Intro to AI: Final Project

Revanth Korrapolu (rrk69) and Vineeth Puli (vrp54)

May 6, 2018

1 Algorithms

We used the UC Berkeley boilerplate code for the main driver of the app, and we implemented the Naive Bayes and Perceptron algorithms by ourselves. In addition, for feature extraction, we used the built-in getPixel method and used the binary value of a pixel (whether it was 0 or greater than 0) to represent whether the pixel was empty or not. We saved this data into a matrix, which we used to store the information from our feature extraction.

To implement the Naive Bayes algorithm, we first trained our data by calculating the probabilities of each component needed for the joint probability model. We achieved this by first counting the number of occurrences of each instance of the possible outputs, and then storing the probability of each output in an dictionary by dividing the number of occurrences by the total number of trials. During this training, we also calculated the P(Feature | Output) of every feature found in the training set by counting each occurrence of the feature and dividing by the previously stored number of occurrences of the corresponding output it was found in, and stored these probabilities in an array of dictionaries "FeatureProb", in which each index represented a different output. We also implemented a function to calculate log probabilities, which returned a dictionary of probabilities corresponding to the log of the product of probabilities for each possible output. We found these values by first finding the natural log of each prior probability, and then for each feature found within a data image that had a binary value of 1, we added the log of the P(Feature | Output) that was previously stored in the data structure "FeatureProb", to the values stored in the dictionary, doing this process for each possible output. When this portion of the algorithm ends, the program picks the greatest value in the dictionary of log probabilities, which represents the output that most closely matched the inputted data given that it had the highest product of probabilities in the Bayes model.

For the Perceptron algorithm, we set up the training process by first populating the weights array, an array of dictionaries that represents the weight of each feature for each possible output, with the indices of the array corresponding to the outputs. We then iterated through the list of given training

data, and initially predicted an output for each data image by taking the dot product of the images features (a list of 0s and 1s) and each list of Weights in the weights array, and choosing the dot product that is highest. This represents the Weights vector that is closest to the features vector of that image, and therefore the output that most closely matches the input data. We compared the predicted output with the given correct output for that image: if the prediction was correct, then we moved onto the next data image; if the prediction was wrong, we updated the Weights of the incorrectly predicted output and the correct output by subtracting the values of the features of the image from the weights of the incorrect output, and adding the values of the features to the weights of the correct output. Over the repetition of all the training data, this process shifted the vectors of the output Weights to the best position possible to classify new data. Once the training is completed, the algorithm then classifies new data by taking the dot product of an images features with all the weights, and choosing the output index with the highest value.

2 Code Usage

For faces dataset:

python dataClassifier.py -d faces -c [naiveBayes, perceptron] -t 450 -k 0.1 -r [45, 90, . . . , 450]

For digits dataset:

python dataClassifier.py -d digits -c [naiveBayes, perceptron] -t 5000 -k 35 -r [500, 1000, . . . , 5000]

To run our program, copy the command above. Replace the first bracket, with either "naiveBayes" or "perceptron" to run the respective algorithm. To change the size of the training data, alter the number within the last bracket. For example, the digits dataset had 5000 total datums. So to run the the Perceptron algorithm on 20% of the data, you could run the following command:

