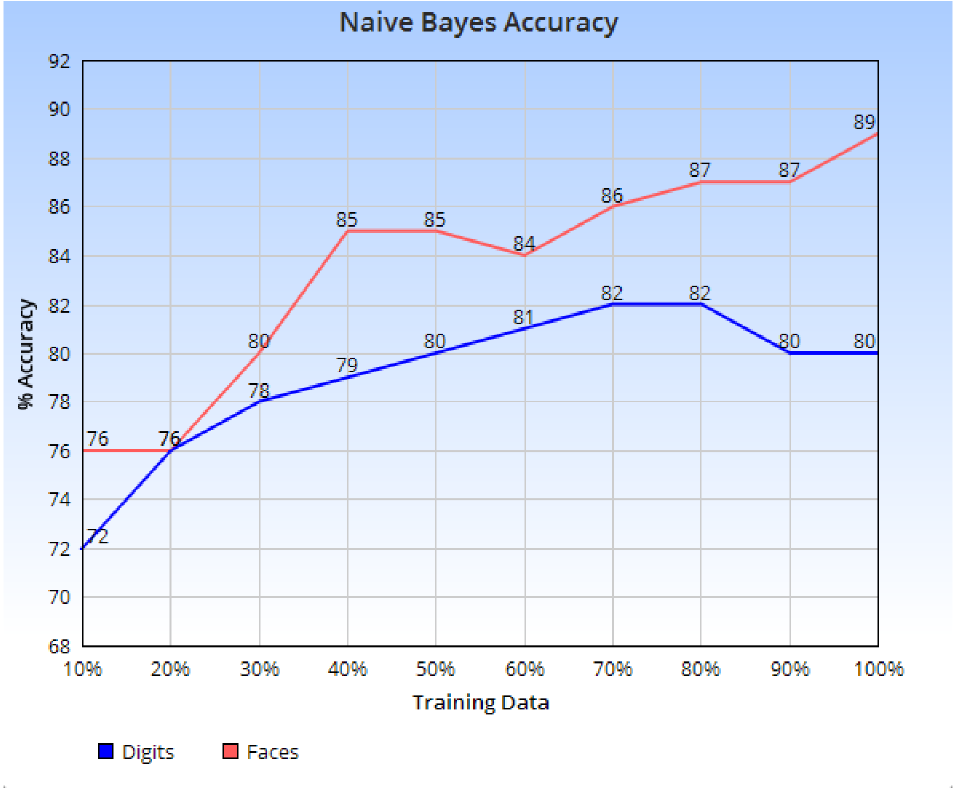
Project: Face and Digit Classification

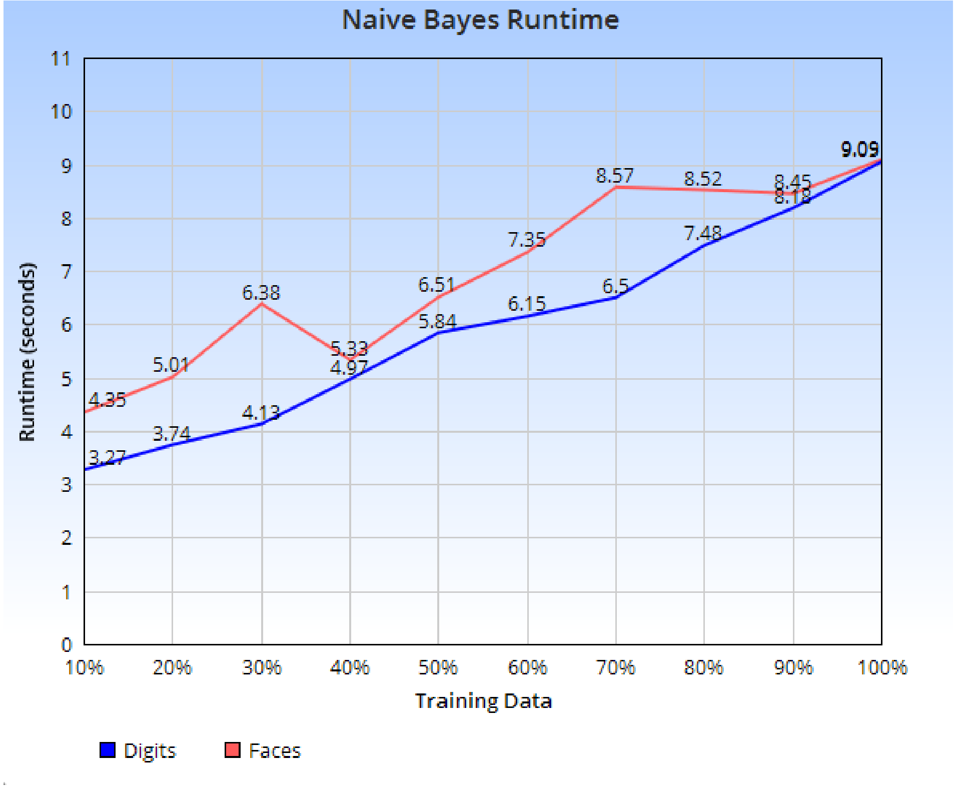
Kevin Nehrbauer RUID: 169005809

Christopher Woolley RUID: 170007112

Naive Bayes

Using the code skeleton provided by Berkeley, Naïve Bayes is implemented under the trainAndTune and calculateLogJointProbabilities methods. In trainAndTune, our code creates three counters to calculate the prior probability, conditional probability, and counts (which is the total data set used to calculate conditional probability). Using a for loop through the training labels gets us the prior probability, and a nested for loop going through each feature contained in training data, adding one to counts([feature, label]) each time, and adding one to conditional probability([feature, label]) if the label is equal to 1 for that particular feature until everything in the dataset has been iterated through. Once the prior is normalized (number of training instances with label y over the number of training instances) it then uses another nested for loop that goes through each value [(feature, label)] in counts and conditional probability, adds 