Face and Digit Classification

Anna Godin, Kimberly Wolak

12/9/19

1 Bayes

a.) Describe the algorithm

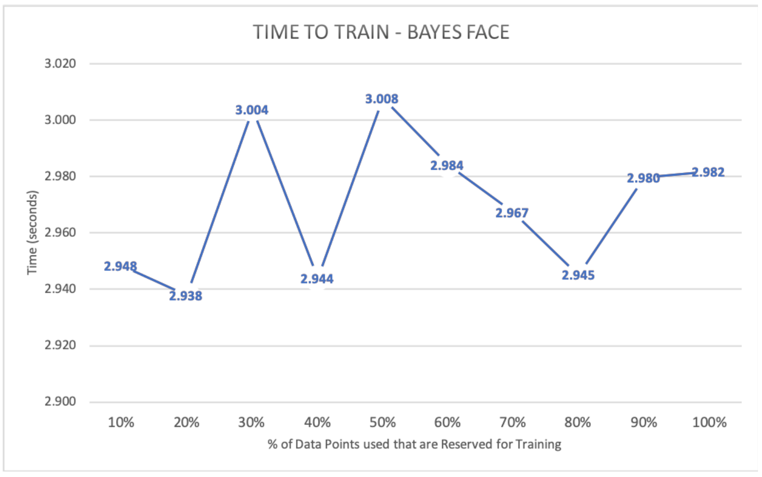
- In Bayes the feature we used was pixel occupancy per region. So first, the algorithm finds the maximum pixel occupancy across all features and all pictures for that training set. With that maximum it sets up a matrix for each unique data point (faces/not faces, 1,2,3 etc). Each row corresponds to a feature and each column is a number between 0 and max. Next, the algorithm tallies and plots the feature and pixel data received from the training data in their corresponding matrix. So for example, if the face picture had 5 pixels in region 3. In the face matrix it would add a +1 to row 3 column 5. Then at the end it would divide each cell by the total number of pictures to come up with percents for each feature and pixel combination. Lastly, when you test it the algorithm will take the features from the testing image and look up the associated percent across all matrices. for each matrix it will keep a score by multiplying each feature percent by each other as given by the testing image feature (each matrix score is independent from each other).The highest percent out of all the matrices will be chosen as the guess.

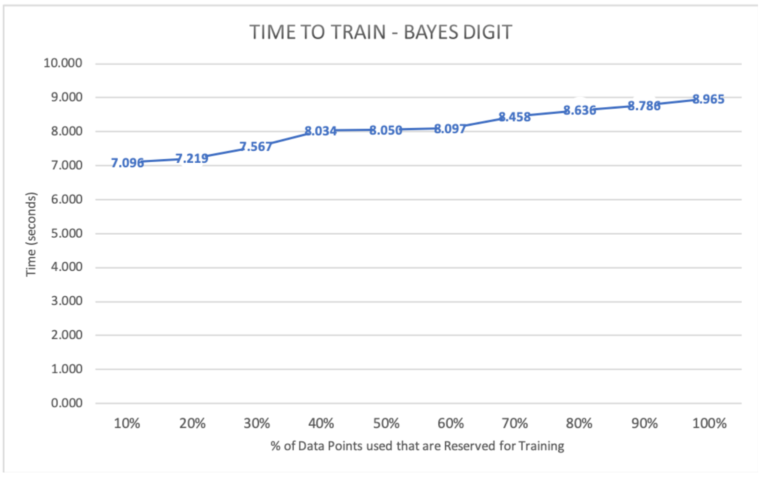
b.) Challenges and lessons learned

- The challenge with this algorithm was finding the right amount of features to make it accurate enough. At first I started with 9 features and that wasn't good enough. I then bumped it up to 25 features and it still wasn't enough. I then tried to adjust the matrix because for the cells not used they were set to be 0.001, so I thought maybe adjusting that would help, but that didn't change anything. Lastly, I bumped up the features to 100 and that ended up working very well. Another thing that was challenging was finding a way to sort the digit data. Unlike faces I had to choose between 10 options instead of 2. So it wasn't as simple as saying if this one is higher, pick this. So what I ended up doing was making a class to keep track of a label and a final percent. I then used a lambda function to sort on a field to find out which percent was the highest to choose for my guess. Both challenges helped me learn how to layout my code and think critically about a solution that would work best for the situation. I also learned a lot by trying all different methods to get it to work properly.

