Implementing the Perceptron Algorithm for Finding the Weights of a Linear Discriminant Function

Marufa Kamal
dept. Computer Science and Engineering
Ahsanullah University of Science and Technology
Dhaka, Bangladesh
160204073@aust.edu

***Abstract*—In machine learning, Perceptron is a linear algorithm used for linear separation in binary classification tasks. In this experiment I attempt on finding the weights of a linear discriminant function using the perceptron algorithm, to take the sample points to a higher dimension so that they can be linearly separable. The number of iterations required to classify the sample points along with their respective bar graphs is shown in comparison when different initial weights are taken.**

***Index Terms*—Perceptron, Perceptron algorithm, Linear discriminant function, Normalization.**

I. Introduction

Perceptron algorithm in machine learning acts as a binary classifier that identifies a sample point to its respective class in between two classes. This algorithm has two categories i.e., batch perceptron and single perceptron. Separating non-linear data linearly can be difficult when they are in a lower dimension, in this case, the perceptron algorithm can take the data to a higher dimension by updating their weights and classify them to their respective classes. It draws a decision boundary that separates the two classes using a hyperplane in the feature space. It is like a neuron taking some inputs and predicting the class it belongs to.

II. Experimental Design / Methodology

The following steps were taken in order to complete the 4 tasks that were given.

A. Plotting Training Sample Points

The training data set consisted of 6 vector samples of which 3 belonged to class 1 and the remaining to class 2. Using the python package '**matplotlib**' I have plotted all sample points from the '*train-perceptron.txt*' file making sure that the samples from the same class are marked with the same color and marker. Storing the values of the 2 classes in separate **numpy** arrays the points are plotted. A linear decision boundary cannot be drawn in this case as shown in figure 1, the data's are scattered in a way that it cannot be separated linearly.

B. Generating High Dimensional Sample Points

To take the sample points to a higher dimension a ϕ function is used which changes the inputs using the following formula:

$$y = [\mathrm{x}_1^2 \ \mathrm{x}_2^2 \ \mathrm{x}_1 * \mathrm{x}_2 \ \mathrm{x}_1\mathrm{x}_2 \ 1]$$

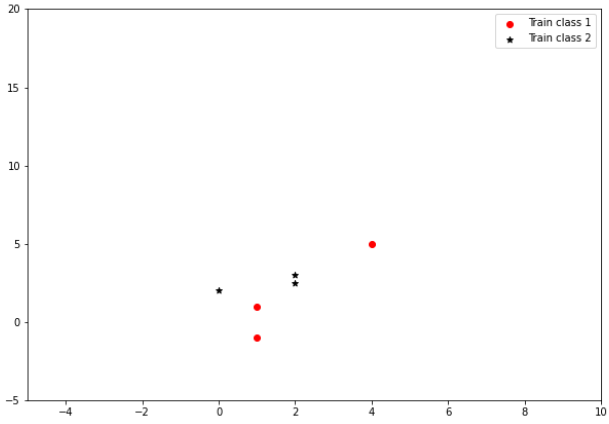

Fig. 1. Plotting Training Sample Points

After computing the second-order sample points to six-dimensional vector space, I further normalize(negate) the inputs of class two for easier computation making it opposite to the initial input.

C. Using Perceptron Algorithm for Finding the Weights

Taking 3 different initial weights, all zero, all one and all random the weights are updated using the following equation:

$$\begin{aligned} \omega(\mathrm{i+1}) &= \omega(\mathrm{i}) + \alpha \mathrm{Y}_m^{(k)}, \\ \text{if, } \omega^T(\mathrm{i})\mathrm{Y}_m^{(k)} &<= 0 \end{aligned} \tag{1}$$

i.e., if $Y_m^{(k)}$ is misclassified. Here α is the learning rate ranging between 0.1 and 1.0 with step size taken 0.1. And for being correctly classified the condition for following equation is met.

$$\begin{aligned} \omega(\mathrm{i+1}) &= \omega(\mathrm{i}), \\ \text{if, } \omega^T(\mathrm{i})\mathrm{Y}_m^{(k)} &> 0 \end{aligned} \tag{2}$$

When the inputs are misclassified the weights are using the two methods i.e., batch update or single update.

III. Result Analysis

The following results are produced, showing the number of iterations for each step size with the varying initial weights, and a comparison between one at a time and many at a time update for the different weights are shown using the bar graphs.

