REPORT

Our code consists of three file: loader.py, perceptron.py, and naive_bayes.py. loader.py has the method that creates the features vector from each image (MNIST or face)—it takes the image file and flattens it into a 2D numpy.array. Since the MNIST images are 28x28 pixels, the features vector for MNIST have a length of $28 * 28 = 784$. The face images are 70x60 pixels, so the features vector for faces have a length of $70 * 60 = 4200$. Each of the features vectors are binary—1 represents a filled pixel, 0 represents an empty pixel. We also remove all the newline characters from the original image in creating the features vectors.

1.) PERCEPTRON ALGORITHM

For our perceptron algorithm, we created a perceptron data structure whose attributes were its weights and bias(es). We initialized the weights and biases as random values between 0 and 1 in their applicable shapes (*np.random.rand(10, 784)* for digits, *np.random.rand(4200)* for faces). In our training method for digits, we iterated through the features vectors and multiplied it by the weights vector, adding the sum of the biases, which yielded the prediction vector. The prediction was a length of 10 by the properties of matrix multiplication, so the each index corresponded to a digit—the index hosting the greatest value in the prediction vector is the digit prediction based on the weights vector. If the prediction was correct, we moved on to the next features vector. If the prediction was incorrect, we added the current features vector to the weight vector corresponding to the correct digit and subtracted the weights vector corresponding to the incorrectly predicted digit by the features vector. We also added 1 to the bias corresponding to the correct digit and subtracted 1 to the bias corresponding to the incorrectly predicted digit.

```
self.weights[label] = self.weights[label] + vector
self.weights[pred_digit] = self.weights[pred_digit] - vector
self.bias[label] = self.bias[label] + 1
self.bias[pred_digit] = self.bias[pred_digit] - 1
```

Note that this was all done with the training data. The algorithm for faces was very similar—the weights array had a shape of 4200 (rather than 10, 784), and there was a single bias instead of an array of 10 biases.

2.) NAIVE BAYES CLASSIFIER ALGORITHM

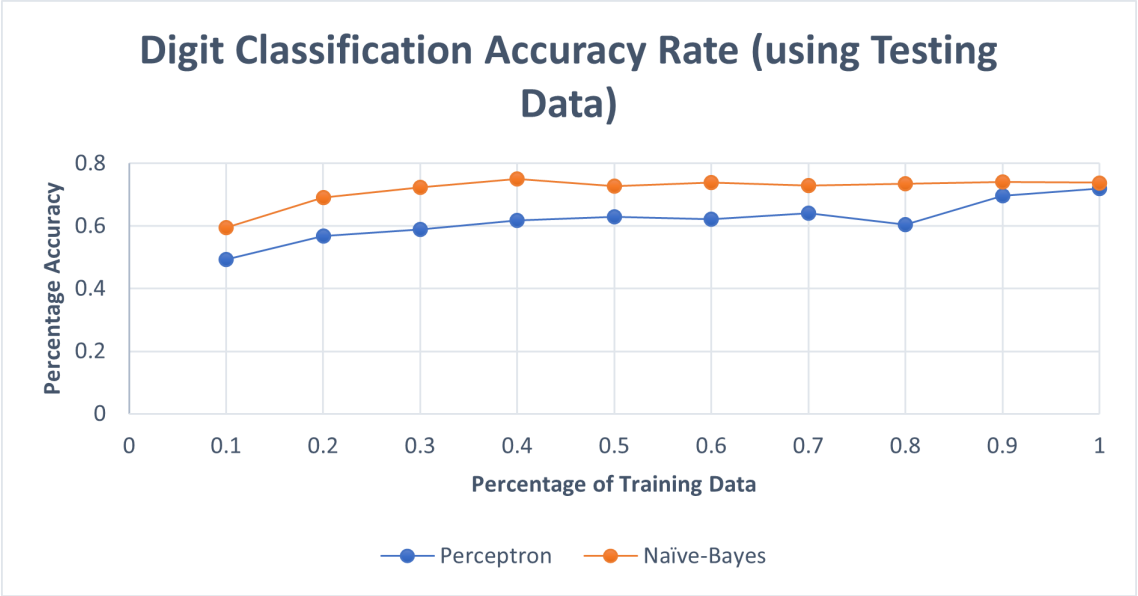
For our Naive Bayes classifier algorithm, we created a data structure that held all the necessary values for us to make calculations while predicting. First, we sifted through the features vectors—for digits, we found the proportions of each digit, and for faces, we found the proportions of face images and non-face images (called *distribution* in the program). Then, we created a matrix to hold the conditional probabilities of each pixel given its label (for instance, for digits, we create a 10x784 matrix, with the vector at matrix[i] containing the conditional probabilities for each pixel for an image that we know is digit i). We calculate this conditional probability by traversing through the list of features vectors, and adding it to its corresponding array in the conditional probability matrix. We then divide each value in the matrix by how many frequently its digit appears in the data set. As the Berkeley website mentions, we must also do Laplace smoothing to make sure that the logarithms we do next can work (if any conditional probability is 0, the logarithm will not work, so we add k to prevent that from happening).