c.) Training

- The following is a report of the time needed for training as a function of the number of data points used for training

- 

- 

- The following is a list of the **Bayes Face** standard deviation of the percent accuracy as a function of the number of data points used for training, with the first item in the list being the standard deviation for 10% of the training data and the last item being the standard deviation for 100% of the training data:

$$0.041, 0.022, 0.035, 0.036, 0.034, 0.029, 0.020, 0.010, 0.014, 0.000$$

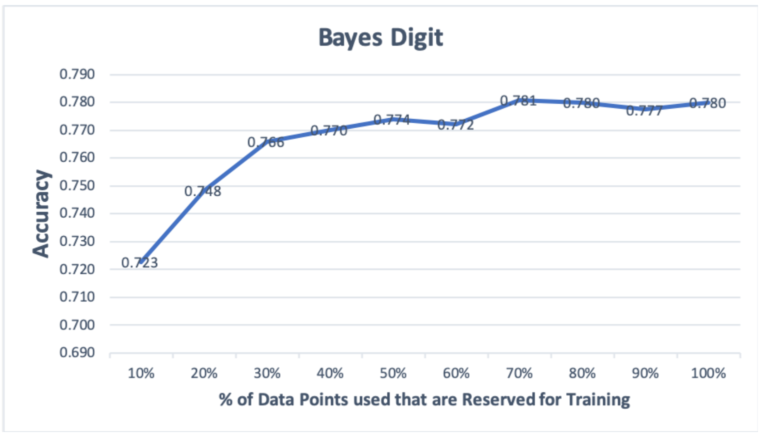
- The following is a list of the **Bayes Digit** standard deviation of the percent accuracy as a function of the number of data points used for training:

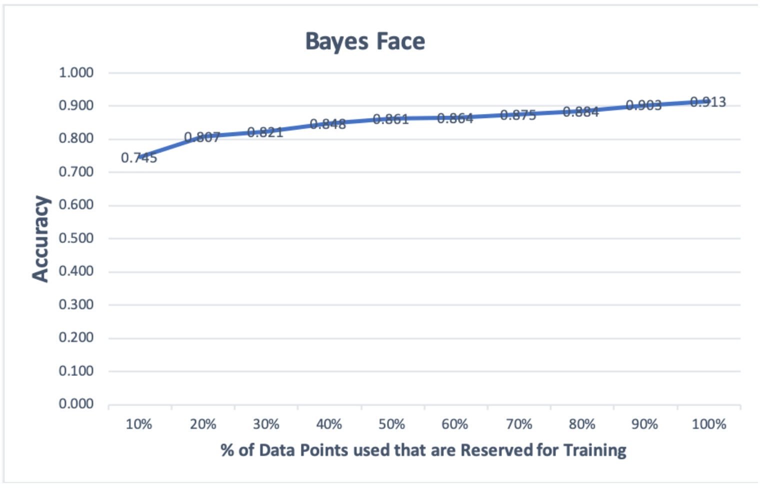
$$0.015, 0.008, 0.012, 0.007, 0.004, 0.010, 0.005, 0.006, 0.004, 0.000$$

- It is evident that the standard deviation of the % accuracy of samples is decreasing for each increment of additional testing data

d.) Performance of Algorithm based on incremental training points

- The following are charts of the average accuracy over 5 iterations of training/testing, as a function of the percentage of data points used that are reserved for training the Naive Bayes algorithm