1 to each conditional value and 2 to each counts value (since the only possible feature values can be 0 or 1. To get the final conditional probability values, a for loop sets each conditional probability[i] is set to (conditionalProb[i] / counts[i]). Once this is done, the Berkeley code calls calculateLogJointProbabilitites, which takes for each item i the log(prior probability) + the sum of the logs of each feature’s conditional probability. Overall, Naive Bayes resulted in somewhat accurate results using little of the training data as shown in the graphs below. As the amount of training data increases, the accuracy for both digit and face classifying increases, displaying a learning curve. From our results, Naive Bayes produced more accurate results when executed on Faces. For Digits, we found that in our case, using 70% or 80% training data resulted in 2% more accurate results than using 100%.

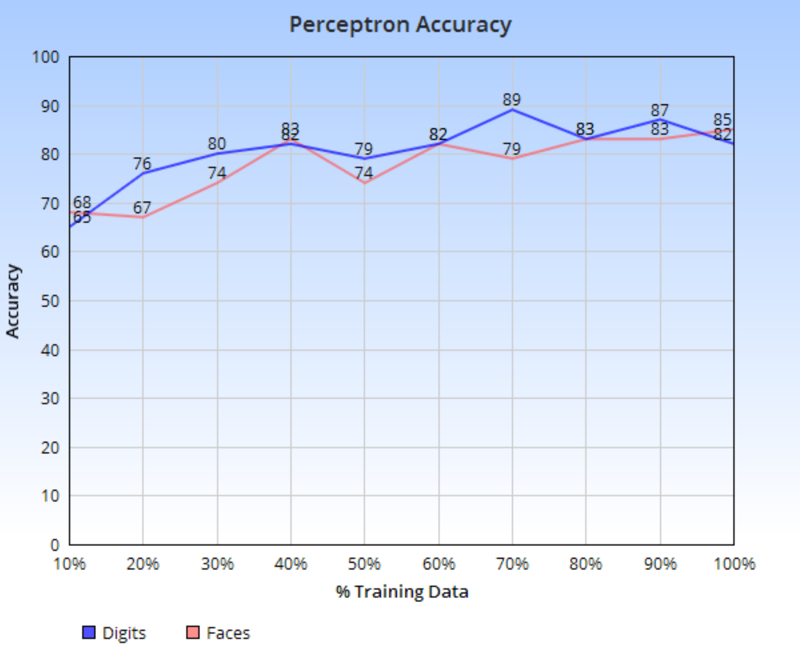

Naive Bayes Accuracy
% Accuracy
92
90
88
86
84
82
80
78
76
74
72
70
68
10%
20%
30%
40%
50%
60%
70%
80%
90%
100%
Training Data
76
76
80
85
85
84
86
87
87
89
72
76
78
79
80
81
82
82
80
80
🟦 Digits
🟥 Faces

