CS 440 MP3
Section Q4
4 Credits

Christian Howard
howard28@illinois.edu

Abstract

Within this MP, software was developed to create and evaluate a Maximum a Posteriori classifier. This generic software was applied to a set of problems such as classifying handwritten digits, classifying face vs. non-face images, classifying Hebrew 'Yes' vs. 'No', and classifying spoken digits. The algorithms developed performed reliably, classifiers reaching at least 80% accuracy with some reaching more than 95% classification accuracy. Within the handwritten digits classifier and face classifier problems, some work was done into investigating the value and time complexity of using various block feature representations based on the original image data. It was found that there is a sweet spot in terms of feature block sizes that will have the best accuracy. Additionally, theoretical complexity trends were validated against experimental time measurements recorded during the training and testing phases of the handwritten digit classification process, helping to better contrast the pros and cons of the different feature representations.

Contents

1 Naive Bayes Formulation

1.1 Bayes Rule Classifier Formulation

Before diving into each of the problems for this MP, a discussion of the general mathematical formulation is an important first step and will allow for easily formulating the problems later.

Let us assume our goal is to develop a classifier that can take some input $x \in \mathcal{X} = \mathcal{V}^d$, where $\mathcal{V} = \{v_1, v_2, \cdots, v_m\}$ are the values a component of x can take, and produce some class $\omega \in \Omega = \{\omega_1, \cdots, \omega_p\}$ that is most likely representative when the input $x = v$, where v is some value in $\mathcal{X}$. We can choose the classifier to be the following:

$$\omega^* = \arg\max_{\omega \in \Omega} P(\omega | x = v) \tag{1}$$

Now using Bayes Rules and a Conditional Independence assumption, we can use the following relationship to compute the classifier:

$$\begin{aligned} P(\omega|x) &= \frac{P(\omega)P(x = v|\omega)}{\sum_{\hat{\omega}\in\Omega} P(x = v|\hat{\omega})P(\hat{\omega})} \\ &= \frac{P(\omega)\prod_{l=1}^{d} P(x_l = v_l|\omega)}{\sum_{\hat{\omega}\in\Omega} P(x = v|\hat{\omega})P(\hat{\omega})} \end{aligned}$$

Since the denominator is just a normalization constant shared between each $P(\omega|x = v)$, we can ignore computing it and just use the numerator to help choose the class. This means the classifier rule actually ends up being:

$$\omega^* = \arg\max_{\omega \in \Omega} P(\omega) \prod_{l=1}^{d} P(x_l = v_l|\omega) \tag{2}$$

To make things more reasonable to compute in practice due to finite precision, without changing the classifier result, we can $\log(\cdot)$ the terms in the classifier:

$$\omega^* = \arg\max_{\omega \in \Omega} \log P(\omega) + \sum_{l=1}^{d} \log P(x_l = v_l|\omega) \tag{3}$$

1.2 Learning with Discrete valued Features

Based on 3, it is obvious that $P(\omega)$ and $P(x_l = \hat{v}|\omega)$ need to be computed $\forall \omega, x_l, \hat{v}$. As discussed in the MP problem statement, usage of Laplace Smoothing will be utilized to smooth the categorical data and make the algorithms we use more robust.

Let us assume we have some training dataset of inputs and label outputs D_T, defined as $D_T \subset D = \{(x, \omega) | x \in \mathcal{X}, \omega \in \Omega\}$. Let us define $G_\omega : \Omega \times D \to \mathcal{P}(\mathcal{X})$ as a map that, given some class and dataset, produces a subset of the input data that corresponds to the class ω. Using this, we can define the computation for each of the desired probabilities as the following:

$$P(\hat{\omega}) = \frac{|G_\omega(\hat{\omega}, D_T)|}{|D_T|} \tag{4}$$

$$P(x_l = \hat{v} | \hat{\omega}) = \frac{\alpha + \sum_{\hat{x} \in G_\omega(\hat{\omega}, D_T)} \mu(\hat{x} \cdot \hat{e}_l, \hat{v})}{\alpha |\mathcal{V}| + |G_\omega(\hat{\omega}, D_T)|} \tag{5}$$

where α is the Laplace Smoothing constant and $\mu : \mathcal{V}^2 \to \mathbb{R}$ is defined as the following:

$$\mu(v_1, v_2) = \begin{cases} 1 & v_1 = v_2 \\ 0 & v_1 \neq v_2 \end{cases} \tag{6}$$

These quantities are computed for each $\hat{\omega} \in \Omega, \hat{v} \in \mathcal{V}$, and each component of x. With these equations, the training can be done quite simply. To get a feel for the space and time complexity, the naive time complexity for computing (4) is $O(pN)$, where $N = |D_T|$, with a memory complexity of $O(p)$. As for (5), this quantity can be computed in $O(dmpN)$ with a space complexity of $O(dmp)$. Additionally, if we want to then evaluate this model after training, we can find the evaluation time complexity to be $O(pd)$ for each input point.