- 

- 

2 Perceptron

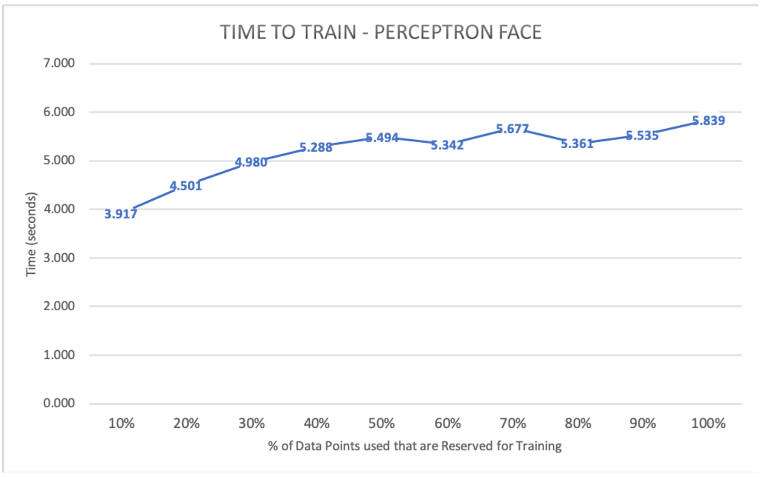
a.) Describe the algorithm

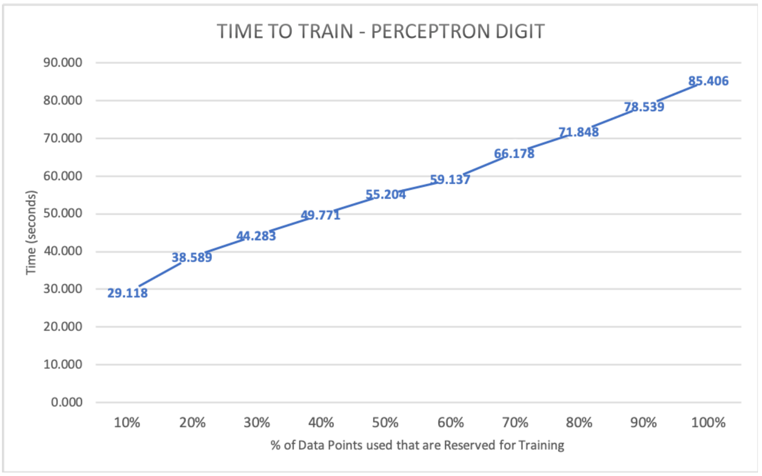
- This algorithm begins with initializing random weights for each feature, which we defined to be the pixel occupancy per region. We had 100 features, because we split the image up into a 10x10 grid. This, each image has 100 weights. The weight vector is calculated by adding up the multiplication of each weight by its corresponding feature pixel density. For faces, if this resulting number is positive, then the prediction for the image is that it is a face. If it is negative, then the prediction is not a face. The algorithm checks the corresponding training label to validate the correctness of the prediction. If the prediction was correct, the algorithm moves on to a new image and calculates the weight vector again. If the prediction was too high (predicted face and it was not face), then each weight for that image is decremented by its corresponding feature. If the prediction was too low, (predicted not face and it was face), then each weight is incremented by its corresponding feature. Our algorithm continues to loop through the training data and update the weights until it hits a threshold of 85% correct guesses.
- The only difference between training faces and digits is that for each image, the algorithm calculates the weight vector for the digits 1-9, and chooses the highest one for its prediction. If the prediction was incorrect, the weights for the incorrect predicted digit are decreased by their respective features, and the weights for the correct digit are increased by their respective features. This process takes quite a long time due to the number of calculations that must be made at each iteration.