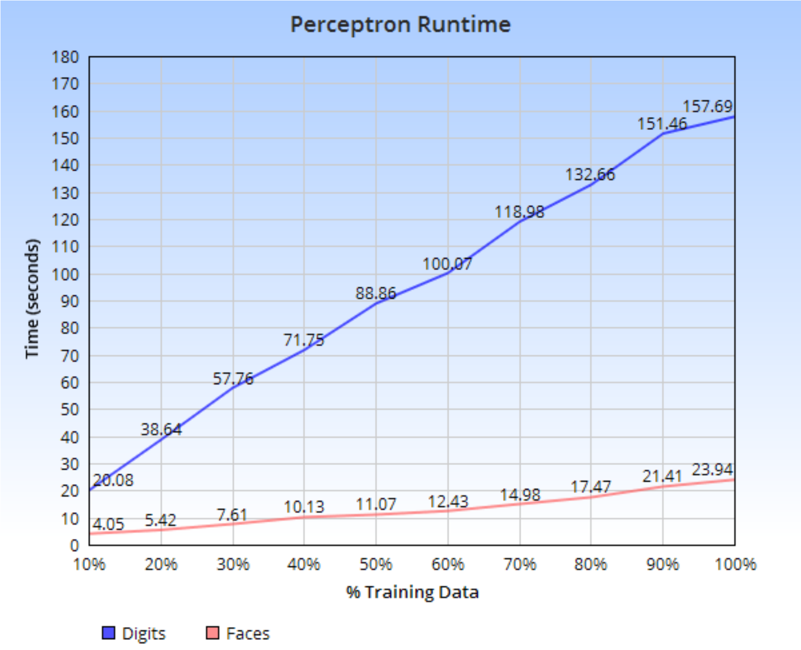

Naive Bayes Runtime
Runtime (seconds)
11
10
9
8
7
6
5
4
3
2
1
0
10%
20%
30%
40%
50%
60%
70%
80%
90%
100%
Training Data
4.35
5.01
6.38
5.33
6.51
7.35
8.57
8.52
8.45
9.09
3.27
3.74
4.13
4.97
5.84
6.15
6.5
7.48
8.18
🟦 Digits
🟥 Faces

Perceptron

Using the code skeleton provided by Berkeley, we were able to implement the Perceptron classification method to get accurate results for both faces and digits. The code only needed to be manipulated in the train function, where it utilized data from the training data, training labels, and the classify function provided by the code skeleton. Using the classify function we can predict the closest matching guess to compare to our label to give it the greatest chance of success. The way we were able to implement this classification was by having a conditional statement that would adjust the weights only if the prediction was incorrect. If it is incorrect it will shift the weight of the predicted value down, and bring the score of the label higher in an attempt to even out the weights for the next time around.

In the graphs below, it will show the learning curves of the perceptron method as we test it with varying amounts of training data. Perceptron had the highest runtime of any of the classifications we used, most likely due to the amount of sifting that is done throughout the training process as it uses the classify function to find the best guess for the labels to be compared to. When using all the training data, digits took 157.69 seconds to go through all 5000 images, while faces only took 23.94 seconds to go through 451 images.

