Aaron Egbert

CS 478

**Perceptron Lab**

**1. The Perceptron Learning Algorithm**

After studying the specific requirements needed for the perceptron learning algorithm, I chose to implement a perceptron learning machine that could distinguish between multiple classes. This machine instantiates the number of perceptrons corresponding to the data’s total number of possible classification outputs.

However, before actually implementing this multi-perceptron machine, I began my work by implementing a single perceptron to use for testing and for sections 2-5. This involved taking the data inputs gathered from the ARFF files as *xi*, the ith input value, multiplying each value by a randomly generated weight and getting a specific output value as shown below:

z= *x1 w1 + x2 w2 + … +xi wi + 1\*wb*

Then, if z > 0, we interpret the output as having generated 1, or otherwise 0 is z <=0, in a binary classification system.

I wanted to make sure that one perceptron could distinguish between two classifications first, so I tested it using the data as shown in section 2. Many of my perceptron’s bugs resulted from simple arithmetic errors and using the wrong indexes in for-loops. The most challenging part of the perceptron learning algorithm was implementing the correct stopping criteria, which will be explained in greater detail in section 3.

Once linearly separable data could be distinguished with good accuracy, I supported quadric perceptrons by adding the second order multiplicative combinations as other inputs to the perceptron.

Once the single perceptron worked well enough by itself for linearly separable and non-linearly separable data sets, I expanded the perceptron learning machine into instantiating a perceptron for each classification output. This meant that I had to assign each perceptron to train on a particular class (treated as a 1 in the target output), and treat every other class as a 0. Then, after training these perceptrons, the perceptron that yielded the highest output for the data inputs would be considered the correct classification for the data. This implementation allowed for greater flexibility as well as accuracy improvement, of which I use fully in section 6.

**2. Two ARFF Files**

**Linearly Separable:**

For the linearly separable data set, I separated eight randomly generated points by a negative sloping line. This allowed me to assign each of the points a class based on what side of the line it was on. Here is the file I used for testing my single perceptron:

@RELATION test

@ATTRIBUTE x Continuous

@ATTRIBUTE y Continuous

@ATTRIBUTE class {Blue, Green}

@DATA

0.0,0.0,Blue

-0.5,0.7,Blue

-0.6,-0.5,Blue

0.4,-0.6,Blue

0.0,1.0,Green

1.0,0.0,Green

0.2,0.5,Green

0.66,-0.1,Green

%

%

**Non-linearly Separable:**

For the non-linearly separable data set, I decided to make a simple separation of the eight points with a quadratic curve. This would allow for simple testing as well as verification of the quadratic capabilities of my perceptron algorithm. The file used for non-linear testing is as follows:

@RELATION test2

@ATTRIBUTE x Continuous

@ATTRIBUTE y Continuous

@ATTRIBUTE class {Blue, Green}

@DATA

0.0,0.0,Blue

-0.3,0.3,Blue

-0.5,-0.8,Blue

0.4,-0.6,Blue

-1.0,-0.85,Green

0.0,1.0,Green

0.5,0.5,Green

0.95,-1.0,Green

%

%

**3. Training on the two ARFF Files**

One of the most difficult parts for me in implementing the perceptron learning algorithm was defining some criteria of when to stop training. The difficulty lies in training the perceptron enough so as to generalize the data set, but not so much as to memorize it. Thus, I decided to include three conditions that must be met in order for the training to stop.

First is that the algorithm must undergo at least 10 epochs. This is so as to move the weights a comfortable distance away from the original random values given it and towards its optimum values.

Second, the training will stop when the accuracy of the algorithm is improving by less than 1%. This helps the perceptron learning algorithm to keep improving until it no longer improves at a significant rate.

Lastly, the current epoch’s weight accuracy needs to be better than the previous two epochs. This condition is to prevent the algorithm from ending on weights that are worse than previously seen as well as avoiding the side-effects of over-adjusting weights.

**Linearly Separable Data**

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Set #** | **Learning Rate** | **# Epochs** | **Training Set Accuracy** |
| 1 | 0.1 | 10 | 0.875 |
| 1 | 0.1 | 10 | 0.875 |
| 1 | 0.1 | 10 | 1 |
| 1 | 0.1 | 10 | 0.75 |
| 1 | 0.1 | 10 | 1 |
| 1 | 0.2 | 10 | 1 |
| 1 | 0.2 | 10 | 1 |
| 1 | 0.2 | 10 | 0.875 |
| 1 | 0.2 | 10 | 0.875 |
| 1 | 0.2 | 10 | 1 |
| 1 | 0.5 | 10 | 0.875 |
| 1 | 0.5 | 10 | 1 |
| 1 | 0.5 | 10 | 1 |
| 1 | 0.5 | 10 | 1 |
| 1 | 0.5 | 10 | 1 |

Here, I have run my perceptron algorithm on the linearly separable data set defined in section 2. Note that the accuracy improves on average when the learning rate increases. I was surprised to find that after 10 epochs, the accuracy didn’t quite reach 100%, since I thought that the weights would converge to their optimum values. I believe that this is most likely caused by the weights being stuck in a local maximum due to the weights starting at random values, in addition to a lack of sufficient data to generalize the boundary line. In this circumstance, a larger learning rate helps move the weights closer to their optimum position quicker, as well as avoiding getting stuck in local maximums.

