CS440 Final Project Report
Members:
Junjie He (jh1285)
Dianliang Guo (dg757)
Xinlong Zhang (xz371)

Naive Bays Classifier:

1. Run and Test the Algorithm:
   python naive_main.py

2. Results:

Iteration	Training Size	Training Type	Average Training Time	Average Correct Prediction	Standard Deviation
1	500	Digit Image	0.351121 Sec	70.47%	0.11%
2	1000	Digit Image	0.748740 Sec	72.1%	0.76%
3	1500	Digit Image	1.147506 Sec	73.74%	0.13%
4	2000	Digit Image	1.618394 Sec	74.6%	0.28%
5	2500	Digit Image	1.829957 Sec	74.74%	0.05%
6	3000	Digit Image	2.126869 Sec	75.54%	0.16%
7	3500	Digit Image	2.536561 Sec	75.84%	0.04%
8	4000	Digit Image	2.901904 Sec	75.76%	0.08%
9	4500	Digit Image	3.264933 Sec	76.32%	0.12%
10	5000	Digit Image	3.490273 Sec	76.7%	0.19%

Iteration	Training Size	Training Type	Average Training Time	Average Correct Prediction	Standard Deviation
1	45	Face Image	0.171817 Sec	50.8%	0.26%
2	90	Face Image	0.358287 Sec	70.3%	3.8%
3	135	Face Image	0.491822 Sec	84.0%	0.65%
4	180	Face Image	0.764415 Sec	86.0%	0.77%
5	225	Face Image	0.816866 Sec	87.3%	0.26%
6	270	Face Image	1.051893 Sec	86.6%	0.0%
7	315	Face Image	1.183084 Sec	87.3%	0.32%
8	360	Face Image	1.276113 Sec	88.0%	0.32%
9	405	Face Image	1.425991 Sec	87.46%	0.26%
10	450	Face Image	1.648034 Sec	86.8%	0.26%

3. Implementation for Digits:
   Step 1: Read the files
   Each digit image has 28 * 28 pixels. We use a list to store images by saving data from each pixel. The pixel is converted to a non-zero number if the pixel contains '#' or '+' (foreground). Otherwise, the pixel is saved as zero (background). However, there is no converting rule for saving data from label files.

   Step 2: Start training
   At first, we shuffle the order of training data and training labels in pairs.

   Second, count the occurrences for each digit from 0 to 9 in the training labels. Then, calculate the prior probability for each digit by using the occurrence divided by the training size.

$$\hat{P}(y) = \frac{c(y)}{n}$$

where c(y) is the number of training instances with label y and n is the total number of training.

Step 3: Calculate the probability for each pixel
First, get the feature from each digit image. The feature is arranged as a long list, which contains 784 (28*28) pixels in one row.

Second, count the occurrence of of each pixel for certain digit is '#' or '+' (foreground). Specifically, we create **Ten** 28*28 list, and initialize the each number in the list is zero to count the occurrence for each pixel in each digit.

Third, calculate the probability for each pixel in each digit by using the occurrence divided by total number.

$$\hat{P}(F_i = f_i | Y = y) = \frac{c(f_i, y)}{\sum_{f_i' \in \{0,1\}} c(f_i', y)}$$

Step 4: Get Prediction
First, get features for each Testing image
Second, Calculate the prediction

- Create 10 lists for 10 digits to store the count.
- Iterate the feature list,
  - if the pixel is a foreground pixel:
    - Iterate over the 10 lists and add the count the log2(probability of certain digit), probability is from Step 3.
  - Else:
    - Iterate over the 10 lists and add the count the log2(1 - probability of certain digit), probability is from Step 3.
- Calculate the probabilities of 10 digits for the image.

Third, return the digit with maximum probability.
In this project, we use Laplace smoothing, which adds k counts to every possible observation value:

$$P(F_i = f_i | Y = y) = \frac{c(f_i, y) + k}{\sum_{f_i' \in \{0,1\}} (c(f_i', y) + k)}$$

If k=0, the probabilities are unsmoothed. As k grows larger, the probabilities are smoothed more and more.

Step 5: Testing and Calculate Correct Prediction Percentage

Compare the predicted results with the true results and count the correct numbers. Then, use the correct numbers divided by training size to get

Correct Prediction Percentage.

Step 6: Calculate the mean value and std.

4. Implementation for Faces:
   Step 1: Read the files
   Each Face image has 70 * 60 pixels. We use a list to store images by saving data from each pixel. The pixel is converted to a non-zero number if the pixel contains '#' or '+' (foreground). Otherwise, the pixel is saved as zero (background). However, there is no converting rule for saving data from label files.

   Step 2: Start training
   At first, we shuffle the order of training data and training labels in pairs.

