2)

The activation function is of the form:

By setting the weights as follows:

We get the following activations based on the values of A and B:

|  |  |  |
| --- | --- | --- |
| A | B |  |
| 0 | 0 | 0 |
| 0 | 1 | -1 |
| **1** | **0** | **1** |
| 1 | 1 | 0 |

We need to select a threshold such that only the third activation above (bolded) exceeds it. We can do that by setting it as:

The XOR operation can be broken up as follows:

We can set the weights for:

As:

We need to select a threshold such that +1 is a high enough activation to trigger the perceptron but 0 isn’t. We can use:

The second perceptron will take the output of the first (called C) as well as A and B:

By setting the weights as:

We need a threshold that allows activation +0.5 to trigger the perceptron but 0 cannot, we can choose:

3)

We can first simply the output equation as follows:

The summation can also swallow up the zero weight if we can find a zero x that satisfies:

We can apply the quadratic equation:

Since we know a valid zero x exists, the output equation can be further simplified to:

Which leads to the rule:

For gradient descent, the error is defined by:

To minimize the error, we can calculate the partial derivative in with respect to weights:

By negating this equation to clean up negatives, the weight update is then:

Giving the full gradient descent training rule as:

The main benefit of the gradient descent rule over the perceptron rule is that it can manage data that isn’t linearly separable.

4)

To perform regression, instead of voting at the leaf nodes during construction, the mean can be taken. For example, if a leaf holds 10 samples of the label 50 and 100 samples of the label 25, the answer could be 27.2727… rather than 100.

For any order polynomial, the equation is:

The mean squared error is given by:

We can solve for the partial derivative with respect to the constants by:

The leaf nodes will try to minimize the error and so we need to set the equation to zero and solve for the function being sought:

This is equal to the mean.

5)

The decision tree algorithm can be made lazy by delaying tree construction until testing time. At training time, the algorithm would simply record the instance vectors. When a testing sample is provided, the algorithm can fetch training instances that more closely match the testing vector in terms of attribute values. These training examples would be used to construct the tree.

The advantage of this approach is that the processing during training time is minimal. If we have lots of training vectors and very few testing samples, this approach may be appropriate. However, the downside is that processing is left up to the testing phase and the algorithm could have to construct a different tree for each testing sample. The more common decision tree algorithm only needs to construct one tree.

6)

Unfortunately, neither KNN and decision trees are the best choices for linearly separable data. If the option were available, I would choose an SVM with a linear kernel. Between the two selections, I would choose the decision tree algorithm because the splitting technique could still create perfectly separable leaf nodes. The KNN algorithm may also do the same but there’s the chance of a nearest neighbour being closest on the other side of the line. If it is known beforehand that most points are not close to the line, I would choose KNN. If performance were a concern in terms of training time, I would also consider KNN more strongly.