1.3 Implementation Discussion

Given the theoretical representation discussed above, the implementation was designed to allow for generic abstraction so I could tackle new problems by just modifying some of the defining quantities, like d, V, Ω. The code basically implemented a Discriminant class that would take some dataset for a class, a Laplace Smoothing constant, and the total amount of data and produce a trained object that could compute class posteriors and likelihoods. Then, a separate function would take a list of these Discriminant objects and would perform the MAP computation for some set of input data.

To keep things separate, I implemented methods to read in the data from disk in a manner that allowed me to pass in some functor that could transform the baseline input data into data with new features. I used this to easily go from the baseline pixel features to Disjoint and Overlapping block features, for example, with lots of code reuse. This allowed me to separate data reading and feature extraction from the classifier model in a very modular way.

2 Problem 1 — Digit Classification

2.1 Single Pixels as Features

2.1.1 Problem Formulation

Within this problem, we are handed a dataset of square images, each with $n = 28$ rows and columns, read into storage with binary values at each pixel. For ease of use, each image is vectorized into a column vector such that $V = \{0, 1\}$ and $d = n^2$, given the definitions in Section 1. Note that we also define $\Omega = \{0, 1, \cdots, 9\}$, where each value is a label for the corresponding hand-written digit. Using this formulation, it is trivial to use the generic algorithms based on the formulation in Section 1 to train and test the classifier.

2.2 Results

After a training run-time of around 0.155 seconds and testing run-time of around 18.08 seconds, the classifier was able to achieve an overall classification accuracy of 77.1%. The individual classification accuracy for each digit can be seen in Table 1. This optimal classification was found with a Laplace Smoothing constant of $\alpha = 0.1$. I found that increasing the Laplace Smoothing constant actually hurt the training performance, which I found interesting.

Digit	Classification Accuracy
0	84.4%
1	96.2%
2	77.6%
3	79.0%
4	76.6%
5	67.3%
6	75.8%
7	72.6%
8	60.1%
9	80.0%

Table 1: Accuracy for individual digit classifiers

If we look at the Confusion Matrix in Figure 1, we can see that most of the misclassifications were when it misclassified 4 as 9, misclassified 5 as 3, misclassified 7 as 9, and misclassified 8 as 3. When thinking about the shapes of these digits, these misclassifications seem reasonable. For each of these misclassification pairs, we see their associated log likelihood and log odds ratio plots in Figures 2, 3, 4, 5.

As a further facet to investigate, Figures 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 show the testing data examples from each class that have the highest and lowest posterior probabilities for each digit. It is interesting to note that the testing data with the lowest posterior probabilities generally barely resembles the digit it is labeled as while the testing data with the highest posterior probabilities definitely resembles a clear version of the digit it is labeled as.

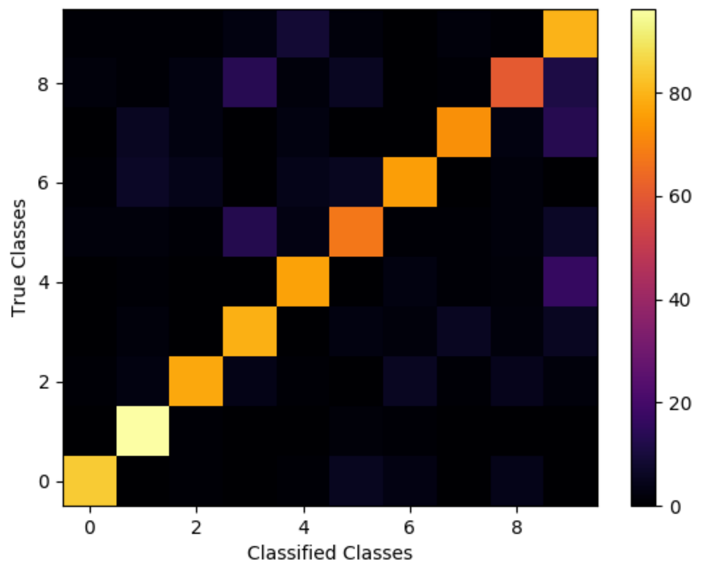


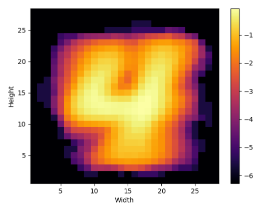
Figure 1: Confusion Matrix for Single Pixel Features