   Second, count the occurrences for face image and non-face image in the training labels. Then, calculate the prior probability for face image and non-face image by using the occurrence divided by the training size.

$$\hat{P}(y) = \frac{c(y)}{n}$$

   where c(y) is the number of training instances with label y and n is the total number of training.

   Step 3: Calculate the probability for each pixel
   First, get the feature from each image. The feature is arranged as a long list, which contains 4200 (70*60) pixels in one row.

   Second, count the occurrence of of each pixel for certain digit is '#' or '+' (foreground). Specifically, we create **Ten** 70*60 list, and initialize the each number in the list is zero to count the occurrence for each pixel in face image and non-face image.

   Third, calculate the probability for each pixel in face image and non-face image by using the occurrence divided by total number.

$$\hat{P}(F_i = f_i | Y = y) = \frac{c(f_i, y)}{\sum_{f_i' \in \{0,1\}} c(f_i', y)}$$

   Step 4: Get Prediction
   First, get features for each Testing image
   Second, Calculate the prediction
   - Create 2 lists for face image and non-face image to store the count.
   - Iterate the feature list,
     - if the pixel is a foreground pixel:
       - Iterate over the 2 lists and add the count the log2(probability of face image and non-face image), probability is from Step 3.

Else:

Iterate over the 2 lists and add the count the log2(1 - probability of face image and non-face image), probability is from Step 3.

Calculate the probabilities of face image and non-face image.

Third, return the result with maximum probability.

In this project, we use Laplace smoothing, which adds k counts to every possible observation value:

$$P(F_i = f_i | Y = y) = \frac{c(f_i, y) + k}{\sum_{f_i' \in \{0,1\}} (c(f_i', y) + k)}$$

If k=0, the probabilities are unsmoothed. As k grows larger, the probabilities are smoothed more and more.

Step 5: Testing and Calculate Correct Prediction Percentage

Compare the predicted results with the true results and count the correct numbers. Then, use the correct numbers divided by training size to get Correct Prediction Percentage.

Step 6: Calculate the mean value and std.

Perceptron Classifier:

1. Run and Test the Algorithm:
   python perceptron_main.py

2. Results:

Iteration	Training Size	Training Type	Training Time	Correct Prediction	Standard Deviation
1	500	Digit Image	0.11083521 Sec	65.32%	7.27317%
2	1000	Digit Image	0.21700818 Sec	68.72%	2.335294%
3	1500	Digit Image	0.33494181 Sec	71.22%	1.99138143 %
4	2000	Digit Image	0.448854970 932 Sec	74.16%	2.738320653 25% %
5	2500	Digit Image	0.723532390	72.9%	4.296044692

			594 Sec		5 %
6	3000	Digit Image	0.724981212616 Sec	76.12%	1.24803846095 %
7	3500	Digit Image	0.788022565842 Sec	74.9%	3.59165699921 %
8	4000	Digit Image	0.903143405914 Sec	75.26%	3.79926308644 %
9	4500	Digit Image	1.03824100494 Sec	75.7%	1.39857069896 %
10	5000	Digit Image	1.1670987606 Sec	77.76%	1.34253491575 %

Iteration	Training Size	Training Type	Training Time	Correct Prediction	Standard Deviation
1	45	Face Image	0.0426871299744 Sec	66.00%	3.45124776148 %
2	90	Face Image	0.0834100723267 Sec	69.33%	7.2418536608 %
3	135	Face Image	0.128484773636 Sec	76.26%	5.34332397587 %
4	180	Face Image	0.172117042542 Sec	74.53%	6.42771775637 %
5	225	Face Image	0.210180568695	71.46%	9.96348890254

			Sec		%
6	270	Face Image	0.260808801651 Sec	82.4%	2.44404037064 %
7	315	Face Image	0.291110372543 Sec	78.8%	5.23916872117 %
8	360	Face Image	0.330620384216 Sec	79.46%	4.3287154882 %
9	405	Face Image	0.379338932037 Sec	80.66%	3.01109061084 %
10	450	Face Image	0.42560839653 Sec	84.93%	1.49666295471 %

3. Implementation for Digits:
   Step 1: Read the files
   Each digit image has 28 * 28 pixels. We use a list to store images by saving data from each pixel. The pixel is converted to a non-zero number if the pixel contains '#' or '+' (foreground). Otherwise, the pixel is saved as zero (background). However, there is no converting rule for saving data from label files.