python dataClassifier.py -d digits -c perceptron -t 5000 -r 1000

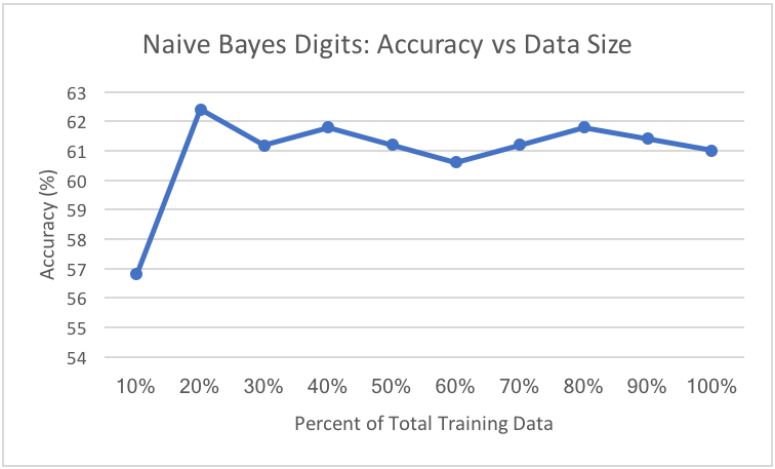
3 Results

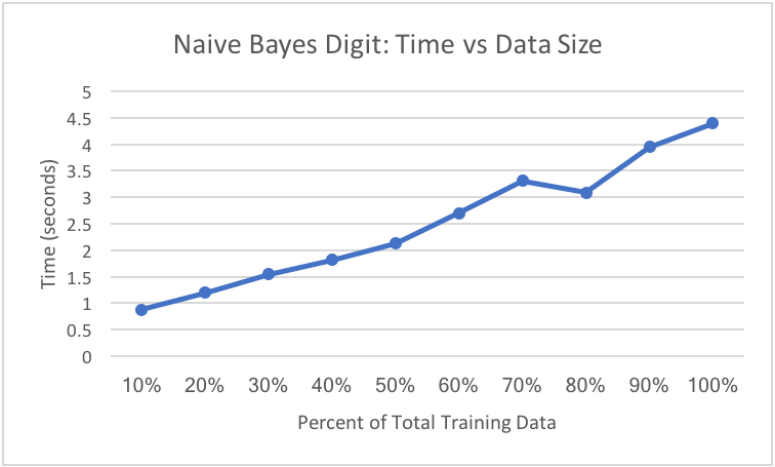
Naive Bayes

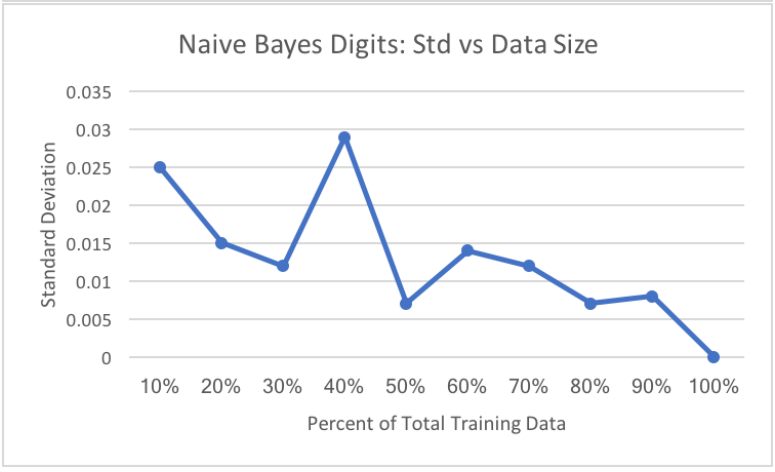
	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
Digits -k 0.1 (datums)	500	1000	1500	2000	2500	3000	3500	4000	4500	5000
Time (s)	0.881	1.196	1.542	1.815	2.130	2.70	3.313	3.082	3.950	4.39
Mean Acc (%)	56.8	62.4	61.19	61.8	61.2	60.6	61.2	61.8	61.4	61.0
Std:	0.025	0.015	0.012	0.029	0.007	0.014	0.012	0.007	0.008	0
Faces -k 35 (datums)	45	90	135	180	225	270	315	360	405	450
Time (s)	0.468	0.648	0.769	0.938	1.097	1.277	1.372	1.502	2.06	1.795
Mean Acc (%)	49.4	49.6	50.6	48.2	56.2	51.0	51.8	60.8	74.4	88.0
Std:	0.029	0.028	0.029	0.024	0.135	0.053	0.059	0.129	0.143	0