2.3 Disjoint Block as Features

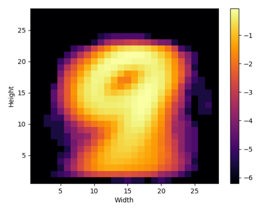
2.3.1 Problem Formulation

Within this problem, we take the same dataset of square images, each with $n = 28$ rows and columns of pixels with binary values, and create new set of features based on taking disjoint subregions of a given image and representing them as a single value. For simplicity, each image $I \in B_I = \{0, 1\}^{n \times n}$ and define $V_{rq} = \{0, 1, \cdots, 2^{rq} - 1\}$. Then let us define the operator $F_{rq} : B_I \rightarrow V_{rq}^{\frac{n}{r} \times \frac{n}{q}}$ as the map that converts an image into a smaller matrix made up of the new, disjoint features. The algorithm representing F_{rq} for computing the new set of features is found in Algorithm 1.

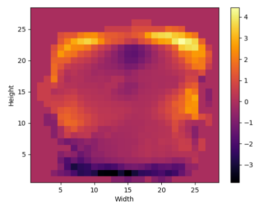
For ease of use, each resulting feature matrix G is vectorized into a column vector such that $V = V_{rq}$ and $d = \frac{n^2}{rq}$ for some chosen pair of r and q, given the definitions in Section 1. Using this formulation, we can again use the generic algorithms based on the formulation in Section 1 to compute the information needed for the classifier.



(a) Digit 4 Log Likelihood

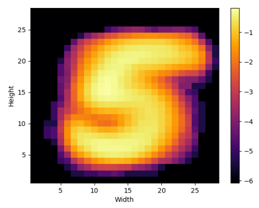


(b) Digit 9 Log Likelihood

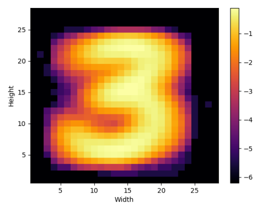


(c) (4, 9) Log Odd Ratio

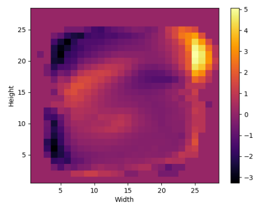
Figure 2: Log Likelihood and Log Odd Ratio for (4, 9) Misclassification Pair



(a) Digit 5 Log Likelihood

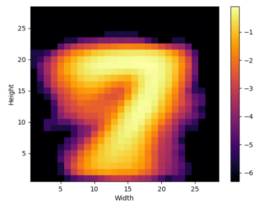


(b) Digit 3 Log Likelihood

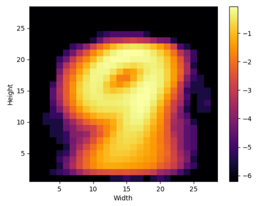


(c) (5, 3) Log Odd Ratio

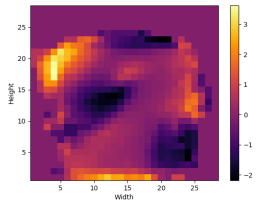
Figure 3: Log Likelihood and Log Odd Ratio for (5, 3) Misclassification Pair



(a) Digit 7 Log Likelihood

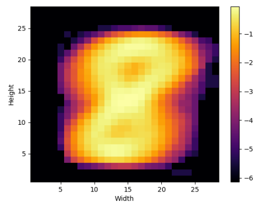


(b) Digit 9 Log Likelihood

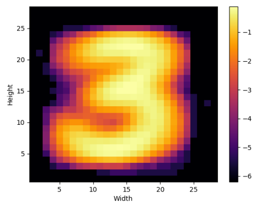


(c) (7, 9) Log Odd Ratio

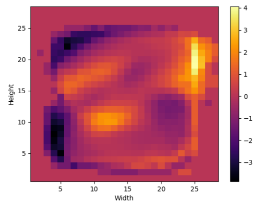
Figure 4: Log Likelihood and Log Odd Ratio for (7, 9) Misclassification Pair



(a) Digit 8 Log Likelihood



(b) Digit 3 Log Likelihood

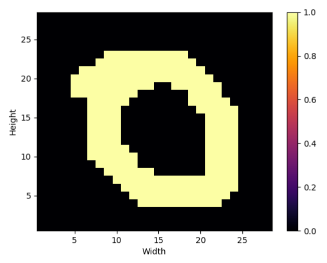


(c) (8, 3) Log Odd Ratio

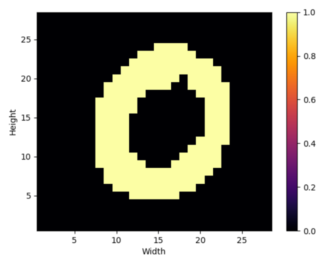
Figure 5: Log Likelihood and Log Odd Ratio for (8, 3) Misclassification Pair

