Perceptron

Algorithm to develop a perceptron:

Initialize the weights and bias to random values. The number of weights should be equal to the number of input features.

For each training example, calculate the weighted sum of the inputs and bias. This can be represented as:

weighted_sum = (weight1 * input1) + (weight2 * input2) + ... + (weightn * inputn) + bias

Apply the activation function to the weighted sum. The most common activation function used with perceptrons is the step function, which outputs 1 if the weighted sum is greater than or equal to zero, and 0 otherwise.

Compare the predicted output to the true output for the training example. If the prediction is incorrect, adjust the weights and bias using the perceptron learning rule, which is:

new_weight = old_weight + learning_rate * (true_output - predicted_output) * input

new_bias = old_bias + learning_rate * (true_output - predicted_output)

Repeat steps for each training example for a specified number of epochs or until the algorithm converges.

Model: Y=w1*x1 + w2*x2 + b

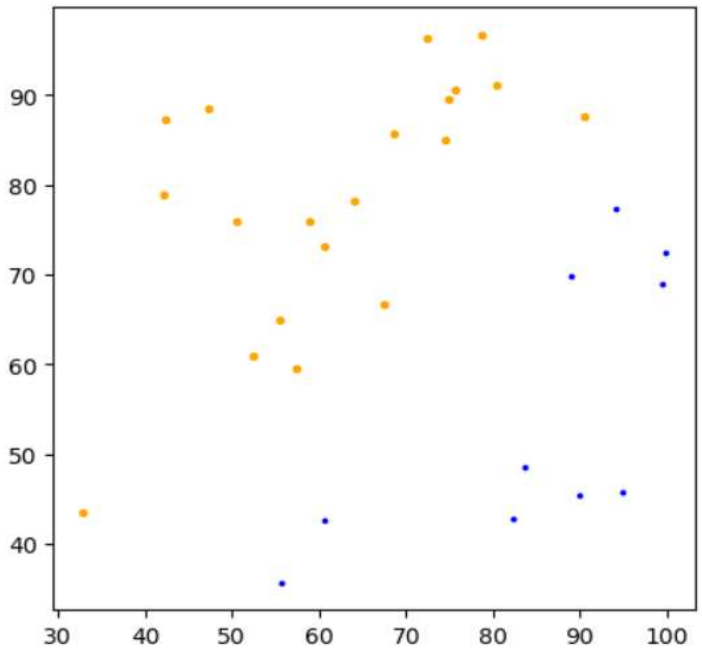
Initialise: w1=-20, w2=-10, b=10 , $\alpha = 0.0001$(these all are selected by hit and trail method so that optimum solution obtain).

If Y>0, update the weights w1 & w2

w1=w1+ α*Y(actual)*x1

w2=w2 + α*Y(actual)*x2

Using the perceptron following graph is obtain on test data after training:



Orange Points Indicates: Test Pass

Blue Points Indicates: Test Fail

Multi-Layer Perceptron

Initialize the weights and biases for each layer to random values. The input layer has no biases and the number of weights is equal to the number of input features.

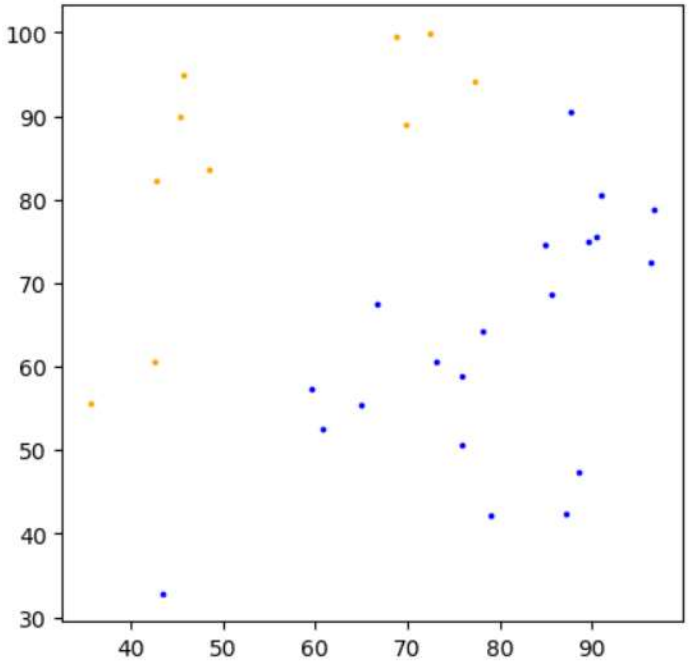
For each training example, forward propagate the inputs through the network. This involves calculating the weighted sum of the inputs and biases for each neuron in each layer, and applying an activation function to the result. The most commonly used activation function is the ReLU (rectified linear unit) function.

