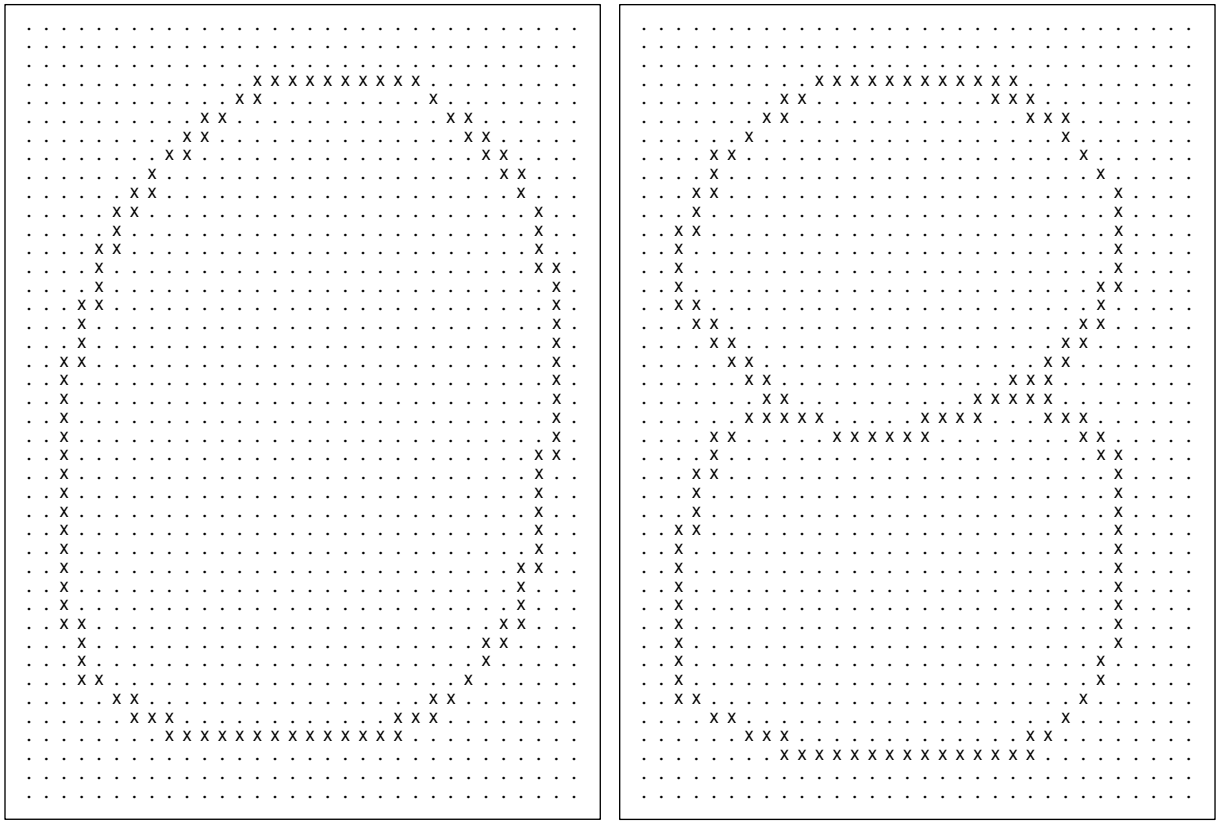

Figure 5. *Two image arrays of two sample input digits – 0 (left) and 8 (right). Dots represent white pixels and X's represent black pixels.*

```
u_0 : 0.41
u_1 : 0.33
u_2 : 0.29
u_3 : 0.38
u_4 : 0.19
u_5 : 0.29
u_6 : 0.39
u_7 : 0.18
u_8 : 0.34
u_9 : 0.23
```

```
u_0 : 0.45
u_1 : 0.29
u_2 : 0.39
u_3 : 0.47
u_4 : 0.25
u_5 : 0.42
u_6 : 0.38
u_7 : 0.23
u_8 : 0.52
u_9 : 0.27
```

Figure 6. *Console output of the memberships of the sample in Figure 5 – 0 (left) and 8(right). u_X represents the membership of the sample in the digit class X.*

