CS 440 ONL: Assignment 3

Team: Asish Balu(asishtb2), Ramya Narayanaswamy(rpn2)

(3-credit students)

Part 1 - Digit Classification

In this part, we are tasked with creating a Naive Bayes classifier that can classify images of digits. The training set is composed of 5000 28x28 images of digits. The test set is composed of 1000 28x28 arrays of pixels. Each pixel can either be empty (indicating that it is part of the background) or filled with a “+” or “#” (indicating that it is part of the foreground).

Note: For this problem, we do not distinguish between the two different foreground values.

Implementation:

Our implementation was comprised of three main steps: *training, testing, and evaluation.*

Training:

In the training step, we parsed the array of training images. For each image, we knew the classification of the digit from the training labels. Using this information, we kept track of the number of times a certain pixel was in the foreground as well as the number of times the pixel was in the background given its digit classification. For instance, we counted how many times pixel [6,8] was in the background for all images classified as “8”. We called this data “value frequency”.

In addition to calculating the frequencies of each pixel value, we also calculated the frequency of each class. This was just the number of images from each digit class (for example, there were about 490 zeros and 510 threes). We called this data “class frequency”.

Using the value frequencies and class frequencies, we could calculate the likelihoods with the following formula:

$$P(F_{ij} = f \mid class) = ((\#\text{ of times pixel (i,j) has value f in training examples from this class}) + L) / ((\text{Total \# of training examples from this class}) + L*2)$$

The “likelihood” is defined as the probability that a certain pixel has a certain value given the class of the image. In this formula, L denotes the Laplacian smoothing factor. This is used so

that during testing, if a pixel has a new value that was never in the training images, the likelihood will not evaluate to zero. This prevents the classification of a digit from failing due to a single pixel value.

Choosing the Laplacian Smoothing Factor:

The Laplacian smoothing factor was chosen as 5.1 using a manual hill climbing algorithm. We ran the program once with a smoothing factor of 1. The accuracy of the classifier was about 75%. Then we slowly increased the smoothing factor by 0.1 and kept track of the accuracy. At a smoothing factor of 5.1, the accuracy peaked at 76.5% and only decreased from then on.

The hill climbing algorithm found a pretty good smoothing factor, but it can be inaccurate when there are lots of local maxima. There could be a better smoothing constant greater than 5.1, but it is unlikely that the accuracy gained will be significant.

Testing:

To test the classifier we used MAP (maximum a posteriori) classification according to our learned Naive Bayes model. For each class, we calculated the probability of a certain test image to be part of that class given the pixel states using the following formula:

$$P(class) \cdot P(f_{1,1} \mid class) \cdot P(f_{1,2} \mid class) \cdot \ldots \cdot P(f_{28,28} \mid class)$$

Then, we took the class with the highest probability as the output. To avoid underflow, we used the log of these values. We compared the outputs to the corresponding test labels to calculate the classification accuracy.

Evaluation:

We evaluated the classifier in two ways:

1) 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.

Note: The main diagonal represents the percentage (in decimal form) of correctly classified images from a particular digit class. For instance, the percentage of 1's classified correctly as 1's is about 84.4 %.