**Non-linearly Separable Data**

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Set #** | **Learning Rate** | **# Epochs** | **Training Set Accuracy** |
| 2 | 0.1 | 10 | 0.75 |
| 2 | 0.1 | 10 | 0.625 |
| 2 | 0.1 | 10 | 0.75 |
| 2 | 0.1 | 10 | 1 |
| 2 | 0.1 | 10 | 0.875 |
| 2 | 0.2 | 10 | 0.75 |
| 2 | 0.2 | 10 | 0.85 |
| 2 | 0.2 | 10 | 1 |
| 2 | 0.2 | 10 | 1 |
| 2 | 0.2 | 10 | 0.875 |
| 2 | 0.5 | 10 | 0.875 |
| 2 | 0.5 | 10 | 1 |
| 2 | 0.5 | 10 | 1 |
| 2 | 0.5 | 10 | 1 |
| 2 | 0.5 | 10 | 0.875 |

After running my perceptron algorithm on a non-linearly separable data set, I noticed a slight improvement in the training set accuracy when the learning rate increases. Similarly to the linearly separable data results, in this data set, the larger learning rate moved the weights more quickly towards their optimum values, however, in this circumstance it seemed to have made it more difficult to get a 100% accuracy due to the large weight changes.

**4. Decision Lines after Training**

**Linearly Separable Data**

From running the perceptron learning algorithm on the linearly separable data, I found that line came out to intercept the y-axis at 0.15 and the x-axis at 0.1. In order to make the line easier to see on the graph, I constructed a larger line that represented the algorithm’s decision line boundary. It’s interesting to note here that the boundary line is closer to the blue dots than the green dots. This is due to the “human-like” nature of the perceptron trying to “learn” the best line that can divide the two data sets.

**Non-linearly Separable Data**

From running the quadric version of the perceptron learning algorithm, I discovered the boundary curve by finding points on the graph that yielded 0 as an output in the perceptron. Interestingly, this boundary curve is pretty accurate.

**5. Learning the Voting Task**

|  |  |  |  |
| --- | --- | --- | --- |
| **Training Iteration** | **# Epochs** | **Training Set Accuracy** | **Test Set Accuracy** |
| 1 | 10 | 0.99068 | 0.92086 |
| 2 | 13 | 0.99379 | 0.93525 |
| 3 | 10 | 0.98136 | 0.94245 |
| 4 | 15 | 0.99379 | 0.90647 |
| 5 | 11 | 0.99068 | 0.94245 |
| Average | 11.8 | 0.99006 | 0.929496 |

In the table above, notice how similar the accuracies turned out to be for the voting task. It’s remarkable to see how having lots of data, in combination with multiple epochs, can yield 90+% accurate results.

As I was recording the data for the voting tasks, I found out that despite the randomness of the initial weights and the order of the data being trained on, the model yielded similar weights for each input to the perceptron. This means that the model learned what inputs were most important in yielding the desired output. For example, when the weights provide a false positive net output, the corresponding inputs’ weights will be decreased, and a false negative output will increase the corresponding weights values. This is the equation that changes these weights:

∆*wi = c(t – z) xi*

*wi* – the ith weight, corresponding to the ith input value

c – Learning Rate

t – target output (binary 1 or 0)

z – net output (z= *x1 w1 + x2 w2 + … +xi wi)*

*xi* – the ith input value

The most important feature for the voting task is this weight updating equation. Without it, there is no way in which the perceptron machine could “learn” and get consistent accuracies in the 90+% range. On the other hand, the least critical feature for the voting task would most likely be the randomization of the initial values for the weights. Since there is so much data and so many epochs being run on the voting task, the starting position of the weights should be insignificant to the output of where the weights end up to be validated.

In looking at this chart, you can see that the more epochs that are applied to this dataset, the better the classification rate becomes. However, with this dataset, this trend levels off to less than 1% accuracy improvement after 4 epochs, so this is another reason as to why I have the perceptron algorithm perform at least 10 epochs of training.

**6. Learning the Iris Dataset**

For my own experiment with perceptrons, I decided to apply my algorithm to the Iris dataset, in order to successfully classify between 3 output values, instead of a binary output. As I explained in section 1, after extensive work on building and testing a multi-perceptron learning algorithm, one that could instantiate a perceptron for every output class, I ran this new algorithm and came up with these results:

|  |  |  |
| --- | --- | --- |
| **Instance** | **# Epochs** | **Training Set Accuracy** |
| 1 | 38 | 0.94667 |
| 2 | 12 | 0.95334 |
| 3 | 17 | 0.66666 |
| 4 | 44 | 0.9533 |
| 5 | 10 | 0.8533 |

What surprised me most about these results was the variance in how many epochs it took for the perceptron machine to stop. I looked closer at many of these iterations to find that most of these epochs gained slight, but consistent improvements in the training set accuracy. Despite the learning rate being only 0.1, the stopping criteria to wait for the accuracy to improve by less than 1% proved to help the algorithm provide better results.