These results look encouraging, however when I computed the overall accuracy (shown in Figure 7a), it was only 53%. This was slightly discouraging, so I tried using just the number of black pixels in each box for the features instead of the average distance. This yielded slightly better results (Figure 7b) at 60%. It seemed to give better accuracy for about half of the digits, so I tried using a combination of both. Where the accuracy was higher for average distance, I used that; where the accuracy was higher for number of black pixels, I used that. This yielded 65% accuracy (Figure 7c). This was the best accuracy I was able to produce.

```
    0 : 60%
    1 : 50%
    2 : 90%
    3 : 86.67%
    4 : 60%
    5 : 3.33%
    6 : 100%
    7 : 0%
    8 : 80%
    9 : 3.33%
Total : 53.33%
```

a)

```
    0 : 3.33%
    1 : 26.67%
    2 : 96.67%
    3 : 56.67%
    4 : 86.67%
    5 : 0%
    6 : 100%
    7 : 33.33%
    8 : 100%
    9 : 96.67%
Total : 60%
```

b)

```
    0 : 50%
    1 : 0%
    2 : 96.67%
    3 : 93.33%
    4 : 73.33%
    5 : 3.33%
    6 : 100%
    7 : 36.67%
    8 : 100%
    9 : 96.67%
Total : 65%
```

c)

Figure 7. *Console output of the overall accuracy of the test dataset for a) average distance, b) number of black pixels, and c) a combination of both (i.e the best method for each digit).*

6. Conclusion

While the accuracy may not have been as good as the paper it's based on, which claims around 98%, my fuzzy digit classifier certainly is fuzzy. This is really all I had hoped to get out of this project anyway.

In the future, I may try to centre the results around 0.5 because I noticed that memberships were never over 0.6 and mostly above 0.2. It would be great if I could shift it such that the correct digit had membership 1, while the others had smaller memberships.

References

[1] M. Hanmandlu, K. R. Murali Mohan, S. Chakraborty, S. Goyal, and D. R. Choudhury, "Unconstrained handwritten character recognition based on fuzzy logic," *Pattern Recognition*, vol. 36, no. 3, pp. 603–623, Mar. 2003.

[2] M. Hanmandlu and O. V. R. Murthy, "Fuzzy model based recognition of handwritten numerals," *Pattern Recognition*, vol. 40, no. 6, pp. 1840–1854, Jun. 2007.

