Title: Implementation of the Perceptron Learning Algorithm.

Objectives:

1. Classify by only using one linear boundary
2. Mimicking the work of a single neuron of brain
3. Applying a one-layer method which is computationally easy than others
4. Focusing on basic or not complex neural network algorithm

Methodology:

Perceptron learning algorithm is basically a weight adjusting algorithm just like most of the other algorithms of neural networks. Here, we take one input at a time and then process the input with respect to the weights and threshold. If we don't get the desired output, we come back and adjust the weights to get the correct output as desired. Without making any delay, let's first look at the pseudo algorithm of the perceptron learning algorithm:

1. **Initialize weights and threshold:** Define $w_i(t), (0 \leq i \leq n)$, to be the weight from input i at time t, and θ to be the threshold value in the output node. Set w_0 to be $-\theta$, the bias, and x_0 to be always 1. Set $w_i(0)$ to small random values, thus initializing all the weights and the threshold.

2. **Present input and desired output:** Present Input $x_0, x_1, x_2, \ldots, x_n$ and desired output $d(t)$.

3. **Calculate actual output:**
$$y(t) = f_h\left[\sum_{i=0}^{n} w_i(t)x_i(t)\right]$$

4. **Adapt weights:**
   If correct, $w_i(t+1) = w_i(t)$
   If output 0, should be 1 (class A), $w_i(t+1) = w_i(t) + x_i(t)$
   If output 1, should be 0 (class B), $w_i(t+1) = w_i(t) - x_i(t)$

Note that weights are unchanged if the net makes the correct decision. Also, weights are not adjusted on input lines which do not contribute to the incorrect response, since each weight is adjusted by the value of the input on that line, $x_i(t)$, which would be zero.

This is the basic perceptron learning algorithm. However, various modifications have been suggested to this basic algorithm. The first is to introduce a multiplicative factor of less than one into the weight adaptation term. This has the effect of slowing down the change in the weights, making the network take smaller steps towards the solution. This alteration to the algorithm entails replacing step 4 with the following:

Adapt weights – modified version:

If correct, $w_i(t+1) = w_i(t)$
If output 0, should be 1 (class A), $w_i(t+1) = w_i(t) + \eta x_i(t)$
If output 1, should be 0 (class B), $w_i(t+1) = w_i(t) - \eta x_i(t)$

where $0 \leq \eta \leq 1$, a positive gain term that controls the adaptation rate.

Another algorithm of similar nature was suggested by Widrow and Hoff. They realized that it would be best to change the weights by a lot when the weighted sum is a long way from the desired value, whilst altering them only slightly when the weighted sum is close to that required to give the correct solution. They proposed a learning rule known as Widrow-Hoff delta rule, which calculates the difference between the weighted sum and the required output, and calls that the error. Weight adjustment is then carried out in proportion to that error. This means that during the learning process, the output from the unit is not passed through the step function. However actual calculation is affected by using the step function to produce the $+1$ or 0 indication as before.

The error term Δ can be writer

$$\Delta = d(t) - y(t)$$

where $d(t)$ is the desired response of the system, and $y(t)$ is the actual response. This takes care of the addition or subtraction, since if the desired output is 1 and the actual output is 0, $\Delta = +1$ and so the weights are increased. Conversely, if the desired output is 0 and the actual output is $+1$, $\Delta = -1$ and so the weights will be decreased. Note that weights are unchanged if the net makes the correct decision, since $d(t) - y(t) = 0$.

The learning algorithm is basically the same as for the basic perceptron, except this time step 4 is replaced by

Adapt weights – Widrow-Hoff delta rule:

$$\Delta = d(t) - y(t)$$
$$w_i(t+1) = w_i(t) + \eta \Delta x_i(t)$$
$$d(t) = \begin{cases} +1, if\ input\ from\ class\ A \\ 0, if\ input\ from\ class\ B \end{cases}$$

where $0 \leq \eta \leq 1$, a positive gain function that controls the adaptation rate.

