University of Nevada, Reno

Department of Computer Science and Engineering

CS 691: Machine Learning

Project 2

Cayler Miley

Instructor: Emily Hand

October 17, 2018

1 Perceptron

1.1 perceptron_train(X, Y)

The `perceptron_train` function computes a weight vector and bias using the given samples (X) and their corresponding labels (Y). For debugging purposes, an optional flag is defaulted to False to suppress printing. The initial weights and bias are set to zero and an epoch iteration limit of 1000 is set (arbitrarily chosen). Next an initial "stale" weight vector and bias are set to determine if the weight and bias vector change during the epoch. Next for each sample in the sample set, the weight vector and bias is computed. The computation is completed using the `update_weights` function which returns the same `w,b` if the computed activation is correct and recomputes `w,b` if it is incorrect. Since this is a helper function for the `perceptron_train` function, it is not included in the write-up. If the weights have not changed for an entire epoch, then the loop breaks and the final `w,b` are returned. The loop also stops if the epoch limit is reached, to guard against data that is not linearly separable. The following is a line-by-line copy of the function in Python:

```
1  def perceptron_train(X, Y, FLAG_debug=False):
2      # initialize weights and bias to zero
3      w = np.zeros(X[0].size)
4      b = 0
5
6      # set limit for non-linearly separable data
7      max_epochs = 1000
8
9      w_stale = np.zeros(X[0].size)
10     b_stale = 0
11
12     for epoch in range(max_epochs):
13         w_update = []
14         for index, sample in enumerate(X):
15             w, b = update_weights(sample, Y[index], w, b)
16
17             # append to list of T/F for weight and bias changes
18             if np.array_equal(w_stale, w) and b_stale == b:
19                 w_update.append(True)
20             else:
21                 w_update.append(False)
22             w_stale = w
23             b_stale = b
24
25         # if weights have not changed for an entire epoch
26         if np.all(w_update):
27             if FLAG_debug:
28                 print("Converged at epoch: {}".format(epoch))
29
30             # break and return
31             break
32         else:
33             if FLAG_debug:
34                 print("Not converged at epoch: {}".format(epoch))
35
36     if FLAG_debug:
37         print("W: {}, B: {}".format(w, b))
38
39     return [w, b]
```

1.2 `perceptron_test(X_test, Y_test, w, b)`

The `perceptron_test` computes the accuracy of the weight vector and bias given on the sample set (X) and its corresponding labels (Y). The function simply computes the activation for each sample and compares it with the label. The correct counter is implemented for each correct activation and then divided by the length of the set X to compute the accuracy. The accuracy is then returned. The following is a line-by-line copy of the function in Python:

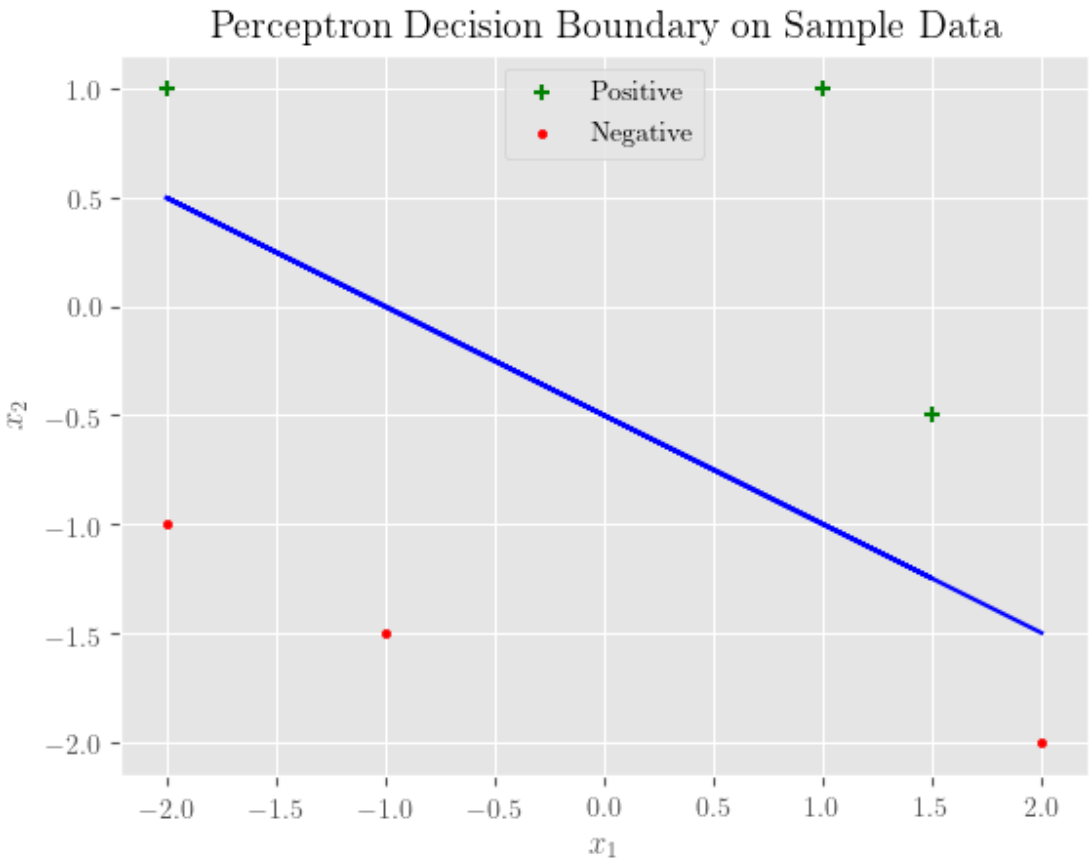
```
def perceptron_test(X_test, Y_test, w, b, FLAG_debug=False):
    correct = 0

    # for each sample in the test set, compare the prediction with the label
    for index, sample in enumerate(X_test):
        activation = np.dot(sample, w) + b
        if activation * Y_test[index] > 0:
            correct += 1

    accuracy = correct / len(X_test)

    if FLAG_debug:
        print("Number correct: {} out of {}".format(correct, len(X_test)))
        print("Testing accuracy: {}".format(accuracy))

    return accuracy
```

1.3 Experimental Results

The `perceptron_train` function outputs `[[2., 4.], 2]` on the following data:

Sample	x_1	x_2	y
1	-2	1	1
2	1	1	1
3	1.5	-0.5	1
4	-2	-1	-1
5	-1	-1.5	-1
6	2	-2	-1

The data and the decision boundary linearly separating the data is as follows:



2 Gradient Descent

2.1 `gradient_descent(grad, x_init, eta)`

The `gradient_descent` function finds the minimum value of a function given its gradient, initial value, and a step size (η). The function is based off of the iterative process of computing the gradient and subtracting a fraction of the gradient from the current minimum, until the gradient is approximately zero (noting an absolute minimum for a convex function). The equation is: $x_i = x_{i-1} - \eta \nabla f(x_{i-1})$ where $x_0 = x_{init}$ and $i = 1, 2, \ldots, n$ such that $||\nabla f(x_n)|| \approx 0$. When the L_2 norm of the gradient is zero, the function is at a minimum at the point x_n. The function itself simply loops while the gradient at the current minimum is greater than a threshold and continuously recomputes the minimum. The function assumes a sufficient step size such that the function will not diverge. The following is a line-by-line copy of the function in Python:

```
1  def gradient_descent(grad, x_init, eta):
2      # use stopping point for gradient value sufficiently close to 0
3      threshold = 1e-5
4
5      # use the initial values as the minimum before descending
6      minimum = x_init
7
8      # while the gradient is effectively non-zero
```

```
    while np.linalg.norm(grad(minimum)) > threshold:
        # min = min - eta*grad
        minimum = np.subtract(minimum, np.multiply(eta, grad(minimum)))

    return minimum
```

2.2 Experimental Results

Given the test function:

$$f(x) = x_1^2 + x_2^2$$

the gradient is computed as:

$$\nabla f(x) =< 2x_1, 2x_2 >$$

When `gradient_descent` is run on the gradient using $\nabla f(x) =< 2x_1, 2x_2 >$, $x_{init} = (5.0, 5.0)$, and $\eta = 0.1$ the output is `[3.13855087e-06 3.13855087e-06]` which is effectively `[0 0]`