Aaron Egbert

CS 478

**Perceptron Lab**

**1. The Perceptron Learning Algorithm**

To implement the perceptron learning algorithm, I first studied the specific requirements needed for its implementation. Since I needed to generate a perceptron that could distinguish between multiple classes, I decided to build a perceptron learning machine that would instantiate the number of perceptrons corresponding to the data’s total number of possible classification outputs.

However, before actually using the multiple perceptrons, I wanted to make sure that one perceptron could distinguish between two classifications first. Many of the 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, I expanded the perceptron learning machine to build 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:

@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:

@RELATION test2

|  |  |  |  |
| --- | --- | --- | --- |
| **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 |

@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 of perceptron learning is to define some criteria of when to stop training. One of the

most important principles surrounding this is to train the algorithm enough so as to generalize the data set, but not so much as to memorize it. Thus, I have made it so that the training algorithm will stop when it has undergone at least 10 epochs AND when the accuracy of the algorithm is improving by less than 1% AND is at least better than the previous two epochs. The first 10 epochs in the training phase are automatically run so as to move the weights a comfortable distance away from the original random values given it and towards its optimum values. This second condition helps the perceptron learning algorithm to keep improving until it no longer improves at a significant rate. The last 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 weight

**Linearly Separable Data**

In the Linearly Separable Data, we see 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 the optimum weights. 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 |

In the Non-linearly Separable Data, we see a slight improvement in the training set accuracy when the learning rate increases. In this data set, the larger learning rate moved the weights more quickly towards the optimum position, but it also makes 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, the line came out to intercept the y-axis at 0.15 and the x-axis at 0.1. From that, I was able to construct a line, representing the algorithm’s decision line boundary.

**Non-linearly Separable Data**

From running the quadric perceptron learning algorithm, the points on the graph that yielded 0 to the output represent the boundary curve.

**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 |

By looking at the weights, the model has learned what inputs are 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 the 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 the weight updating equation. Without it, there is no way in which the perceptron machine could “learn” and get 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, 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 a 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 testing to make sure that the perceptrons could pick up each of their corresponding outputs, I 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.