Appendix A – Code Listing

Test.java:

```
import java.io.File;
import java.text.DecimalFormat;
import java.util.Vector;

// Test is a class for testing the digit classifier.
public class Test {

	public static void main(String[] args) {
		// For rounding decimals to 2 decimal points
		DecimalFormat decimals = new DecimalFormat("#.##");

		// Extract features from training set (if haven't already) and store them
		// in a csv file to avoid having to extract them every run.
		for (int i = 0; i < 10; i++) {
			if (!(new File("./training/" + i + ".csv").exists()))
				FeatureExtractor.extractFeatures("./training/" + i + ".gif",
					"./training/" + i + ".csv");
		}

		// Create new "classifiers" for each digit.
		// Each "classifier" loads features previously extracted and provides
		// methods to calculate the membership in the digit it represents.
		// The name "classifier" is probably the wrong choice of words.
		// Each of these are really a sub-classifier that give some measure
		// of "closeness" to the digit they represent. However, I decided to keep it
		// consistent with the architecture of my previously written (non-fuzzy)
		// digit classifier.
		Classifier zero = new Classifier(0, "./training/0.csv");
		Classifier one = new Classifier(1, "./training/1.csv");
		Classifier two = new Classifier(2, "./training/2.csv");
		Classifier three = new Classifier(3, "./training/3.csv");
		Classifier four = new Classifier(4, "./training/4.csv");
		Classifier five = new Classifier(5, "./training/5.csv");
		Classifier six = new Classifier(6, "./training/6.csv");
		Classifier seven = new Classifier(7, "./training/7.csv");
		Classifier eight = new Classifier(8, "./training/8.csv");
		Classifier nine = new Classifier(9, "./training/9.csv");

		// ---------------------------------------------------------
		// TESTING
		// ---------------------------------------------------------

		// ------------------
		// Single Digit Test
		// ------------------

		// 0
		// ------------------
		Region region = new
			Region(ImageUtils.thresholdImage2D(ImageUtils.loadImage(
				"./test-images/single0.gif"), 100), 0);
		double[] x = FeatureExtractor.getFeatures(region.getArray());
		double[] memberships = new double [10];
```

```
System.out.println("Input image:");
System.out.println(region.toString());
System.out.println("Memberships:");

memberships[0] = zero.membership(x);
memberships[1] = one.membership(x);
memberships[2] = two.membership(x);
memberships[3] = three.membership(x);
memberships[4] = four.membership(x);
memberships[5] = five.membership(x);
memberships[6] = six.membership(x);
memberships[7] = seven.membership(x);
memberships[8] = eight.membership(x);
memberships[9] = nine.membership(x);

for (int i = 0; i < 10; i++) {
	System.out.println("    u_" + i + " : "
		+ decimals.format(memberships[i]));
}
System.out.println();

// 2
// ------------------
region = new Region(ImageUtils.thresholdImage2D(ImageUtils.loadImage(
		"./test-images/single2.gif"), 100), 0);
x = FeatureExtractor.getFeatures(region.getArray());
memberships = new double [10];

System.out.println("Input image:");
System.out.println(region.toString());
System.out.println("Memberships:");

memberships[0] = zero.membership(x);
memberships[1] = one.membership(x);
memberships[2] = two.membership(x);
memberships[3] = three.membership(x);
memberships[4] = four.membership(x);
memberships[5] = five.membership(x);
memberships[6] = six.membership(x);
memberships[7] = seven.membership(x);
memberships[8] = eight.membership(x);
memberships[9] = nine.membership(x);

for (int i = 0; i < 10; i++) {
	System.out.println("    u_" + i + " : "
		+ decimals.format(memberships[i]));
}
System.out.println();

// 8
// ------------------
region = new Region(ImageUtils.thresholdImage2D(ImageUtils.loadImage(
		"./test-images/single8.gif"), 100), 0);
x = FeatureExtractor.getFeatures(region.getArray());
memberships = new double [10];

System.out.println("Input image:");
System.out.println(region.toString());
System.out.println("Memberships:");

memberships[0] = zero.membership(x);
```

```
memberships[1] = one.membership(x);
memberships[2] = two.membership(x);
memberships[3] = three.membership(x);
memberships[4] = four.membership(x);
memberships[5] = five.membership(x);
memberships[6] = six.membership(x);
memberships[7] = seven.membership(x);
memberships[8] = eight.membership(x);
memberships[9] = nine.membership(x);

for (int i = 0; i < 10; i++) {
	System.out.println("    u_" + i + " : "
		+ decimals.format(memberships[i]));
}
System.out.println();

// ------------------
// Accuracy
// ------------------

System.out.println("Overall Accuracy:");

// Keep track of total number of correct guesses (i.e. highest membership is
// in the correct digit) for all digits
double totalAccuracy = 0;

// Keep track of the total number of test cases (should be 30 * 10 digits = 300)
int testSize = 0;

// For each digit, i
for (int i = 0; i < 10; i++) {

	// Extract the regions in the test image (i.e. split the grid and
	// convert to binary array)
	Vector regions = ImageUtils.findRegionsTest(
		ImageUtils.thresholdImage2D(
			ImageUtils.loadImage(
				"./test-images/" + i + ".gif"),
					100));

	// Keep track of the number of correct guesses (i.e. highest
	// membership is in the correct digit) for current digit (i)
	int correct = 0;

	// For each test sample
	for (int j = 0; j < regions.size(); j++) {
		// Extract the features of the digit image
		x = FeatureExtractor.getFeatures(
			((Region)regions.elementAt(j)).getArray());

		// Calculate the memberships
		memberships[0] = zero.membership(x);
		memberships[1] = one.membership(x);
		memberships[2] = two.membership(x);
		memberships[3] = three.membership(x);
		memberships[4] = four.membership(x);
		memberships[5] = five.membership(x);
		memberships[6] = six.membership(x);
		memberships[7] = seven.membership(x);
		memberships[8] = eight.membership(x);
		memberships[9] = nine.membership(x);
```

```
                        // Find the digit with the maximum membership
                        int max = 0;
                        for (int k = 0; k < 10; k++){
                                if (memberships[k] >= memberships[max]) {
                                        max = k;
                                }
                        }

                        // If the digit with the digit with the maximum membership is
                        // the correct digit (i), add to total
                        if (max == i) {
                                correct++;
                        }
                } // repeat for all test sample

                // Print the accuracy for the digit
                System.out.println("      " + i + " : "
                        + decimals.format(((double)correct/regions.size())*100) + "%");
                // Add the total number of correct guesses and the total number
                // of samples to the running totals
                totalAccuracy += correct;
                testSize += regions.size();
        } // repeat for all digits 0-9

        // Print the overall accuracy
        System.out.println("  Total : " + decimals.format(
                (totalAccuracy/testSize)*100) + "%");
    }

}
```