Neuron units using this learning algorithm were called ADALINEs (adaptive linear neurons) by Widrow, who also connected many of them together into a many-ADALINE structure, or MADALINE.

Another alternative proposed is to use inputs that are not 0 or 1 (binary), but are instead -1 and +1, known as bipolar. Using binary inputs means that input values with 0's on them are not trained, whereas bipolar value allow all the inputs to be trained each time. This simple alteration helps to speed up the convergence process but often leads to confusion in the literature as some authors discuss binary inputs and others bipolar ones. Effectively, they are equivalent and the use of one or the other is usually a matter of personal preference.

Now, let's look at the basic view of the model,

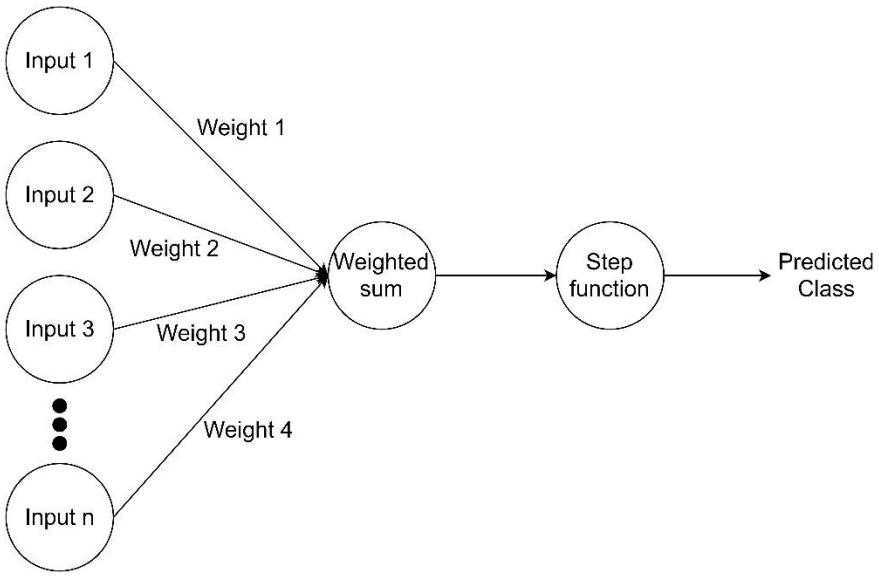

Figure: Model view of Perceptron Learning Algorithm

Basically, the perceptron learning algorithm finds out the best linear boundary to separate two classes. The important feature of this algorithm is that it only works for binary classes. To understand more clearly let's see an example of illustrated boundary using perceptron learning algorithm:

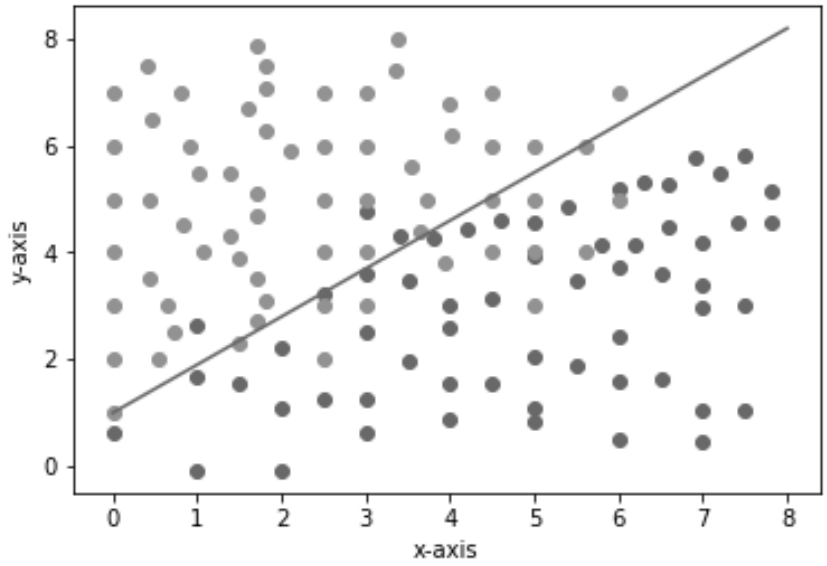

Figure: Illustrating the output of perceptron learning algorithm

In the next page a flowchart of the perceptron learning algorithm has been shown for better understanding.

