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

Anika Tanzim
Dept. Computer Science and Engineering
Ahsanullah University of Science and Technology
Dhaka, Bangladesh
160204072@aust.edu

***Abstract*—The objective of this experiment is to implement the Perceptron algorithm for finding the weights of a Linear Discriminant function.The Perceptron algorithm is a two-class (binary) classification machine learning algorithm. This means that it learns a decision boundary that separates two classes using a line (called a hyperplane) in the feature space. We are implementing two types of Perceptrons (Single update and Batch update) using different weight vectors and varying the hyperparameter (learning rate). By performing the train data, we will compare the results of the two types of updates.**

***Index Terms*—Perceptron algorithm, learning rate, hyperparameter, Ssingle update, batch update.**

I. Introduction

The Perceptron is a linear classification algorithm. This means that it learns a decision boundary that separates two classes using a line (called a hyperplane) in the feature space. As such, it is appropriate for those problems where the classes can be separated well by a line or linear model, referred to as linearly separable. But if the data are not linearly separable, ϕ function is used to take the data in higher dimension. By taking in higher dimension, perceptron algorithm can be applied on them.

There are two type of process to update the weights by perceptron algorithm: single update (one at a time) and batch update (many at a time). The discriminant function i.e. used here,

$$\overline{w}(i+1) = \overline{w}(i) + \alpha\, \overline{y_m}^k \qquad \text{if } \overline{w}^T(i)\overline{y_m}^k <= 0$$

$$\text{i.e. if } \overline{y_m}^k \text{ missclasified.}$$

$$\overline{w}(i+1) = \overline{w}(i) \qquad \text{if } \overline{w}^T(i)\, \overline{y_m}^k > 0$$

Here, w is the weight vector, y is the feature vector, α is the learning rate where, $0 < \alpha <= 1$ ans T denotes the transpose of a vector. We need to update the weight until all data are correctly classified.

II. Experimental Design / Methodology

A. *Plotting and observing the train dataset*

A train dataset is given. All the points are classified into two classes - Class 1 and Class 2. Using Matplotlib, all the points from Class 1 are plotted using blue dot marker and the points from Class 2 are plotted using green star marker in figure 1.

After observing the figure, it is said that no decision boundary can be drawn to distinguish between the classes. The data are not linearly separable. And perceptron algorithm cannot be applied in non-linear data. So ϕ function is used to get a higher dimension of sample points.

B. *Generating the high dimensional sample points*

Considering the case of a second order polynomial discriminant function, the high dimensional sample points y is generated. To generate, the following formula is used:

$$[\ x_1^2\ \ x_2^2\ \ x_1 * x_2\ \ x_1\ \ x_2\ \ 1\ ]$$

By using this ϕ function, sample points go to the six dimensional space. After that one of the two classes needs to be negated to normalize data. It is a process to take one class complete to the opposite and this is why if $\overline{w}^T(i)\ \overline{y_m}^k <= 0$ it is misclassified. Here, Class 2 is negated. Now perceptron algorithm can be applied on these data.

C. *Using Perceptron Algorithm (both one at a time and many at a time) for finding the weight coefficients*

Now, the weights that helps to classy all the data, are calculated for both batch update ad single update using different learning rate from 0.1 to 1 with step size 0.1.

While single update is used, one point at a time is evaluated. When one point is misclassified, the weights are getting updated for next point. Thus, until all the points are classified correctly, the process is iterated.

Again, While batch update is used, all the points are evaluated together. That is, for one set of weight values, all the points are evaluated first. And then, if there is/are misclassified data, the summation of y are taken and weights get updated. Thus, until all the points are classified correctly, the process is iterated.

Thus we will get a set of weight values for which all the point will give positive values for each updating process. According to the algorithm this set of values is a valid weight values. So we can draw decision boundary with the weight values.

D. *Generating results and bar-chart for different weights and learning rates*

Now, for different initial weights (all-zero, all-one, randomly-initialized) and for for different learning rate, two updating process are evaluated. The comparison tables and bar-charts are generated for each initial weight vector. The results are analyzed hereafter.