2.3.2 Results

For the results generated, we are looking at features with dimension pairs (2,2), (2,4), (4,2), and (4,4). With this set of disjoint block feature dimensions, Table 2

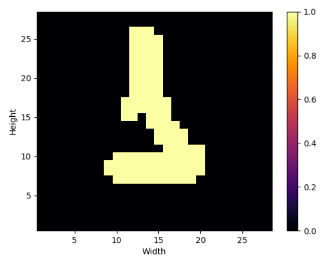


(a) Min Posterior Test Data

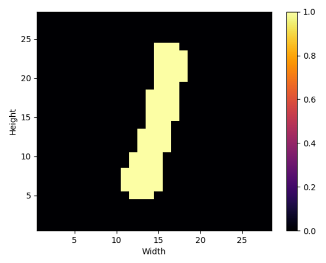


(b) Max Posterior Test Data

Figure 6: Min and Max Posterior Test Data for 0 Digit

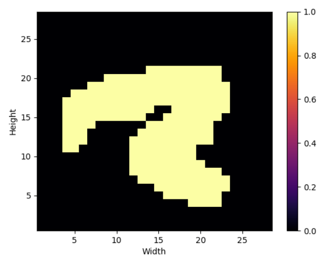


(a) Min Posterior Test Data

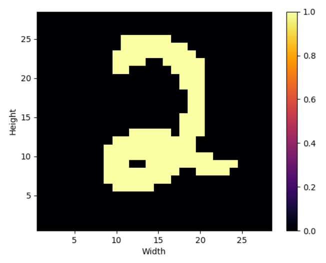


(b) Max Posterior Test Data

Figure 7: Min and Max Posterior Test Data for 1 Digit



(a) Min Posterior Test Data

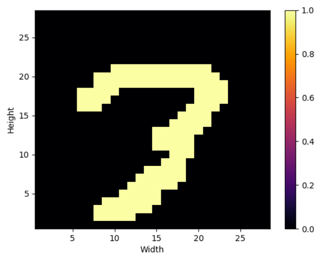


(b) Max Posterior Test Data

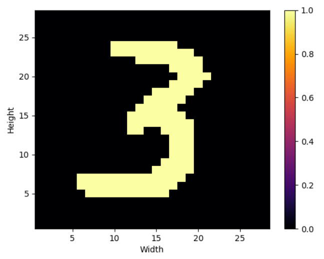
Figure 8: Min and Max Posterior Test Data for 2 Digit

includes the accuracy and training run-time for each pair, including (1,1) which represents the data collected in Section 2.1.

In terms of trends, especially when looking at the features based on individual pixels from Section 2.1, there are some notable ones. First, we can see that

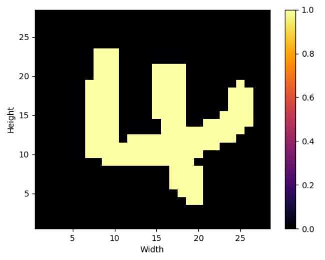


(a) Min Posterior Test Data

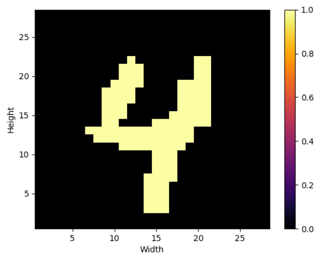


(b) Max Posterior Test Data

Figure 9: Min and Max Posterior Test Data for 3 Digit

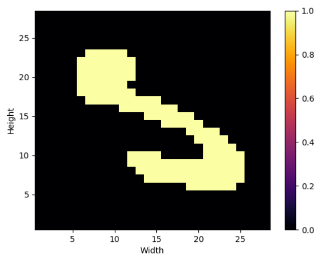


(a) Min Posterior Test Data

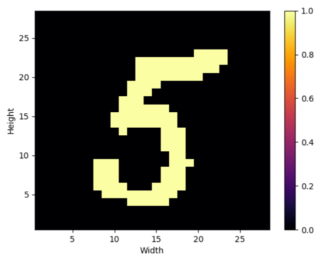


(b) Max Posterior Test Data

Figure 10: Min and Max Posterior Test Data for 4 Digit



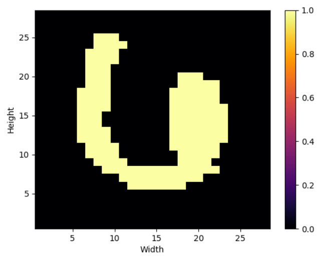
(a) Min Posterior Test Data



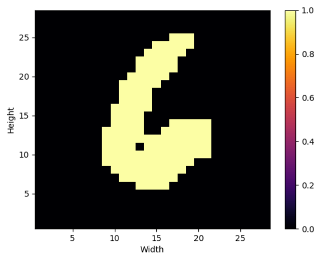
(b) Max Posterior Test Data

Figure 11: Min and Max Posterior Test Data for 5 Digit

using the disjoint block features, in general, improve performance relative to the individual pixel features. We can also see, however, that making the Disjoint Blocks enclose too many pixels can actually hurt performance, likely because too large of blocks result in too much loss of information and make it hard to