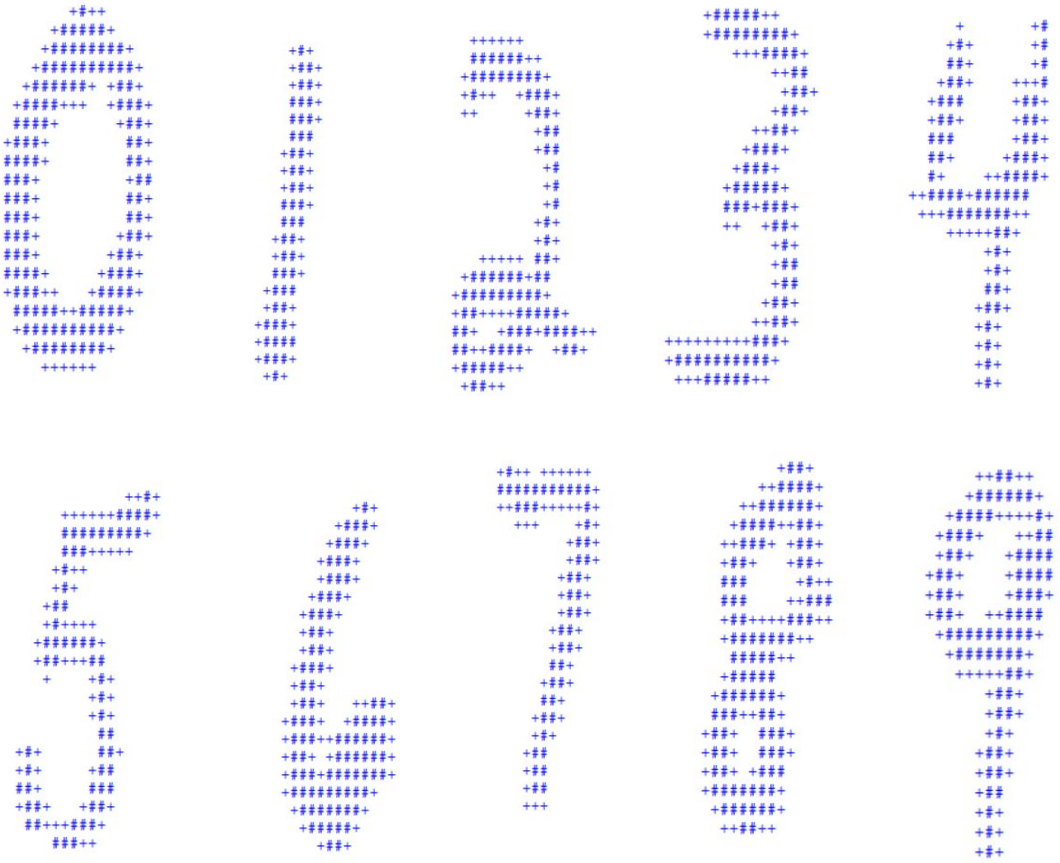
0.844	0.000	0.011	0.000	0.000	0.067	0.044	0.000	0.033	0.000
0.000	0.954	0.000	0.000	0.000	0.019	0.009	0.000	0.019	0.000
0.019	0.029	0.738	0.049	0.010	0.010	0.058	0.010	0.058	0.019
0.000	0.020	0.000	0.780	0.000	0.060	0.010	0.050	0.020	0.060
0.000	0.009	0.009	0.000	0.738	0.000	0.028	0.009	0.019	0.187
0.022	0.011	0.011	0.109	0.022	0.717	0.011	0.011	0.022	0.065
0.011	0.055	0.055	0.000	0.055	0.088	0.714	0.000	0.022	0.000
0.000	0.057	0.028	0.000	0.028	0.000	0.000	0.717	0.028	0.142
0.010	0.010	0.029	0.126	0.019	0.078	0.000	0.010	0.631	0.087
0.010	0.010	0.010	0.020	0.080	0.020	0.000	0.020	0.020	0.810

The classification rates (accuracies) of each digit are as follows:
0) 84.4 %
1) 95.4%
2) 73.8%
3) 78.0%
4) 73.8%
5) 71.7%
6) 71.4%
7) 71.7%
8) 63.1%
9) 81.0%

OVERALL : 76.5 %

For each digit, we found the test image that the classifier predicted most confidently (highest posterior probability), as well as the image that the classifier predicted least confidently (lowest posterior probability)

MOST CONFIDENT FOR EACH DIGIT:



LEAST CONFIDENT FOR EACH DIGIT:

2) Odds Ratios

The four pairs of digits with the highest confusion rates were:

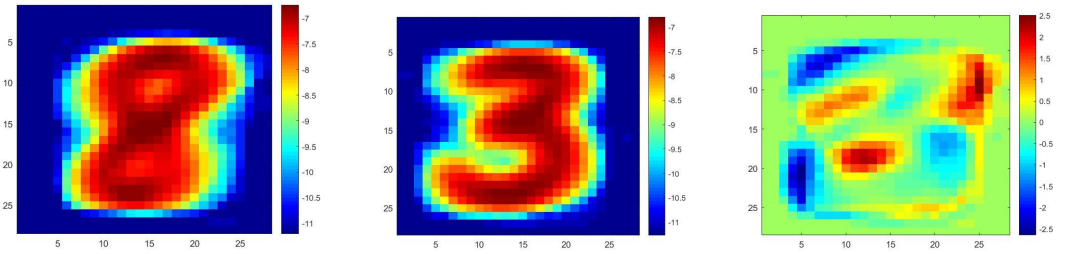
A) 8 being misclassified as a 3
B) 5 being misclassified as a 3
C) 7 being misclassified as a 9
D) 4 being misclassified as a 9

For each pair, we calculated the odds ratio using the following formula:

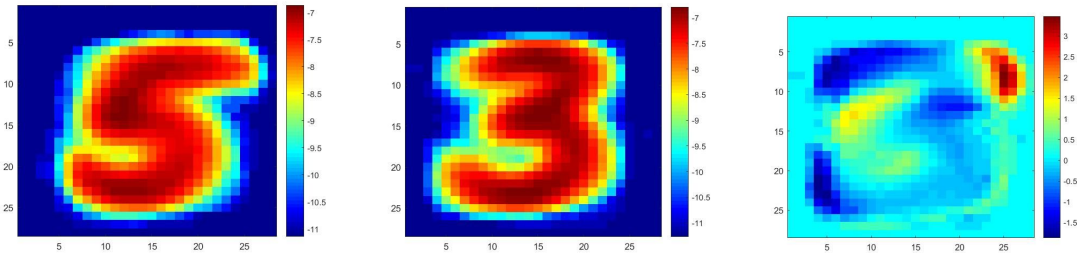
$$\mathbf{odds(F_{ij}=1,\ c_1,\ c_2) = P(F_{ij}=1 \mid c_1) \ / \ P(F_{ij}=1 \mid c_2).}$$

Then we plotted the log-likelihood graphs as well as the log-odds graph. Here are the results for each pair:

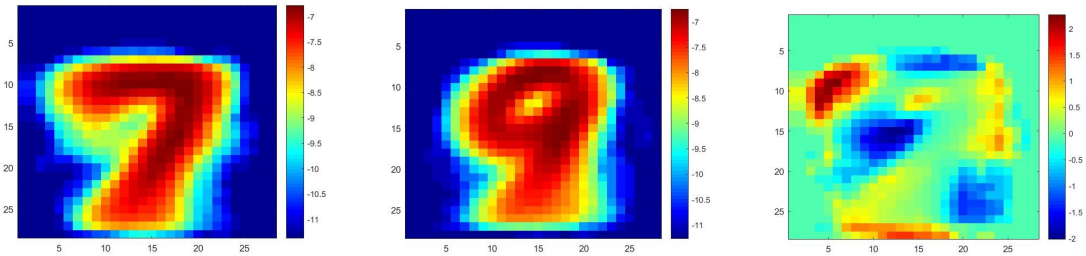
A)



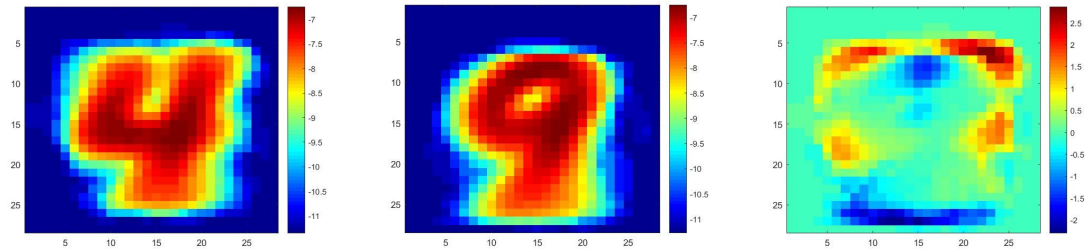
B)



C)



D)



Part1 Codes: All the Part1 contents are in the folder Part1.The main classifier was written in Python. How to run code: python DigitClassification.py. The odd ratios, highest and lowest posterior probabilities were plotted using matlab Plotter.m.

Part 2 - Audio Classification

The classification task is to identify the Hebrew words “yes” and “no” from the given spectrogram of the audio file. A Naive Bayes binary classifier was implemented. Different phases of the classifier framework are described below

Training Phase:

The spectrogram of a single training data has binarized text image and each pixel is considered as a feature. The text image has 25x10 pixels, and each pixel represents a high frequency or a low frequency. Thus, there are 250 features and each feature(pixel) could take 2 distinct values. Brief implementation details are provided below

Step1: Initialization of data structures: Our implementation uses 4 Python dictionaries (yeshigh, yeslow, nohigh,nolow) to calculate the likelihoods of Naive-Bayes classifier. Each dictionary has 250 keys corresponding to the pixel location of text image. The value of each key is the cumulative count of a feature representing high or low frequency for a given class (yes or high). For instance, yeshigh[(12,11)] = 20, represents that the pixel at (12,11) for the class “yes” has 20 occurrences of high-frequency in the training set. We also keep track of number of training samples belong to class “yes” and “no” for apriori calculation.

Step2: Update the data structures : The given “yes” and “no” training files are parsed and the the dictionary values are updated for every feature for all of the training data. At the end of step2, all measurements required for likelihood and apriori are available.