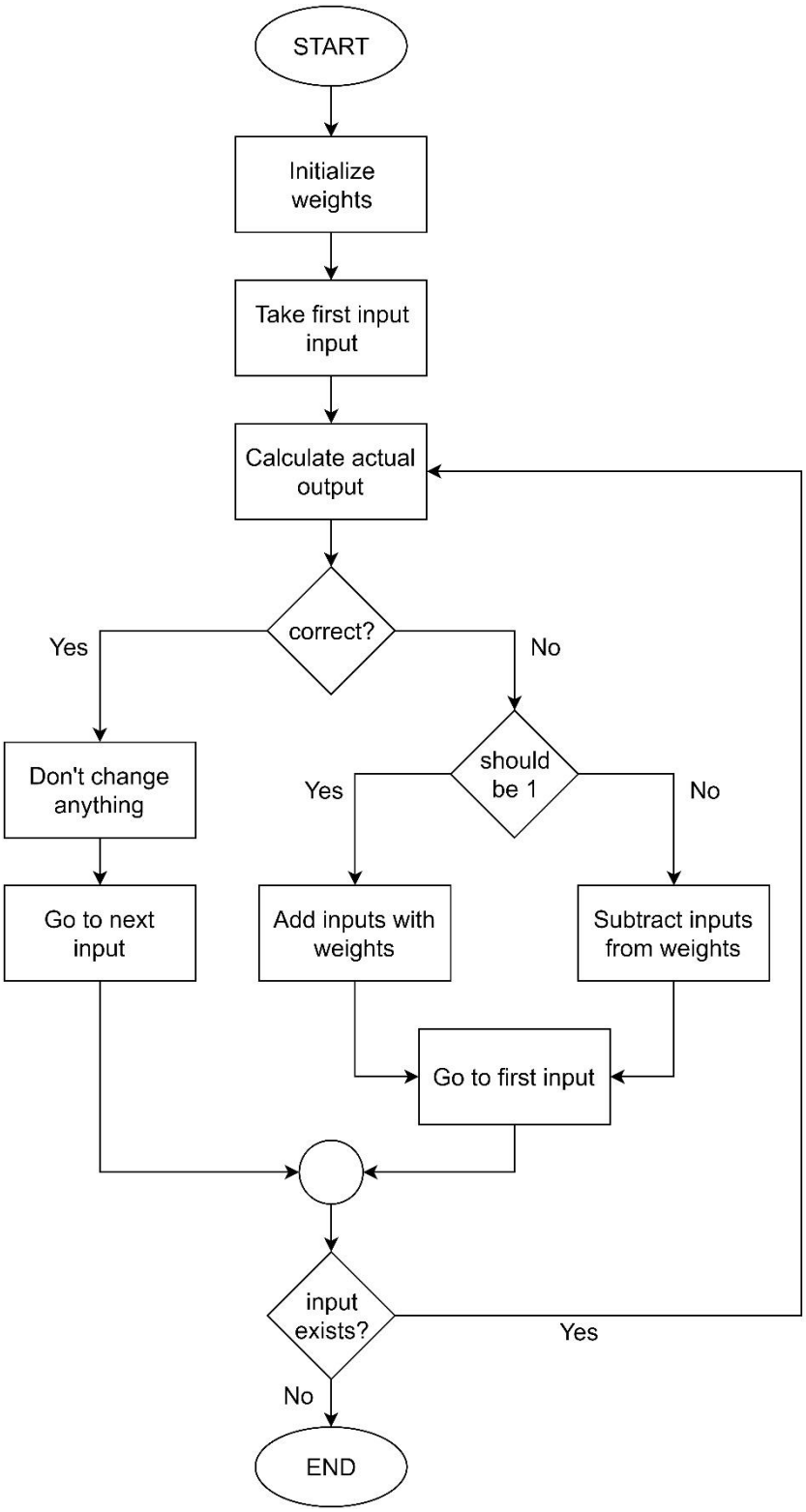

Figure: Flowchart for perceptron learning algorithm.

