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

Rashedun Nobi Chowdhury
ID: 160204039
Group: A2
Email: 160204039@aust.edu

***Abstract*—Data classification is one of the basic application of Machine Learning. Of the two types of model used for data classification naming discriminant approach and generative approach, discriminant approaches estimate the discriminant function explicitly. It does not require a probability distribution. The discriminant function is used to predict which class a data point falls in. In this experiment, I implement the perceptron algorithm to find the weights of a Linear Discriminant Function. I also do some experiments to compare the number of iterations it takes for the algorithm to converge for a given learning rate.**

***Index Terms*—Perceptron, Linear Discriminant Function, Binary Classification, Machine Learning**

I. Introduction

In classification techniques of Machine Learning, we harness the power of pattern recognition to find patterns in data. Based on the patterns, we use different approaches that grant us the ability to effectively find dissimilarities of data belonging to different class to predict a future unknown data. One such approach is discriminant approach. In this approach, we first determine the form of discriminant which we can use to separate the two class of data. For this experiment we are dealing with a linear discriminant function which can be written in the form

$$g(X) = w^t x + w_0 \tag{1}$$

Here, **w** determines the orientation of the hyperplane whereas w_0 determines its location. In order to function effectively it is necessary to properly estimate the values of **w** and w_0 . One thing we have to keep in mind is that in order to separate the two class of data using a linear discriminant, the data must be linearly separable. It can be noted that perceptron algorithm helps us to determine these values. Perceptron is the simplest type of neural network. It first starts with a random or predefined value of weights (w). It then updates the values in each iteration until all the data are properly classified. The rest of the report is organized as follows, section 2 contains the experimental design. In section 3 I briefly describe the results. In section 4 I conclude the report with some discussions on the advantages and disadvantages of the algorithm. Finally in section 5, I attach a snapshot of my implemented code.

II. Experimental Design / Methodology

As mentioned earlier, discriminant functions 1 are used to describe a data point and determine its class. Converting 1 to homogeneous form we get,

$$g(x) = a^t y \tag{2}$$

where, a^t is the modified weight vector and y is the augmented feature vector. The decision rule then can be described as follows, if $g(x) > 0$, then y_i falls in class 1 else if $g(x) < 0$ then y_i falls in class 2. However, if we normalize one of the two classes the equation,

$$a^t y_i > 0 \tag{3}$$

can be used to determine if a data is properly classified or not. For normalizing, we keep one class of data as it is while performing a negation operation on the other class.

At first I plot all the data points in *fig 1* . It can be seen from the figure that, the data of two classes are not linearly separable.

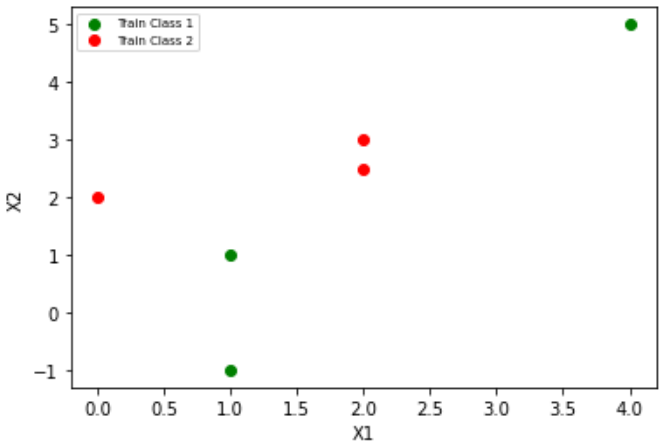

Fig. 1. Distribution of Data Points

We know that, for a linear discriminant to function, the data must be linearly separable. However, the given data is not linearly separable in two dimensional space.
To tackle this issue, we must map the data to a higher

dimensional space so they become linearly separable. I use the following polynomial discriminant function

$$y = [x_1^2 \quad x_2^2 \quad x_1 * x_2 \quad x_1 \quad x_2 \quad 1] \tag{4}$$

to take all the data points in the train dataset to a higher dimension. I have followed two strategies for perceptron algorithm for finding the correct weight vectors. First one is *One at a Time*.