b.) Challenges and lessons learned

- Perceptron ended up being the algorithm that took the longest to train the data. This is because I kept track of a counter of correct training predictions, and terminated the training once a certain number of images has been processed and the percentage of correct predictions was over the threshold of 85% out of every image processed in training. One of the main lessons I learned while implementing perceptron is how much longer it took to train digits rather than faces, due to the fact that guessing a face is a binary decision, while digits are not. Calculating each f(x) weight vector for the digits definitely took longer than calculating just one f(x) weight vector for faces. This takes up more run time as well as memory.

c.) Training

- The following is a report of the time needed for training as a function of the number of data points used for training
- 

- 

- The following is a list of the **Perceptron Face** standard deviation of the percent accuracy as a function of the number of data points used for training, with the first item in the list being the standard deviation for 10% of the training data and the last item being the standard deviation for 100% of the training data:

$$0.041, 0.016.0.059, 0.030, 0.021, 0.067, 0.013, 0.030, 0.011, 0.017$$

- The following is a list of the **Perceptron Digit** standard deviation of the percent accuracy as a function of the number of data points used for training:

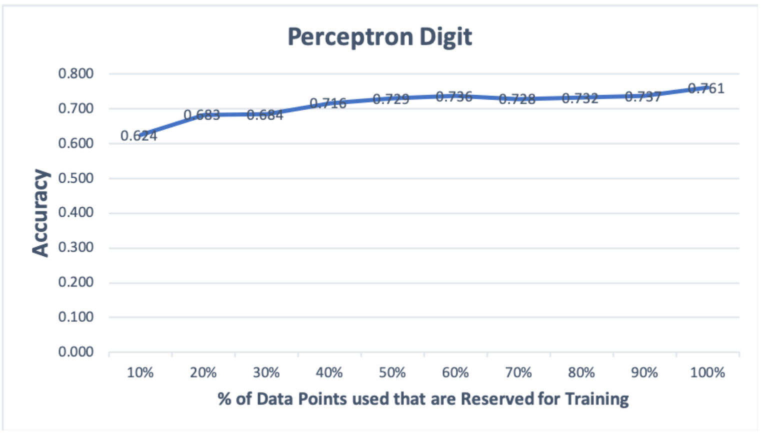
$$0.056, 0.031, 0.056, 0.013, 0.056, 0.030, 0.030, 0.033, 0.040, 0.019$$

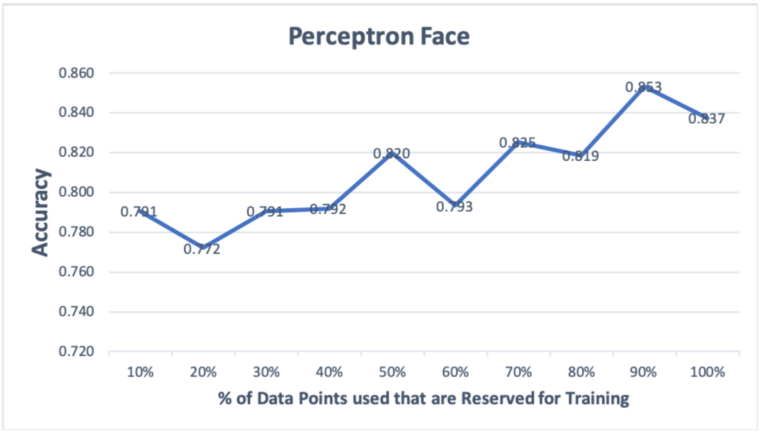
- It is evident that the standard deviation of the % accuracy of samples is decreasing for each increment of additional testing data

d.) Performance of Perceptron based on incremental training points

- Based on the below results, this algorithm performed fairly well for digits as well as faces. For digits, average performance based on incremental training points was quite steady, and did not have drastic spikes or falls. However, in the faces graph, it is evident that each iteration of adding more data resulted in fluctuations in the average accuracy. While there is a general upwards trend as more training data is being introduced, there are more evident spikes in the accuracy. This could be because there are two elements of randomness incorporated into each run of training the algorithm, which are random data points every time and random initialization of weights for each run.