Implementation:

To implement perceptron learning algorithm,

1. First, we generated a dataset.
2. Break down the dataset into train set and test set.
3. Take the train set and apply the algorithm on it.
4. Take the test set and applying the adjusted weights, find out the accuracy.

The code itself has multiple parameters. The number of input lines can be modified as will of user. The same thing goes for the multiplicative factor. Also, the value of the threshold can also be modified. Besides, the percentage of data to be in train set can be modified either. This code can be easily modified for the other two modified version pretty easily. The code is given below. It contains: data generation, train-test set generation, training and testing – all in one.

Code:

```
"""
Algorithm Title: The Perceptron Learning Algorithm
Author: Azmain Yakin Srizon
"""

# importing necessary libraries
from numpy import binary_repr # for converting into binary
import random # for generating random numbers

# Specifications
input_line = 7 # defining number of input lines
print('Input Lines: ' + str(input_line)) # printing number of inputs
number_of_elements = pow(2,input_line) # calculating maximum number of inputs

# Making inputs
inputs = []
classes = []

i = 0
for i in range(0,number_of_elements):
    tmp = list(binary_repr(i,input_line)) # convert each number to binary list
    if tmp[0]=='0':
        classes.append('0') # defining class A
    else:
        classes.append('1') # defining class B
    inputs.append(tmp) # adding class
    i = i+1

# Making Train & Test Sets
train = []
train_class = []
test = []
test_class = []

train_percentage = 0.6 # set the train percentage
test_percentage = 1-train_percentage # calculating test percentage

tmp = int(train_percentage*(number_of_elements)) # calculating how many in train
print('In Train: ' + str(tmp)) # print number of train elements
print('In Test: ' + str(number_of_elements-tmp)) # print number of test elements
i = tmp
while i!=0:
    a = random.randint(0,tmp) # generating random numbers
    if inputs[a] not in train:
```

```
        train.append(inputs[a]) # adding to train data
        train_class.append(classes[a]) # adding class to train class
        i = i-1

train.sort() # sorting train data
train_class.sort() # sorting train class

c=0
for i in inputs:
    if i not in train: # checking who is not in train
        test.append(i) # adding data to test data
        test_class.append(classes[c]) # adding class to test class
    c=c+1

# Applying the algorithm to train
weights = []
for i in range(0,input_line):
    a = random.randint(1,10) # taking random value between 1 to 10
    weights.append(float(a)/10.0) # initializing weights
print('Initial Weights: ' + str(weights))

i = 0
f_h = 0.5 # initializing threshold
count = 0
flag = 0
eeta = 0.2 # initializing multiplicative factor
while i<tmp:
    count = count + 1
    y_t = 0.0
    c = 0
    for j in train[i]:
        y_t = y_t + float(j)*weights[c] # claculating summation
        c = c+1
    if y_t < f_h: # checking with threshold
        tmp_class = 0 # assigning temporary class A
    else:
        tmp_class = 1 # assigning temporary class B

    if int(train_class[i])!=int(tmp_class): # error caused
        flag = 1
        if tmp_class == 0: # should be 1
            cc = 0
            for j in train[i]:
                if int(j)==1:
                    weights[cc]=weights[cc]+eeta*1.0 # update the weigths
                cc = cc+1
        else: # should be 0
            cc = 0
            for j in train[i]:
                if int(j)==1:
                    weights[cc]=weights[cc]-eeta*1.0 # update the weights
                cc = cc+1
        continue
    else:
        if flag==1: # checking if need to start from the beginning again or not
            i=0
            flag=0
            continue
        i = i+1

print('Weights after adjustment: ') # printing the adjusted weights
s = ''
for i in weights:
    s = s + str("{0:.1f}".format(i)) + ' '
print(s)
print('Steps required: ' + str(count)) # printing the number of steps required

# It's time to test
```

```
i = 0
f_h = 0.5 # same threshold
wrong = 0 # variable to calculate number of wrongs
while i<number_of_elements-tmp:
    y_t = 0.0
    c = 0
    for j in test[i]:
        y_t = y_t + float(j)*weights[c] # calculating summations
        c = c+1
    if y_t < f_h:
        tmp_class = 0 # predicting class A
    else:
        tmp_class = 1 # predicting class B

    if int(test_class[i])!=int(tmp_class):
        wrong = wrong + 1
    i = i+1

accuracy = (1.0-float(wrong)/float(number_of_elements-tmp))*100.0 # calculating
accuracy
print('Prediction Accuracy: ' + str("{0:.2f}".format(accuracy)) + '%') #printing
accuracy
```