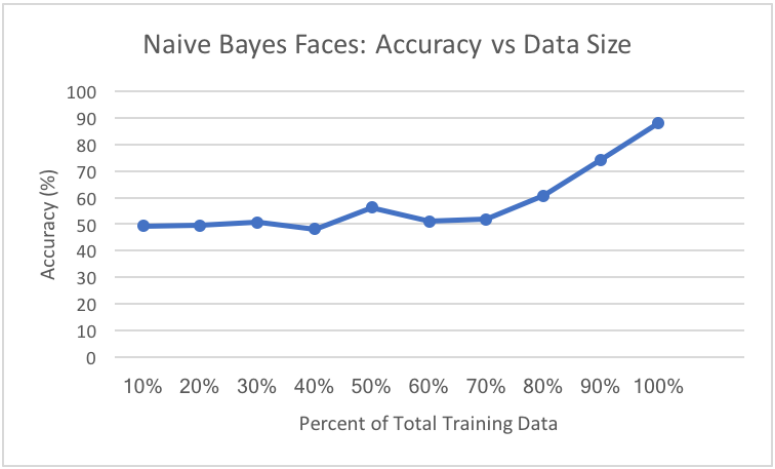
Perceptron

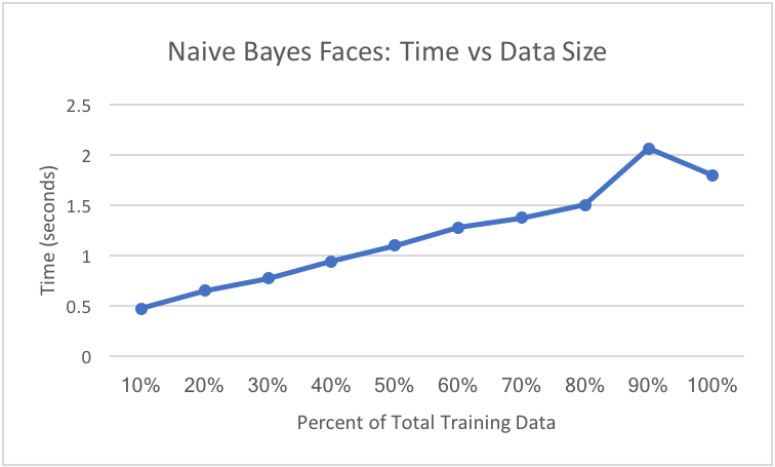
	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
Digits -k 0.1 (datums)	500	1000	1500	2000	2500	3000	3500	4000	4500	5000
Time (s)	19.830	37.704	57.628	74.457	99.088	116.26	134.65	159.52	169.92	201.70
Mean Acc (%)	76.20	76.8	77.6	79.80	82.4	81.0	80.60	81.6	81.8	81.0
Std:	0.0365	0.032	0.042	0.0172	0.0249	0.0228	0.0135	0.0313	0.0312	0.0
Faces -k 35 (datums)	45	90	135	180	225	270	315	360	405	450
Time (s)	2.07	4.704	6.449	8.049	13.85	12.438	16.616	16.965	18.907	23.585
Mean Acc (%)	65.6	78.4	76.6	81.2	81.4	86.0	84.6	83.0	86.2	85%
Std:	0.103	0.036	0.033	0.059	0.037	0.0126	0.0162	0.032	0.018	0.0

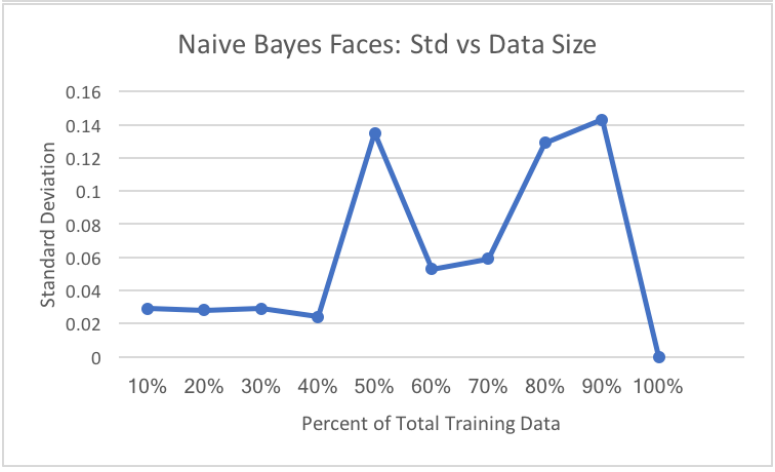

Naive Bayes Digits: Accuracy vs Data Size
Accuracy (%)
63
62
61
60
59
58
57
56
55
54
10%
20%
30%
40%
50%
60%
70%
80%
90%
100%
Percent of Total Training Data