ImagUtils.java:

```
import java.awt.Image;
import java.awt.Toolkit;
import java.awt.image.PixelGrabber;
import java.util.Vector;

// ImagUtils contains methods related to loading digit samples from an image.
public class ImageUtils {

    // Takes an image file and stores it in a 3D array (row, col, rgb value)
    public static int[][][] loadImage(String filename) {
        Image img = null;
        int[][][] imageAsArray = null;

        // This method only works for gifs and jpgs
        if (filename.endsWith(".gif") || filename.endsWith(".jpg")
                || filename.endsWith(".jpeg")) {
                img = Toolkit.getDefaultToolkit().getImage(filename);
        } else {
                System.out.println("Unsupported image format.");
        }

        // If the image is a gif or a jpg
        if (img != null) {
```

```
                ImageIcon icon = new ImageIcon(img);

                int height = icon.getIconHeight();
                int width = icon.getIconWidth();

                int[] flattenedImage = new int [width * height];

                PixelGrabber grabber = new PixelGrabber(img, 0, 0, width,
                        height, flattenedImage, 0, width);
                try {
                        grabber.grabPixels();

                        int[][] rgb = new int [flattenedImage.length][4];

                        // Extract RBG values from the flattened image
                        for (int i = 0; i < flattenedImage.length; i++) {
                                rgb[i][0] = (flattenedImage[i] & 0x00ff0000) >> 16;
                    rgb[i][1] = (flattenedImage[i] & 0x0000ff00) >> 8;
                    rgb[i][2] =  flattenedImage[i] & 0x000000ff;
                    rgb[i][3] = (flattenedImage[i] & 0xff000000) >> 24;
                 }

                        imageAsArray = new int [height][width][4];

                        int i = 0;
                        for (int row = 0; row < height; row++) {
                                for (int col = 0; col < width; col++) {
                                        imageAsArray[row][col][0] = rgb[i][0];
                                        imageAsArray[row][col][1] = rgb[i][1];
                                        imageAsArray[row][col][2] = rgb[i][2];
                                        imageAsArray[row][col][3] = rgb[i][3];

                                        i++;
                                }
                        }
                } catch (InterruptedException e) {e.printStackTrace();}
        }

        return imageAsArray;
}

// Takes a 3D array (created by the loadImage method) along with a threshold
// and converts it to black and white.
public static int[][] thresholdImage2D (int[][][] RBGImage, int threshold) {
        int width = RBGImage.length;
        int height = RBGImage[0].length;

        int [][] greyImage = new int [width] [height];

        for (int row = 0; row < width; row++) {
                for (int col = 0; col < height; col++) {
                                int greyPixel = (RBGImage[row][col][0]
                                        + RBGImage[row][col][1]
                                        + RBGImage[row][col][2]) / 3;
                                if (greyPixel >= threshold) {
                                        greyImage[row][col] = 1;     //white
                                } else {
                                        greyImage[row][col] = 0;     //black
                                }
                }
        }
```

```
		return greyImage;
	}

	// Takes an image and splits it into smaller regions for each sample digit.
	// In order to simplify the segmentation, the input images used have been
	// entered electronically (using a tablet) into a grid of 10 x 10 rectangles
	// of size 42 x 32 each.
	public static Vector findRegions (int[][] img) {
		// List of regions
		Vector regions = new Vector();
		int[][] regionArray = new int [42] [32];
		int regionNum = 0;

		for (int i = 0; i < 10; i++) {
			for (int j = 0; j < 10; j++) {
				regionArray = new int [42] [32];
				for (int row = 0; row < 42; row++) {
					for (int col = 0; col < 32; col++) {
						regionArray[row][col] =
						img[row + i + (i*42)][col + j + (j*32)];
					}
				}
				regions.add(new Region(regionArray, regionNum));
				regionNum++;
			}

		}

		return regions;
	}

	// Same as findRegions, but used for testing the accuracy with a reduced sample
	// size (3 x 10 rectangles instead of 10 x 10)
	public static Vector findRegionsTest (int[][] img) {
		Vector regions = new Vector();
		int[][] regionArray = new int [42] [32];
		int regionNum = 0;

		for (int i = 0; i < 3; i++) {
			for (int j = 0; j < 10; j++) {
				regionArray = new int [42] [32];
				for (int row = 0; row < 42; row++) {
					for (int col = 0; col < 32; col++) {
						regionArray[row][col] =
							img[row + i + (i*42)][col + j + (j*32)];
					}
				}
				regions.add(new Region(regionArray, regionNum));
				regionNum++;
			}

		}

		return regions;
	}

}
```

