RUTGERS

CS440: Project 2

Image Classification

Alborz Jelvani

Professor: Rui Wang

Department of Computer Science

Rutgers, The State University of New Jersey

August 2020

For further references on code, please refer to: `https://github.com/Jelvani/Rutgers_CS440_Intro_to_AI/tree/master/Image_classification`

1 Introduction

In this project, we will be performing the task of image classification through machine learning methods. Our tests and analysis are performed on a data set of images, which are from: `https://inst.eecs.berkeley.edu/~cs188/sp11/projects/classification/`, which this project is based from.

The goal of this project is to show that by inputting training data into a set of algorithms, the classes from the data can be 'learned' by the model, and be used for inference, with a certain degree of accuracy. The 2 classification techniques we use are by the *Naive Bayes* algorithm and *perceptron* algorithm, which will be discussed in more detail in later parts.

2 Experimental Setup

All tests are performed on the following machine:

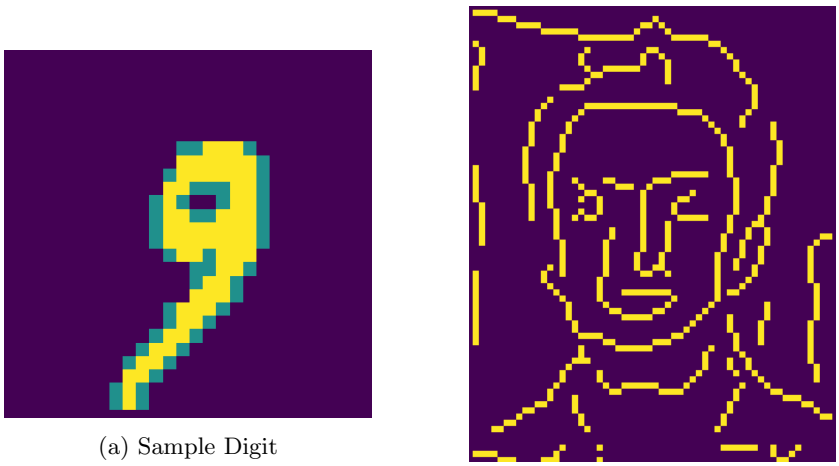
- OS: Windows 10 Enterprise version 1903
- CPU: Intel Core i7-4790 @ 3.60 GHZ
- RAM: 16.0 GB
- Python Version: 3.8.2 (32-Bit)

All run times shown are in seconds, and timing is calculated with Pythons built-in `time` module. For run-time testing, any print functions that are used in-between timings are disabled, as well as any plotting functions. For each training instance, Python's built-in `random` module is used to pick a random index from a percentage of the training data. For our tests, we perform 5 iterations of inference for every percentage of training data increment (done in increments of 10% from 10% to 100%) and average the accuracy's for the graphs. The same is done for train time calculation.

3 Given Data

We are given 2 data sets which will be used for this entire project. They are both image data sets, one consisting of labeled hand-written digits, and another consisting of contour edges of human faces. The data sets are also accurately labeled for our use in this project. The data is converted into a `numpy` array of integers, where for our digits, the edges are represented as 2, and interior areas are 1. Any white space is considered a 0. On the other hand, for our faces, the contours are represented as 1, and any white space is 0. This method will consist of our features for our classifiers, where each pixel in the image represents an integer, where 0 pixels are locations of no data, and non-zero pixels are the locations of our data, for every single sample.

We then extract these features as a feature vector using the `flatten` function in `numpy`.



(a) Sample Digit

(b) Sample Face

Figure 1: Sample Data

4 Naive Bayes

In this section, we will discuss the Naive Bayes classifier and look at our experimental results.

A Naive Bayes classifier is nothing more than the computation of probabilities, and choosing the highest probability as the algorithms guess. The basis of the classifier sits on top of *Bayes Theorem*, which states:

$$P(A \mid B) = \frac{P(B \mid A) \cdot P(A)}{P(B)} \tag{4.1}$$

This equation is often the basis of many machine learning techniques, and has a few parts.

$P(A \mid B)$ is known as our *posterior*, and is what the user is trying to compute. For example, we want to compute the probability of the digit 3 (A), given our feature vector (B).

$P(B \mid A)$ is known as *likelihood* and this will be calculated from our training data. An example would be, the probability of these features (B), belonging to digit 3 (A).

