Digit Classification

Naive Bayes Classifier

David Young, Noah Prince

October 2015

Contents

1 Introduction

1.1 Submission Overview

This writeup summarizes the procedure and results of a maximum a posteriori (MAP) classification of test digits according to a learned Naive Bayes model. It also contains discussion of said results and attempts to provide some insight and reflection on the behavior of implemented algorithms. This report was produced for course "CS-440: Artificial Intelligence" at University of Illinois Urbana Champaign.

1.2 Problem Definition.

- **Features**: The basic feature set consists of a single binary indicator feature for each pixel. Specifically, the feature $F_{i,j}$ indicates the status of the (i,j)th pixel. Its value is 1 if the pixel is foreground (no need to distinguish between the two different foreground values), and 0 if it is background. The images in the training set are of size 28*28, so there are 784 features in total.

- **Training**: The goal of the training stage is to estimate the likelihoods $P(F_{i,j}|$ class) for every pixel location (i,j) and for every digit class from 0 to 9. The likelihood estimate is defined as

$$P(F_{i,j} = f| \text{ class}) = \frac{\text{\# of times pixel (i,j) has value f in training examples from this class}}{\text{Total \# of training examples from this class}} \tag{1}$$

  Likelihoods must be smoothed using Laplace smoothing. Laplace smoothing is a very simple method that increases the observation count of every value f by some constant k. This corresponds to adding k to the numerator above, and k*V to the denominator (where V is the number of possible values the feature can take on). The higher the value of k, the stronger the smoothing.

- **Testing**: The implementation here performs a maximum a posteriori (MAP) classification of test digits according to the learned Naive Bayes model. Suppose a test image has feature values $f_{1,1}, f_{1,2}, ..., f_{28,28}$. According to this model, the posterior probability (up to scale) of each class given the digit is given by

$$P(class) * P(f_{1,1}|class) * P(f_{1,2}|class) * ... * P(f_{28,28}|class) \tag{2}$$

- **Evaluation**: The true class labels of the test images from a testlabels file are used to check the correctness of the estimated label for each test digit. Performance is reported in terms of the classification rate for each digit (percentage of all test images of a given digit correctly classified). Also reported is a confusion matrix. This is a 10x10 matrix whose entry in row r and column c is the percentage of test images from class r that are classified as class c. Also included for each digit class, are the test examples from that class that have the highest and the lowest posterior probabilities according to the classifier. Think of these as the most and least "prototypical" instances of each digit class (and the least "prototypical" one is probably misclassified).

- **Odds ratios**: When using classifiers in real domains, it is important to be able to inspect what they have learned. One way to inspect a naive Bayes model is to look at the most likely features for a given label. Another tool for understanding the parameters is to look at odds ratios. For each pixel feature $F_{i,j}$ and pair of classes c1, c2, the odds ratio is defined as

$$odds(F_{i,j} = 1, c_1, c_2) = P(F_{i,j} = 1|c_1)/P(F_{i,j} = 1|c_2). \quad (3)$$

  This ratio will be greater than one for features which cause belief in c1 to increase over the belief in c2. The features that have the greatest impact on classification are those with both a high probability (because they appear often in the data) and a high odds ratio (because they strongly bias one label versus another).

  Four pairs of digits that have the highest confusion rates according to your confusion matrix are selected. For each pair, the maps of feature likelihoods are displayed for both classes as well as the odds ratio for the two classes.

2 Background

2.1 Learned Models

A learned model can be formed by allowing a program to establish probabilities of a particular feature given the object under scrutiny is of a certain class. After observing several objects, the program can then utilize this learned model to classify new objects. An illustration of this process can be seen here:

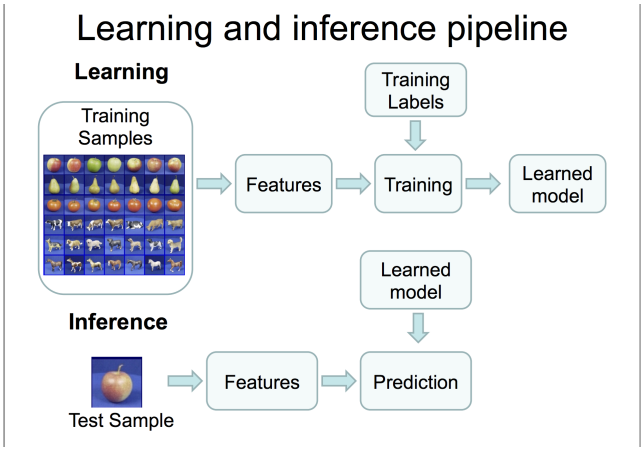

Figure 1: Learned Model Pipeline

3 Overview of Source

Obtaining the source code
The entirety of the code written for this project can be found at the following repository:

https://github.com/dcyoung/DigitClassification

Summary of source code
The following source files were written from scratch. All code is well commented with Javadocs; it should be no burden to browse for specific details.

Filename	Description
AccuracyStates.java	Builds and returns confusion matrix and other statistics.
DigitOrganizer.java	Organizes source digits and calculates likelihoods.
Digit.java	Holds the pixel and numeral data for a single digit.
FileReader.java	Reads properly formatted .txt files into a list of Digits.
HeatMapGenerator.java	Generates a heatmap for the four most confused pairs in a given confusion matrix.
TestRunner.java	Ties all of the above together.

4 Implementation

4.1 Representing Digits

A Digit class represents each digit from the files, and consists of that digit's true value and it's image data (a 2D array of 1's and 0's).

4.2 Reading the Data From Files

This is done through a function that takes as input a label and image data filename and returns a list of Digit objects. Using Java's file scanner, for every digit, It retrieves the true digit value from the label file, and the data from the image data file. It uses that data to build an object of Digit class, and add that to the list which it returns.

4.3 Training

The DataOrganizer class takes as input all of the training data. Using the training digits, this class computes the likelihood of a pixel being '1' given the digit's true value was of some class (0 through 9). This is placed in a 2D likelihood array for each class (a likelihood for every pixel in every class).

Getting the posterior probabilities of a single Digit being each class, then, involves adding the logs of the likelihood that a pixel has a given feature value (easily computable from the likelihood array for each class). The class with the maximum posterior probability is the Digit's most likely true value, as predicted by the algorithm.

The first task to train is to take all of the Digits in the list provided by the file reader and group them based on class. A list of digit classes with each entry containing a list of all Digits for that class is much more convenient to work with. This can be seen here:

```
1
public void groupDigits(){
3    int numDigits = this.allDigits.size();
     for(int i = 0; i < numDigits; i++ ){
5        groupedDigits.get(this.allDigits.get(i).getTrueValue()).add(this.allDigits.
       get(i));
     }
7 }
```

:

With the digits grouped in this way, it is trivial to loop through every digit and calculate the likelihoods based on the training data in that class. The probability of a pixel ($F_{i,j}$) having value f given a digit class is: $P(F_{i,j} = f|class) = \frac{\text{\# of times pixel (i,j) has value f in training examples from this class}}{\text{Total \# of training examples from this class}}$

To get the number of times a pixel (I, j) has value f in this case means looping through all training data for this digit and calculating the sum of the pixel values at (I, j). This can all be seen in the calculateLikelihoods function:

```
public void calculateLikelihoods(){
    int sum;
    float likelihood;

    float k  = 25; //constant
    int V = 2; //number of possible values a feature can take (here binary 0,1  so 2
      values)

    //for digit 0->9
    for(int d = 0; d < likelihoods.size(); d++ ){

       //for every pixel i,j
       for(int i = 0; i < 28; i++ ){
          for(int j = 0; j < 28; j++ ){
             //for each test img
             sum = 0;
             for(Digit tempDig : this.getGroupedDigits().get(d)){
                sum += tempDig.getPixelData()[i][j];
             }
             //P(Fij = f | class) = (# of times pixel (i,j) has value f in training
     examples from this class) / (Total # of training examples from this class).


              // smooth the likelihoods to ensure that there are no zero counts
              // Laplace smoothing is a very simple method that increases the
     observation count of every value f
              // by some constant k. This corresponds to adding k to the numerator
     above, and k*V to the
              // denominator (where V is the number of possible values the feature
     can take on).
              // The higher the value of k, the stronger the smoothing

             likelihood = (float) ((k+sum)/(k*V+this.getGroupedDigits().get(d).size
     ()));
             likelihoods.get(d)[i][j] = likelihood;
          }
       }
    }
}
```