Region.java:

```
// A Region is simply an object to store the 2D array representing a sample digit.
// It provides a simple way to print the sample to the strdout for testing.
public class Region {

	int[][] region = null;
	int id = -1;

	public Region(int[][] regionArray, int regionNum) {
		region = regionArray;
		id = regionNum;
	}

	public String toString () {
		String outString = "    ";
		for(int row = 0; row < region.length; row++) {
			for (int col = 0; col < region[0].length; col++) {
				if (region[row][col] == 1) {
					outString += ". ";
				} else {
					outString += "X ";
				}
			}
			outString += "\n    ";
		}
		return outString;
	}

	public int[][] getArray() {
		return region;
	}
}
```

FeatureExtractor.java:

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Vector;

// FeatureExtractor is responsible for extracting and storing the features from a sample
// or a set of samples.
public class FeatureExtractor {

	// Extracts the features from all digits found in "input" and stores them in "output."
	public static void extractFeatures (String input, String output) {
		FileWriter fstream = null;
		try {
			fstream = new FileWriter(output);
		} catch (IOException e1) {e1.printStackTrace();}
		BufferedWriter out = new BufferedWriter(fstream);

		// Get all the samples in the input image.
		Vector regions = ImageUtils.findRegions(
					ImageUtils.thresholdImage2D(
						ImageUtils.loadImage(input), 100));

		// These represent the imaginary lines in a grid over the image (as used
```

```
		// in the Box Approach outlined by Hanmandlu et al).
		int[] rows = {0,7,14,21,28,35};
		int[] cols = {0,8,16,24};

		// for each sample digit
		for (int i = 0; i < regions.size(); i++){
			Region reg = (Region)regions.elementAt(i);

			// for every imaginary box in the grid
			for (int row : rows) {
				for (int col : cols) {
					try{
						// Get the average distance to the origin of
						// all the black pixels and write it to "output"
						out.write(getAverageDistance(reg.getArray(),
							row, col) + ",");

					} catch (Exception e){
						System.err.println("Error getting "
							+ "AverageDistance for " + row + ", "
								+ col);
					}
				}
			}

			// Next sample
			try{
				out.write("\n");

			} catch (Exception e) {System.err.println("Error adding newline.");}
		}

		try {
			out.close();
		} catch (IOException e) {e.printStackTrace();}
	}

	// Get the feature vector of a test sample image
	public static double[] getFeatures(int[][] img) {
		// The feature vector
		double[] x = new double [24];

		int currentFeature = 0;

		// These represent the imaginary lines in a grid over the image (as
		// used in the Box Approach outlined by Hanmandlu et al).
		int[] rows = {0,7,14,21,28,35};
		int[] cols = {0,8,16,24};

		// for every imaginary box in the grid
		for (int row : rows) {
			for (int col : cols) {
				// Get the average distance to the origin of all the
				// black pixels and put it in the array
				x[currentFeature] = getAverageDistance(img, row, col);
				currentFeature++;
			}
		}

		// Return the feature vector
		return x;
```

```
	}

	// Get the average (Euclidean) distance to the origin of a given rectangle in
	// a given sample.
	private static double getAverageDistance (int[][] img, int row, int col) {
		// Keep track of the total distance of all the black pixels
		double totalDistance = 0;
		// Keep track of the overall number of black pixels
		int totalBlackPixels = 0;

