Implementing the Perceptron algorithm for finding the weights of a Linear Discriminant function

Mayeesha Humaira
dept. Computer Science and Engineering
Ahsanullah University of Science and Technology
Dhaka, Bangladesh
160204008@aust.edu
section - A1

***Abstract*—The main aim of this experiment is to utilize the Perceptron algorithm to take data points to higher dimension and hence finding ideal weights of a Linear Discriminant Function.**

***Index Terms*—Perceptron, many at a time, one at a time, weight**

I. Introduction

Data points that can not be separated in lower dimension can be separated in higher dimension. The perceptron algorithm which is a linear classification algorithm that takes data points to higher dimension to separate data points of two classes. This algorithm minimizes the misclassified samples by updating weights. As weights are being updated the orientation of the hyperplane keeps on changing until all the training samples are correctly classified.

II. Experimental Design / Methodology

A dataset named train-perceptron.txt was provided which contain data points of two classes. Data in the dataset are shown in Table I. The following task were performed on this dataset.

TABLE I
Data points in the train-perceptron.txt dataset.

x1	x2	class
1	1	1
1	-1	1
2	2.5	2
0	2	2
2	3	2
4	5	1

1) **Task-1:**
Take input from "train-perceptron.txt" file. Plot all sample points from both classes, but samples from the same class should have the same color and marker. Observe if these two classes can be separated with a linear boundary.

Solution:-
Data points were plotted using matplotlib as shown in Fig 1. Here the red circles represent data points of class-1 and the the blue star marks represent data points of class-2.

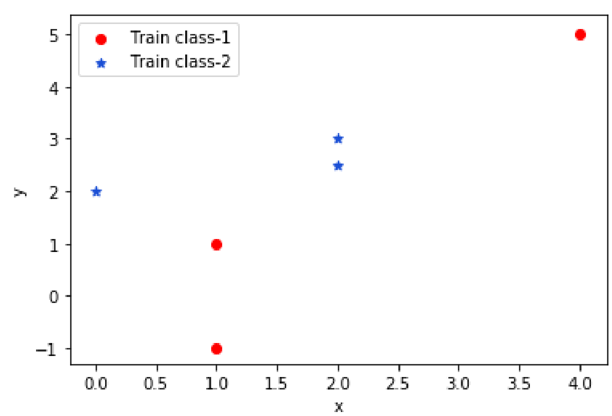

Fig. 1. Graphical representation of training data using matplotlib.

2) **Task-2:**
Consider the case of a second order polynomial discriminant function. Generate the high dimensional sample points y, as discussed in the class. We shall use the following formula:

$$y = \begin{bmatrix} x_1^2 & x_2^2 & x_1 * x_2 & x_1 & x_2 & 1 \end{bmatrix} \qquad (1)$$

Also, normalize any one of the two classes.

Solution:- These data points cannot be separated in 2D which can be seen from Fig 1. As a result, I took them to higher dimension using the Eq 1. this equation takes these data point to 6D. Furthermore, these data points were normalized by negating 6D data of class-2. The resulting 6*6 matrix of both classes are shown in Eq 2. There first three rows represent 6D data of class-1 and the last three rows represent 6D data of class-2.

$$y = \begin{bmatrix} 1. & 1. & 1. & 1. & 1. & 1. \\ 1 & 1. & -1. & 1. & -1. & 1. \\ 16. & 25. & 20. & 4. & 5. & 1. \\ -4. & -6.25 & -5. & -2. & -2.5 & -1. \\ -0. & -4. & -0. & -0. & -2. & -1. \\ -4. & -9. & -6. & -2. & -3. & -1. \end{bmatrix} \quad (2)$$

3) **Task-3:**
Use Perceptron Algorithm (both one at a time and many at a time) for finding the weight coefficients of the discriminant function (i.e., values of w) boundary for your linear classifier in task 2. Here α is the learning rate and $0 < \alpha \leq 1$.