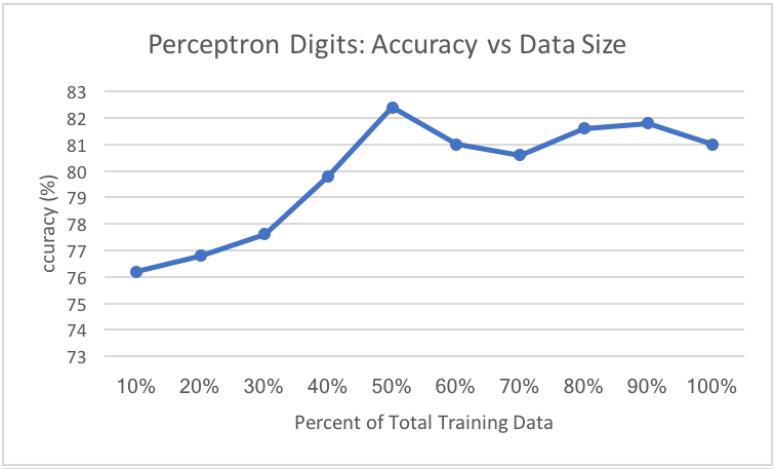

Naive Bayes Digit: Time vs Data Size
Time (seconds)
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
10%
20%
30%
40%
50%
60%
70%
80%
90%
100%
Percent of Total Training Data

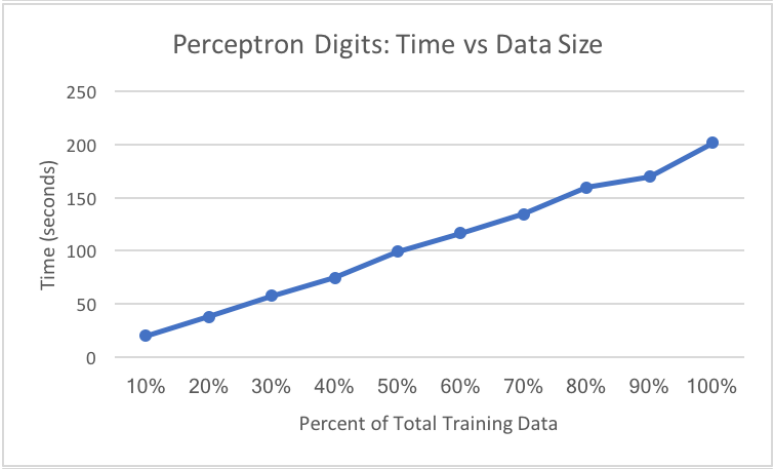

Naive Bayes Digits: Std vs Data Size
Standard Deviation
0.035
0.03
0.025
0.02
0.015
0.01
0.005
0
10%
20%
30%
40%
50%
60%
70%
80%
90%
100%
Percent of Total Training Data

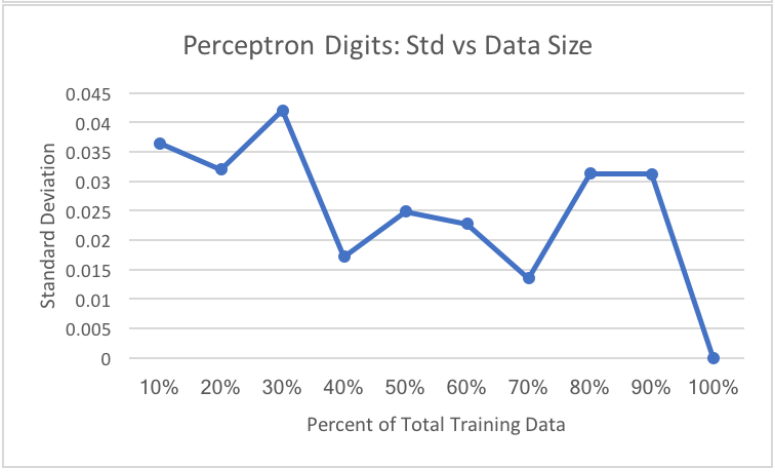

Naive Bayes Faces: Accuracy vs Data Size
Accuracy (%)
100
90
80
70
60
50
40
30
20
10
0
10% 20% 30% 40% 50% 60% 70% 80% 90% 100%
Percent of Total Training Data

