Siddharth Hegde

RUID: 218000972

Final Project Write-up

Algorithm Implementation:

The Naïve Bayes Classifier is implemented as a probabilistic model that learns the conditional probabilities of features given a label, with the prior probabilities of each label. For data representation, the classifier works with binary data, in which each data point is represented as a util.counter, which maps each feature index to a binary value (0 or 1). The labels are discrete values, which are from 0-9 and the classifier considers the probability distributions of each label and its feature. During the training, the classifier initially collects all the unique features across the training data. By doing this, it ensures the model knows the full set of features to estimate the probability. In addition, the training step also counts how often each label appears (P(label)) as well as how often each feature is on, or when value = 1, given a particular label. In the implementation, the code uses util. Counter objects to accumulate these counts, such as: labelCounts[label] storing number of samples having a given label and featureOnCounts [(feature, label)] storing number of samples with a given label that has feature = 1 (estimating P (feature = 1 | label)). After that, Naïve Bayes uses Laplace smoothing to avoid zero probabilities. The formula is as follows, given a parameter k: P(feature=1|label) = (featureOnCounts[(f,label)] + k) / (count_of_samples_with_label + 2k). The denominator accounts for the smoothing for both outcomes, ensuring probability is never zero. By having probabilities that are not zero, this will not cause issues when taking logarithms for future computations. Once the model is trained, the classifier store both the P(Label) and P(feature = 1 | label) . Then, the classifier computes the log-likelihood for each label, using the formula:

$$\log P(label) + \sum_{f \in features} [x_f \log P(f = 1|label) + (1 - x_f) \log(1 - P(f = 1|label))]$$

The log probability of a label given a datum uses this formula, which then the classifier chooses the label with the highest log probability. To classify a new sample, the classifier iterated over all the labels, and computes the joint probability, which then picks the label with the max value.

In contrast to the Naïve Bayes algorithm, the Perceptron Classifier is a linear model, which is consistent with a weight vector for each possible label. The weights are stored in util.Counter objects, and unlike Naïve Bayes which store probabilities, the Perceptron classifier stores linear weights, which are used to score each input. Initially, the classifier starts with initializing all the weights to zero. Each label has its own set of weights, and if there is an instance of multiple labels, such as digits for example where the labels were 0-9, there would be multiple linear classifiers, one for each label. The classifier then makes a number of full passes over the entire training set, otherwise known as the epochs, which is initialized in the main.py through the variable max_iterations. For each training sample, the classifier computes a score for each label by taking the dot product of the sample's features and its respective weight vector, which can be represented by the formula:

$$score(label) = \sum_{f} w_{label,f} \cdot x_f$$

It then predicts the label with the highest score. In the case of the prediction being incorrect, it updates the weights through the formulas:

$$w_{true\_label} \leftarrow w_{true_{label}} + n \cdot x$$
$$w_{predicted\_label} \leftarrow w_{predicted\_label} - n \cdot x$$

In this formula, n, is the learning rate. The learning rate controls the magnitude of each weight update when the model makes a misclassification. A smaller learning rate results in improving stability as it makes more gradual weight adjustments, whereas a larger learning rate makes bigger jumps, which can speed up the learning process but risk instability. In addition, the updates push the weights of the correct label's vector in a direction which makes it a higher probability to score higher in the future. This process of correction through iteration continues till all the epochs are completed.

Moreover, both models use util.Counter to use as a baseline of representation of features and labels. However, Naïve Bayes and Perceptron have different methodologies when it comes to their approach. Naïve Bayes uses probabilities, frequencies, and makes predictions by computing log-probabilities. Training using this classifier involves counting the number of occurrences and normalizing the data. Perceptron uses iterative correction and a learning rate to adjust weights when it misclassifies a sample. Through multiple passes of data (epochs), it can be sensitive the order of samples as well as the learning rate. Eventually after multiple iterations of computing label scores, the Perceptron classifier linearly separates the data and selects the label with the highest score.