Output:

Input Lines: 7
In Train: 76
In Test: 52
Initial Weights: [0.1, 0.5, 1.0, 0.5, 0.1, 0.2, 0.9]
Weights after adjustment:
0.9 -0.1 0.0 -0.1 -0.1 0.0 0.1
Steps required: 495
Prediction Accuracy: 100.00%

Performance Analysis:

From the sample output a lot of things can be illustrated. Before doing performance analysis, let's break down the output. So, number of input lines were 7. Hence total possible inputs are 128. As the train percentage was 0.2, 76 samples are in train class and 52 samples are in test class. Now, we train by the train samples and get the adjusted weights when multiplicative factor was set to 0.2 and threshold was set to 0.5 while training. It needed 495 steps to complete the weight adjustment.

After that, test samples are being predicted as class A or class B by using the adjusted weights keeping threshold 0.5 again. And the result shows that the accuracy is 100% which tells that all the samples of the test set has been predicted perfectly. So, in terms of accuracy the algorithm does a pretty good job.

However, let's see what happens when we modify the parameters. If we increase the value of train percentage, the accuracy obviously increases although number of steps also increase. But as there are only two classes, the weight adaptation doesn't need to struggle more if train size is increased, hence, increasing of steps can be negligible. While the train percentage is approximately 0.4, the algorithm gives a great result. But when it decreases more, accuracy decreases. The main reason behind is a smaller number of inputs in train class. Also, there are some problems regarding random samples in train sets. If the number of samples from A or B class dictate the train set, then it gets biased and accuracy decreases as it can't predict the other class then.

If the number of input line is increases, the number of sample increase. That means a very small number of train percentage like 0.1 or 0.2 is enough to perfectly predict the classes of test samples. So, for a larger number of input lines, the perceptron learning algorithm works like a charm.

If we increase the value of multiplicative factor, the number of steps required for weight adjustment decreases but the weight adjustment can be messy and accuracy can drop also. With a smaller value of multiplicative factor, the accuracy increases dramatically but the process gets slow down as more steps are required for weight adaptation now.

The value of threshold has not a lot to do with the accuracy. But if the threshold is super high or super low, the accuracy changes a bit. Moreover, more steps are required for weight adaptation if this happens. The reason is that the weights have to get biased in order to support the very high or very low value of threshold.

In a word, all these parameters are connected with one another and that's why changing any value causes impact on others and overall accuracy. But overall, the algorithm itself works great for linear boundary problems.

There some limitations of this algorithm however. For example:

1. The algorithm doesn't work if more than one boundary is needed for classification.
2. If no-linear boundary is needed then this algorithm has to be ignored.
3. XOR problem can't be solved via this algorithm as it needs two decision boundaries.

Apart from these limitations, the perceptron learning algorithm is one of the best in practice for linear boundary problems.