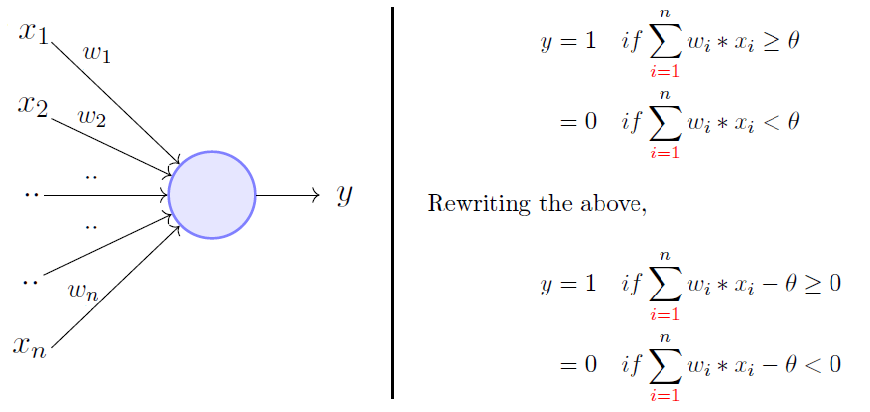
***Practical 9***

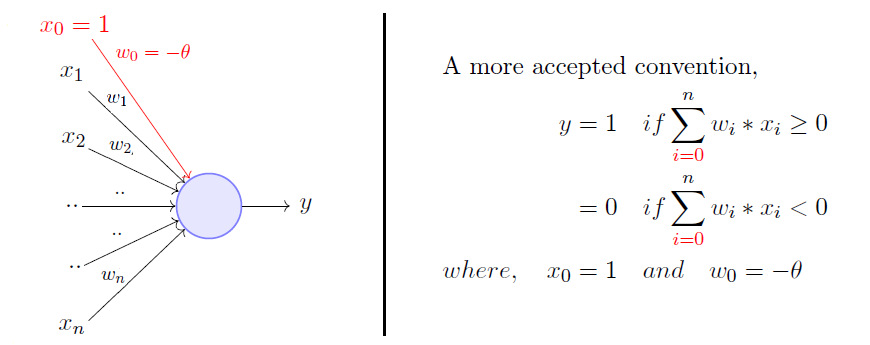
**Aim:** *Implement the Perceptron Algorithm.*

***Theory:***

*A perceptron is not the Sigmoid neuron we use in ANNs or any deep learning networks today.*

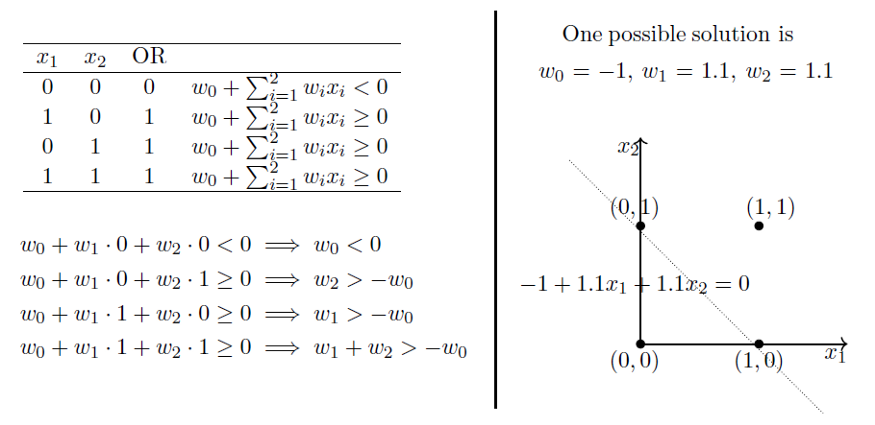


*The perceptron model is a more general computational model than McCulloch-Pitts neuron. It takes an input, aggregates it (weighted sum) and returns 1 only if the aggregated sum is more than some threshold else returns 0. Rewriting the threshold as shown above and making it a constant input with a variable weight, we would end up with something like the following:*



*A single perceptron can only be used to implement****linearly separable****functions. It takes both real and boolean inputs and associates a set of****weights****to them, along with a****bias****(the threshold thing I mentioned above). We learn the weights, we get the function. Let's use a perceptron to learn an OR function.*

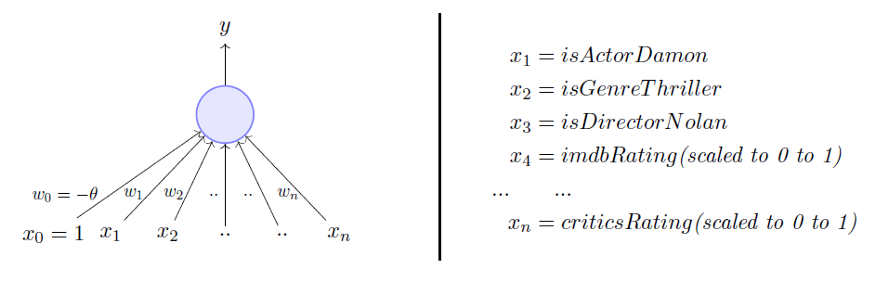
***OR Function Using A Perceptron***



*What’s going on above is that we defined a few conditions (the weighted sum has to be more than or equal to 0 when the output is 1) based on the OR function output for various sets of inputs, we solved for weights based on those conditions and we got a line that perfectly separates positive inputs from those of negative.*