:

Getting the posterior probability can be seen here:

```
1
public double getPixelLikelihood(int digClass, int pixRow, int pixCol, int
     featureVal){
3    double likelihoodPixelIsOne = this.getLikelihoods().get(digClass)[pixRow][pixCol
     ];
     if(featureVal == 1){
5       return likelihoodPixelIsOne;
     }
7    else{
        return 1-likelihoodPixelIsOne;
9    }
}
11
public ArrayList<Double> getPosteriorProbabilities(Digit digit){
13   ArrayList<Double> postProbs = new ArrayList<Double>();
     int[][] testImg = digit.getPixelData();
15   double tempProb;
     double PijGivenClass;
17
     //for each digClass 0->9
19   for(int digClass = 0; digClass < 10; digClass++){
        tempProb = Math.log(getProbabilityOfDigitClass(digClass));
21      //for each pixel in the testimg
        for(int i = 0; i < 28; i++ ){
23         for(int j = 0; j < 28; j++ ){
              PijGivenClass = Math.log(getPixelLikelihood(digClass, i, j, testImg[i][
     j]));
25            tempProb += PijGivenClass;
           }
27      }
        postProbs.add(tempProb);
29   }

31   return postProbs;
}
```

:

4.4 Classifying Test Data

Test data is classified by choosing the class with the maximum posterior probability (using the training DataOrganizer) for each digit. This can be seen here:

```
FileReader fr = new FileReader();
String imgDataFilename = "digitdata/trainingimages";
String labelFilename = "digitdata/traininglabels";
ArrayList<Digit> trainingDataDigits = fr.readDigitData(imgDataFilename,
    labelFilename);

imgDataFilename = "digitdata/testimages";
labelFilename = "digitdata/testlabels";
ArrayList<Digit> testDataDigits = fr.readDigitData(imgDataFilename, labelFilename);

DataOrganizer trainingData = new DataOrganizer(trainingDataDigits);
DataOrganizer testData = new DataOrganizer(testDataDigits);

AccuracyStats stats = new AccuracyStats();
for(int digClass = 0; digClass < 10; digClass++){
   for(int i = 0; i < testData.getGroupedDigits().get(digClass).size(); i++){
      Digit digit = testData.getGroupedDigits().get(digClass).get(i);
      ArrayList<Double> postProbs = trainingData.getPosteriorProbabilities(digit);
      stats.addDatapoint(digClass, postProbs.indexOf(Collections.max(postProbs)));
   }
}
```

:

4.5 Acquiring the Confusion Matrix

An AccuracyStats class exists to build the confusion matrix. This class takes datapoints one at a time; where a datapoint consists of a Digit's actual class and the class predicted by the maximum posterior probability. A non-normalized confusion matrix, where each entry is the number of datapoints of a particular value, is updated for every new datapoint. For example, if a datapoint of actual = 9, classifiedAs = 10 comes through, row 9, column 10 in the non-normalized confusion matrix is incremented by 1.

To build the actual confusion matrix after all data has been processed involves normalizing the non-normalized confusion matrix by the sum of the value in each row.

addDatapoint can be seen here:

```
public void addDatapoint(int actual, int classifiedAs) {
    this.confusionMatrixNonNormalized[actual][classifiedAs]++;
}
```

:

Normalizing can be seen here:

```
public double[][] getConfusionMatrix() {
    // Normalize confusion matrix
    int[] rowTotals = new int[10];
    for (int i = 0; i < rowTotals.length; i++) {
        for (int j = 0; j < 10; j++) {
            rowTotals[i]+=confusionMatrixNonNormalized[i][j];
        }
    }

    double[][] confusionMatrix = new double[10][10];
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < 10; j++) {
            confusionMatrix[i][j] = 1.0 * confusionMatrixNonNormalized[i][j]/
    rowTotals[i];
        }
    }

    return confusionMatrix;
}
```

:

It can be noted that the diagonal of this confusion matrix is the correct classification rates for each class.