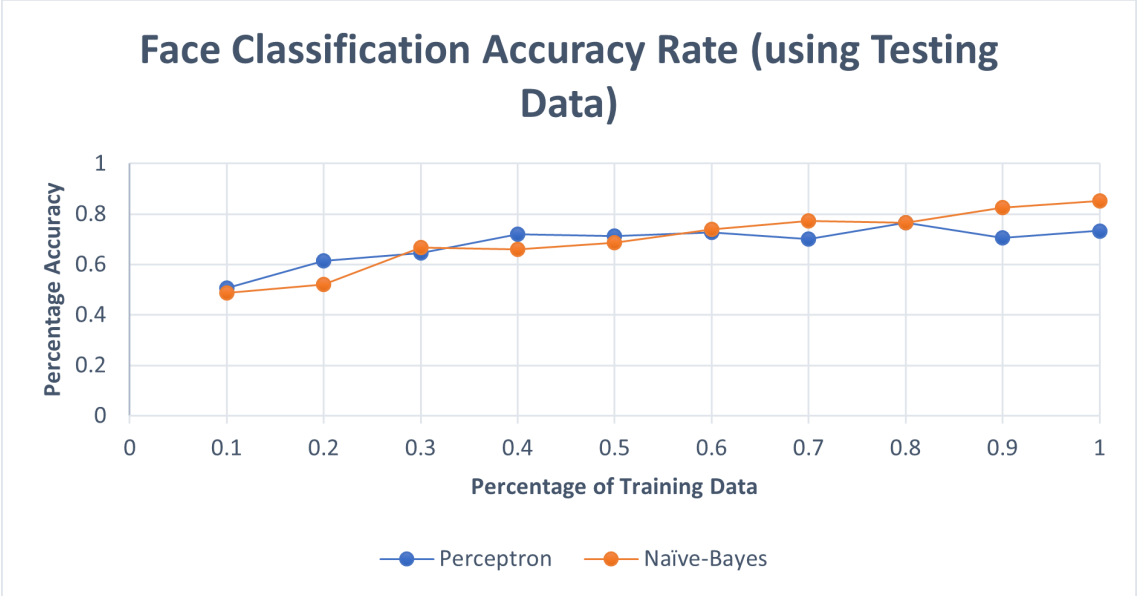
```
for i in range(10):
    for j in range(self.vector_len):
        cond_prob = (counts[i][j]+0.001)/(self.digits_count[i]+0.001)
```

self.cond_probs[i][j] = cond_prob

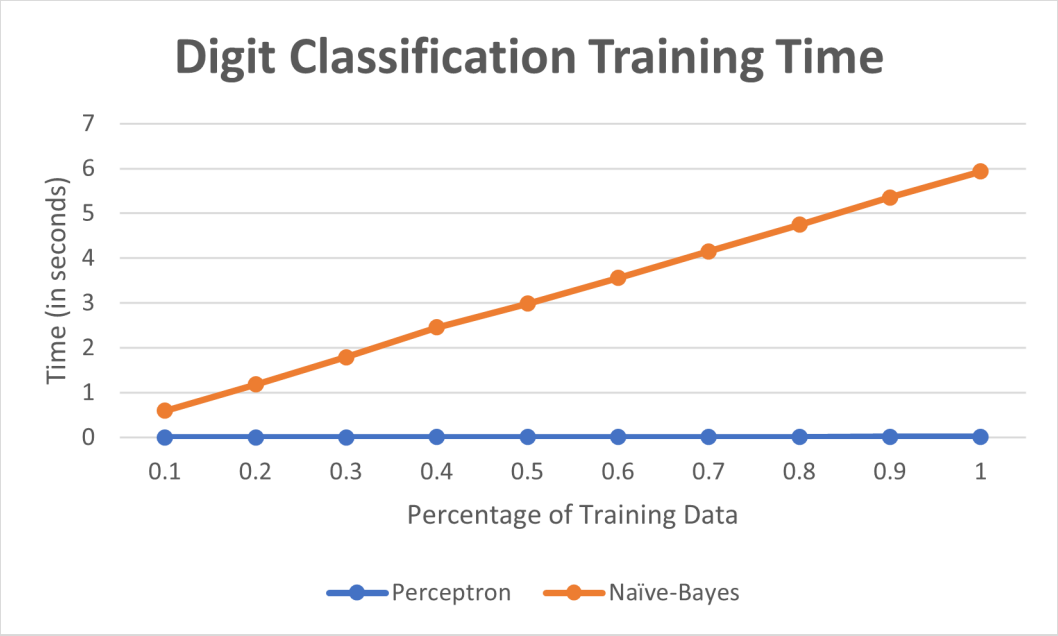
Once we have the conditional probability matrix, we can start predicting. First we create an array to hold the prediction values for each possibility (for digits, the list would have a length of 10, for faces, 2). As we iterate through the features vectors, for each vector, we iterate through and if we encounter a filled pixel, we add the logarithm of the conditional probability of the corresponding pixel to the prediction list. If we encounter an empty pixel, we iterate through the conditional probabilities corresponding to that pixel and subtract 0.1 if they are equal to 1. Then, we add the logarithm of one minus the conditional probability of the corresponding pixel to the prediction list. After traversing the vector, we iterate through the prediction vector and add the logarithm of the corresponding digit's distribution to finalize the prediction vector. We return the index of the maximum value, which is the label the algorithm predicts for the given features vector. We do this until we have gone through every vector in the data set.