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

TABLE I
Initial Vector All One

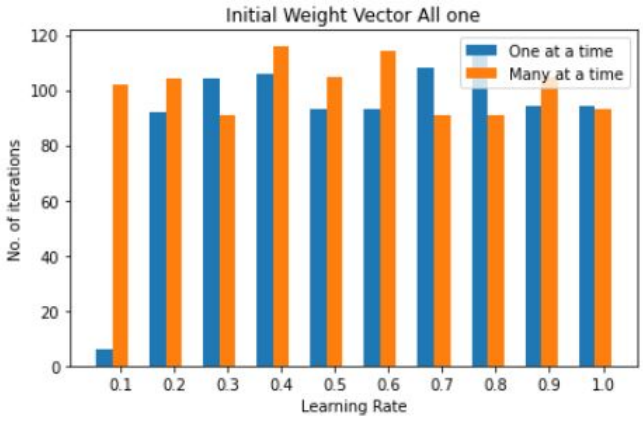

Fig. 2. Initial Vector All One

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

TABLE II
Initial Vector All Zero

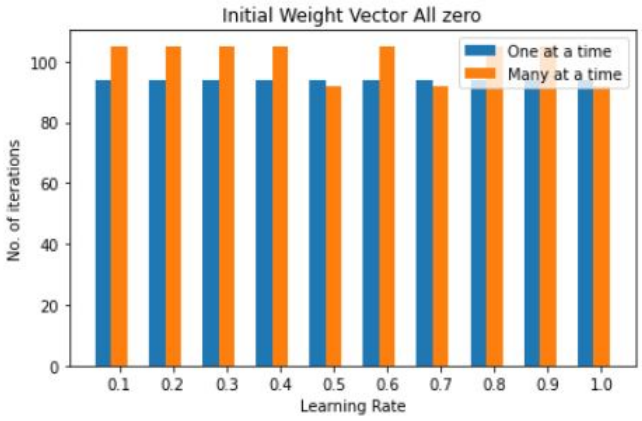

Fig. 3. Initial Vector All zero

Alpha(learning rate)	one at a time	many at a time
0.1	109	125
0.2	100	100
0.3	94	126
0.4	98	102
0.5	98	115
0.6	112	127
0.7	109	101
0.8	108	116
0.9	108	116
1.0	115	116

TABLE III
Initial Vector All Random

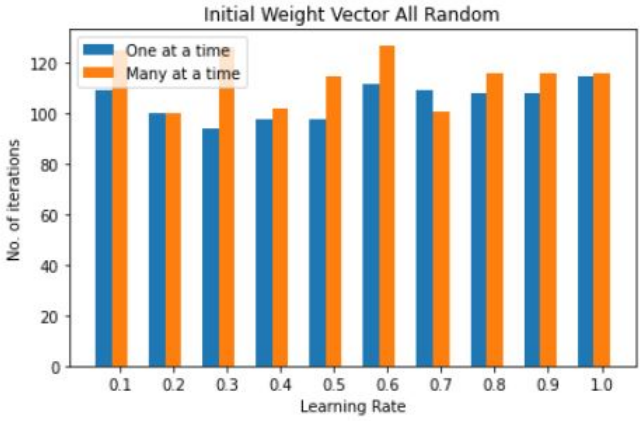

Fig. 4. Initial Vector All Random

IV. Question Answering

- Taking the points to higher dimension held to draw decision boundary for non-linear data. For non-linear data in lower dimension the accurate hyperplane cannot be drawn to classify them. That is why it is taken to a higher dimension using the ϕ function.
- For the three initial weight cases and for each learning rate the no of updates taken before converging are shown in the bar charts along with the values in the tables.

V. Conclusion

Perceptron algorithm is quite popular in binary classification. Implementing this algorithm helps to classify nonlinear data. Moreover, it is seen that for only 6 vector points the single update takes fewer iterations compared to the batch update method. so for higher sample points single update will

take much lesser iterations and work more efficiently. We can conclude from this experiment that, for this type of binary classification problem one at a time method performs better in comparison to many at a time update. Moreover taking sample points to higher dimensions leads to easier computation.