   Step 2: Start training
   At first, we shuffle the order of training data and training labels in pairs and get features for each Testing image.
   Second, Calculate the weight.
   Create 10 lists for 10 digits to store the weight for each digit.
   Initialize weight and bias for each digit.
   Iterate the feature list, calculate score for each Training digit by making dot product between features for Testing image and weight, then sum the dot product with bias.
   Find predicted label with highest score and compare it with the true label.
   if equal:
   continue and do nothing
   Else:
   weight(predicted) = weight(predicted) - feature
   weight(true label) = weight(true label) + feature

   bias[predicted] = bias[predicted] -1
   bias[true label] = bias[true label] +1

Step 3: Get Prediction

Create a local list for storing answer.

Iterate the feature list, calculate score for each digit by making dot product between features for Testing image and weight, then sum the dot product with bias and store it in local list.

Then find the predicted answer with comparing all the value in local list. The index of highest value is the predicted answer.

Step 4: Testing and Calculate Correct Prediction Percentage

Compare the predicted results with the true results and count the correct numbers. Then, use the correct numbers divided by training size to get Correct Prediction Percentage.

Step 5: Calculate the mean value and std.

4. Implementation for Faces:

Step 1: Read the files

Each Face image has 70 * 60 pixels. We use a list to store images by saving data from each pixel. The pixel is converted to a non-zero number if the pixel contains '#' or '+' (foreground). Otherwise, the pixel is saved as zero (background). However, there is no converting rule for saving data from label files.

Step 2: Start training

At first, we shuffle the order of training data and training labels in pairs and get features for each Testing image. The feature is arranged as a long list, which contains 4200 (70*60) pixels in one row.

Second, Calculate the weight.

Create 2 lists for face image and non-face image to store the count.

Initialize weight and bias for image.

Iterate the feature list, calculate score for each Training image by making dot product between features for Testing image and weight, then sum the dot product with bias.

Find predicted label with highest score and compare it with the true label.

```
if equal:
    continue and do nothing
Else:
    weight(predicted) = weight(predicted) - feature
    weight(true label) = weight(true label) + feature

    bias[predicted] = bias[predicted] -1
    bias[true label] = bias[true label] +1
```

Step 3: Get Prediction

Create a local list for storing answer.
Iterate the feature list, calculate score for each digit by making dot product between features for Testing image and weight, then sum the dot product with bias and store it in local list.
Then find the predicted answer with comparing all the value in local list. The index of highest value is the predicted answer.

Step 4: Testing and Calculate Correct Prediction Percentage
Compare the predicted results with the true results and count the correct numbers. Then, use the correct numbers divided by training size to get Correct Prediction Percentage.

Step 5: Calculate the mean value and std.

Mira Classifier:

5. Run and Test the Algorithm:
   python mira_main.py

6. Results:

Iteration	Training Size	Training Type	Training Time	Correct Prediction	Standard Deviation
1	500	Digit Image	0.11837277 Sec	60.04%	4.74156092 %
2	1000	Digit Image	0.22842578 Sec	66.48%	6.80217612 %
3	1500	Digit Image	0.33798799 Sec	70.24%	3.73983956 %
4	2000	Digit Image	0.53519120 Sec	70.8%	3.20312347 %
5	2500	Digit Image	0.62968411 Sec	74.28%	0.73047929 %
6	3000	Digit Image	0.67372856 Sec	74.1%	2.2099773 %
7	3500	Digit Image	0.798119831085 Sec	75.16%	4.11562874905 %

8	4000	Digit Image	0.927905893326 Sec	76.92%	4.83545240903 %
9	4500	Digit Image	1.02087340355 Sec	75.9%	2.84042250378 %
10	5000	Digit Image	1.13420724869 Sec	74.58%	2.57013618316 %

Iteration	Training Size	Training Type	Training Time	Correct Prediction	Standard Deviation
1	45	Face Image	0.0438269615173 Sec	63.6%	6.23395718803 %
2	90	Face Image	0.0960245132446 Sec	71.2%	5.47154254098 %
3	135	Face Image	0.129696369171 Sec	73.33%	5.91795760565 %
4	180	Face Image	0.17216925621 Sec	78.66%	0.73029674334 %
5	225	Face Image	0.212182188034 Sec	69.73%	9.81246372959 %
6	270	Face Image	0.253668451309 Sec	81.73%	1.23648246607 %
7	315	Face Image	0.29950056076 Sec	78.13%	8.29886203712 %
8	360	Face Image	0.344541263	84.26%	2.686592223

			58 Sec		95 %
9	405	Face Image	0.383156824 112 Sec	79.86%	6.764613810 12 %
10	450	Face Image	0.393366575 241 Sec	81.73%	3.991101212 56 %

7. Implementation for Digits:
   Step 1: Read the files
   Each digit image has 28 * 28 pixels. We use a list to store images by saving data from each pixel. The pixel is converted to a non-zero number if the pixel contains '#' or '+' (foreground). Otherwise, the pixel is saved as zero (background). However, there is no converting rule for saving data from label files.

