Erick Galvez
Erica Koplitz
Julian Romero

# 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 $\sqrt{(\sum_{i=1}^{n=5}(xi - x)^2)/ n-1}$ |
|---|---|---|---|---|
| 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, 86 | 80.4 | 3.006 |
| 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 $\sqrt{(\sum_{i=1}^{n=5}(xi - x)^2)/ n-1}$ |
|---|---|---|---|---|
| 45 | 0.39, 0.37, 0.38, 0.37, 0.38 | 53, 70, 53, 53, 52 | | |
| 90 | 0.78, 0.70, | 63, 66, 67, 63, 76 | | |

| | | | | |
|---|---|---|---|---|
| | 0.72, 0.74, 0.74 | | | |
| 135 | 1.13, 1.04, 1.11, 1.12, 1.11 | 83, 75, 83, 77, 83 | | |
| 180 | 1.54,1.47, 1.33, 1.43, 1.45 | 86, 80, 92, 84, 87 | | |
| 225 | 1.91, 1.74, 1.79, 1.81, 1.81 | 84, 90, 83, 85, 86 | | |
| 270 | 2.27, 2.28, 2.15, 2.11, 2.13 | 86, 84, 86, 87, 84 | | |
| 315 | 2.61, 2.55, 2.40 | 86, 87, 84 | | |
| 360 | 3.09 | 86% | | |
| 405 | 3.34 | 88% | | |
| 451 | 3.71 | 89% | | |

**Implementation**:
Naive Bayes was implemented by iterating over the possible labels and summing the number of times each feature was 1 for that label. This was used to calculate the conditional probability that the feature is 1:
**What was learned:**

**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 $\sqrt{(\sum_{i=1}^{n=5}(xi - x)^2)/ n-1}$ |
|---|---|---|---|---|
| 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 $\sqrt{(\sum_{i=1}^{n=5}(xi - x)^2)/ n-1}$ |
|---|---|---|---|---|
| 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(notface/notdigit), but it was really label 1(face/digit). Or you predicted label 1(face/digit), but you got label 0(notface/notdigit). 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 of these 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 $\sqrt{(\sum_{i=1}^{n=5}(xi - x)^2)/ n-1}$ |
|---|---|---|---|---|
| 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 $\sqrt{(\sum_{i=1}^{n=5}(xi - x)^2)/ n-1}$ |
|---|---|---|---|---|
| 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.

During this process I understood and learned one of the most important skills to have when going into the CS field. That skill is reading other people's code and working off that because if I were to be honest most of my time was spent looking and understanding how to use the functions and classes given to implement MIRA. The algorithm itself is not hard to understand it took watching a few youtube videos about it to grasp the concept. For the code

however, that took days of reading and putting relations together to finally understand what was happening. It's like a puzzle but once I got a lot of the pieces to fit and saw the bigger picture I realized how useful the given code was because it took care of a lot of the situations with making an image classifier.

In order to fully grasp how the code worked I basically would print out everything, see what type it was, what values it would hold, what was the key and what was the actual value and test some of my calculations on the windows terminal before I would try it on the code itself so that it would not take much time. In my code I write a detailed explanation into how I would set the bias weights and how I would initialize the weights themselves. Then how I went about calculating T or tau and explained the use of my helper functions. One would give back the total weights for each legalLabel and from there I would take the argMax() and see if my guess was correct or not. 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.

To check if I was actually adjusting the weights I would print the total weight for each legal label to see if there was a solid difference. In the end I was able to implement MIRA effectively. Now all that's left is to find the mean and standard deviation which I was able to do by making an array holding the amount of training data from 10% to 90% not accounting the 100% because that would just be inefficient since it would use all training data anyways. So I had two for loops the outer loop to go through each value in the array and the inner to run test and train five times for each given value. From there I would jot down the accuracies displayed on the test data and then just write a simpler python function to calculate the standard deviation.

In conclusion, this project was pretty stressful being that it took a long time to understand the given code, the type value for each variable since python does not show the type and with other projects, exams and quizzes looming overhead. But, even with all that I am proud of what I accomplished because through my own time and effort I finally understood the code, got the implementation down and was able to help my team in order for us to finish.

**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. However, this can be due to different coding methods used between Julian and I (Erick). 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 due to the fact that 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 we understand the concept and come up with an implementation for it.