4.6 Generating Heat Maps

Here heatmaps are generated for the four highest mis-classified pairs in the confusion matrix (one using the odds function on both digits, two for the likelihoods of each digit).

The first issue is to programmatically pick out the four highest values from the confusion matrix. This is done by adding each pair (row, column) in the confusion matrix to a Max Priority Queue data structure. This priority queue uses the value confusion matrix value to establish priority. While priority queue implementations exist in java, the comparator to establish priority had to be built like so:

```
class ConfusionComparator implements Comparator<Pair<Integer,Integer>>{
    double[][] confusionMatrix;

    public ConfusionComparator(double[][] confusionMatrix) {
        this.confusionMatrix = confusionMatrix;
    }

    // Overriding the compare method to sort the age
    public int compare(Pair<Integer, Integer> first, Pair<Integer, Integer> second)
    {
        double one = confusionMatrix[first.getKey()][first.getValue()];
        double two = confusionMatrix[second.getKey()][second.getValue()];
        if (one > two) return -1;
        if (one < two) return 1;
        return 0;
    }
}
```

:

The priority queue was built using this code:

```
PriorityQueue<Pair<Integer,Integer>> pq = new PriorityQueue<Pair<Integer,Integer
    >>(10*10, new ConfusionComparator(confusionMatrix));
for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 10; j++) {
        // don't include diagonal
        if (i != j) {
            pq.add(new Pair<Integer, Integer>(i, j));
        }
    }
}
```

:

With the priority queue established, the next step is to pop/poll four pairs from the queue and use them to generate heatmaps. Heatmaps were generated using the Java jheatchart library. This can be seen here:

```

for (int i = 0; i < 4; i++) {
    Pair<Integer,Integer> indices = pq.poll();
    int r = indices.getKey();
    int c = indices.getValue();
    System.out.println(i + " Using r = " + r + " c = " + c);

    generateHeatMap(likelihoods.get(r), "Heat chart for r=" + r, "r=" + r + "-heat-
    chart.png");
    generateHeatMap(likelihoods.get(c), "Heat chart for c=" + c, "c=" + c + "-heat-
    chart.png");

    double[][] odds = getSpecifiedOdds(likelihoods.get(r), likelihoods.get(c));
    generateHeatMap(odds, "Heat chart for odds of r=" + r + " vs c=" + c, r + "-" +
     c + "-odds-heat-chart.png");
}
```

:

The odds function can be seen here:

```

  public double[][] getSpecifiedOdds(double[][] likelihoods1, double[][]
    likelihoods2){
    double[][] odds = new double[28][28];
    for(int i = 0; i < 28; i++ ){
      for(int j = 0; j < 28; j++ ){
        odds[i][j] = likelihoods1[i][j]/likelihoods2[i][j];
      }
    }
    return odds;
  }
```

:

4.7 Tying it All Together

All of these components are tied together in the TestRunner class. This class uses all of the components to print the classification rates, confusion matrix, and to build the heatmaps. The printing and heatmap generation can be seen here:

```
System.out.println("Confusion Matrix:");
stats.printConfusionMatrix();

System.out.println();
System.out.println("Classification Rates by digit:");
double[] classRates = stats.getClassificationRates();
for(int i = 0; i < 10; i++){
    System.out.println("Digit " + i + ": " + df.format(classRates[i]));
}
System.out.println("Average Classification Rate Across all Digit Classes: \n" + df.
    format(stats.getAverageClassificationRate()));

try {
    HeatMapGenerator hmg = new HeatMapGenerator(stats, trainingData.getLikelihoods()
    );
} catch (IOException e) {
    System.out.println("Failed to create/save heat maps");
    e.printStackTrace();
}
```

:

5 Results

5.1 Classification Rates By Digit (k=1):

- Digit 0: 0.84
- Digit 1: 0.96
- Digit 2: 0.78
- Digit 3: 0.79
- Digit 4: 0.77
- Digit 5: 0.67
- Digit 6: 0.76
- Digit 7: 0.73
- Digit 8: 0.60
- Digit 9: 0.81
- Average: 0.77

5.2 Confusion Matrix

	0	1	2	3	4	5	6	7	8	9	
	0.84	0.00	0.01	0.00	0.01	0.06	0.03	0.00	0.04	0.00	0
	0.00	0.96	0.01	0.00	0.00	0.02	0.01	0.00	0.00	0.00	1
	0.01	0.03	0.78	0.04	0.01	0.00	0.06	0.01	0.05	0.02	2
	0.00	0.02	0.00	0.79	0.00	0.03	0.02	0.06	0.02	0.06	3
	0.00	0.01	0.00	0.00	0.77	0.00	0.03	0.01	0.02	0.17	4
	0.02	0.02	0.01	0.13	0.03	0.67	0.01	0.01	0.02	0.07	5
	0.01	0.07	0.04	0.00	0.04	0.05	0.76	0.00	0.02	0.00	6
	0.00	0.06	0.03	0.00	0.03	0.00	0.00	0.73	0.03	0.13	7
	0.02	0.01	0.03	0.14	0.02	0.06	0.00	0.01	0.60	0.12	8
	0.01	0.01	0.01	0.03	0.09	0.02	0.00	0.01	0.01	0.81	9

5.3 Heat Charts of Four Most Confused Numbers

4 and 9

.

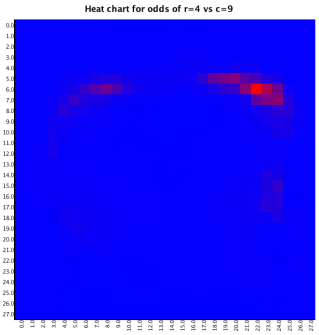

Figure 2: c = 4 r = 9 Odds

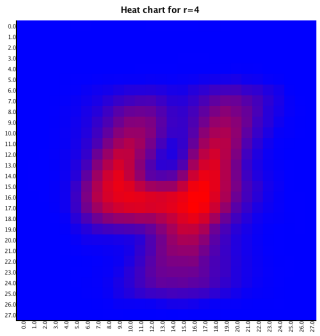

Figure 3: r = 4 likelihoods

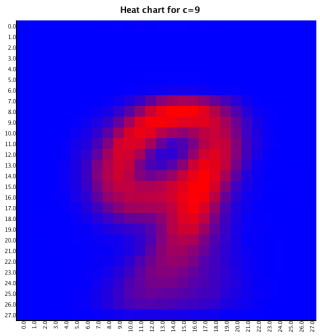

Figure 4: c = 9 likelihoods

8 and 3

.

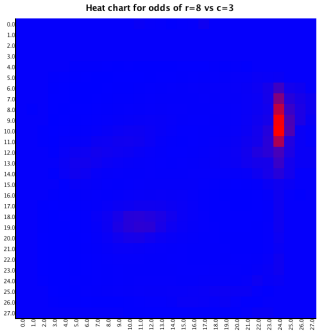

Figure 5: c = 8 r = 3 Odds

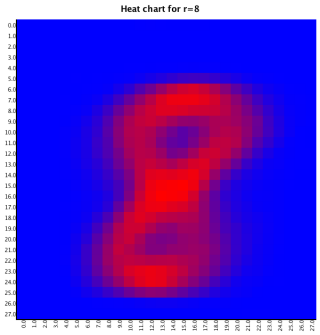

Figure 6: r = 8 likelihoods

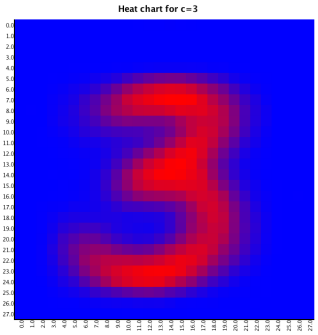

Figure 7: c = 3 likelihoods

7 and 9

.