VI. Algorithm Implementation / Code

```
# -*- coding: utf-8 -*-
"""PR_assm2.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/16
    VuN8VB2oVv1n-cw1B2wfbG7CdV3D82i
"""

#1
import matplotlib.pyplot as plt
import numpy as np
train_data =np.loadtxt('/content/drive/MyDrive/4.2/
    pattern lab /assm2/train-perceptron.txt',dtype='
    float64');

print(train_data);
#print(test_data);
print(train_data[1][1]);
train_data_c1=[];
train_data_c2=[];
for item in train_data:
    if item[2]==1:
        #print(item[2]);
        train_data_c1.append(item)
    elif item[2]==2:
        train_data_c2.append(item)
train_data_class1= np.array(train_data_c1);
train_data_class2= np.array(train_data_c2);
#
print(train_data_class1);
print(train_data_class2);
x1,y1= train_data_class1[:,0],train_data_class1
    [:,1];
x2,y2= train_data_class2[:,0],train_data_class2
    [:,1];
print("x1",x1);
print("x2",x2);
fig=plt.figure(1, figsize=(10,7))
chart1= fig.add_subplot()
chart1.scatter(x1,y1,marker='o',color='r',label='
    Train class 1');
chart1.scatter(x2,y2,marker='*',color='k',label='
    Train class 2');

chart1.axis([-5,10,-5,20]);
chart1.legend()#
plt.savefig('TrainClass.png')

#2
classVal=[]
for a in train_data_class1:
  classVal.append(np.array([a[0]*a[0],a[1]*a[1],a
    [0]*a[1],a[0],a[1],1]))
for a in train_data_class2:
  classVal.append(np.array([a[0]*a[0],a[1]*a[1],a
    [0]*a[1],a[0],a[1],1])*-1)#normalized class2
for x in classVal:
    print(x)

#3
a=0
eta= np.array([0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7,
    0.8, 0.9, 1.0])
```

```
def batchProcessing(weight):
  for z in range(10):
    w=weight
    for count in range(99999999):
      Ysum=np.zeros_like(classVal[0])
      product=np.zeros_like(classVal[0])
      misclass=0 #all classified at first
      for i in range(6): #0 to 5
        wTy= np.dot(classVal[i], w)
        if (wTy <= 0.0):
          misclass=1
          Ysum=classVal[i]+Ysum
      product=Ysum*eta[z]
      w=w +product
      if misclass==0:
        iteration.append( count+1)
        break
#single update
def singleUpdate(weight):
    for z in range(10):
      w=weight
      for count in range(9999999999):
          misclass=0
          for i in range(6):
            wTy= np.dot(classVal[i], w)
            if (wTy <= 0.0):
                misclass=1
                Ysum = np.zeros_like(classVal[0])
                Ysum= Ysum + classVal[i]
                product=Ysum*eta[z]
                w=w +product
          if misclass==0:
            iteration1.append( count+1)
            break

#one
print("Initial Weight Vector = All one")
iteration =[]
iteration1 =[]
v=0.1
w=np.ones_like(classVal[0])
batchProcessing(w)
singleUpdate(w)
print("Alpha(Learning Rate)"+"\t\t"+"One at a Time"+
    "\t\t"+'Many at a Time')
for x in range(10):
  print(format(v, '.1f'),'\t\t\t\t',iteration1[x], "
    \t\t\t" ,iteration[x])
  v=v+0.1
iteration7 =iteration
iteration8 =iteration1

#zero
print("Initial Weight Vector = All Zero")
v=0.1
iteration =[]
iteration1 =[]
w=np.zeros_like(classVal[0])
batchProcessing(w)
singleUpdate(w)
print("Alpha(Learning Rate)"+"\t\t"+"One at a Time"+
    "\t\t"+'Many at a Time')
for x in range(10):
  print(format(v, '.1f'),'\t\t\t\t',iteration1[x], "
    \t\t\t" ,iteration[x])
  v=v+0.1
iteration3 =iteration
iteration4 =iteration1

#random
print("Initial Weight Vector = All random")
iteration =[]
iteration1 =[]
v=0.1
```

```
np.random.seed(1)
w = np.random.uniform(0, 1, len(classVal[0]))
batchProcessing(w)
singleUpdate(w)
print("Alpha(Learning Rate)"+"\t\t"+"One at a Time"+
    "\t\t"+'Many at a Time')
for x in range(10):
  print(format(v, '.1f'),'\t\t\t\t',iteration1[x], "
    \t\t\t" ,iteration[x])
  v=v+0.1
iteration5 =iteration
iteration6 =iteration1

bar_width = 0.3
index = np.arange(10)
plt.title('Initial Weight Vector All one')
plt.bar(index, iteration8 , bar_width,label='One at
    a time')
plt.bar(index + bar_width, iteration7 , bar_width,
    label='Many at a time')
plt.xlabel('Learning Rate')
plt.ylabel('No. of iterations')
plt.xticks(index + bar_width, ('0.1', '0.2', '0.3',
    '0.4', '0.5', '0.6', '0.7', '0.8', '0.9','1.0'))
plt.tight_layout()
plt.legend()
plt.show()
plt.savefig('one.png')
###
bar_width = 0.3
index = np.arange(10)
plt.title('Initial Weight Vector All zero')
plt.bar(index, iteration4 , bar_width,label='One at
    a time')
plt.bar(index + bar_width, iteration3 , bar_width,
    label='Many at a time')
plt.xlabel('Learning Rate')
plt.ylabel('No. of iterations')
plt.xticks(index + bar_width, ('0.1', '0.2', '0.3',
    '0.4', '0.5', '0.6', '0.7', '0.8', '0.9','1.0'))
plt.tight_layout()
plt.legend()
plt.show()
plt.savefig('zero.png')
###
bar_width = 0.3
index = np.arange(10)
plt.title('Initial Weight Vector All Random')
plt.bar(index, iteration6, bar_width,label='One at a
     time')
plt.bar(index + bar_width, iteration5 , bar_width,
    label='Many at a time')
plt.xlabel('Learning Rate')
plt.ylabel('No. of iterations')
plt.xticks(index + bar_width, ('0.1', '0.2', '0.3',
    '0.4', '0.5', '0.6', '0.7', '0.8', '0.9','1.0'))
plt.tight_layout()
plt.legend()
plt.show()
plt.savefig('random.png')
```