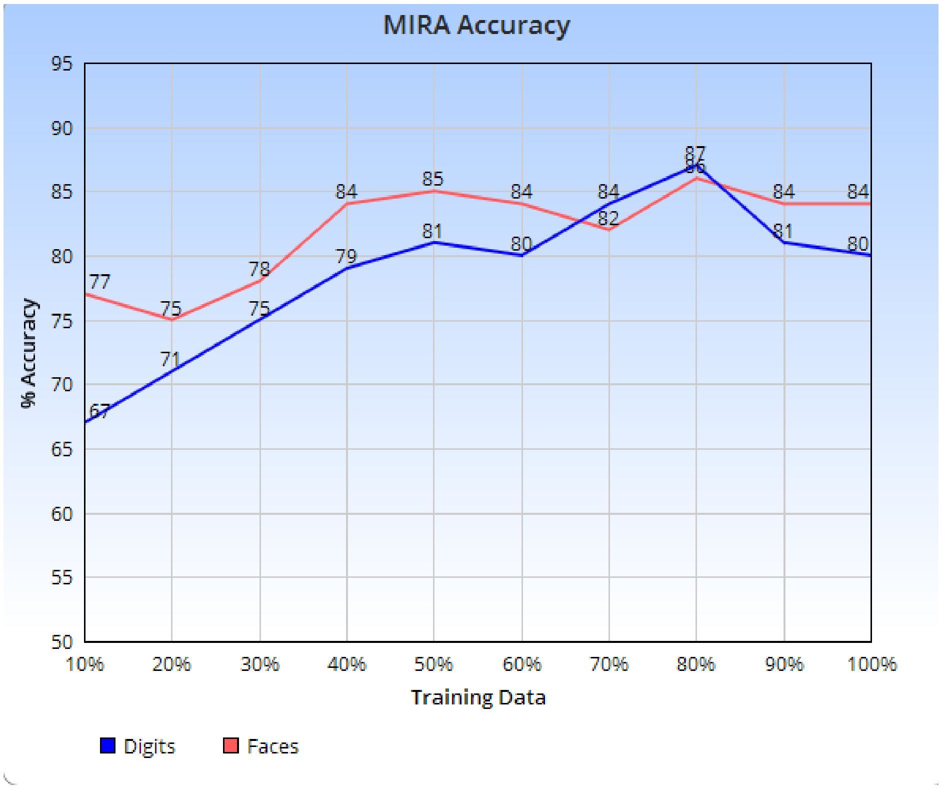


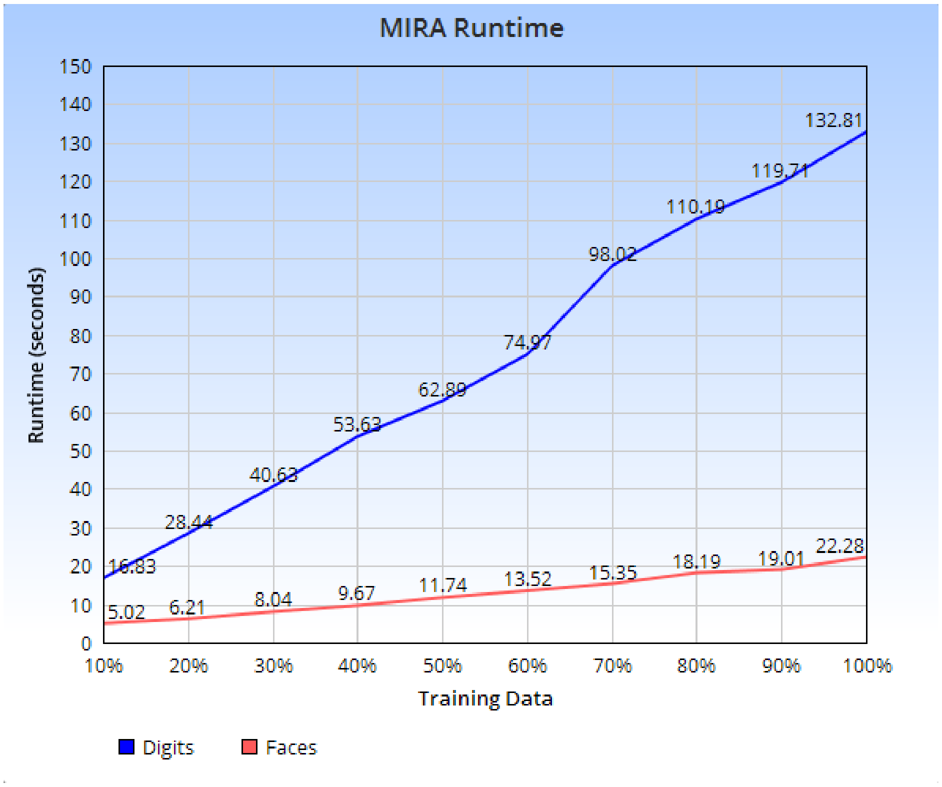


MIRA

We implemented MIRA following the code skeleton provided by Berkeley. Similar to Perceptron, MIRA keeps a weight vector for each label y. Then each label from trainingLabels is compared to their predictedLabel. If they are equal, then the instance is correct and nothing is done, else, we need to update the weight vectors. To update them, we calculate tau by taking the min of {C (where c is 50), ((self.weights[predictedLabel] - self.weights[label]) * datum + 1) / (2 * (datum * datum))])} where datum is all the features of y and y*. The variable diff (differences) is used to update the weight vectors which is calculated by taking all the features and multiplying their values by tau. Finally, the weights are updated using +diff and -diff. These weights are then used by the Berkeley code to classify the data.