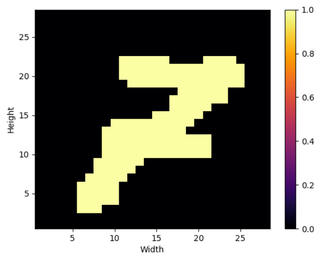


(a) Min Posterior Test Data

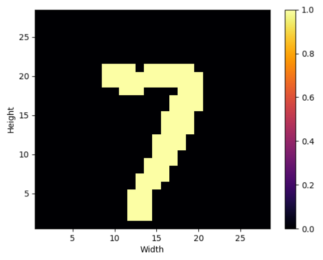


(b) Max Posterior Test Data

Figure 12: Min and Max Posterior Test Data for 6 Digit

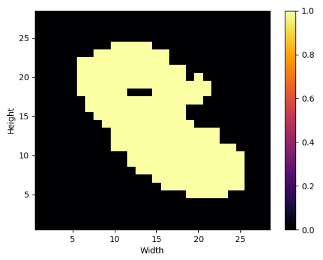


(a) Min Posterior Test Data

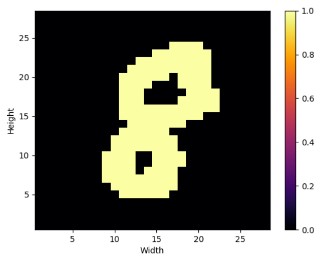


(b) Max Posterior Test Data

Figure 13: Min and Max Posterior Test Data for 7 Digit



(a) Min Posterior Test Data

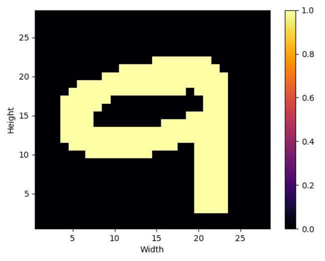


(b) Max Posterior Test Data

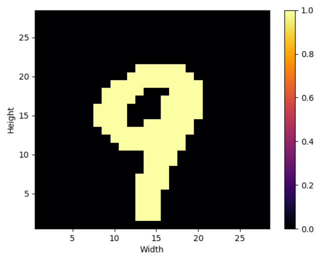
Figure 14: Min and Max Posterior Test Data for 8 Digit

make meaningful classifications from that resulting feature dataset.

If one looks at run-time for training, we can see the block size of the features increases exponentially as the number of pixels per block are increased. This makes sense because, based on the discussion from Section 1.2, we can see the



(a) Min Posterior Test Data



(b) Max Posterior Test Data

Figure 15: Min and Max Posterior Test Data for 9 Digit

run-time complexity for the disjoint block features will be $O\left(\frac{n^2}{rq}2^{rq}pN\right)$. If we hold other terms constant, we can see that the run-time complexity grows by $\frac{2^{rq}}{rq}$ as the product of the disjoint block dimensions, rq, grows.

For large product values for rq, one would expect the exponential numerator to greatly dominate, in turn making large disjoint block features take much longer to train. This analysis validates why the (4,4) feature is much more time consuming to train than the (1,1) feature, while the (2,2) feature is not much slower to train relative to the (1,1) feature.

As for the testing run-time, we know from Section 1.2 that the complexity will be $O\left(p\frac{n^2}{rq}\right)$. This means that as rq increases, we should expect the run-time to decrease dramatically due to the $(rq)^{-1}$ factor. Looking at the results from Table 2, we can see this reduction occurs as the disjoint blocks span a wider area of pixels, validating the theoretical analysis.