- The following are charts of the average accuracy over 5 iterations of training/testing, as a function of the percentage of data points used that are reserved for training the Perceptron algorithm

- 

- 

3 K-Nearest

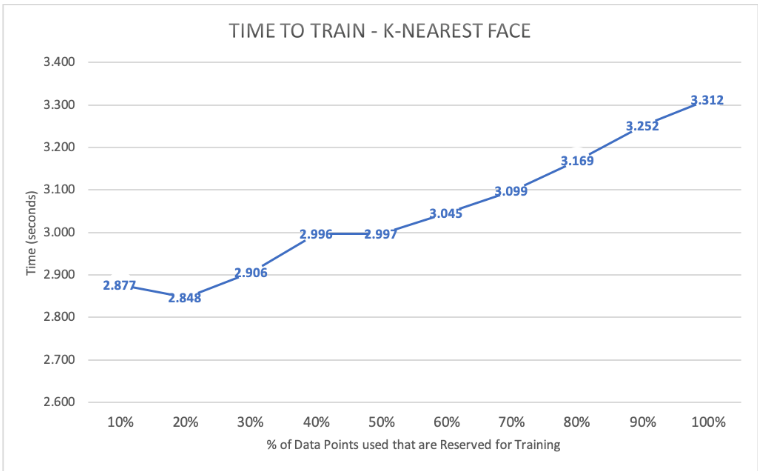
a.) Describe the algorithm

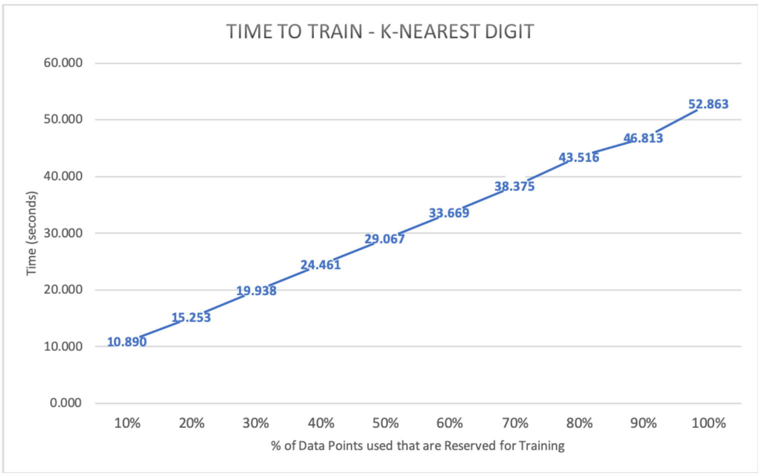
- This algorithm works by taking the euclidean distance of the features in the testing image against all of the features of the images in the training data. So for example image 1 of the testing data would be compared against all 5000 images in the training data by finding the euclidean distance for each paring. Then all of that data is saved into an array and sorted from lowest to highest. After that the first 3 cells are chosen to vote. the most frequently seen within those 3 cells is chosen for the guess. This process repeats for each image in the training set.

b.) Challenges and lessons learned

- When I first did this algorithm I couldn't come up with a great accuracy for digits. The first thing I adjusted was the amount of neighbors I looked at. This increased the accuracy slightly, but not enough to be relevant. So I then started looking into using other distances besides euclidean. I found that chi squared distance was supposed to be more accurate, but after trying for a long time I couldn't get the algorithm to work with our data points. Lastly after increasing the amount of features I used per image the accuracy increased by a lot. I learned a lot about the algorithm and ways to enhance it besides using euclidean. In general euclidean distance in KNN is very slow so by using chi square it speeds up the process and makes it a lot more accurate. It compensates for the shortcomings of the euclidean distance. I also learned different ways to handle the data, do distances and sort the information while trying to get this algorithm to work.

c.) Training

- The following is a report of the time needed for training as a function of the number of data points used for training
- 

