**Image Classification Report**

Note might need to use (py -2 m dataClassifier ) to run code if on python 3

**Naive Bayes:**

*Digits*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Amount of training Data used | Avg Time to train data | Percentage accuracy on test data given 5 trials and column amount of random training data | Mean Accuracy | STD |
| 500 | 0.79s | 77, 77, 74, 75, 81 | 76.8 | 2.4 |
| 1000 | 1.55s | 78, 73, 80, 74, 76 | 76.2 | 2.56 |
| 1500 | 2.34s | 79, 80, 76, 76, 76 | 77.4 | 1.74 |
| 2000 | 3.19s | 80, 76, 79, 79, 80 | 78.8 | 1.47 |
| 2500 | 3.92s | 81, 79, 78, 78, 79 | 79 | 1.22 |
| 3000 | 4.83s | 77, 75, 79, 77, 76 | 76.8 | 1.327 |
| 3500 | 5.67s | 77, 77, 80, 77, 75 | 77.2 | 1.6 |
| 4000 | 6.40s | 78, 78, 80, 78, 78 | 78.4 | 0.8 |
| 4500 | 7.246s | 78, 79, 78, 80, 78 | 78.6 | 0.8 |
| 5000 | 7.962 | 79, 79, 79, 79, 79 | 79 | 0 |

*Faces*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Amount of training data used | Time to train data (sec) | Percentage accuracy on test data given column amount of random training data | Mean accuracy | STD |
| 45 | 0.378s | 53, 70, 53, 53, 52 | 56.2 | 7.73 |
| 90 | 0.736s | 63, 66, 67, 63, 76 | 67 | 5.34 |
| 135 | 1.102s | 83, 75, 83, 77, 83 | 80.2 | 3.90 |
| 180 | 1.444s | 86, 80, 92, 84, 87 | 85.8 | 4.38 |
| 225 | 1.812s | 84, 90, 83, 85, 86 | 85.6 | 2.7 |
| 270 | 2.188s | 86, 84, 86, 87, 84 | 85.4 | 1.34 |
| 315 | 2.498s | 86, 87, 84, 86, 85 | 85.6 | 1.14 |
| 360 | 2.912 | 86, 88, 86, 91, 87 | 87.6 | 2.07 |
| 405 | 3.226s | 88, 88, 88, 88, 88 | 88 | 0 |
| 451 | 3.752s | 89, 89, 89, 89, 89 | 89 | 0 |

**Implementation**:

Naive Bayes was implemented by first iterating over the possible labels and summing the number of times each feature (pixel coord) was 1 for that label. This was used to calculate the conditional probability that the feature is 1 for each label (P(feature=1|label)), which represented the probability a specific feature was 1 for every image with a given label. Because the feature can only be in the set {0,1}, P(feature=0|label) can be calculated using 1-P(feature=1|label). The prior distribution was also calculated over the training set, which represents what fraction of the set is made up of each label, i.e. the chance that any given image from the set has that label (P(label)).

After training, the classifier can guess what label to assign an image based on these two calculated data sets. In theory, the probability of each label can be determined using P(label)\*P(feature1=image.feature1|label)\*P(feature2=image.feature2|label)...\*P(featuren=image.featuren|label). The probabilities of each label can then be compared for the specific image, and the label that is most likely (i.e. the one that maximizes the previous function) is guessed by the algorithm. However, because multiplying the probabilities might result in underflow, the same comparison was made using the summed logs of each of the probabilities in our code.

**What was learned:**

Because Naive Bayes does not iterate over the training data multiple times, it takes a relatively small amount of time to train and evaluate conditional probability and prior distribution. The results were fairly accurate: when 100% of the training data was used, digits were able to consistently be predicted correctly 79% of the time, and faces were accurately guessed closer to 89%. Even with smaller training sets, if the distribution of features is representative enough of the test data as a whole, Naive Bayes can be reasonably successful. Because of this, while Bayes trends upwards in success rate as percentage of sample size increases, it is not guaranteed to increase from step to step, as randomly selected sets of training data vary in how well they represent the proportions of the entire set.

The standard deviation of the iterations also decreased until reaching zero, because randomizing the order of the training images does nothing to affect their probability when the same set is being chosen every time. The greater the percentage of the training set being analyzed, the more likely that the smaller set shares components with other training sets. The conclusion is that, while increasing the size of the training set generally results in more accurate results, no amount of iterating over the same training set increases the accuracy of the algorithm.