For the feature extraction portion of this project, I used samples.py (inspired by the Berkeley code). This file is responsible for reading in raw image data, in which using ASCII text representations of digits or faces, it converts the data into a structured internal format. For instance, the function loadDataFile(filename, n , width, height), n reads the lines of ASCII image data. The image is also represented with a height and width of characters, in which characters being ‘ ‘, +, and # for digits or faces. The characters are then converted into integers such as 0,1 or 2 using helper functions live convertToInteger, which indicates different scales of pixel intensity or edge detection. The results are then stored in a util.Counter object, which maps each pixel to a numerical feature, 0,1,2, in which 0 indicates a blank pixel, 1 for a light pixel, and 2 for a dark pixel. After samples.py returns these items, each sample is an object of Counter of features, which are indexed by pixel coordinates. This is one step of the feature extraction implementation, where it turns ASCII characters and translates them into a numerical representation so that the learning algorithms can train on in a smooth way.

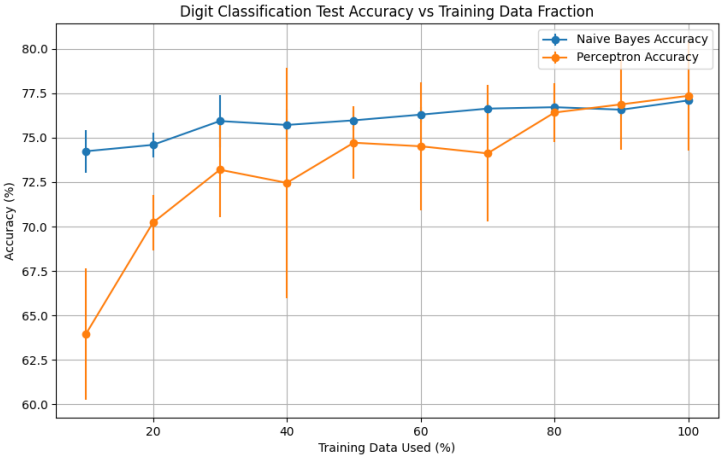
After samples.py returns an object of Counters, which are mapping coordinates to integer pixel values, main.py also provides another layer of feature processing to convert the raw pixel data into binary features. One instance of this is the toBinaryFeatures(data, mode) in main.py, where the function iterates over each Counter object. The function then decides which data it is processing: if it is digit data, each pixel in the original image can be 0, 1, or 2 or blank, gray, or black respectively. The code then simplifies this by mapping zero to zero and the non-zero values to 1. This then creates a binary image, where every non-blank pixel is treated as a feature. For face data, the logic attributes a pixel as “on” or 1, if its value is 2, which corresponds to ‘#’ in the ASCII representation, which is usually an edge or a strongly present feature. Otherwise, it remains “off” or 0. This logic just differentiates the features of interest for face detection. Finally, after binarizing the data, it is passed into the classifiers’ train and classify functions to then train the model.

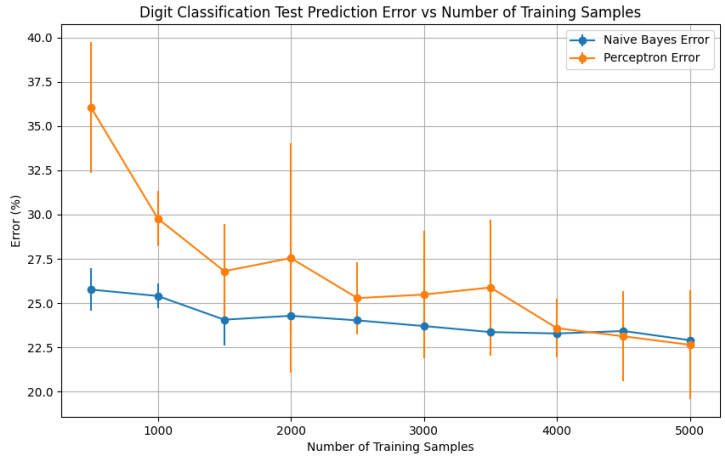
Digit Classification:

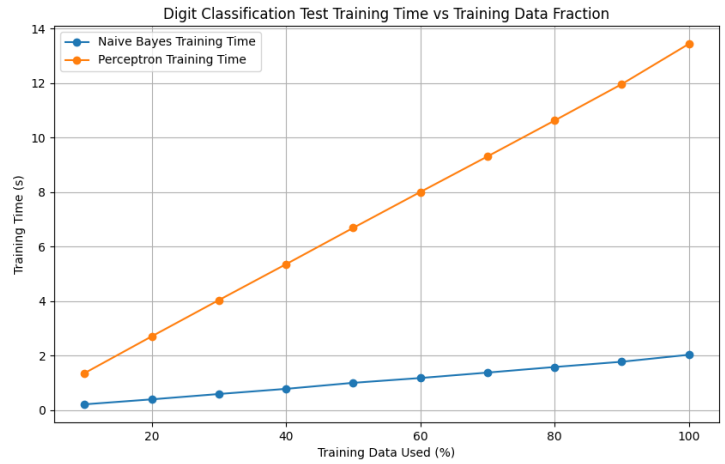
The Naïve Bayes classifier achieves a relative stable accuracy across different training set sizes. Originally, with only 10% training size, it achieves 74.38% accuracy and progressively improves in small increments, and finally ends up at around 77% accuracy (100% data). This indicates that Naïve Bayes is very quick to capture underlying patterns in the digit dataset and benefits only partially from additional data after a certain point. On the other hand, the Perceptron classifier begins with a lower accuracy (61.82%) on only 10% of the data, which means that it struggles more than Naïve Bayes with limited data. Subsequently, as more samples were fed into the model, the Perceptron's accuracy increases, in which sometimes even exceeded the performance of Naïve Bayes. However, the accuracy remains variable, which reflects Perceptron's greater sensitivity to data availability.

When examining the prediction error, a similar pattern forms. Naïve Bayes maintains a relatively low and stable error rate, plateauing around the mid to lower 20 percentage. On the other hand, the perceptron's error starts off higher with the limited number of samples (35-40%), but as it is fed more data it converges to a range similar to Naïve Bayes. Based on this experiment, it indicates that the Perceptron Classifer needs more data to reduce its error rate and stabilize its performance, whereas Naïve Bayes stays relatively similar no matter the training set size.

Both the Naïve Bayes Classifier and the Perceptron Classifier increase at a relatively linear rate as more training samples are used. Initially, at 10% of the training sample, the Naïve Bayes starts out at 0.21 seconds on average whereas Perceptron starts roughly around 1.35 seconds. This significant difference means that the Perceptron's iterative weight updates and adjustments consume more time than Naïve Bayes' probability computation. As the training size grows, Naïve Bayes scales only minimally, increasing to just under 2 seconds with 100% of the data. This stable growth can be attributed to the classifier's straight forward estimation methods. On the other hand, the Perceptron's training time increases significantly, which surpasses 13 seconds with all 5,000 samples. This indicates the Perceptron's complex procedure, in which accumulates a large number of operations as the dataset grows larger.