- 

- The following is a list of the **K-Nearest Neighbors Face** standard deviation of the percent accuracy as a function of the number of data points used for training, with the first item in the list being the standard deviation for 10% of the training data and the last item being the standard deviation for 100% of the training data:

$$0.051, 0.024, 0.019, 0.025, 0.028, 0.029, 0.030, 0.005, 0.007, 0.000$$

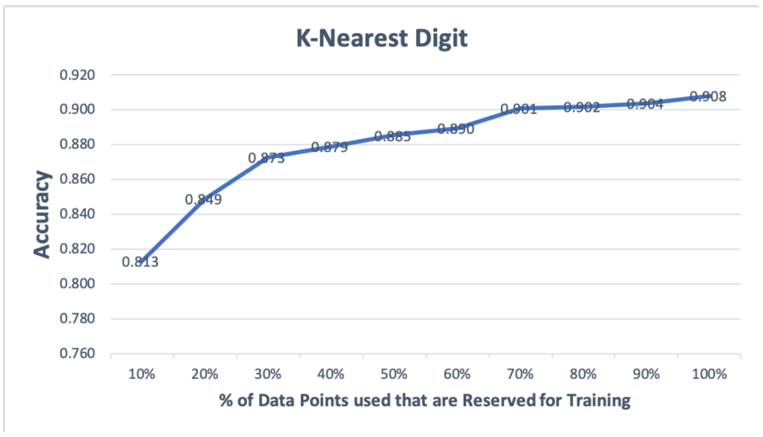
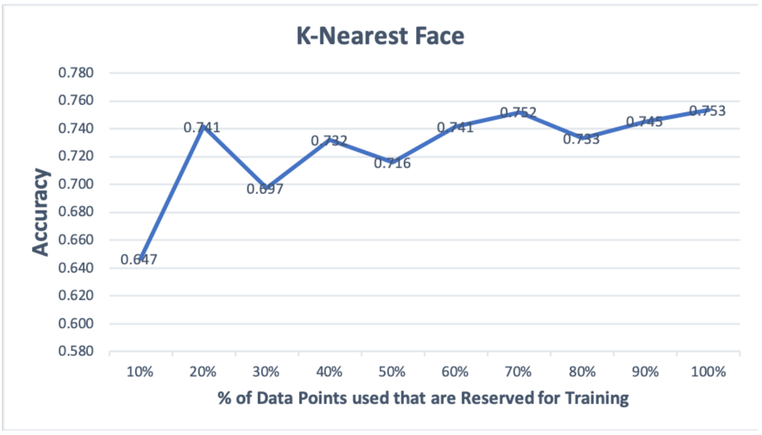
- The following is a list of the **K-Nearest Neighbors Digit** standard deviation of the percent accuracy as a function of the number of data points used for training:

$$0.015, 0.012, 0.005, 0.006, 0.006, 0.007, 0.006, 0.004, 0.002, 0.001$$

- It is evident that the standard deviation of the % accuracy of samples is decreasing for each increment of additional testing data

d.) Performance of Algorithm based on incremental training points

- The following are charts of the average accuracy over 5 iterations of training/testing, as a function of the percentage of data points used that are reserved for training the K-Nearest Neighbors algorithm

- 
K-Nearest Digit
Accuracy
% of Data Points used that are Reserved for Training
0.760
0.780
0.800
0.820
0.840
0.860
0.880
0.900
0.920
10%
20%
30%
40%
50%
60%
70%
80%
90%
100%
0.813
0.849
0.873
0.879
0.885
0.890
0.901
0.902
0.904
0.908
- 
K-Nearest Face
Accuracy
% of Data Points used that are Reserved for Training
0.580
0.600
0.620
0.640
0.660
0.680
0.700
0.720
0.740
0.760
0.780
10%
20%
30%
40%
50%
60%
70%
80%
90%
100%
0.647
0.741
0.697
0.732
0.716
0.741
0.752
0.733
0.745
0.753