E. *Answering the questions*

1) In task 2, we need to take the sample points to a high dimension because the given data were not linearly separable. We know, when the data are not linearly separable, the decision boundary can not drawn in a way that it can distinguish all the data properly. Again, perctron algorithm can work on non-linear data. To use perceptron on these data, at first the sample points are needed to take in the higher dimension.So, ϕ function is used to get the high dimension. And so it will be easier to have the appropriate weight that willl help to classify all the points.
2) In each of the three initial weight cases and for each learning rate, number of updates, the algorithm takes before converging is shown in the tables and bar-charts. And from that we can see, for each inital weight vector, in most of the cases, one at a time takes less updates to converge.

III. Result Analysis

From, output 1, we can see, the sample points are non-linear.From, output 2, we can see the sample points are taken to the 6 dimensional space using ϕ function.From output 4, we can see the comparision between the updaing process. Here we can clearly see, batch updates takes more updates to converge than single update process.

The final output of this program is as follows-

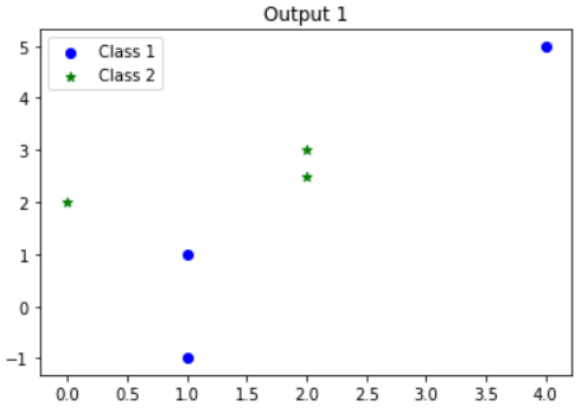

Fig. 1. Task 1 Output: Train dataset points

.

```
Output 2:
[[ 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.  ]]
```

Fig. 2. Task 2 Output: Train dataset points using ϕ function

TABLE I
Sample Output (Initial Weight Vector All Zero):

Alpha(Learning Rate)	One at a Time	Many at a Time
0.1	94	105
0.2	94	105
0.3	94	92
0.4	94	105
0.5	94	92
0.6	94	105
0.7	94	105
0.8	94	105
0.9	94	105
1.0	94	92

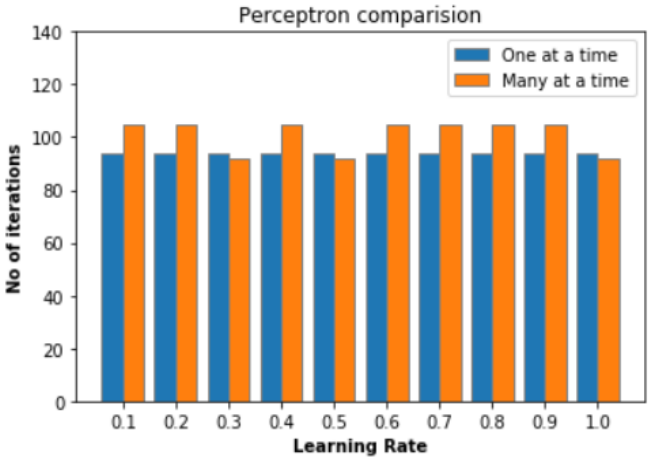

Fig. 3. Task 4 Output :Comparision with initial weight vector all zero

TABLE II
Sample Output (Initial Weight Vector All One):

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

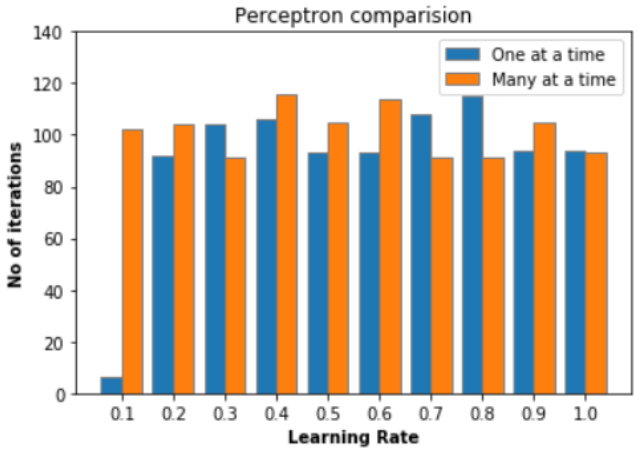

Fig. 4. Task 4 Output :Comparision with initial weight vector all one

TABLE III
SAMPLE OUTPUT (INITIAL WEIGHT VECTOR ALL RANDOM(SEED):