*Doesn’t make any sense? Maybe now is the time you go through that*[*post*](https://towardsdatascience.com/4d8c70d5cc8d)*I was talking about. Minsky and Papert also proposed a more principled way of learning these weights using a set of examples (data). Mind you that this is NOT a Sigmoid neuron and we’re not going to do any Gradient Descent.*

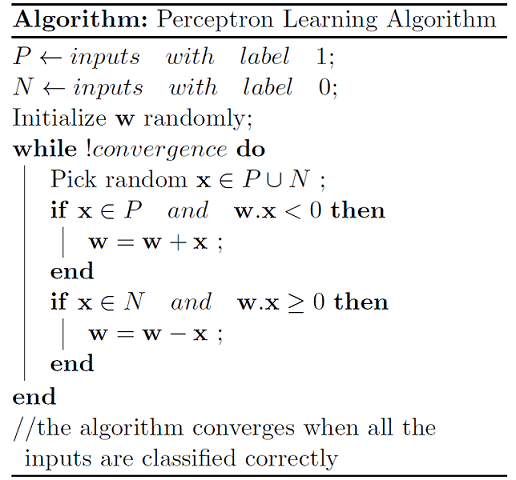
***Setting Up The Problem***



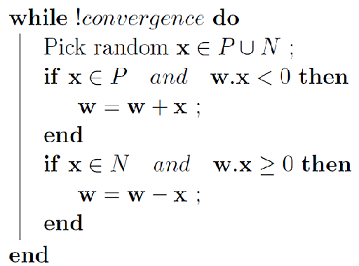
*We are going to use a perceptron to estimate if I will be watching a movie based on historical data with the above-mentioned inputs. The data has positive and negative examples, positive being the movies I watched i.e., 1. Based on the data, we are going to learn the weights using the perceptron learning algorithm. For visual simplicity, we will only assume two-dimensional input.*

***Perceptron Learning Algorithm***

*Our goal is to find the****w****vector that can perfectly classify positive inputs and negative inputs in our data. I will get straight to the algorithm. Here goes:*



*We initialize****w****with some random vector. We then iterate over all the examples in the data, (P U N) both positive and negative examples. Now if an input****x****belongs to P, ideally what should the dot product****w.x****be? I’d say greater than or equal to 0 because that’s the only thing what our perceptron wants at the end of the day so let's give it that. And if****x****belongs to N, the dot product MUST be less than 0. So if you look at the if conditions in the while loop:*

**

***Case 1:****When****x****belongs to P and its dot product****w.x****< 0****Case 2:****When****x****belongs to N and its dot product****w.x****≥ 0*

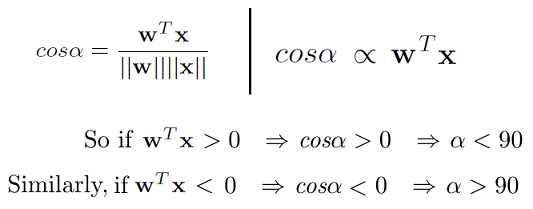
*Only for these cases, we are updating our randomly initialized****w****. Otherwise, we don’t touch****w****at all because Case 1 and Case 2 are violating the very rule of a perceptron. So we are adding****x****to****w****(ahem vector addition ahem) in Case 1 and subtracting****x****from****w****in Case 2.*

***Why Would The Specified Update Rule Work?***

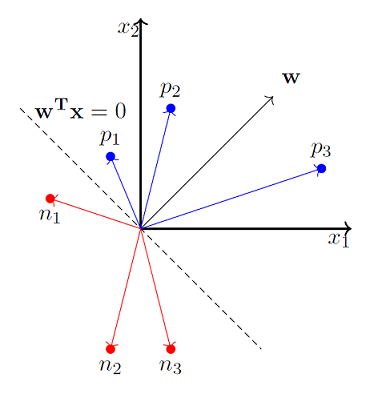
*But why would this work? If you get it already why this would work, you’ve got the entire gist of my post and you can now move on with your life, thanks for reading, bye. But if you are not sure why these seemingly arbitrary operations of****x****and****w****would help you learn that perfect****w****that can perfectly classify P and N, stick with me.*

*We have already established that when****x****belongs to P, we want****w.x****> 0, basic perceptron rule. What we also mean by that is that when****x****belongs to P, the angle between****w****and****x****should be \_\_\_\_\_ than 90 degrees. Fill in the blank.*