**Perceptron Data:**

*Digits*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Amount of training Data used | Avg Time to train data | Percentage accuracy on test data given 5 trials and column amount of random training data | Mean Accuracy | STD |
| 500 (10%) | 17.549 | 69,70, 68, 65,69 | 68.2 | 1.72 |
| 1000 (20%) | 38.328 | 84,78, 78, 82,73 | 79 | 3.795 |
| 1500 (30%) | 49.801 | 81, 77, 74, 70,80 | 78.8 | 3.544 |
| 2000 (40%) | 72.666 | 80, 83, 74, 74, 80 | 78.2 | 3.6 |
| 2500 (50%) | 84.277 | 87,88, 86, 84, 77 | 84.4 | 3.929 |
| 3000 (60%) | 111.817 | 86, 87, 71, 79, 80 | 80.6 | 5.748 |
| 3500 (70%) | 121.314 | 80, 75, 75, 82, 76 | 77.6 | 2.871 |
| 4000 (80%) | 139.483 | 86, 78, 83, 86, 77 | 82 | 3.847 |
| 4500 (90%) | 157.374 | 83, 83, 86, 84,73 | 81.8 | 4.534 |
| 5000 (100%) | 177.491 | 83,84, 75,82, 87 | 82.2 | 3.969 |

*Faces*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Amount of training data used | Time to train data (sec) | Percentage accuracy on test data given 5 trials and column amount of random training data | Mean accuracy (%) | STD |
| 45 (10%) | 3.7335 | 77,71, 66, 75, 65 | 70.8 | 4.750 |
| 90 (20%) | 5.9021 | 80, 83, 70, 76, 76 | 77 | 4.382 |
| 135 (30%) | 12.518 | 81, 77, 84, 82, 76 | 80 | 3.0332 |
| 180 (40%) | 12.672 | 75, 83, 75,85, 78 | 79.2 | 2.4819 |
| 225 (50%) | 16.175 | 83,85, 85, 83, 84 | 84 | 0.8944 |
| 270 (60%) | 40.860 | 80,80,83, 84, 84 | 82.2 | 1.8330 |
| 315 (70%) | 41.858 | 79, 83,84, 88, 84 | 83.6 | 2.8705 |
| 360 (80%) | 47.764 | 80, 86, 84, 82, 89 | 84.2 | 3.1241 |
| 405 (90%) | 49.058 | 91, 86, 89, 78, 88 | 86.4 | 4.4989 |
| 451 (100%) | 41.928 | 86, 91,86,82, 86 | 86.2 | 2.8566 |

**Perceptron:**

**Implementation**:

Perceptron is an algorithm that uses weights and features to score its decisions. The perceptron uses two types of weights: regular weights that are set at random values based on the training data and a bias weight that is a random arbitrary whole number value. These two types of weights are included in two different ways to the score. The regular weights are computed by getting the dot product of a list of features and their associated weights. The bias weight is then added to this score. (The bias weight does not get multiplied by a feature) This is done for each legal label in the training set (i.e. 0 and 1). From these two labels, the greatest score is taken, and this score’s label value would be the predicted score.

The predicted score is then compared with the true label of the training set. If it matches the true label you have a good predicted score and the next data point is iterated. If it does not match the true label the weights are updated. The labels can be wrong based on two ways: Either you predicted a label 0(not face/not digit), but it was really label 1(face/digit). Or you predicted label 1(face/digit), but you got label 0(not face/not digit). The regular weights and the bias weights are updated based on the true labels and the predicted labels.

The true label’s regular weights are updated by adding the features of the instance from the training data. The true labels bias weight is just updated by adding one to the weight. This is equivalent to rewarding the program to get this label right the next instance. The predicted labels regular weights are updated by subtracting the features of the instance from the training data. The predicted labels bias weight is just updated by subtracting one to the weight. Both are equivalent to scolding the program to not get this label wrong again the next instance. This is then iterated for every training data instance. Once done it can be iterated again through the same training data updating the weights and getting a better prediction with more iterations.

**What was learned:**

What I learned from perceptron is that iteration and updating is a very powerful technique to predict some label in a dataset. I also learned that picking your features as well is very important as it can make the difference from your algorithm being effective or not. One can imagine this algorithm could work for many other types of data sets not just faces or digits. With this said I can imagine one could use this to predict protein conformations as a feature could be whether certain conformations bind at a certain free energy or if they do not. Something in hindsight to is that sometimes the most basic features can be the most efficient. As well as training data size is very dependent on whether your algorithm can learn.

**Results:**

I can see from the results for faces and digits that over 50 % of data for perceptron shows much more consistent results as supposed to less than that. Perceptron seems to be also dependent on the amount of iterations done on the training data. As more iterations can give a more accurate result. There also seems to be some inconsistencies when increasing the data set. The trend should have a correlation when increasing the data set as it should lead to a more accurate prediction. This is not always the case, however.