Alpha(Learning Rate)	One at a Time	Many at a Time
0.1	105	98
0.2	109	104
0.3	93	105
0.4	108	117
0.5	96	106
0.6	94	106
0.7	97	101
0.8	84	105
0.9	94	122
1.0	105	88

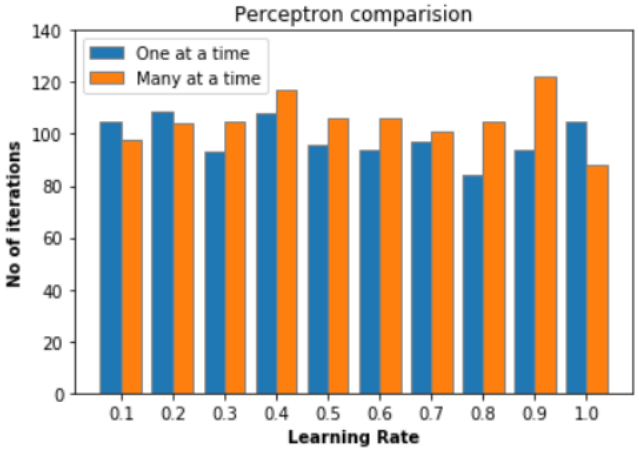

Fig. 5. Task 4 Output :Comparision with initial weight all random(seed)

IV. CONCLUSION

The perceptron algorithm is a very commonly used algorithm.In this experiment, we can learn that one at a time is better than many at a time to converge early.

V. ALGORITHM IMPLEMENTATION / CODE

```
# In[1]:


import matplotlib.pyplot as plt

import numpy as np

a=np.loadtxt('train-perceptron.txt')
#print(a)

#### task 1
class1=[]
class2=[]
for i in a:
    if i[2]==1:
        class1.append(i)
    else:
        class2.append(i)
class1=np.array(class1)
class2=np.array(class2)

x1=class1[:,0]
y1=class1[:,1]
x2=class2[:,0]
y2=class2[:,1]

fig,ax=plt.subplots()
plt.title("Output 1")
ax.scatter(x1, y1, marker='o',color='b',label='Class
    1')
ax.scatter(x2, y2, marker='*',color='g',label='Class
    2')
ax.legend()




# In[2]:


### task 2
def phii(x1,x2):
    return np.array([x1*x1, x2*x2, x1*x2, x1, x2,
    1])

y=[]
# phii func of class 1
for i in range(len(x1)):
    y.append(phii(x1[i],y1[i]))
# phii funcof class 2
for i in range(len(x2)):
    temp = phii(x2[i],y2[i])
    #negating class 2 for normalization
    y.append(np.dot(temp,-1))

y=np.array(y)
print("Output 2:")
print(y)


# In[3]:


### task 3
```