Step3: Apriori and likelihood calculations: Apriori for “yes” and “no” classes are calculated as the fraction of training samples belonging to respective classes. Calculation of likelihood is performed for every feature and the dictionary values are updated i.e the raw count of each feature is now transformed to represent likelihoods. The likelihood for a feature represented by its location (i,j) is

P(F (i,j) = f | class = c) = (#of times (i,j) has value f in training in class c) + k /((# of training samples in c) + k*2))

Here k is the Laplacian smoothing factor. The 2 in the denominator denotes each feature could take 2 distinct values (high or low frequency)

Testing Phase

The model generated by training is subject to identifying the classes in the test data set. The implementation uses 2 dictionaries, truelabel and testlabel. The key is the image id ranging from 0 to MaxImages -1 and value is true label or the predicted label from classification task respectively.

The input test files are parsed and for each test image MAP classification is performed. The posterior probability of each class is given by P(class). P(f1,1| class). …...P(f25,10 | class). As suggested in the guidelines, summation of logarithms were used to avoid underflow.

Evaluation Phase

The Laplacian smoothing factor K was swept for values ranging from 0.1 to 10 in increments of 0.5. The confusion matrix and accuracy for each K was obtained for the given test set. The value of K that gives the best accuracy is 3.1

The overall accuracy is 97% for K = 3.1. The "Yes" accuracy is 98% and "No" accuracy is 96%. The confusion matrix is given below. The rows represent the true label and columns represent the predicted label.

	Yes(Predicted Label)	No(Predicted Label)
Yes (True Label)	0.98	0.02
No (True Label)	0.04	0.96

Extra Credit for Part 2

1)Using unsegmented spectrogram file

The classifier framework for this section is similar to Part2, except for the training phase which has changes w.r.t. training data parsing. Data-structures for likelihood and priori calculations remain the same. A brief summary of changes is given below

The samples in the training data set were unsegmented. Each training file has a text image with 25 rows and 150 columns, representing 8 consecutive occurrences of Yes or No. the columns are split into blocks of 10 columns each. The resultant 15 blocks are interpreted as alternating blocks of TrainingSample(T) and Noise(B). It was assumed that noise occured when two word samples were joined. The unsegmented sequence is TBTBTBTBTBTBTBT. The class label of each training sample was encoded in the file name. The parsing logic updated the likelihood dictionaries by identifying T from the given unsegmented spectrogram and class details from the respective file name. The testing phase had a minor change as the input test files were provided in individual text files rather than a consolidated files as in Part2.

The smoothing factor did not affect the accuracy rate when it was swept from 0.1 to 10 and K was fixed to 1. **Overall Accuracy of the model is 96%.**The "Yes" accuracy is 98% and "No" accuracy is 94%.

The confusion matrix is given below. The rows represent the true label and columns represent the predicted label.

	Yes(Predicted Label)	No(Predicted Label)
Yes(True Label)	0.98	0.02
No(True Label)	0.06	0.94

2) **Sum of columns as features**

The original dataset from Part2 was used. A similar classification framework was built with a few changes as described below. Each text image is represented by 25 features, reducing the feature count from 250. The value of every feature is sum of high frequency elements in every row. The number of possible values are (0/10,1/10,2,3...10/10). The denominator of 10 was omitted in the calculations as it is a common scaling factor.

Training involved updating likelihood dictionaries Yes and No. The keys of the each dictionary are (row_identifier, sum_of_columns) and value is cumulative count of number of times that the given sum for a particular row occurs for the Yes or No training sets. At the end of training apriori and likelihoods are calculated. The likelihood is given by
P(F (i,j) = f | class = c) = (#of times (i,j) has value f in training in class c) + k /((# of training samples in c) + k*11)), note that the denominator is updated to represent 11 distinct values that a feature could take.

There was not much change in accuracy when K was swept from 0.1 to 10. K was fixed to 1 **The overall accuracy rate is 95%**, which is slightly less than individual pixels as features (97%) as expected. The “Yes” accuracy is 98% and “No” accuracy is 92%.

The confusion matrix is given below. The rows represent the true label and columns represent the predicted label.

	Yes (Predicted Label)	No(Predicted Label)
Yes (True Label)	0.98	0.02
No (True Label)	0.08	0.92

Part2 Codes: All files for part2 are within the folder Part2. The individual subfolders represent all the sections described above. Training.py codes training phase, NaiveBayesTest.py codes the testing phase. The top-level is TestTraining.py. How to run : python TestTraining.py

Individual contribution statement:
Part1 : Asish Balu
Part2 and extra credits : Ramya Narayanaswamy