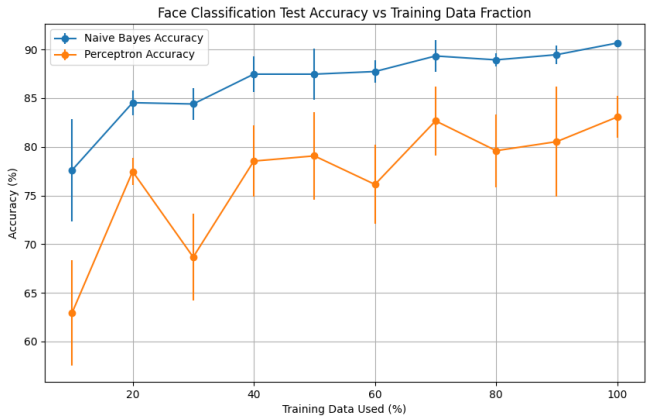




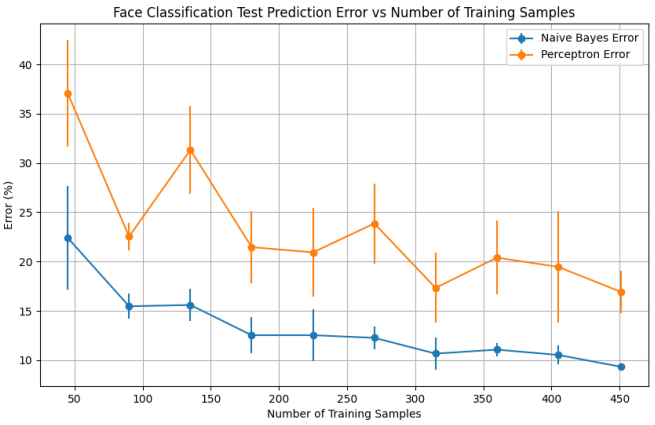


Face Classification:

Both Naïve Bayes and Perceptron classifiers show a clear improvement in accuracy as more training data is fed to the model. Naïve Bayes starts out strong with above 70% accuracy at 10% of the data and ultimately reaches an astounding 90.67% accuracy at full training capacity. Throughout its progression, the accuracy maintains its stability as seen through the low variation as depicted by its standard deviations. This consistency indicates that Naïve Bayes has a quick grasp of the patterns necessary for identifying faces and continues to refine its recognition through the passing of more data. The Perceptron classifier in comparison to its performance with the Digits data, starts off weaker with 62.93% accuracy at only 10% of the data. However, as more samples are trained in to the model, the Perceptron steadily improves, drawing closer to the performance of Naïve Bayes. Ultimately, after using 100% of the dataset, it achieves approximately an accuracy of roughly 83% with a moderate standard deviation. Even though it does not match Naïve Bayes' performance, its ability to reduce its error rate through larger datasets is proof of its ability to learn effectively.



When measuring the prediction error, both models' prediction error declines as the datasets grow. Naïve Bayes' transition from 22.4% at 10% data to just under 10% shows its strong and effective learning curve. In addition, Perceptron shows the most change as its error rate was much higher, being roughly 37% at 10% data to approximately 17% at full capacity. Even though the error rate is higher than Naïve Bayes', the delta between the initial data to full data shows the Perceptron's strong ability to improve as more data is introduced. Moreover, both classifiers benefitted immensely from increased training data in terms of accuracy and training data. Naïve Bayes showed stronger performance whereas Perceptron showed stronger improvement based on initial error rate to final error rate, showing that larger data sets is beneficial to its linear decision-making algorithm.



Similarly to the Digit task, both the Naïve Bayes and Perceptron classifiers show an increase in training time as the training data grows. However, the consistency of the growth of time differs notably between the two classifiers. At 10% of the data, both the Naïve Bayes and Perceptron train quickly, both being under 0.2 seconds. As more data is introduced Naïve Bayes' training time scales up smoothly, reaching just over 1 second at the full training set, meaning the estimation methods remain efficient. However, the Perceptron's training time grows more noticeably as the dataset grows. At 100% of the data, it reaches 1.49 seconds, and although it is not a large amount of time, the growth in training time is more noticeable than the time of Naïve Bayes. In comparison to the digit task, the iterative weight updates that it performs after encountering misclassifications contribute to the time increase. Overall, both classifiers become slower as they process larger training sets of face data, however Naïve Bayes grows in a moderate and predictable increase, while the Perceptron requires more time due to its computational efforts as the data set grows.