Algorithm 1 One at a Time strategy in Parceptron

1) Read the train dataset
2) Take the data to higher dimension using 4
3) Set the weight vector as all zero
4) Set α=0.1
5) For all data point y, compute $g(x)$ using 2
6) If $g(x) > 0$ data is properly classified, else misclassified
7) Update weight vector w if misclassified, using

$$w(i+1) = w(i) + \alpha y_m^k \tag{5}$$

Otherwise,

$$w(i+1) = w(i) \tag{6}$$

8) Repeat step 7 after each data until all are properly classified
9) Set new value of α = α+0.1 if α <1, go to 5
10) Set new weight vector, go to 4

Many at a time is quite similar to 1. In many at a time strategy, we compute y_m^k by summing all the misclassifed data. The update also occurs after all the data have been iterated.

III. Result Analysis

In this section, I will compare the number of iterations it took for the perceptron algorithm to converge for different learning rates α and weight vectors w. In the first case, I set the values of weight vector to 0. The number of iterations it took for the algorithm to converge is given in I,

Learning Rate (α)	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	92
0.7	94	92
0.8	94	105
0.9	94	105
1	94	92

TABLE I
Comparison of Iterations to converge for weight zero

It can be seen that the varying the learning rate has no impact on the number of iterations in case of One at A time. However the same is not true for Many at a Time strategy. The number of iterations taken to converge is either 92 or 105.

A visual representation of the comparison is given in 2

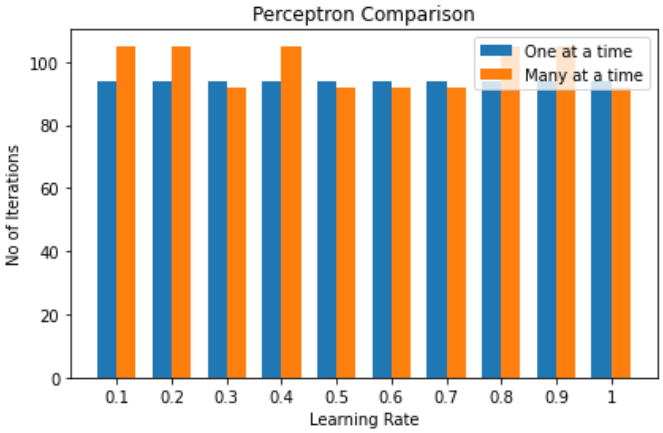

Fig. 2. Comparison of Iterations to converge for weight zero

Next we set the initial weight vector to 1. We then perform similar experiment $i.e$ vary the learning rate between 0.1 and 1 and note how many iterations were taken for the algorithm to converge.

Learning Rate (α)	One at a Time	Many at a Time
0.1	6	101
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	114	91
0.9	94	105
1	94	93

TABLE II
Comparison of Iterations to converge for weight one

When initial weight is set to 1, one at a time strategy converges fastest α = 0.1. However it can take upto 114 iterations (α = 0.8) for convergence in this strategy. For Many at a Time strategy, number of interations for convergence varies between 91 and 114.

A bar chart representation of table II is given in figure 3
For the last comparison, I set the weight vector randomly.

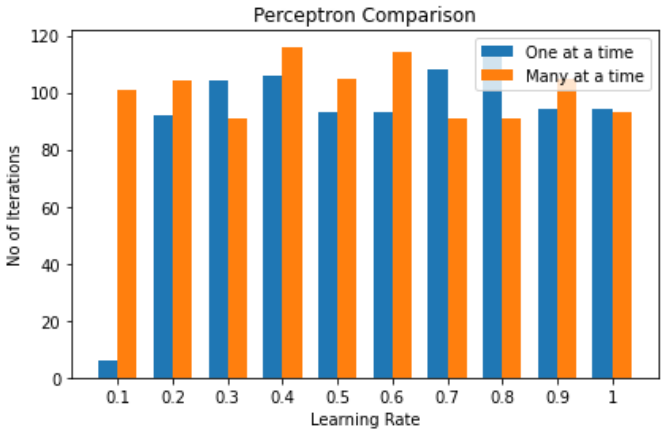

Fig. 3. Comparison of Iterations to converge for weight one

I then perform the similar experiments as the previous two. The findings are presented in table III

