Project 2 Report

Allen Zhang axz3
Joseph Hawkins jah463
Anthony De Rose atd78

Part 0 - Reading Images

Upon looking at the images, we determined that all face images were 70 lines long. All digits were 20 lines long, but with varying amounts of whitespace in between digits. We decided for faces to just read 70 lines and concatenate the lines without any newlines. For digits, we would search for a line that contained non-space characters, then read the next 20 lines. We determined that faces could be represented as a string of 4200 characters, and digits were strings of 560 characters. Each index in the string would represent a corresponding “pixel” in the image.

Part 1 - Structure and Implementation of Code

We used 2 python files to implement the project. One is called naive.py for naive bayes, and the other is perceptron.py, for perceptron. The programs for each take in 3 arguments: “d” or “f” for data type, “m” or “w” for operating system, and an integer 0-100 for the % of the training set used. The program will randomly pick a percentage of the training data provided based on the user’s input. For faces, the label “0” represents not a face, and “1” represents a face. For digits, the numbers 0-9 correspond to actual digits.

Choosing features is an important part of both algorithms. For Naive Bayes, we wanted to compare the number of images in the training dataset that had the same character at the same pixel as the test image. For example, Φ_5 is the number of images in the training set that had the same character at index 5 of the test image. In order to do this, we first iterate through the training dataset and keep a count of the total number of characters seen at each index, or “pixel”, for each label type. Then, we return a 3D array that contains the number of characters for each label, each character type, at each position.

```
for imageNo, trainingImage in enumerate(trainingImages):
    if imageNo>=trainingSize:
        break
    # loops through a single training image
    for index, item in enumerate(trainingImage):
        if item == " ":
            allTrainingData[trainingLabels[imageNo]][0][index] += 1
        elif item == "#":
            allTrainingData[trainingLabels[imageNo]][1][index] += 1
        else:
            allTrainingData[trainingLabels[imageNo]][2][index] += 1
```

In our actual method for calculating Naive Bayes, we must find P(X | Y) * P(Y). We find the product of all of the features in order to get P(X|Y). We do not need to divide the features by the number of occurrences of the label since it is a constant. However, there is potential for an overflow error so we take the sum of the logs of the features. Once we calculate this for the label types, we find the label with the higher value of P(Y|X), which is our guess.

```
while classNo < numClasses:  # cycles though the different classes to get likelihood
    probXgivenY = 0
    # finds the probability an image belongs in a class given its features
    for index, item in enumerate(testImage):
        # print(probXgivenY)
        if item == " ":
            probXgivenY = math.log(
                (trainingData[classNo][0][index]+1)/(labelCount[classNo]+1)) + probXgivenY
        elif item == "#":
            probXgivenY = math.log(
                (trainingData[classNo][1][index]+1)/(labelCount[classNo]+1)) + probXgivenY
        elif item == "+":
            probXgivenY = math.log(
                (trainingData[classNo][2][index]+1)/(labelCount[classNo]+1)) + probXgivenY
    classNo += 1  # move to next class
    probability.append(probXgivenY)
```

For perceptron we stopped training after updating the weight function 25 times. We chose a higher number because we wanted the weights to be more accurate and reduce the effects of randomness on the results. For features, we had each feature correspond to an index in the string for the training image. The value of each feature depended on the character at that location. Spaces were 1, hashtags were 2, and plusses were 3. We calculate the F value by finding the sum of the weights multiplied by the features. We find the F values of all the label values and pick the largest one, which is our guess. Then we check to see if the guess is correct. If it is correct, we continue the loop. If it is incorrect, we update the weights for the label with the incorrect guess. We also check to see if the correct value had a positive F-value. If it did not, we also need to update the weights for that label. If the correct value had a positive but smaller F-value than the actual guess, we still don't change it.

```
for i, features in enumerate(trainingFeatures):  # for every example on our training set
    if i>= trainingSize:
        break
    y_i = labels[index]  # ground truth for training image label
    guesses = np.zeros(numClasses)      #list of F vals for each set of weights, <0 is false, >0 is true, index max value is the guess
    for classNo, f in enumerate(allF):
        F = calculateF(allF[classNo], features)
        guesses[classNo] = F
    ##print(guesses)
    y = np.argmax(guesses)    #find max of guess
    ##print(y)
    if guesses[y] > 0 and y == y_i:  #check to see if it matches with y_i
        index += 1
        continue        #if true, continue

    if guesses[y] > 0 and y != y_i:  # we incorrectly guessed
        allF[y] = updateF(allF[y], features, False) #update for the incorrect value guessed
    if guesses[y_i] < 0:
        allF[y_i] = updateF(allF[y_i], features, True)  #update the value that should have been guessed if it is less than 0
```

Once we finish training, then we simply iterate through each test image and calculate the F value based off of its features for each label value. We take the largest F value and the corresponding label value is our best guess.