		// For each pixel in the given rectangle
		for (int i = row; i < row + 7; i++) {
			for (int j = col; j < col + 8; j++) {
				if (img[i][j] == 0) {
					totalBlackPixels++;
					// Euclidean Distance to origin (0,0)
					totalDistance += Math.sqrt(i^2 +j^2);
				}
			}
		}

		// If there are no black pixels, the average distance is undefined (NaN),
		// so return 0.
		if (totalBlackPixels == 0) {
			return 0;
		}

		// Return the average distance
		//return totalDistance/totalBlackPixels;

		//Return the total number of black pixels (found to have better results)
		return totalBlackPixels;
	}

}
```

Classifier.java:

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.Vector;

// A classifier for a digit. I use the name "classifier" in a non-traditional sense.
// A classical classifier would take an image and classify it as one of c classes.
// However, this is a fuzzy classifier so it provides a membership function for 1 digit that
// will return a number between 0 and 1, representing the closeness of a test sample digit to
// a set of training samples.
public class Classifier {

	// The digit this classifier represents
	int digit = -1;

	// The training samples.
	Vector samples = new Vector();

	// The means of each feature in the training data
	double means[] = new double[24];
```

```
// The variances of each feature in the training data
double variances[] = new double[24];

// Constructor
public Classifier(int num, String samplesFile) {
	digit = num;
	loadSamples(samplesFile);
	calculateMeans();
	calculateVariances();
}

// Takes a CSV file of training data (extracted by the feature extractor) and
// puts it in the samples Vector.
private void loadSamples(String samplesFile) {
	try {
		BufferedReader in = new BufferedReader(new FileReader(samplesFile));
		String text = in.readLine();

		// For every sample
		while (text != null) {
			// The raw String values
			String[] values = text.split(",");
			// Stores the feature vector for the current sample
			double[] x = new double [values.length];

			// Parse the data values into the feature vector
			for (int i = 0; i < values.length; i++) {
				x[i] = Double.parseDouble(values[i]);
			}

			// Add to the set of samples
			samples.add(x);

			// Next sample
			text = in.readLine();
		}

		in.close();
	} catch (FileNotFoundException e) {
		// TODO Auto-generated catch block
		e.printStackTrace();
	} catch (IOException e) {
		// TODO Auto-generated catch block
		e.printStackTrace();
	}
}

// Calculates the averages of each of the features and stores them in the means array
private void calculateMeans() {
	double total = 0;
	int numSamples =  samples.size();

	for (int i = 0; i < 24; i++) { // for each feature (box)
		for (int j = 0; j < numSamples; j++) { // for each sample
			// add feature (box) i from sample j to total
			total += ((double[])samples.elementAt(j))[i];
		}
		means[i] = total / numSamples; // mean of feature (box) i
		total = 0;
	}
```

```
	}

	// Calculates the variances of each of the features and stores them in the means array
	private void calculateVariances() {
		double total = 0;
		int numSamples =  samples.size();

		for (int i = 0; i < 24; i++) { // for each feature (box)
			for (int j = 0; j < numSamples; j++) { // for each sample
				// add square of difference of feature (box) i from
				// sample j and the mean of feature i to total
				total += Math.pow(((double[])samples.elementAt(j))[i]
						- means[i], 2);
			}
			variances[i] = total / numSamples; // mean of feature (box) i
			total = 0;
		}
	}

	// Calculates the membership in the digit that this Classifier represents.
	// From the paper:
	//	membership(x) = (1/c) * sum_j=1_to_c (e ^ - abs(x_j - mean_j) / variance_j)
	//		where  c is the number of fuzzy sets (24 features)
	//			x_j is the value of the jth feature of sample x
	//			mean_j is the training set mean of the jth feature
	//			variance_j is the training set variance of the jth feature
	//			e is euler's constant
	public double membership(double[] x) {
		double total= 0;

		for (int i = 0; i < 24; i++) {
			if (variances[i] != 0)
				total += Math.exp(-1 * (Math.abs(x[i] - means[i])
						/ variances[i]));
		}

		return total / 24;
	}
}
```