2.4 Overlapping Block as Features

2.4.1 Problem Formulation

Similarly to the Disjoint Block features, we take the same dataset of square images, each with $n = 28$ rows and columns of pixels with binary values, and create new set of features based on taking overlapping subregions of a given image and representing them as a single value. For simplicity, each image $I \in B_I = \{0,1\}^{n\times n}$ and define $V_{rq} = \{0, 1, \cdots, 2^{rq} - 1\}$. Then let us define the operator $H_{rq} : B_I \to V_{rq}^{(n-r+1)\times(n-q+1)}$ as the map that converts an image into a smaller matrix made up of the new, overlapping features. The algorithm

```
Algorithm 1: $F_{rq}$ Definition
  Data: Image, $I \in B_I$
  Data: Number of rows in subblock of image used for a feature, $r$
  Data: Number of columns in subblock of image used for a feature, $q$
  Result: Output matrix of new features, $G \in V_{rq}^{\frac{n}{r} \times \frac{n}{q}}$

  // Initialize output feature matrix, $G$
  $G_{ij} = 0 \ \forall i \in \{1, 2, \cdots, \frac{n}{r}\}, j \in \{1, 2, \cdots, \frac{n}{q}\}$

  // Construct the feature values
  for $i \in \{1, 2, \cdots, \frac{n}{r}\}$ do
      for $j \in \{1, 2, \cdots, \frac{n}{q}\}$ do
          for $k \in \{1, 2, \cdots, r\}$ do
              for $l \in \{1, 2, \cdots, q\}$ do
                  $t = 1 + (i - 1)r + k$
                  $s = 1 + (j - 1)q + l$
                  $G_{ij} = G_{ij} + 2^{(l-1)+q(k-1)} I_{ts}$
              end
          end
      end
  end

  return $G$
```

representing H_{rq} for computing the new set of features is found in Algorithm 2.

```
Algorithm 2: $H_{rq}$ Definition
  Data: Image, $I \in B_I$
  Data: Number of rows in subblock of image used for a feature, $r$
  Data: Number of columns in subblock of image used for a feature, $q$
  Result: Output matrix of new features, $G \in V_{rq}^{(n-r+1) \times (n-q+1)}$

  // Initialize output feature matrix, $G$
  $G_{ij} = 0 \ \forall i \in \{1, 2, \cdots, (n - r + 1)\}, j \in \{1, 2, \cdots, (n - q + 1)\}$

  // Construct the feature values
  for $i \in \{1, 2, \cdots, (n - r + 1)\}$ do
      for $j \in \{1, 2, \cdots, (n - q + 1)\}$ do
          for $k \in \{1, 2, \cdots, r\}$ do
              for $l \in \{1, 2, \cdots, q\}$ do
                  $t = (i + k)$
                  $s = (j + l)$
                  $G_{ij} = G_{ij} + 2^{(l-1)+q(k-1)} I_{ts}$
              end
          end
      end
  end

  return $G$
```

Feature Dimension Pair	Training Run-time	Testing Run-time	Accuracy
(1, 1)	0.155s	18.08s	77.1%
(2, 2)	0.194s	4.082s	84.3%
(2, 4)	1.609s	2.156s	82.7%
(4, 2)	1.734s	2.036s	84.7%
(4, 4)	183.6s	0.975s	75.3%

Table 2: Results information for Disjoint Block Features

As before, each resulting feature matrix G is vectorized into a column vector such that $V = V_{rq}$ and $d = (n - r + 1)(n - q + 1)$ for some chosen pair of r and q, given the definitions in Section 1. Using this formulation, we can again use the generic algorithms based on the formulation in Section 1 to compute the information needed for the classifier.

2.4.2 Results

For the results generated, we are looking at features with dimension pairs (2,2), (2,4), (4,2), (4,4), (2,3), (3, 2), (3,3). With this set of overlapping block feature dimensions, Table 3 includes the accuracy and training run-time for each pair, including (1,1) which represents the data collected in Section 2.1.

Feature Dimension Pair	Training Run-time	Testing Run-time	Accuracy
(1, 1)	0.155s	18.08s	77.1%
(2, 2)	0.739s	14.43s	85.4%
(2, 4)	11.13s	16.75s	85.4%
(4, 2)	11.75s	13.59s	85.9%
(4, 4)	2,516s	13.26s	80.3%
(2, 3)	2.869s	13.33s	86.0%
(3, 2)	2.610s	13.42s	86.9%
(3, 3)	20.52s	13.02s	83.7%

Table 3: Results information for Overlapping Block Features

When looking at the features based on individual pixels from Section 2.1 and the disjoint feature results in Section 2.3.2, there are some notable trends. First, we can see that the Overlapping Block features achieve more robust accuracies, all block sizes achieving 80%+ performance. This trend makes sense since the Overlapping Block features ensure we don't miss much information, even with larger block sizes, due to the overlapping behavior. We also note that there is a sweet spot in terms of block dimensions that achieves performance of just under 87%. What I find interesting is how, based on reviewing the (2,4), (4,2), (2,3), (3,2) results, it appears that features that represent blocks with more rows perform better than blocks with more columns, implying vertical information is less important than the horizontal information.