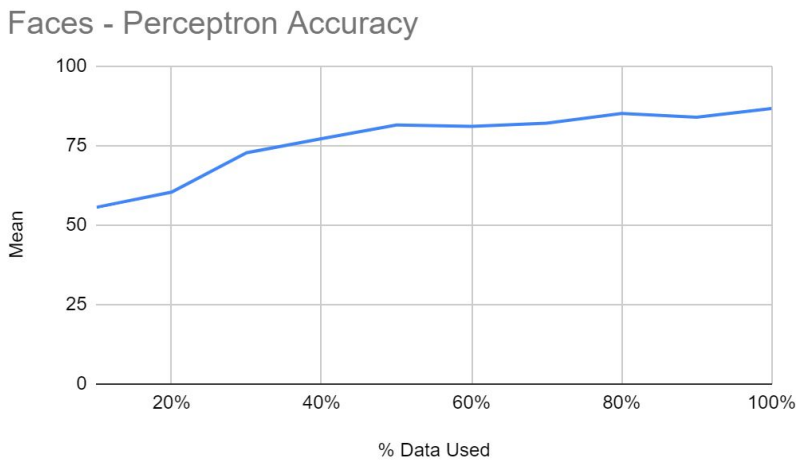
```
for testImage in testImages:
    fx = getFeatures(testImage)
    guesses = []
    for weights in allWeights:
        fxiw = calculateF(weights, fx)
        guesses.append(fxiw)

    sol = np.argmax(guesses)
    results.append(sol)
```

Part 2 - Results

Perceptron Faces

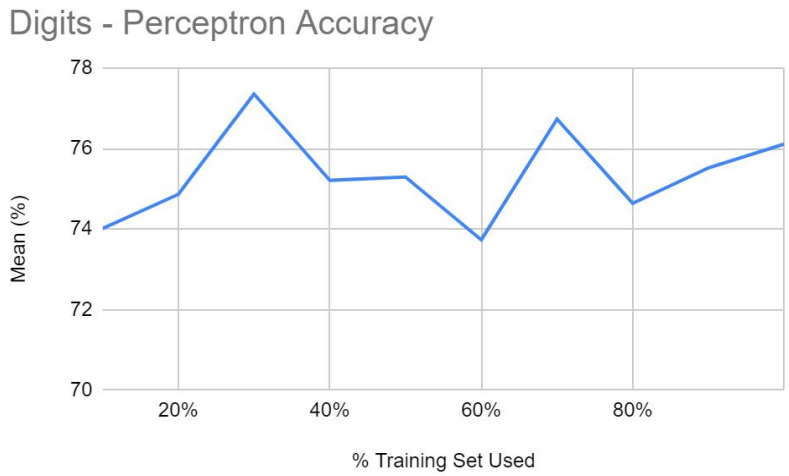
% Training Set Used	Mean (%)	Standard Deviation (%)	Training Time
10%	55.53	5.89	1.24
20%	60.33	11.25	1.69
30%	72.73	11.90	3.31
40%	77.13	9.66	4.18
50%	81.47	6.69	3.20
60%	81.00	7.89	3.71
70%	82.00	5.18	3.76
80%	85.06	4.79	5.15
90%	83.93	6.96	2.85
100%	86.66	2.78	5.76



Perceptron Digits

% Training Set Used	Mean (%)	Standard Deviation (%)	Training Time
10%	74.01	1.67	5.55
20%	74.86	3.57	14.46
30%	77.35	1.56	21.62