Learning Rate (α)	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	101	150

TABLE III
COMPARISON OF ITERATIONS TO CONVERGE FOR RANDOM WEIGHT

Again, a bar chart representation of table III is given in figure 4

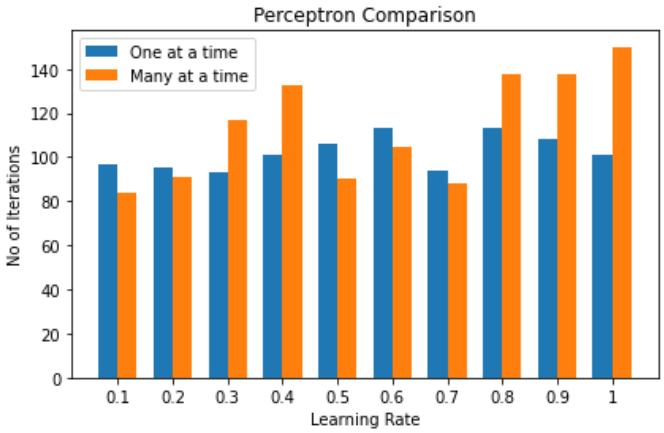

Fig. 4. Comparison of Iterations to converge for random weight

Based on all these experiments, I find that the algorithm converges fastest when α is set to **0.1**, initial weights are set to **1** and **One at a Time** strategy is used.

IV. CONCLUSION

In this experiment, I implemented the Perceptron algorithm for finding the weights of a Linear Discriminant Function. The given dataset was the not linearly separable in its original form. I therefore used the polynomial discriminant function to take the data to a higher dimension and then performed the experiment. I also found that the learning rate and initial weight vector has an impact on the number of iterations taken for the algorithm to converge. However I found that in the case of one at a time strategy, the order of data after normalization also impacts the number of iterations required. I feel this is a drawback of the approach as in most cases researchers will not keep an eye on the order of data in a given dataset and it may often take a long time to converge.

V. ALGORITHM IMPLEMENTATION / CODE

A snapshot of my implementation of the algorithm in python is given below,

```
np.random.seed(10)
weight=[np.ones(shape=(1,6)),np.zeros(shape=(1,6))

num_of_data = len(dataset.index)

weight_zero_iterations =[]
weight_one_iterations = []
weight_rand_iterations = []

for x in range (3):
    temp_list_one = []
    temp_list_many = []
    for alp in range (10):
        alpha = (alp+1)*0.1

        #many at a time
        w = weight[x]
        proper_classified = 0
        itr = 0
        while(proper_classified != num_of_data ):
            itr = itr + 1
            proper_classified = 0
            y_sum = np.zeros(shape = (1,6))
            for i in range (num_of_data):
                wTy=np.matmul(w,y[i,:].reshape(6,1
                if(wTy>0):
                    proper_classified =
proper_classified + 1
                else:
                    y_sum=y_sum+y[i,:].reshape(1,6

            w = w + alpha*y_sum
        temp_list_many.append(itr)

        #many at a time
        w = weight[x]
```

```
        proper_classified = 0
        itr = 0
        while(proper_classified != num_of_data ):
            itr = itr + 1
            proper_classified = 0
            y_sum = np.zeros(shape = (1,6))
            for i in range (num_of_data):
                wTy = np.matmul(w, y[i,:].reshape(6,1))
                if(wTy>0):
                    proper_classified =
proper_classified + 1
                else:
                    w = w + alpha*y[i,:].reshape(1,6)

        temp_list_one.append(itr)

    if(x==1):
        weight_zero_iterations.append(temp_list_one)
        weight_zero_iterations.append(temp_list_many)
    elif (x==0):
        weight_one_iterations.append(temp_list_one)
        weight_one_iterations.append(temp_list_many)
    else:
        weight_rand_iterations.append(temp_list_one)
        weight_rand_iterations.append(temp_list_many)
```