If one looks at run-time for training, we can see the block size of the features increases exponentially as the number of pixels per block are increased. This makes sense because, based on the discussion from Section 1.2, we can see the run-time complexity for the overlapping block features will be $O(n^2 2^{rq} pN)$. If we hold other terms constant, we can see that the run-time complexity grows proportionally to 2^{rq} as the product of the overlapping block dimensions, rq, grows. Note that this growth is larger than the growth seen in the disjoint block as a function of rq. This model explains why the (4,4) feature is much more time consuming to train than the (1,1) feature, while the (2,2) feature is not much slower to train relative to the (1,1) feature. The larger time complexity growth seen by the overlapping block as a function of rq also explains why the run-time is much slower for each block feature dimension when contrasting it to the corresponding disjoint features.

As for the testing run-time, we know from Section 1.2 that the complexity will be $O\left(pn^2\right)$. This means that as the dimensions of the overlapping block features change, we should not expect the run-time to change much. Looking at the results from Table 2, we can see that for most of the overlapping features, the testing run-time is similar to each other, validating the theoretical analysis.

3 Problem 1 Extra Credit — Face Classification

The problem being tackled here is taking dataset of labeled thresholded images, based on some edge detection computer vision algorithm, and learning to classify images as either a Face (1) or a Non-face (0).

3.0.1 Problem Formulation

This problem was formulated identically to the formulation found in Section 2.3.1 for the disjoint block features. All work done there carries over into here. One thing to note is that $\Omega = \{0, 1\}$ where, as stated before, the face label is 1 and the non-face label is 0. Additionally, a disjoint block size of (2,2) was used to reduce the resolution of the input matrix of data while also keeping a sufficient amount of information. The Laplace Smoothing constant was set to $\alpha = 0.1$.

3.0.2 Results

After training and testing, it was found that using (2,2) disjoint features produced a classification accuracy on the testing dataset of 98%. The confusion matrix for this test dataset can be seen in Figure 16, making it obvious this classifier performed well against the training dataset.

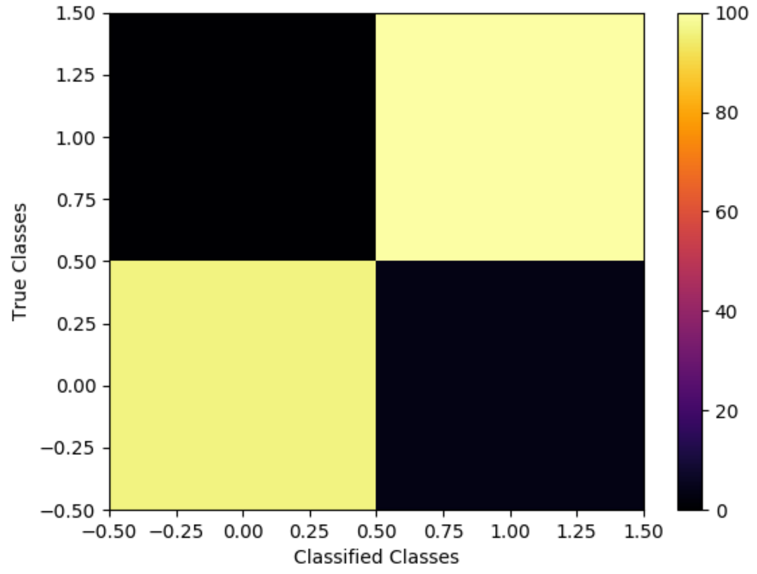


Figure 16: Confusion Matrix for Face Dataset using (2,2) Disjoint Block Features

4 Part 2 — Audio Classification

4.1 Classification of Hebrew 'Yes' and 'No'

Within this problem, the goal is to create a classifier that can distinguish between recordings saying the words 'Yes' and 'No' in Hebrew. The sounds are all represented as thresholded spectrograms, meaning that again the data will be read into memory with binary feature values.

4.1.1 Problem Formulation

Within this problem, we take a dataset of thresholded spectrograms matrices, each with $n_r = 25$ rows and $n_c = 10$ columns of binary values, and vectorize this matrix for use in our framework defined in Section 1. Doing this gives us $V = \{0, 1\}$ and $d = n_c n_r$, given the definitions in Section 1. Note that we also define $\Omega = \{0, 1\}$, where 0 is a label for 'No' and 1 is a label for 'Yes'. Using this formulation, it is straight forward to use our generic algorithms to build and test a classifier for this problem.

4.1.2 Results

After training and testing the classifier based on the spectrogram data, it was found that the classifier correctly classified 96% of the testing data. This is after messing around with the Laplace Smoothing constant, α, and finding that the performance really didn't change as a function of the constant, so I chose $\alpha = 0.1$. The confusion matrix for the results can be found in Figure 17. Recall again that the 0 label is for 'No' and the 1 label is for 'Yes'. It is visually obvious from the confusion matrix that there is minimal errors in the classification. I suspect that the Hebrew 'Yes' and 'No' words have sufficiently different pronunciations to make the classifier this strong.