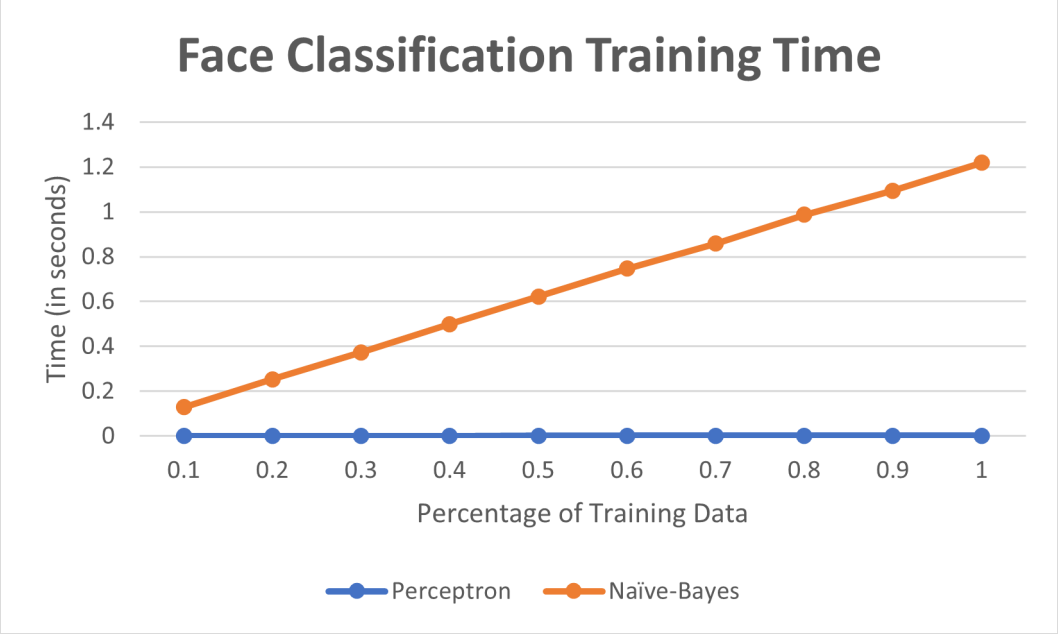
3.) GRAPHS AND EXPLANATIONS





Perceptron Accuracy Std.		Naïve-Bayes Accuracy Std.	
DIGITS	FACES	DIGITS	FACES
0.19	0.195	0.13	0.298
0.23	0.33	0.1234	0.2585
0.247	0.129	0.3584	0.179
0.336	0.338	0.3672	0.0557
0.33	0.099	0.338	0.08
0.144	0.0568	0.376	0.221
0.228	0.145	0.21	0.2579
0.3452	0.2538	0.332	0.254
0.156	0.024	0.142	0.0729
0.151	0.2811	0.38	0.32





The averages of the accuracy of the predictions are in the first two graphs. The first graph shows the average accuracy rates between the perceptron and Naive Bayes classifier for digits, and the second graph shows the same for faces (I could not figure out how to have standard deviations display on the graph in the form of error bars so I just included them in table-format below the first two graphs). Based on the first graph, we can see that the Naive Bayes classifier has a lower base success-rate, which is likely because it accesses the selected data prior and calculates the conditional probabilities for each pixel based on its label, whereas the perceptron's training quality depends on whether the data given is a good representation of the testing data. Regarding the standard deviations, they appear to lack any distinctive pattern—though Naive Bayes seems to generally have lower standard deviation rate in comparison to perceptron. Generally, the both algorithms increase in accuracy the more training data they are given, though perceptron seems to plateau after 75%, and Naive Bayes plateaus after around 80%. We can improve these accuracy rates by improving the algorithms.

In regards to the last two graphs about training time, the Naive Bayes classifier runs for far longer than the perceptron algorithm runs for. The perceptron hovers at around 0.01 seconds even when given 100% of the data, whereas the Naive Bayes classifier's training time increases linearly with the amount of training data.