$P(A)$ is known as our *prior*, and is very easily computable from our training data. It is simply the probability of getting class A, for example, the probability of the digit 3. You could think this could just be set to 1/10, since 10 digits exist in our decimal system, but it is often better to not make assumptions and use the actual probability from the training data, which we do

Finally, $P(B)$ is known as our *marginalization*, and is simply the probability of having those feature vectors for each digit or face.

Now that we have looked at Bayes Theorem, we can begin to discuss what a Naive Bayes classifier is doing.

In simple terms, all we are doing is using Bayes Theorem to compute the probability of each class (each digit or face), and then choose the highest probability as our guess. More formally:

$$P(class \mid f_1, ..., f_m) = \frac{P(f_1, ..., f_m \mid class) \cdot P(class)}{P(f_1, ..., f_m)} \tag{4.2}$$

Which will give us the probability of one class, given feature vector F. We need to compute this for each $class \in Y$, where Y is the set of all possible classes:

$$argmax\ P(Y \mid f_1, ..., f_m) = \frac{P(f_1, ..., f_m \mid Y) \cdot P(Y)}{P(f_1, ..., f_m)} \tag{4.3}$$

We can see that our denominator can be seen as a simply scaling factor, and therefore can be ignored, since it is common to all values we take the maximum of:

$$argmax\ P(Y \mid f_1, ..., f_m) = P(f_1, ..., f_m \mid Y) \cdot P(Y) \tag{4.4}$$

Finally, we can use the chain rule to to expand the probabilities of our feature vector, using the assumption that the features are conditional independent (which is the reason we call this *naive* Bayes):

$$argmax\ P(Y \mid f_1, ..., f_m) = P(Y) \prod_{i=1}^{m} P(f_i \mid Y) \tag{4.5}$$

Now that we know what we must implement, I will discuss how we computed our likelihood, $P(f_i \mid Y)$.

For our digits, we are given a bunch of images, and have 10 classes (0-9). When training our Bayes classifier, we keep a count for each possible value of a feature we encounter for each class. For example, in our digits classifier, we keep a count of how many 0, 1's, or 2's we encounter for each class. We need this since the formula for conditional probability states:

$$P(f_i|class) = \frac{P(class \cap f_i)}{P(class)} \tag{4.6}$$

This means that when we are done training, we have the total number of times f_i was encountered in a class and can multiply this probability for every feature when we are testing our classifier, for each class.

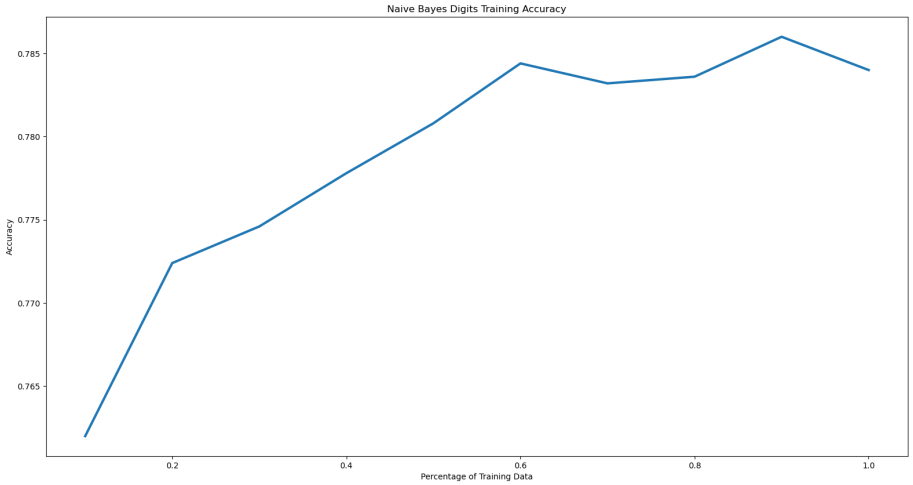
One thing to note, is that the product of many small probabilities can lead to underflow in binary computing systems, and therefore, we will be taking the log probability, rather than production, which translates to the addition of each likelihood probability after taking its log, shown:

$$argmax\ P(Y) \prod_{i=1}^{m} P(f_i \mid Y) = argmax\ \log P(Y) + \sum_{i=1}^{m} \log P(f_i \mid Y) \tag{4.7}$$