$\underline{w}(i+1) = w(i) + \alpha \underline{\tilde{y}}_m^k$ **if** $\underline{w}^T(i)\underline{\tilde{y}}_m^k \leq 0$

(**i.e, if** $\underline{\tilde{y}}_m^k$ **is misclassified**)

= $\underline{w}$(i) **if** $\underline{\tilde{y}}_m^k > 0$

Solution:- Perceptron algorithm **One at a time** has the folowing equation for finding the weight coefficients of the discriminant function.

$$w(t+1) = w(t) + \eta y \quad (3)$$

where w(t) is the weight of current input, w(t+1) is the weight of the next input, η is the learning rate and y is the sample that is misclassified.
Perceptron algorithm **Many at a time** has the following equation for finding the weight coefficients of the discriminant function.

$$w(t+1) = w(t) + \eta \sum_{misclassified} y \quad (4)$$

where w(t) is the weight of current input, w(t+1) is the weight of the next input, η is the learning rate and $\sum y$ are the samples that are misclassified.

4) **Task-4:**
Three initial weights have to be used (all one, all zero, randomly initialized with seed fixed). For all of these three cases vary the learning rate between 0.1 and 1 with step size 0.1. Create a table which should contain your learning rate, number of iterations for one at a time and batch Perceptron for all of the three initial weights. You also have to create a bar chart visualizing your table data.

Solution:-
Three initial weights each of dimension 6 were initialized. First set of weights were six ones second set of weights were six zeros and the third set of weights were random values having seed equal to 10. All three sets of weights are shown below.

$$\begin{bmatrix} [1. & 1. & 1. & 1. & 1. & 1.] \\ [0. & 0. & 0. & 0. & 0. & 0.] \\ [0.771 & 0.0207 & 0.633 & 0.748 & 0.498 & 0.224] \end{bmatrix}$$

The learning rate was varied from 0.1 to 1 having step size of 0.1. As a result, 10 different learning rates $[0.1 \;\; 0.2 \;\; 0.3 \;\; 0.4 \;\; 0.5 \;\; 0.6 \;\; 0.7 \;\; 0.8 \;\; 0.9 \;\; 1.0]$ were achieved. These setup were taken for both one at a time and many at a time and the iteration at which there were no misclassifed samples were saved.

III. Result Analysis

After implementing the Perceptron algorithm following bar charts and tables were obtained.

TABLE II
Number of iterations required for the Perceptron algorithm to correctly classify all data points when weights are initialized to all ones for both One at a time and Many at a time having different learning rates.

alpha(learning rate)	One at a Time	Many at a Time
0.1	6.	102.
0.2	92.	104.
0.3	104.	91.
0.4	106.	116.
0.5	93.	105.
0.6	93.	114.
0.7	108.	91.
0.8	115.	91.
0.9	94.	105.
1.0	94.	93.

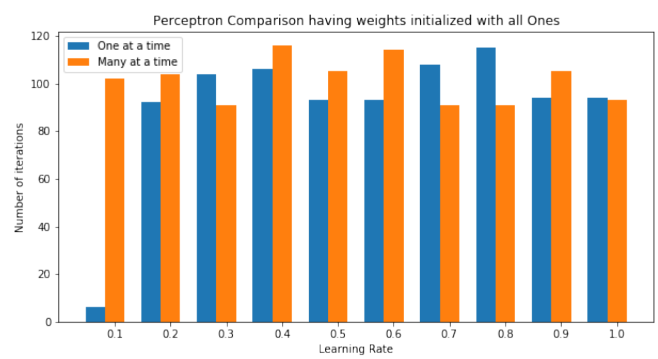

Fig. 2. Graphical representation of the number of iterations required for the Perceptron algorithm to correctly classify all data points when weights are initialized to all ones for both One at a time and Many at a time having different learning rates. using matplotlib.

IV. Question Answering

1) In task 2, why do we need to take the sample points to a high dimension?
Ans: I took sample points to higher dimension as these data points are not linearly separable in lower dimension.