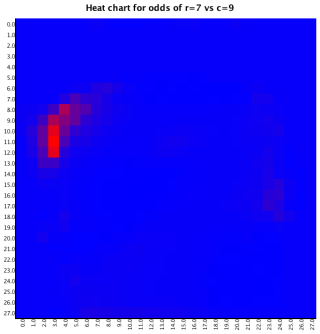

Figure 8: c = 9 r = 9 Odds

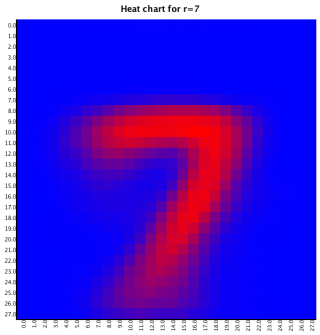

Figure 9: r = 7 likelihoods

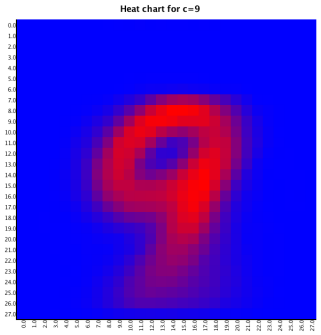

Figure 10: c = 9 likelihoods

5 and 3

.

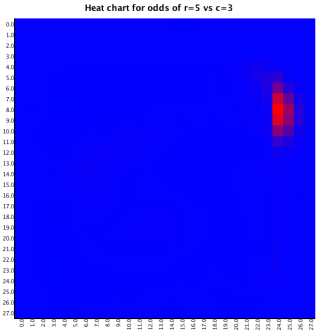

Figure 11: c = 5 r = 3 Odds

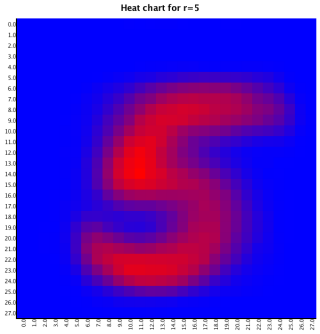

Figure 12: r = 5 likelihoods

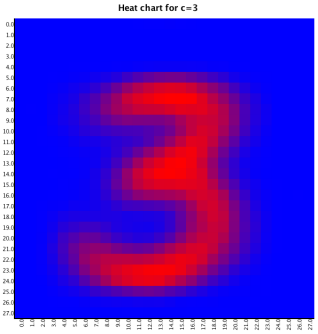

Figure 13: c = 3 likelihoods

6 Analysis & Discussion

Overall, classification rate was high. Viewing the top confused models, it's pretty clear the numbers that are likely to be confused; for example 8 and 9, which are basically a single stroke apart. The heatmaps help to illuminate this difference.

6.1 Potential Improvements

There are many more accurate models than the Naive Bayes Model; a different model could yield better results.

There are also better ways to identify features. Is a feature a single pixel, or is it better identified as a group of pixels? Are there certain pixels that are more important (IE edges of the number), and in certain areas? Modern digit classifying algorithms likely consider more than the naive model used here.

Generally, a way to increase accuracy may be to use a different 'k' for laplacian smoothing. Through testing, the classifier used k = 1 because it had the best results on average classification rates. With k=50, average rate was 0.7179. With k=1 it was 0.7711. The average classification rate with k can be seen here:

k	rate	k	rate
1	0.7711	26	0.7385
2	0.7671	27	0.7364
3	0.7630	28	0.7355
4	0.7600	29	0.7313
5	0.7568	30	0.7293
6	0.7567	31	0.7283
7	0.7557	32	0.7273
8	0.7536	33	0.7263
9	0.7546	34	0.7252
10	0.7567	35	0.7263
11	0.7515	36	0.7242
12	0.7514	37	0.7233
13	0.7492	38	0.7231
14	0.7494	39	0.7220
15	0.7484	40	0.7200
16	0.7452	41	0.7210
17	0.7440	42	0.7219
18	0.7449	43	0.7219
19	0.7440	44	0.7210
20	0.7438	45	0.7210
21	0.7450	46	0.7189
22	0.7448	47	0.7189
23	0.7436	48	0.7189
24	0.7426	49	0.7179
25	0.7406	50	0.7169