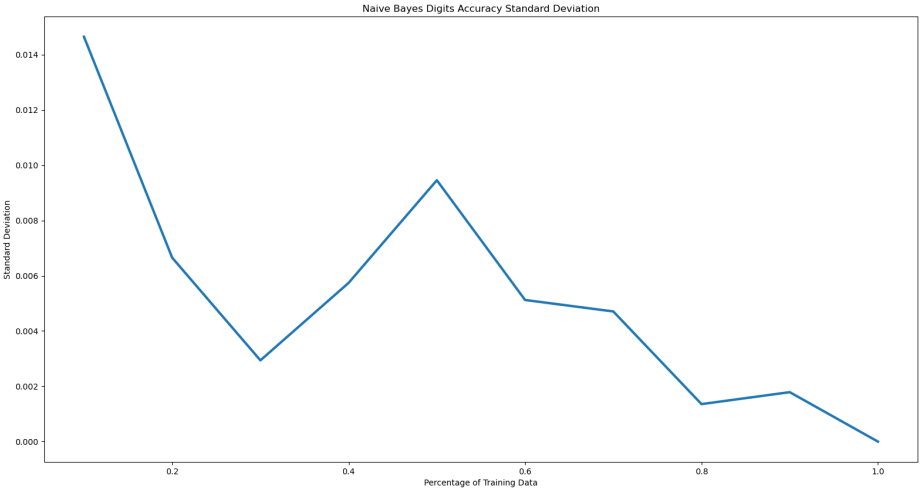
In addition, what do we do if the probability is 0? This is an issue because the logarithm of 0 is undefined. We will fix this through a smoothing technique called *Laplace Smoothing*, by adding a constant k to our observations:

$$P(f_i|class) = \frac{P(class \cap f_i) + k}{P(class) + k} \tag{4.8}$$

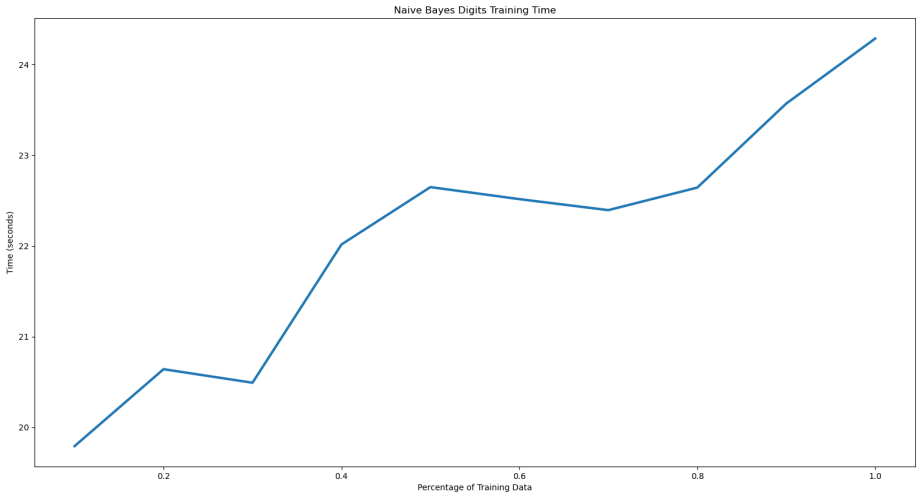
Below is presented our results for our Naive Bayes classifier for the digits data set:



(a) Accuracy



(b) Standard Deviation

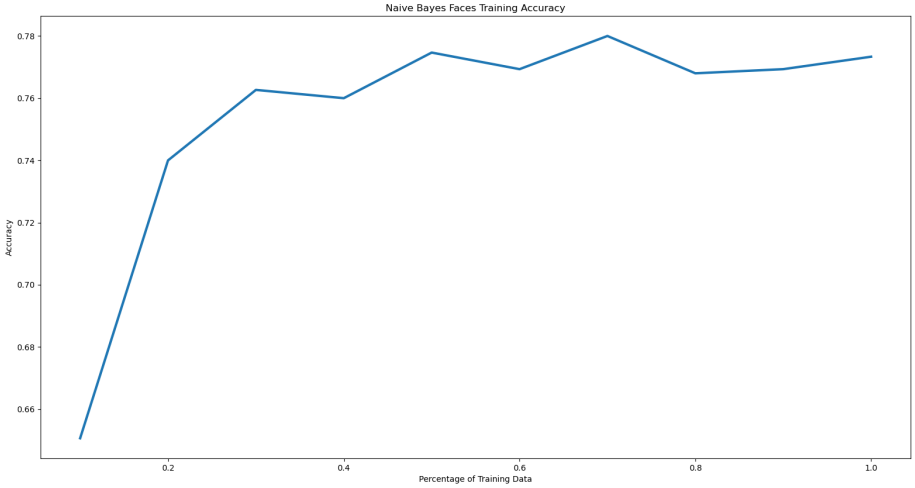


(c) Time

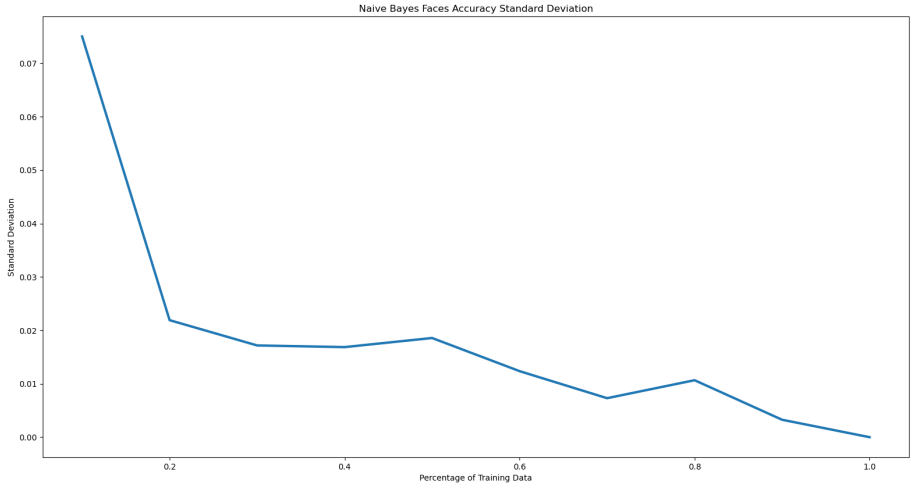
Figure 2: Naive Bayes Digit Training Results

From our results in our digit training set, we can see we were able to achieve approximately, 79% accuracy and we see our standard deviation for 5 trials at each training data percentage has an overall decrease. Our time to train also increases, as expected, due to the larger amount of training data.

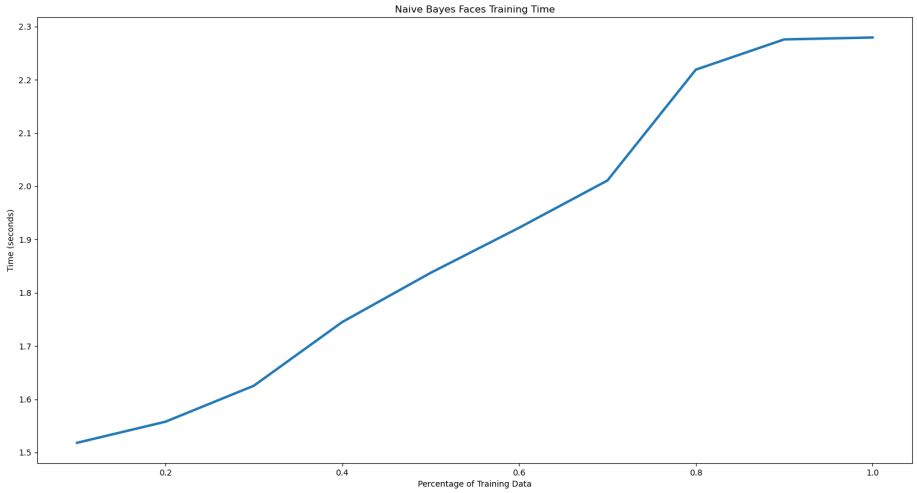
We can also see similar performance for our face training data below:



(a) Accuracy



(b) Standard Deviation



(c) Time

Figure 3: Naive Bayes Face Training Results

5 The Perceptron

Unlike our Naive Bayes classifier, the perceptron algorithm has the goal of finding a linear function that can separate classes of data. In fact, you can just think of the perceptron algorithm as trying to find a linear decision function of *n-dimensions* that best separates data. An example can be seen below from our class slides:

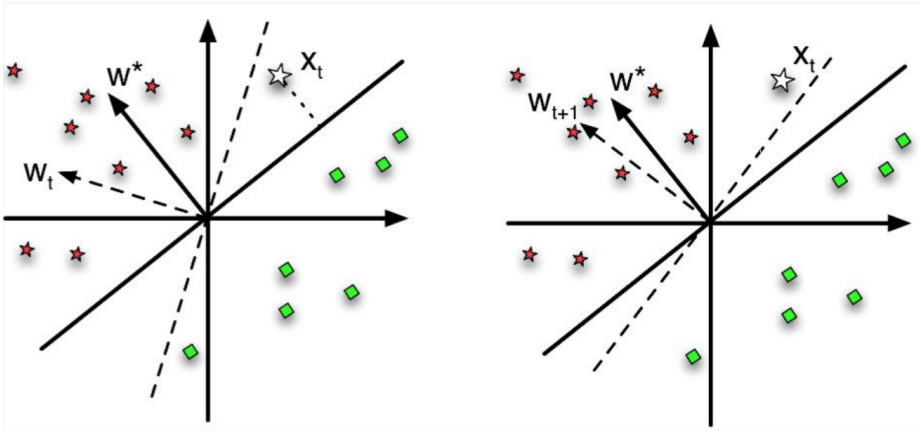


Figure 4: Perceptron updating its linear decision boundary for 2-dimensions

In figure 4, you can see that the linear function is only for 2-dimensions, but the perceptron algorithm applies for *n-dimensions*, which makes it very powerful.

At the heart of a perceptron, lies a vector of weights, which are just the slopes for each dimension of the linear decision function. And these weights, are what are being updated after each pass of data. We can formally represent the score for a perceptron for each feature, f, and class, y, as:

$$score(f, y) = \sum_i f_i w_i^y \tag{5.1}$$

Now we can think, well we get a score for one class, for example the digits 1, but what does that mean for a set of features, which can be thought of as random variables?

Well if we think about it, each vector of weights, which we have for every class present, is best tuned for that certain class, meaning, the goal of the weights is to produce the largest score based on the optimal input. Therefore, we can follow a similar method to our Naive Bayes classifier, and compute the score for each class, and choose the largest score as our guess.

For our faces we are just concerned with a binary results, true or false. For this, we define an activation function, which I chose as just a regular binary step function:

$$Activation = \begin{cases} True & \text{if score(f,y)} \geq 0 \\ False & \text{if score(f,y)} < 0 \end{cases} \tag{5.2}$$

Where below a 0 means no face, and otherwise means a face was detected.

It is important to define the method of choosing our class from our score, because this plays a role in how we tune the weights for each class.

For our digit classifier, every time we receive an training image, we extract a vector of features, who's size determines the dimensions of our perceptron. For example, in our digits we receive an image of size 28 X 28. Therefore, our feature vector is of size 784, and we also have 784 weights, and one bias, which we keep at 0 for our tests.

We initialize these weights to initially 0, and after every iteration of an image, we see if our guess is correct. If not, then we update our weights, as below:

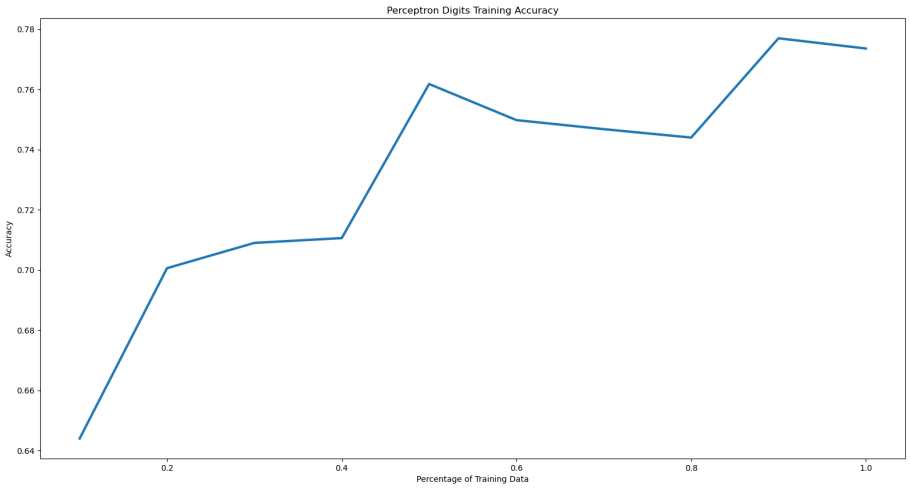
$$\begin{aligned} w^{y} &= w^{y} + f \\ w^{y'} &= w^{y'} - f \end{aligned} \tag{5.3}$$