Calculate the error between the predicted output and the true output for the training example.

Backpropagate the error through the network to update the weights and biases. This involves calculating the derivative of the activation function, multiplying it by the error and the input for each neuron, and updating the weights and biases using a learning rate. This is repeated for each layer in reverse order.

Repeat steps for each training example for a specified number of epochs or until the algorithm converges.

Below graph is obtained on test data:



Orange Points Indicates: Test Pass

Blue Points Indicates: Test Fail

Logistic Regression

Initialize the weights w to random values

Repeat until convergence:

a. Calculate the predicted output y_hat for each example using the current weights w:

y_hat_i = sigmoid(w^T x_i)

where x_i is the input feature vector for example i, and sigmoid is the logistic function

b. Calculate the gradient of the logistic loss function with respect to the weights w:

grad_w = (1/n) * sum_i (y_hat_i - y_i) * x_i

where y_i is the true output variable for example i

c. Update the weights using gradient descent:

w = w - alpha * grad_w

Return the final set of weights w

Note that the logistic loss function is given by:

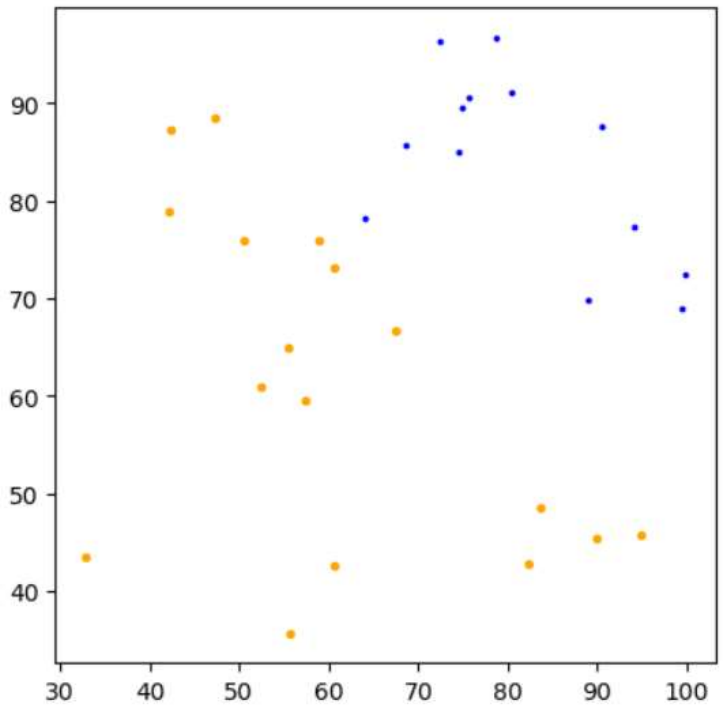
L(w) = - (1/n) * sum_i (y_i log(y_hat_i) + (1-y_i) log(1-y_hat_i))

where y_i is the true output variable for example i, and y_hat_i is the predicted output variable for example i.

The logistic function (sigmoid) is given by:

sigmoid(z) = 1 / (1 + exp(-z))

where z is a real-valued input.



Graph obtained on test data on applying logistic regression.

Orange Points Indicates: Test Pass

Blue Points Indicates: Test Fail