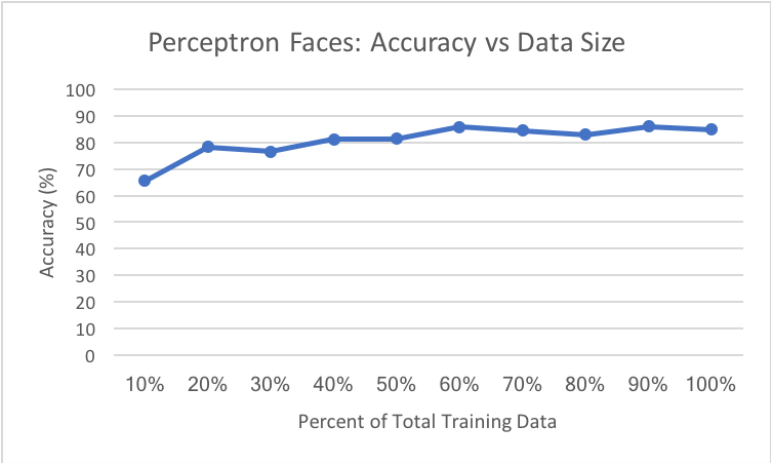

Naive Bayes Faces: Time vs Data Size
Time (seconds)
2.5
2
1.5
1
0.5
0
10% 20% 30% 40% 50% 60% 70% 80% 90% 100%
Percent of Total Training Data

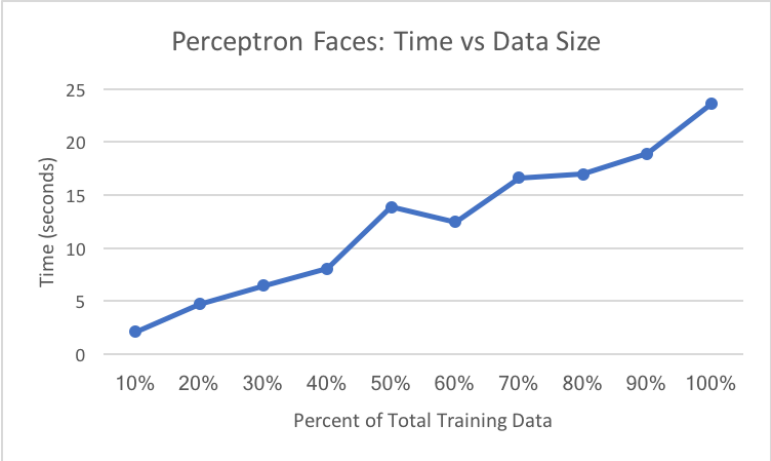

Naive Bayes Faces: Std vs Data Size
Standard Deviation
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
10% 20% 30% 40% 50% 60% 70% 80% 90% 100%
Percent of Total Training Data

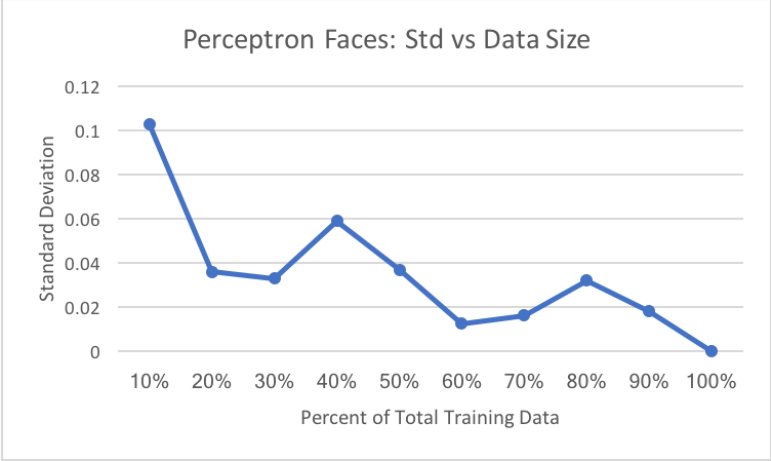

Perceptron Digits: Accuracy vs Data Size
ccuracy (%)
83
82
81
80
79
78
77
76
75
74
73
10% 20% 30% 40% 50% 60% 70% 80% 90% 100%
Percent of Total Training Data