Where we subtract our feature vector from our weight vector for the wrong guess, and also add our feature vector to our actual class that is from the label of the image.

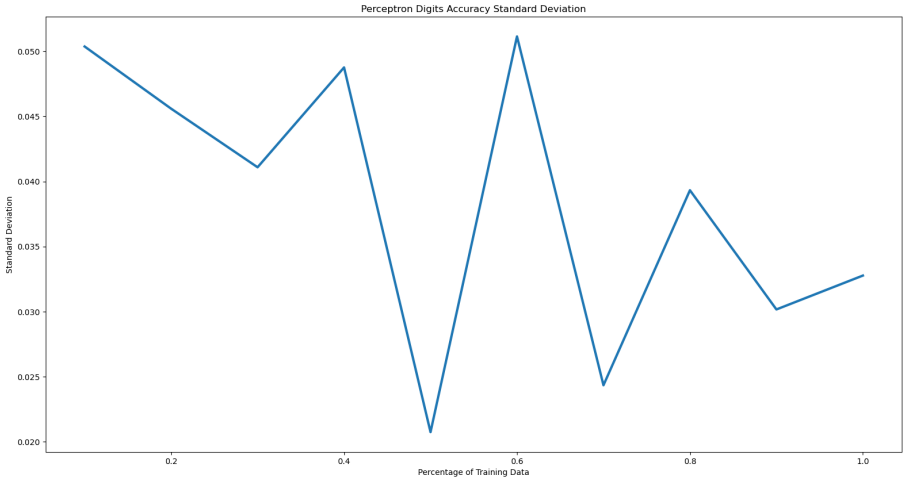
By doing this, with enough training data, the weights of those pixels that consist mostly of locations for a class, will increase, while other pixel weights will not increase by much. Therefore, when we compute the score with the dot product of 2 vectors, we will get a corresponding score.

Now that we have talked about the basics behind the perceptron algorithm, we can take a look at our experimental results.

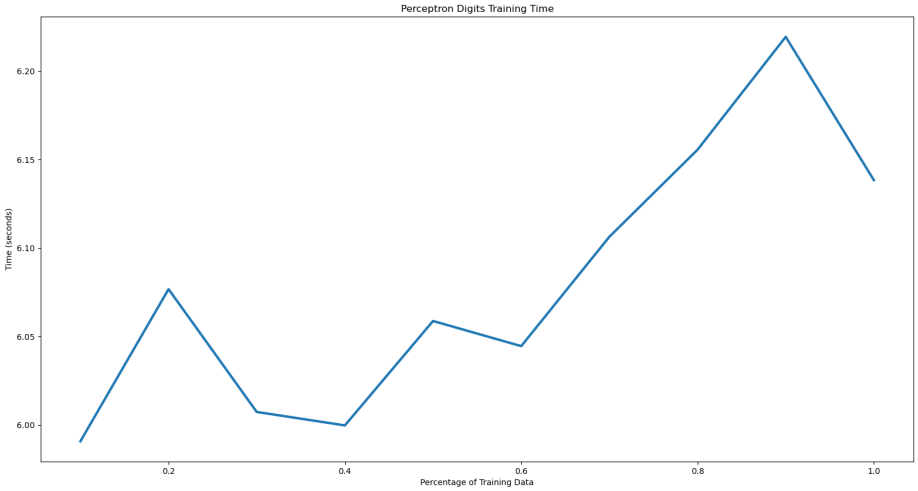
First we present the results for our digit classifier:



(a) Accuracy



(b) Standard Deviation

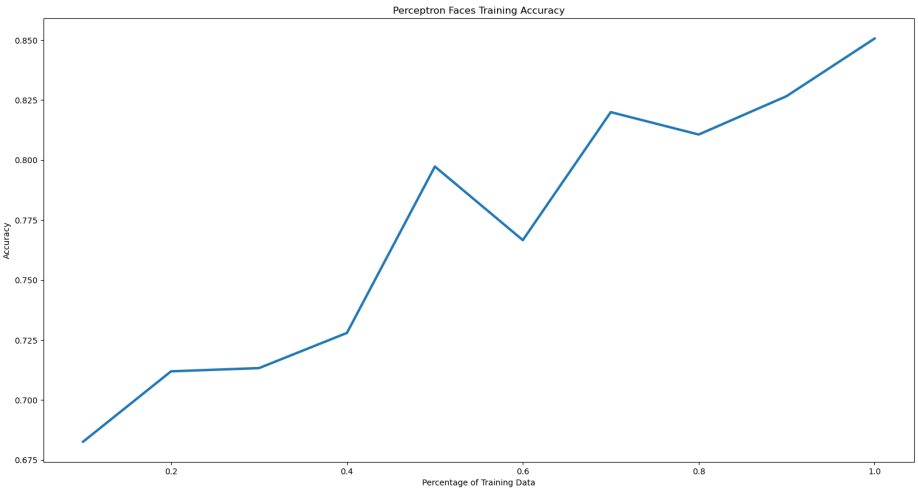


(c) Time

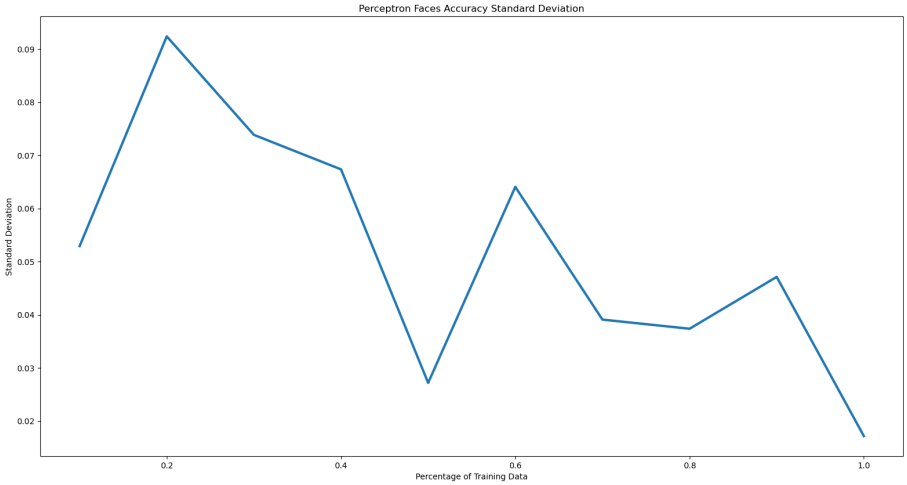
Figure 5: Perceptron Digit Training Results

Our results properly follow what we would assume, especially for our accuracy with more training data, and our time needed for training, although a few disruptions in our linear increases could be due to caching involved with the system reading data, and other similarities. We do see some issues with standard deviation, but we will discuss this in the next section.

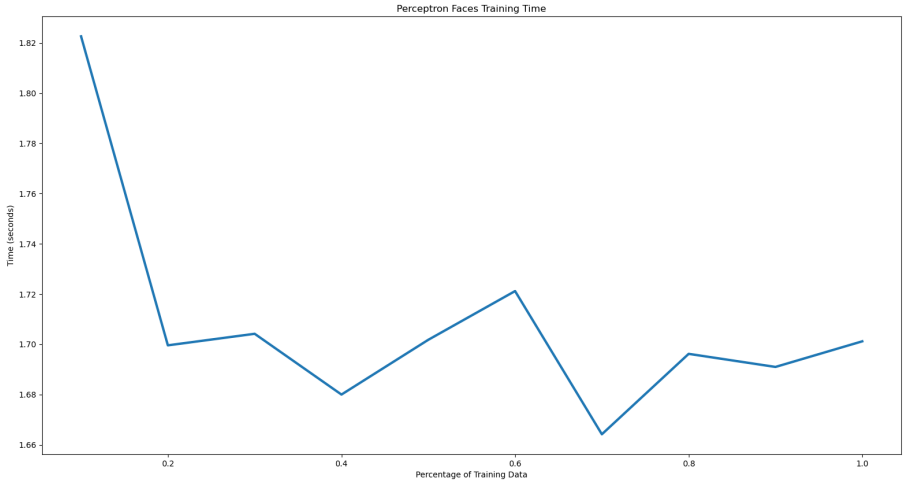
We can also see our training results for our face classifier:



(a) Accuracy



(b) Standard Deviation



(c) Time

Figure 6: Perceptron Face Training Results

We see similar performance in our accuracy for the face classifier, but we do see something odd with the time. An explanation for this can be due to the small range of time in our collected data, which makes the time jumps look larger than it actually is. The difference between the time is around 200 milliseconds, which can not really reflect our training data, especially since we are dealing with only our binary classes for the face classifier. Also, the results could reflect the use of a non-realtime kernel (the Windows NT kernel), as well as caching for our data set file, which is explained by our long run time for our fist iteration at 10% training data.

6 Lessons Learned

In this section, we will briefly go over some other factors that may lead to inconsistencies in our training experiments.

One important thing to note is that in our Naive Bayes classifier, when we are using 100% of our training data, we will always have a standard deviation of 0, since the order we read the data has no impact on our finally probabilities, as long as we use all of it.

On the other hand, for our percpetron algorithm, the order we read the data, even at 100% training, has an impact on our accuracy and standard deviation, since we may have cases where a certain inference is correct, and that image does not contribute to updating the wights. And this of course depends on the order we read the input training images. We see this in our standard deviation chart.

To help guarantee randomness, we used Python's built-in `random` module, which is supposed to gives us a random integer as the index of the training data to choose for each pass. However, I found that the random module is not really *random*, as not anything that uses math equations can truly be random. It is rather *pseudo-random*. Below is a chart of the distribution of integers picked from a range of 0 to 500 for 10 million iterations on the test machine:

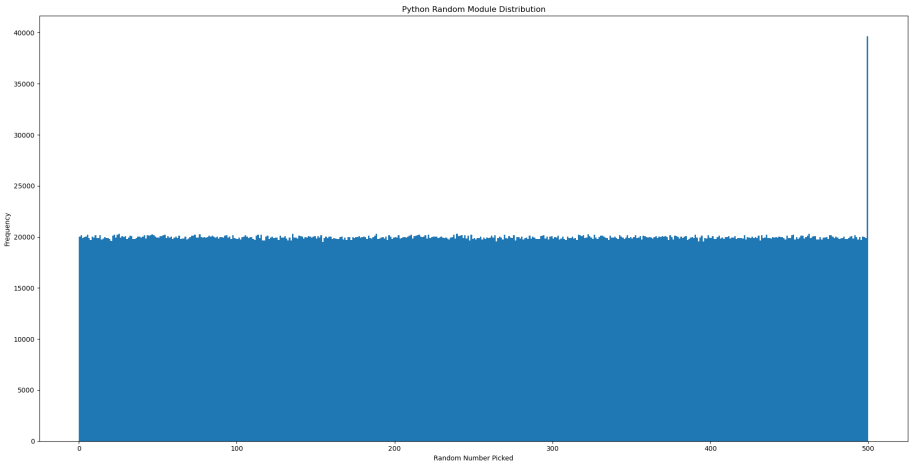


Figure 7: Distribution of integers using Pythons Random module, for the range [0,500] for 10,000,000 iterations

Oddly enough, we do see the random numbers are preferred to be close to our upper bound, near 500.

Due to this error in our random number generation, we can expect our standard deviation to not truly reflect our model, and may therefore be skewed, especially for our perceptron algorithm, which relies on the order we pick our training data.

So how do we fix this? Well on Intel CPU's, they have hardware random number generators, which do not rely on math algorithms, and are therefore not deterministic. This can be used with the Intel IA32 ISA: `RDRAND`. To use the hardware random number generator in Python, we can use the `os` module,

which gives us access operating system functions. We can use the function: `os.urandom()`, which will give us a string of random bytes from the hardware generator. This would be interesting to test in the future, and add on to this assignment.

References

[1] D. Klein and J. DeNero. Course: Berkeley CS188

[2] `https://software.intel.com/content/www/us/en/develop/articles/intel-digital-random-number-generator-drng-software-implementation-guide.html`

[3] `https://en.wikipedia.org/wiki/RDRAND`