TABLE III
NUMBER OF ITERATIONS REQUIRED FOR THE PERCEPTRON ALGORITHM TO CORRECTLY CLASSIFY ALL DATA POINTS WHEN WEIGHTS ARE INITIALIZED TO ALL ZEROS FOR BOTH ONE AT A TIME AND MANY AT A TIME HAVING DIFFERENT LEARNING RATES.

alpha(learning rate)	One at a Time	Many at a Time
0.1	94.	105.
0.2	94.	105.
0.3	94.	105.
0.4	94.	105.
0.5	94.	92.
0.6	94.	105.
0.7	94.	92.
0.8	94.	105.
0.9	94.	105.
1.0	94.	92.

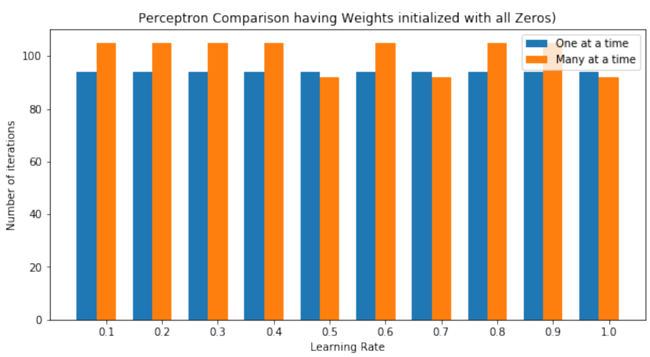

Fig. 3. Graphical representation of the number of iterations required for the Perceptron algorithm to correctly classify all data points when weights are initialized to all zeros for both One at a time and Many at a time having different learning rates. using matplotlib.

TABLE IV
NUMBER OF ITERATIONS REQUIRED FOR THE PERCEPTRON ALGORITHM TO CORRECTLY CLASSIFY ALL DATA POINTS WHEN WEIGHTS ARE INITIALIZED TO ALL RANDOM VALUES FOR BOTH ONE AT A TIME AND MANY AT A TIME HAVING DIFFERENT LEARNING RATES.

alpha(learning rate)	One at a Time	Many at a Time
0.1	97.	84.
0.2	95.	91.
0.3	93.	117.
0.4	101.	133.
0.5	106.	90.
0.6	113.	105.
0.7	94.	88.
0.8	113.	138.
0.9	108.	138.
1.0	101.	150.

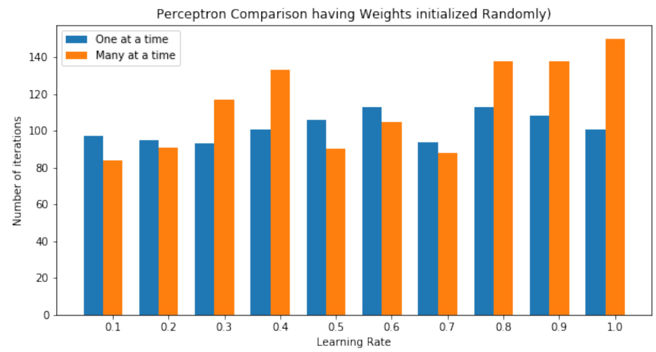

Fig. 4. Graphical representation of the number of iterations required for the Perceptron algorithm to correctly classify all data points when weights are initialized to all random values for both One at a time and Many at a time having different learning rates. using matplotlib.

Points that are not separable in lower dimensions are separable in higher dimensions.

2) In each of the three initial weight cases and for each learning rate, how many updates does the algorithm take before converging?
Ans: Table II, Table III and Table IV shows the number of iterations the algorithm takes before it converges for weights initialized to ones, zeros and random values respectively. Number of iterations represent number of updates required.

V. CONCLUSION

Perceptron algorithm classifies different classes using a linear discriminant function.In this experiment I implemented perceptron algorithm by taking data points to higher dimension where two classes can be separated linearly. The maximum number of iteration this algorithm took to converge was 150 for Many at a time and 115 for One at a time. As result it can be said that One at a time learns faster than Many at a time.