**MIRA Data:**

In the table below it shows the accuracy of MIRA based

*Digits*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Amount of training Data used | Avg Time to train data | Percentage accuracy on test data given 5 trials and column amount of random training data | Mean Accuracy | STD |
| 500 | 023.710s | 57, 70, 70, 68, 73 | 67.6% | 6.189 |
| 1000 | 027.431s | 66, 68, 70, 66, 66 | 67.2% | 1.789 |
| 1500 | 067.090s | 64, 77, 70, 70, 77 | 71.6% | 5.505 |
| 2000 | 086.672s | 78, 75, 68, 77, 64 | 72.4% | 6.107 |
| 2500 | 107.460s | 82, 80, 75, 72, 78 | 77.4% | 3.975 |
| 3000 | 125.622s | 74, 69, 70, 80, 75 | 73.6% | 4.393 |
| 3500 | 151.248s | 73, 80, 84, 79, 77 | 78.6% | 4.037 |
| 4000 | 116.198s | 64, 84, 76, 77, 77 | 75.6% | 7.232 |
| 4500 | 197.549s | 78, 74, 83, 73, 77 | 77% | 3.937 |
| 5000 | 194.735s | 83, 78, 79, 78, 88 | 81.2% | 4.324 |

*Faces*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Amount of training data used | Time to train data | Percentage accuracy on test data given 5 trials and column amount of random training data | Mean accuracy | STD |
| 45 | 02.861s | 70, 60, 68, 70, 67 | 67% | 4.123 |
| 90 | 06.634s | 75, 63, 76, 71, 75 | 72% | 5.385 |
| 135 | 08.310s | 50, 79, 74, 74, 77 | 70.8% | 11.819 |
| 180 | 12.898s | 73, 58, 76, 57, 80 | 72.8% | 11.52 |
| 225 | 14.227s | 75, 77, 72, 79, 80 | 76.6% | 3.209 |
| 270 | 17.383s | 75, 81, 77, 79, 72 | 77.6% | 3.606 |
| 315 | 19.133s | 84, 81, 77, 71, 82 | 79% | 5.148 |
| 360 | 17.984s | 80, 80, 85, 87, 81 | 82.6% | 3.209 |
| 405 | 20.444s | 77, 84, 85, 83, 80 | 81.8% | 3.271 |
| 451 | 27.638s | 86, 86, 74, 77, 83 | 81.2% | 5.45 |

**MIRA -**

This algorithm is basically an improved upon version of perceptron where perceptron does Wy’ = Wy’ - X and Wy = Wy + X after it makes a wrong prediction, where y’ is for weights of predicted class and y is for weights of the true class. MIRA follows almost the same format except it has an additional variable T called tau and tau is a constantly adapting and learning. So, the MIRA update equation looks like this Wy’ = Wy’ - TX and Wy = Wy + TX where

T = min (C, ([(Wy’ - Wy)^2 \* x + 1] / 2x^2)) so with this T is given the smaller between the two values so that it does not over adjust the weights which is also a bad thing! With C being some constant which in my code is 0.001 so that even if the equation used wants to adjust the weights by a large margin the value C is put in place, so it does not do that.

I also implemented a way of taking the most efficient Cgrid value if autotune was enabled. By using a helper function and going through a small amount of data I would find which Cgrid value had the highest accuracy and would then set that value to self.C.

**Algorithm comparisons**

When it comes to perceptron and MIRA they are both very similar performance wise since they are more or less different versions of the same algorithm. However, even though mira is supposed to be more accurate given the random training data the resulting accuracy after running through test data proves otherwise. Perceptron would just slightly outperform MIRA in accuracy and time. I was expecting that the difference in time for training would be a lot different in the sense that perceptron would take noticeably longer to train because not only is perceptron going through X amount of training data it is going through that amount of training data 3 or so times since it does not have that learning rate to adjust the weights slightly it has to go multiple rounds in order to end up with decent weights. As for accuracy, I can see why MIRA could have been outdone since perceptron it going through all that training data multiple times while MIRA just goes through it once but does a more efficient job of adjusting the weights in that one iteration.

For the standard deviation there weren’t too many large differences between the test accuracies except for MIRA when it was training face data. This can be since the random data selected was more heavily of one type than the other which would have trained the weights understand what to do for that one type of data but less aware of how to handle the other type.

When comparing naive bayes to perceptron or MIRA it’s overall accuracy stays roughly the same not seeing drastic improvement after giving it X amount of training data unlike perceptron and MIRA which initially give off accuracies in the low 60s then steadily increase into the low 80s. Naive bayes on the other hand consistently stay in the mid to high 70s when comparing the digit data. Also, a huge difference between naive bayes and the other two algorithms is its runtime. When getting into the upper thousands of training data for MIRA and perceptron the time it takes to train the weights is drastically longer than just taking X amount of training data from the beginning and calculating a probability for each. Leading to an extremely faster performing algorithm.

To conclude, all three algorithms have their pros and cons but naive bayes definitely has a lot more pros being the fastest and most consistent overall. In the end it all depends on how well you understand the concept and come up with an implementation for it.