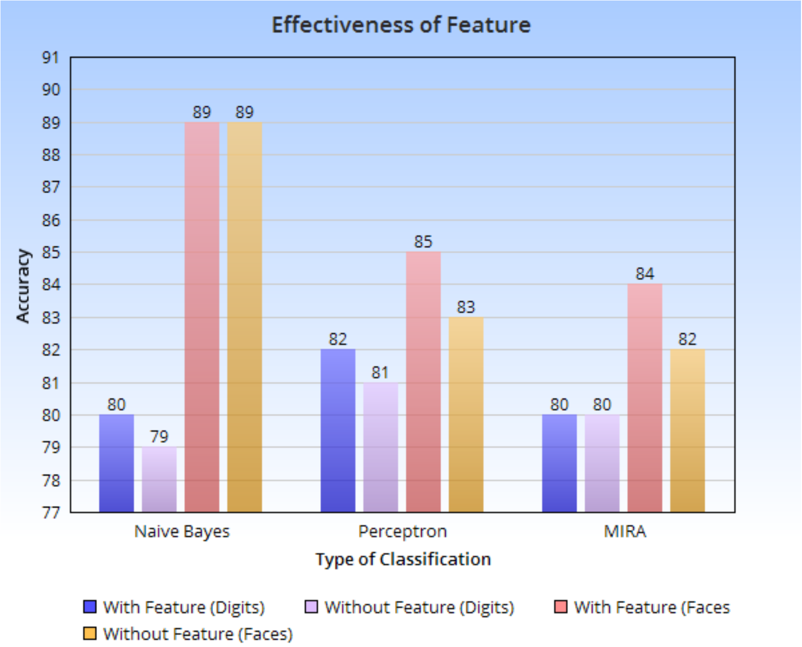
As shown in the graphs below, MIRA produces accurate results even at 10% training data. Overall, both digits and faces produced similar learning curves. The large difference between faces and digits is the runtime. On 100% training data MIRA takes 132 seconds to run on digits, whereas on faces, it takes 22 seconds. Since there are 5000 items in digits vs 451 in faces, there are just more items to run through.


MIRA Accuracy
% Accuracy
95
90
85
80
75
70
65
60
55
50
Training Data
10%
20%
30%
40%
50%
60%
70%
80%
90%
100%
77
75
78
84
85
84
82
86
84
84
67
71
75
79
81
80
84
87
81
80
Digits
Faces


MIRA Runtime
Runtime (seconds)
150
140
130
120
110
100
90
80
70
60
50
40
30
20
10
0
Training Data
10%
20%
30%
40%
50%
60%
70%
80%
90%
100%
16.83
28.44
40.63
53.63
62.89
74.97
98.02
110.19
119.71
132.81
5.02
6.21
8.04
9.67
11.74
13.52
15.35
18.19
19.01
22.28
Digits
Faces

Features Implemented

We implemented one of our own features that helped give as a more reliable and higher accuracy on both digits and tests that varied in effectiveness for each classification. The feature that we added was very similar to the example that was shown in the YouTube videos supplied by the professor, where he divided each image into twelve boxes and counted the amount of filled in pixels in each area. Instead of that, we simply checked to see if there were any pixels in that section at all, if the box was empty (most likely in the corners of the image if at all) then we know that the image is centralized in the frame and it can help distinguish certain numbers and especially weed out a lot of the non faces in the faces training data. Below is a comparison of accuracy when using and not using our implemented feature:


Effectiveness of Feature
Accuracy
Naive Bayes
Perceptron
MIRA
Type of Classification
With Feature (Digits)
Without Feature (Digits)
With Feature (Faces
Without Feature (Faces)