IV. CONCLUSION

The perceptron algorithm is a very commonly used algorithm.In this experiment, we can learn that one at a time is better than many at a time to converge early.

```
def Batch( weight,alpha):
    i=0
    while(i<200):
        flag =0; #initilally all classified
        sum = np.zeros_like(y[0])
        wy=[]
        for j in range(len(y)):
            wy.append(np.dot(weight,y[j])) #
    multiplication of wTy
            if(wy[j] <= 0):
                flag=1 #misclassified detected
                sum = sum + y[j] #summation of
    misclassified data

        weight = weight + (alpha * sum)
        if(flag == 0):
                return i+1
        i=i+1
    return i+1 #if iteration exists 200 , it will
    return 200



# In[4]:


def Single( weight,alpha):
    i=0
    while(i<200):
        flag =0; #initilally all classified

        wy=[]
        for j in range(len(y)):
            wy.append(np.dot(weight,y[j])) #
    multiplication of wTy
            if(wy[j] <= 0):
                flag=1 #misclassified detected
                sum = np.zeros_like(y[0])
                sum = sum + y[j] #summation of
    misclassified data
                weight = weight + (alpha * sum)

        if(flag == 0):
                return i+1
        i=i+1
    return i+1 #if iteration exists 200 , it will
    return 200



# In[7]:


### task 4
####### weight - All Zero
w=np.zeros_like(y[0])
one=[]
many=[]
alpha_arr =[]
print("Weight- All Zero")
print("Alpha(Learning Rate)\tOne at a Time\tMany at
    a Time")

for alpha in np.arange(0.1,1.1,0.1):
    alpha_arr.append(alpha)
    one.append(Single(w,alpha))
    many.append(Batch(w,alpha))
    print("\t{:.1f}".format(alpha) + "\t\t" + str(
    Single(w,alpha))+ "\t\t" + str(Batch(w,alpha)))


### bar-chart
barWidth = 0.4
br1 = np.arange(10)
br2 = [x + barWidth for x in br1]

# Make the plot
plt.bar(br1, one, width = barWidth,
        edgecolor ='grey', label ='One at a time')
plt.bar(br2, many, width = barWidth,
        edgecolor ='grey', label ='Many at a time')

# Adding Xticks
plt.title('Perceptron comparision')
plt.xlabel('Learning Rate', fontweight ='bold')
plt.ylabel('No of iterations', fontweight ='bold')
plt.xticks([r + (barWidth/2) for r in range(10)],
    [0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0])
plt.ylim(0, 140)
plt.legend()
plt.show()


# In[14]:


####### weight - All one
w1=np.ones_like(y[0])
one=[]
many=[]
alpha_arr =[]
print("Weight - All One")
print("Alpha(Learning Rate)\tOne at a Time\tMany at
    a Time")
for alpha in np.arange(0.1,1.1,0.1):
    alpha_arr.append(alpha)
    one.append(Single(w1,alpha))
    many.append(Batch(w1,alpha))
    print("\t{:.1f}".format(alpha) + "\t\t" + str(
    Single(w1,alpha))+ "\t\t" + str(Batch(w1,alpha))
    )


### bar-chart
barWidth = 0.4
br1 = np.arange(10)
br2 = [x + barWidth for x in br1]

# Make the plot
plt.bar(br1, one, width = barWidth,
        edgecolor ='grey', label ='One at a time')
plt.bar(br2, many, width = barWidth,
        edgecolor ='grey', label ='Many at a time')

# Adding Xticks
plt.title('Perceptron comparision')
plt.xlabel('Learning Rate', fontweight ='bold')
plt.ylabel('No of iterations', fontweight ='bold')
plt.xticks([r + (barWidth/2) for r in range(10)],
    [0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0])
plt.ylim(0, 140)
plt.legend()
plt.show()


# In[16]:


np.random.seed(0)
# weight vector Random
w2 = np.random.rand( len(y[0]))
print(w2)
one=[]
many=[]
alpha_arr =[]

print("Weight - Random")
```

```
print("Alpha(Learning Rate)\tOne at a Time\tMany at
    a Time")

for alpha in np.arange(0.1,1.1,0.1):
    alpha_arr.append(alpha)
    one.append(Single(w2,alpha))
    many.append(Batch(w2,alpha))
    print("\t{:.1f}".format(alpha) + "\t\t" + str(
    Single(w2,alpha))+ "\t\t" + str(Batch(w2,alpha))
    )

### bar-chart
barWidth = 0.4
br1 = np.arange(10)
br2 = [x + barWidth for x in br1]

# Make the plot
plt.bar(br1, one, width = barWidth,
        edgecolor ='grey', label ='One at a time')
plt.bar(br2, many, width = barWidth,
        edgecolor ='grey', label ='Many at a time')

# Adding Xticks
plt.title('Perceptron comparision')
plt.xlabel('Learning Rate', fontweight ='bold')
plt.ylabel('No of iterations', fontweight ='bold')
plt.xticks([r + (barWidth/2) for r in range(10)],
    [0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0])
plt.ylim(0, 140)
plt.legend()
plt.show()
```