VI. ALGORITHM IMPLEMENTATION / CODE

```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

#Plotting training data
x,y,z = np.loadtxt('train-perceptron.txt',unpack=
    True, delimiter=' ')
plt.xlabel('x')
plt.ylabel('y')
for m in range(len(z)):
    if z[m]==1:
        xc1=plt.scatter(x[m], y[m], color='r')
    elif z[m]==2:
        xc2=plt.scatter(x[m], y[m], marker='*',
    color='#184DD5')

plt.legend([xc1, xc2], ["Train class-1", "Train
    class-2"])
plt.show()

print('
    --------------------------------------------------
    ')
#two classes
```

```python
xcl1=[]
ycl1=[]
xcl2=[]
ycl2=[]

for m in range(len(z)):
    if z[m]==1:
        xcl1.extend([x[m]])
        ycl1.extend([y[m]])
    elif z[m]==2:
        xcl2.extend([x[m]])
        ycl2.extend([y[m]])
print(xcl1,ycl1)
print(xcl2,ycl2)



print('-------------------------------------------------')
#yd (6,6)
yd = np.zeros((len(y),6))
print(yd,  yd.shape, yd.ndim)
print('-------------------------------------------------')


for i in range(len(ycl1)):
    yd[i][0]=np.power(xcl1[i],2)
    yd[i][1]=np.power(ycl1[i],2)
    yd[i][2]=xcl1[i]*ycl1[i]
    yd[i][3]=xcl1[i]
    yd[i][4]=ycl1[i]
    yd[i][5]=1

#negating class 2
for i in range(len(ycl2)):
    yd[len(xcl1)+i][0]=-(np.power(xcl2[i],2))
    yd[len(xcl1)+i][1]=-(np.power(ycl2[i],2))
    yd[len(xcl1)+i][2]=-(xcl2[i]*ycl2[i])
    yd[len(xcl1)+i][3]=-xcl2[i]
    yd[len(xcl1)+i][4]=-ycl2[i]
    yd[len(xcl1)+i][5]=-1

print(yd)
print('-------------------------------------------------')



weight = [[0]*6 for i in range(3)]

weight[0]=np.ones(6)
weight[1]=np.zeros(6)
np.random.seed(10)
weight[2] = np.random.random((6))

weight=np.array(weight)
print(weight)
print('-------------------------------------------------')

#update
for k in range(3):
    cnt=0
    table = np.zeros(shape=(10,3))
    for alpha in np.arange(1,11,1)/10:
        table[cnt][0]=alpha
        w = weight[k]
        iteration=0

        #single update
        for j in range(500):
            size=0
            for i in range(len(z)):
                g=np.dot(yd[i,:],w.T)
                if g<=0:
                    w=w+alpha*yd[i,:]
                else:
                    size=size+1
            if(size==len(z)):
                iteration=j+1
                break
        table[cnt][1]=iteration

        w = weight[k]
        iteration=0
        wtemp=0

        #batch update
        for j in range(500):
            size=0
            wtemp=0
            for i in range(len(z)):
                g=np.dot(yd[i,:],w.T)
                if g<=0:
                    wtemp=wtemp+yd[i,:]
                else:
                    size=size+1
            if(size==len(z)):
                iteration=j+1
                break
            w=w+alpha*wtemp
        table[cnt][2]=iteration
        cnt=cnt+1

    print(np.array_str(table, suppress_small=True))

    f, ax = plt.subplots()
    f.set_figheight(5)
    f.set_figwidth(10)
    index = np.arange(10)
    bar_width = 0.35

    if k==0:
        plt.title('Perceptron Comparison having weights initialized with all Ones')
    elif k==1:
        plt.title('Perceptron Comparison having Weights initialized with all Zeros)')
    else:
        plt.title('Perceptron Comparison having Weights initialized Randomly)')

    plt.bar(index, table[:,1], bar_width,label='One at a time')
    plt.bar(index + bar_width, table[:,2], bar_width, label='Many at a time')
    plt.xlabel('Learning Rate')
    plt.ylabel('Number of iterations')
    plt.xticks(index + bar_width, table[:,0])
    plt.legend()
    plt.show()
```