Perceptron Digits: Time vs Data Size
Time (seconds)
250
200
150
100
50
0
10% 20% 30% 40% 50% 60% 70% 80% 90% 100%
Percent of Total Training Data


Perceptron Digits: Std vs Data Size
Standard Deviation
0.045
0.04
0.035
0.03
0.025
0.02
0.015
0.01
0.005
0
10% 20% 30% 40% 50% 60% 70% 80% 90% 100%
Percent of Total Training Data


Perceptron Faces: Accuracy vs Data Size
Accuracy (%)
100
90
80
70
60
50
40
30
20
10
0
10% 20% 30% 40% 50% 60% 70% 80% 90% 100%
Percent of Total Training Data


Perceptron Faces: Time vs Data Size
Time (seconds)
25
20
15
10
5
0
10% 20% 30% 40% 50% 60% 70% 80% 90% 100%
Percent of Total Training Data


Perceptron Faces: Std vs Data Size
Standard Deviation
0.12
0.1
0.08
0.06
0.04
0.02
0
10% 20% 30% 40% 50% 60% 70% 80% 90% 100%
Percent of Total Training Data

4 Analysis

We used the same feature recognition property for both the Faces and Digits data set, since our classifiers gave us 80% accuracy and above for all cases other than Naive Bayes for digits. However, for both algorithms, we added varying levels of Laplace smoothing to boost our accuracy.

$$P(F_i = f_i \mid Y = y) = \frac{c(f_i, y) + \boldsymbol{k}}{\Sigma_{f' \epsilon \{0,1\}}(c(f_i', y) + \boldsymbol{k})}$$

If k = 0, the probabilities are unsmoothed. As k grows larger, the probabilities are smoothed more and more. We made sure K was some non-zero, positive number. This allowed us to solve the zero frequency problem when calculating the log joint probabilities for the Naive Bayes algorithm. However, more refined smoothing boosted accuracy quite a bit by reducing overfitting. We found that smoothing factors (of 0.1 for Digits and 35 for Faces) increased our success rate significantly. The optimal smoothing level seemed to be unique to the data set.

Naive Bayes performed faster (as expected), however, there seemed to be a weakness in its prediction accuracy. Particularly in the digits dataset, the Naive Bayes algorithm showed little improvement with respect to size of training data. This could be a result of poor feature extraction, however, the same features proved to be accurate for the Perceptron algorithm. This leads us to believe that the features are not probabilistically independent which results in the poor prediction accuracy. As we know, the Naive Bayes algorithm operates under the implicit assumption that all the attributes are mutually independent. This allows us to multiply the class conditional probabilities in order to compute the outcome probability using Bayes theorem. However, we should have assessed the features to identify if they were independent. One such statistical method would be the chi-squared test for independence. For future exploration, we would look at different feature extraction for digits, such as edge detection or concavity detection.

The Perceptron showed consistent increase in accuracy as the training dataset increased, specifically for the faces dataset, as seen by the increase from 65% accuracy when training on 10% of the dataset, to 86% accuracy when training on 90% of the dataset. For the digits dataset, we found that accuracy increased only slightly from 10% to 50%, and remained consistent 50% to 100% around an accuracy value of 81%. Similar to the accuracy of digits in the Naive Bayes algorithm over an increasing dataset, we believe the Perceptron algorithm has hit a wall in accuracy due to the limitations of the feature extraction. Overall, the Perceptron algorithm resulted in high accuracies (81% for digits and 85% for faces when trained on the full dataset), but its main weakness was its large overhead, as seen by the increase in time with increasingly larger training datasets. For both the digits and faces datasets, the Perceptron algorithm was significantly slower than the Naive Bayes algorithm, taking 23.585 seconds to

train for the full faces dataset and 201.70 seconds to train for the full digits dataset.