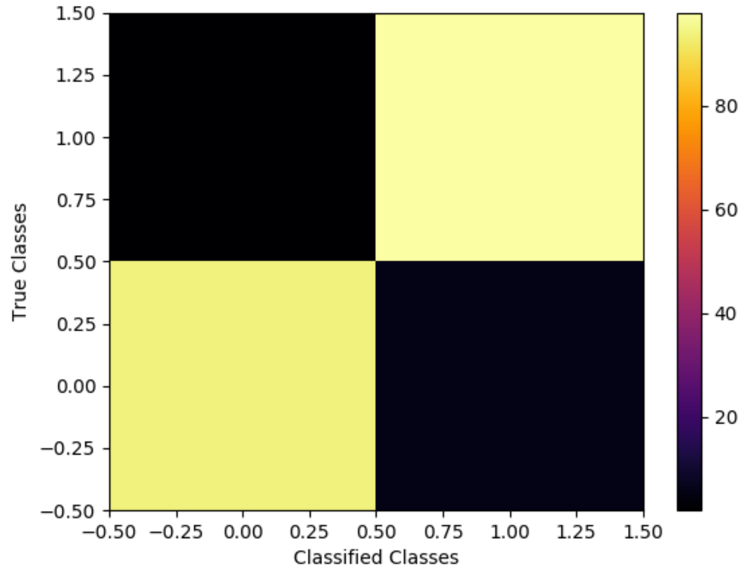


Figure 17: Confusion Matrix for Hebrew 'Yes' vs 'No' Classification

4.2 Classification of Spoken Digits

Within this problem, the goal is to create a classifier that can distinguish between different digits, particularly digits 1 through 5, based on recordings from a set of different speakers. The data given to us is a labeled set of thresholded Mel-frequency Cepstrum Coefficients (MFCCs). The goal is to create a classifier that can achieve greater than 80% overall classification accuracy.

4.2.1 Problem Formulation

For practical purposes, the MFCCs we are given are essentially matrices with binary values for each element of the matrices. If we take each matrix, with $n_r = 30$ rows and $n_c = 13$ columns of binary values, we can vectorize this matrix for use in our framework defined in Section 1. Doing this gives us $V = \{0, 1\}$ and $d = n_c n_r$, given the definitions in Section 1. Note that we also define $\Omega = \{1, 2, 3, 4, 5\}$, where each of the labels correspond to the spoken digit we are classifying. Using this formulation, it is straight forward to use our generic algorithms to build and test a classifier for this problem.

4.2.2 Results

After training and testing the classifier based on the spectrogram data, it was found that the classifier correctly classified 85% of the testing data. The confusion matrix for the results can be found in Figure 18. Based on the confusion matrix, it appears the digits that were toughest to correctly classify were 4 and 5. Particularly, there was a large misclassification error where the classifier interpreted some of the data labeled 4 as a 2. Additionally, noticeably large misclassification errors occurred with 5 where the classifier interpreted some of the data labeled 5 as 1 or 3.

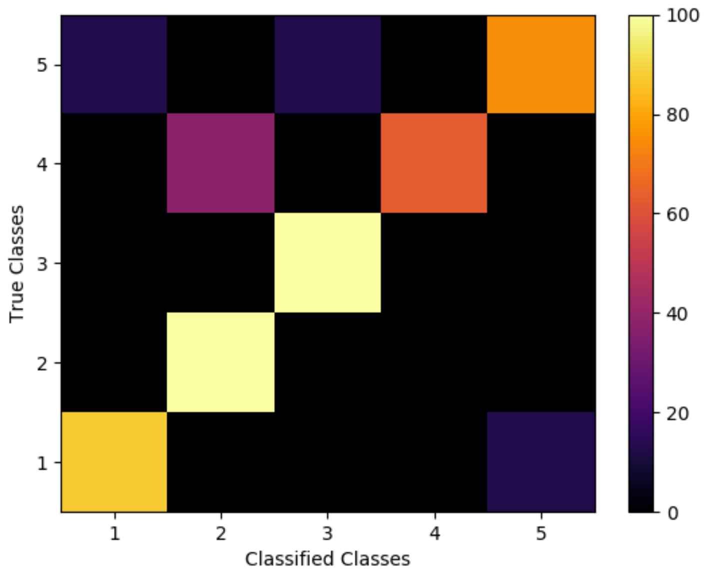


Figure 18: Confusion Matrix for Spoken Digit Classification

5 Contributions

- Christian Howard
  - Did everything since worked on this MP alone

6 List of Potential Extra Credit

- Face Classification for Part 1 of the assignment

Appendices

A Open Source Software Used

Some system python libraries, as well as the **numpy** package, were used to help develop the codes used in this MP.