   Step 2: Start training
   At first, we shuffle the order of training data and training labels in pairs and get features for each Testing image.
   Second, Calculate the weight.
   Create 10 lists for 10 digits to store the weight for each digit.
   Initialize weight and bias for each digit.
   Iterate the feature list, calculate score for each Training digit by making dot product between features for Testing image and weight, then sum the dot product with bias.
   Find predicted label with highest score
   Calculate the coefficient by taking minimum between a constant and ((weight[predicted] - weight[true label]) * feature +1)/(2*(feasture)^2).
   in our alogorithm we set the constont to 0.001.

   Compare predicted label with the true label.
   if equal:
   continue and do nothing
   Else:
   weight(predicted) = weight(predicted) - coeff*feature
   weight(true label) = weight(true label) + coeff*feature

   bias[predicted] = bias[predicted] -coeff
   bias[true label] = bias[true label] +coeff

   Step 3: Get Prediction
   Create a local list for storing answer.
   Iterate the feature list, calculate score for each digit by making dot product

between features for Testing image and weight, then sum the dot product with bias and store it in local list.
Then find the predicted answer with comparing all the value in local list. The index of highest value is the predicted answer.

Step 4: Testing and Calculate Correct Prediction Percentage
Compare the predicted results with the true results and count the correct numbers. Then, use the correct numbers divided by training size to get Correct Prediction Percentage.

Step 5: Calculate the mean value and std.

8. Implementation for Faces:
Step 1: Read the files
Each Face image has 70 * 60 pixels. We use a list to store images by saving data from each pixel. The pixel is converted to a non-zero number if the pixel contains '#' or '+' (foreground). Otherwise, the pixel is saved as zero (background). However, there is no converting rule for saving data from label files.

Step 2: Start training
At first, we shuffle the order of training data and training labels in pairs and get features for each Testing image. The feature is arranged as a long list, which contains 4200 (70*60) pixels in one row.
Second, Calculate the weight.
Create 2 lists for face image and non-face image to store the count.
Initialize weight and bias for image.
Iterate the feature list, calculate score for each Training image by making dot product between features for Testing image and weight, then sum the dot product with bias.
Find predicted label with highest score and compare it with the true label.
if equal:
continue and do nothing
Else:
weight(predicted) = weight(predicted) - feature
weight(true label) = weight(true label) + feature

bias[predicted] = bias[predicted] -1
bias[true label] = bias[true label] +1

Step 3: Get Prediction
Create a local list for storing answer.
Iterate the feature list, calculate score for each digit by making dot product between features for Testing image and weight, then sum the dot product with bias and store it in local list.
Then find the predicted answer with comparing all the value in local list. The

index of highest value is the predicted answer.

Step 4: Testing and Calculate Correct Prediction Percentage
Compare the predicted results with the true results and count the correct numbers. Then, use the correct numbers divided by training size to get Correct Prediction Percentage.

Step 5: Calculate the mean value and std.

Discussion:

Comparison for digit

	Average Training Time	STD	Average Correct Prediction
Naive Bayes	3.490273 Sec	0.19%	76.7%
Perceptron	1.167098 Sec	1.34%	77.76%
Mira	1.134207 Sec	2.57%	74.58%

Comparison for face

	Average Training Time	STD	Average Correct Prediction
Naive Bayes	1.648034 Sec	0.26%	86.8%
Perceptron	0.425608 Sec	1.49%	84.93%
Mira	0.393366 Sec	3.99%	81.73%

When we are comparing data from the table above, we observe that the relationship for average training time between those 3 algorithms is Mira = Perceptron<Naive Bayes.(sligltly difference between perceptron and mira)
Mira has the smallest training time. And Naive Bayes have the longest training time. Overall, algorithms for face runs faster than that for digit. Besides, the training time is directly related to the number of training size. The larger training size, the longer training time.

When talking about correct prediction, all the algorithm has average correct prediction higher than 70%. The overall correct prediction for digit is around 73% for all 3 algorithms. The overall correct prediction for face is a little bit higher which is 79% for all 3 algorithms. For each iteration while running algorithms, we find that the accuracy hits 70% after certain amount of training data. The required size for reaching 70% accuracy is around 1000 for digits and around 90 for faces. Before training enough data, we observe that the correct prediction has the tendency to inescrase. While correct prediction hits 70%, we can see that there’s no directed relationship between training size and correct predictions the algorithms make, There’s a certain upper bound and this bound varies for different training data. But we guess the certain upper bound is around 80% for our algorithm.

STD Comparison: As the data shown above, Naive Baye Algorithm has the least std value, then the Perception and Mira, which means the error for the Naive Baye is the smallest. Generally, the std values for three algorithms are acceptable.