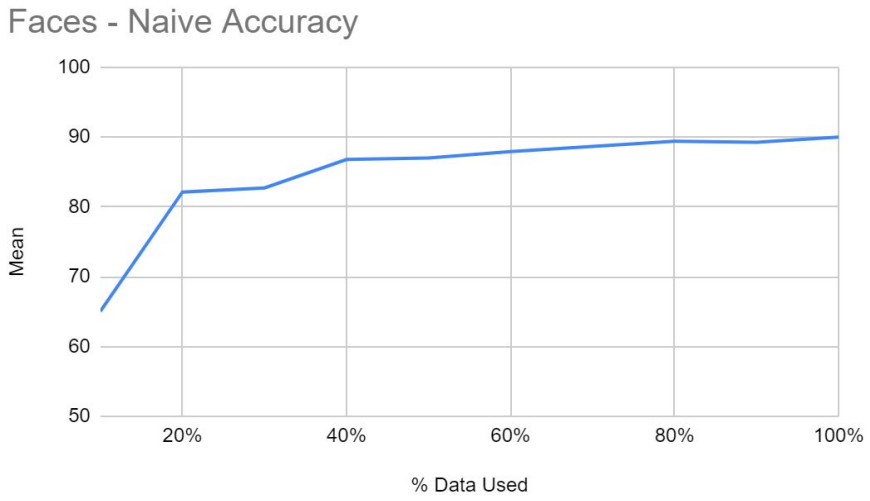
40%	75.21	2.05	29.11
50%	75.29	3.83	36.39
60%	73.73	2.96	43.83
70%	76.73	2.09	49.83
80%	74.64	3.45	57.27
90%	75.52	2.98	63.82
100%	76.11	1.75	70.52



Naive Faces

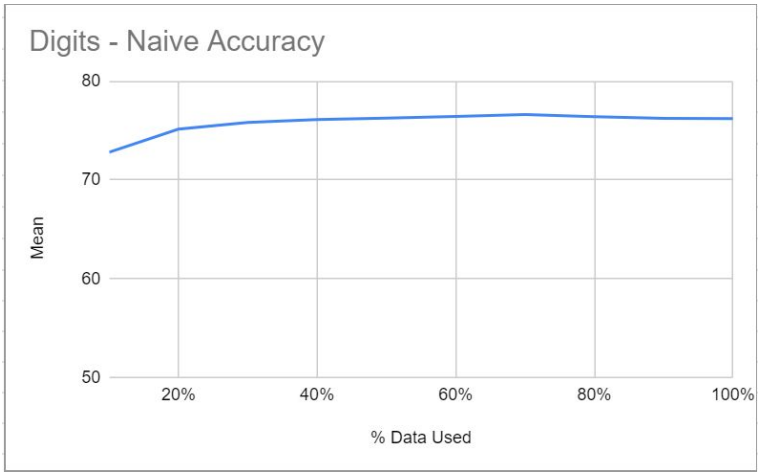
% Training Set Used	Mean (%)	Standard Deviation (%)	Training Time
10%	65.06667	8.71	0.17
20%	82.13333	3.06	0.33
30%	82.73333	2.64	0.53
40%	86.8	1.83	0.68
50%	87	2.04	0.88
60%	87.93333	2.11	0.97
70%	88.66667	1.33	1.15

80%	89.4	1.31	1.29
90%	89.26667	1.31	1.48
100%	90	0	1.62



Naive Digits

% Training Set Used	Mean (%)	Standard Deviation (%)	Training Time
10%	72.81	0.69	0.26
20%	75.15	0.96	0.50
30%	75.81	0.95	0.77
40%	76.11	0.75	1.00
50%	76.25	0.64	1.24
60%	76.41	0.60	1.57
70%	76.61	0.35	1.84
80%	76.4	0.25	2.00
90%	76.22	0.37	2.25
100%	76.2	0	2.46



As seen in the results, for all 4 cases there is a positive relationship between the % of the training set used and the accuracy of the algorithm. There is a tradeoff between accuracy and time, however. As seen in the training times, the larger the training set, generally the larger the training time. This makes sense, since we must iterate through more loops and functions during the training. There is some deviation that prevents the data from being perfect, most likely due to the randomness of the training set chosen, and the randomness in choosing weights in perceptron. Furthermore, it was noted that perceptron was generally slower than naive, due to the need for multiple training sessions, and digits was generally slower than faces, since digits contained more label values and a much larger training set.

Part 3 - Improvements

For improvements we could reduce the effect of randomness even more, since the relationship between accuracy and training size is moderately positive, but could be stronger for some cases such as perceptron digits. However, our results were relatively good for both algorithms for both data types. We surpassed the 60% and 70% threshold for digits and faces respectively. We could make our algorithm a bit faster while iterating through loops, so it would take less time to train.