Table 1: k versus average classification rate

7 Extending Implementation to Face Detection:

The digit classifier worked by treating pre-processed pixels as features, where feature's indicated a pixel to be either a background, edge or interior pixel of a digit. Probability statistics about pixel features for the various digit classes (0-9) yielded a classification algorithm. A simple face detector can be constructed by altering the digit classifier slightly.

Pixels are still features, but instead of a pixel values representing interior, background or edge, pixels only represent the presence or absence of an edge. Reprocessing face images with an edge detector and then isolating just the pixels belonging to the edges yields a simple low resolution matrix format for a face. (See Figure 14)

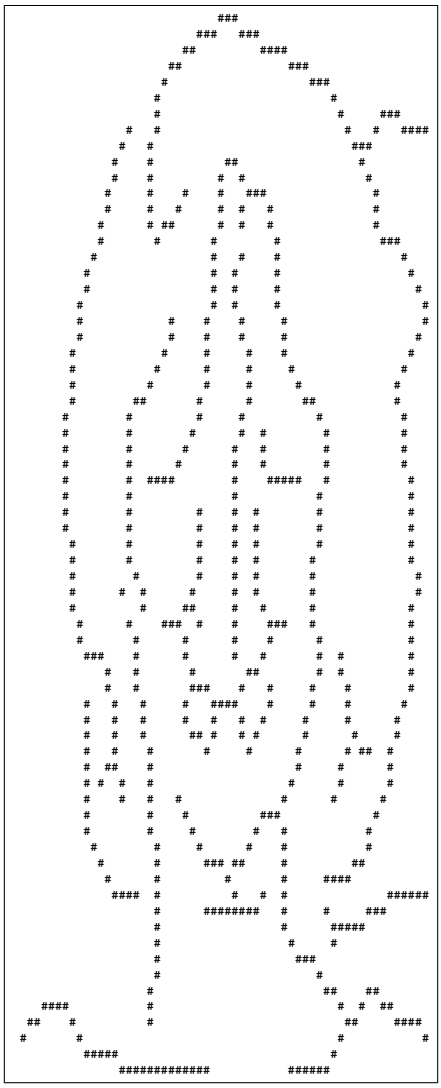

Figure 14: Example Face Representation

For digits, there were 10 classes considered during classification: [0,1,2,3,4,5,6,7,8,9]. With a comprehensive face dataset that was labeled with detailed information, it would be possible to conduct face classification similar to the digits. Classes might include gender, race, different emotions etc. However for a the simple face detector implemented here, there are only two classes: [Face, Not a Face].

The Digit class was replaced by a Face class that uses a larger matrix to store pixel features and the true digit value was replaced by a boolean flag indicating a true face. The FileReader was modified to read faces from the preprocessed text files into Face objects. The DataOrganizer now separates faces into the two classes (face & not face) instead of the 10 digit classes, and then calculates likelihoods similar to the digit implementation. The AccuracyStats class was modified to support the smaller quantity of classes. The TestRunner class contains the logic of face detection given the probabilities generated in the DataOrganizer.

The system was trained using a dataset of 451 images. Of those 451 images, 217 were actual faces, while 234 were not. The results are as follows.

$$\begin{array}{c} \\ Face \\ NotaFace \end{array} \begin{array}{c} \begin{array}{cc} Face & NotaFace \end{array} \\ \begin{pmatrix} 0.93 & 0.07 \\ 0.03 & 0.97 \end{pmatrix} \end{array}$$

Figure 15: Confusion Matrix for Face Detection

	Classification Success Rate
Input is a Face	0.9316239
Input is not a Face	0.9677419

Table 2: Success Rate for Face Detection Given Input

The code for this face detector can be viewed on the "FaceClassifier" branch of the source repository:

https://github.com/dcyoung/DigitClassification

References