*Answer: The angle between****w****and****x****should be less than 90 because the cosine of the angle is proportional to the dot product.*

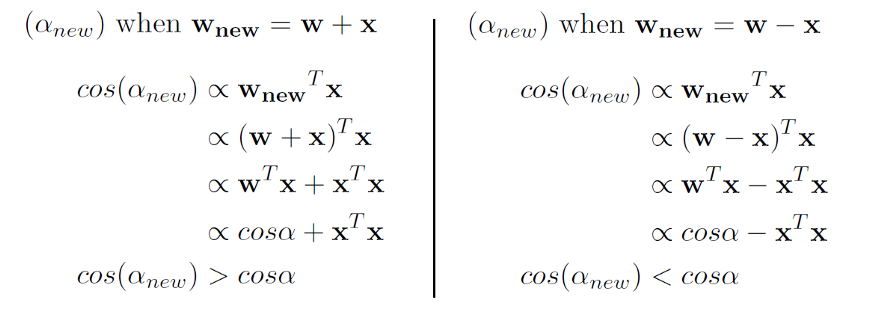
**

*So whatever the****w****vector may be, as long as it makes an angle less than 90 degrees with the positive example data vectors (****x****E P) and an angle more than 90 degrees with the negative example data vectors (****x****E N), we are cool. So ideally, it should look something like this:*

**

*x\_0 is always 1 so we ignore it for now.*

*So we now strongly believe that the angle between****w****and****x****should be less than 90 when****x****belongs to P class and the angle between them should be more than 90 when****x****belongs to N class. Pause and convince yourself that the above statements are true and you indeed believe them. Here’s why the update works:*

**

*So when we are adding****x****to****w****, which we do when x belongs to P and****w.x****< 0 (Case 1), we are essentially****increasing the cos(alpha)****value, which means, we are****decreasing the alpha value****, the angle between****w****and****x****,****which is what we desire****. And the similar intuition works for the case when****x****belongs to N and****w.x****≥ 0 (Case 2).*

*Here’s a toy simulation of how we might up end up learning****w****that makes an angle less than 90 for positive examples and more than 90 for negative examples.*

***Code:***

from sklearn import datasets

import numpy as np

import matplotlib.pyplot as plt

X, y = datasets.make\_blobs(n\_samples=1100,n\_features=2,centers=2,cluster\_std=1.05,random\_state=2)

#Plotting

fig = plt.figure(figsize=(10,8))

plt.plot(X[:, 0][y == 0], X[:, 1][y == 0], 'r^')

plt.plot(X[:, 0][y == 1], X[:, 1][y == 1], 'bs')

plt.xlabel("feature 1")

plt.ylabel("feature 2")

plt.title('Random Classification Data with 2 classes')

def step\_func(z):

        return 1.0 if (z > 0) else 0.0

def perceptron(X, y, lr, epochs):

    # X --> Inputs.

    # y --> labels/target.

    # lr --> learning rate.

    # epochs --> Number of iterations.

    # m-> number of training examples

    # n-> number of features

    m, n = X.shape

    # Initializing parapeters(theta) to zeros.

    # +1 in n+1 for the bias term.

    theta = np.zeros((n+1,1))

    # Empty list to store how many examples were

    # misclassified at every iteration.

    n\_miss\_list = []

    # Training.

    for epoch in range(epochs):

        # variable to store #misclassified.

        n\_miss = 0

        # looping for every example.

        for idx, x\_i in enumerate(X):

            # Insering 1 for bias, X0 = 1.

            x\_i = np.insert(x\_i, 0, 1).reshape(-1,1)

            # Calculating prediction/hypothesis.

            y\_hat = step\_func(np.dot(x\_i.T, theta))

            # Updating if the example is misclassified.

            if (np.squeeze(y\_hat) - y[idx]) != 0:

                theta += lr\*((y[idx] - y\_hat)\*x\_i)

                # Incrementing by 1.

                n\_miss += 1

        # Appending number of misclassified examples

        # at every iteration.

        n\_miss\_list.append(n\_miss)

    return theta, n\_miss\_list

def plot\_decision\_boundary(X, theta):

    # X --> Inputs

    # theta --> parameters

    # The Line is y=mx+c

    # So, Equate mx+c = theta0.X0 + theta1.X1 + theta2.X2

    # Solving we find m and c

    x1 = [min(X[:,0]), max(X[:,0])]

    m = -theta[1]/theta[2]

    c = -theta[0]/theta[2]

    x2 = m\*x1 + c

    # Plotting

    fig = plt.figure(figsize=(10,8))

    plt.plot(X[:, 0][y==0], X[:, 1][y==0], "r^")

    plt.plot(X[:, 0][y==1], X[:, 1][y==1], "bs")

    plt.xlabel("feature 1")

    plt.ylabel("feature 2")

    plt.title("Perceptron Algorithm")

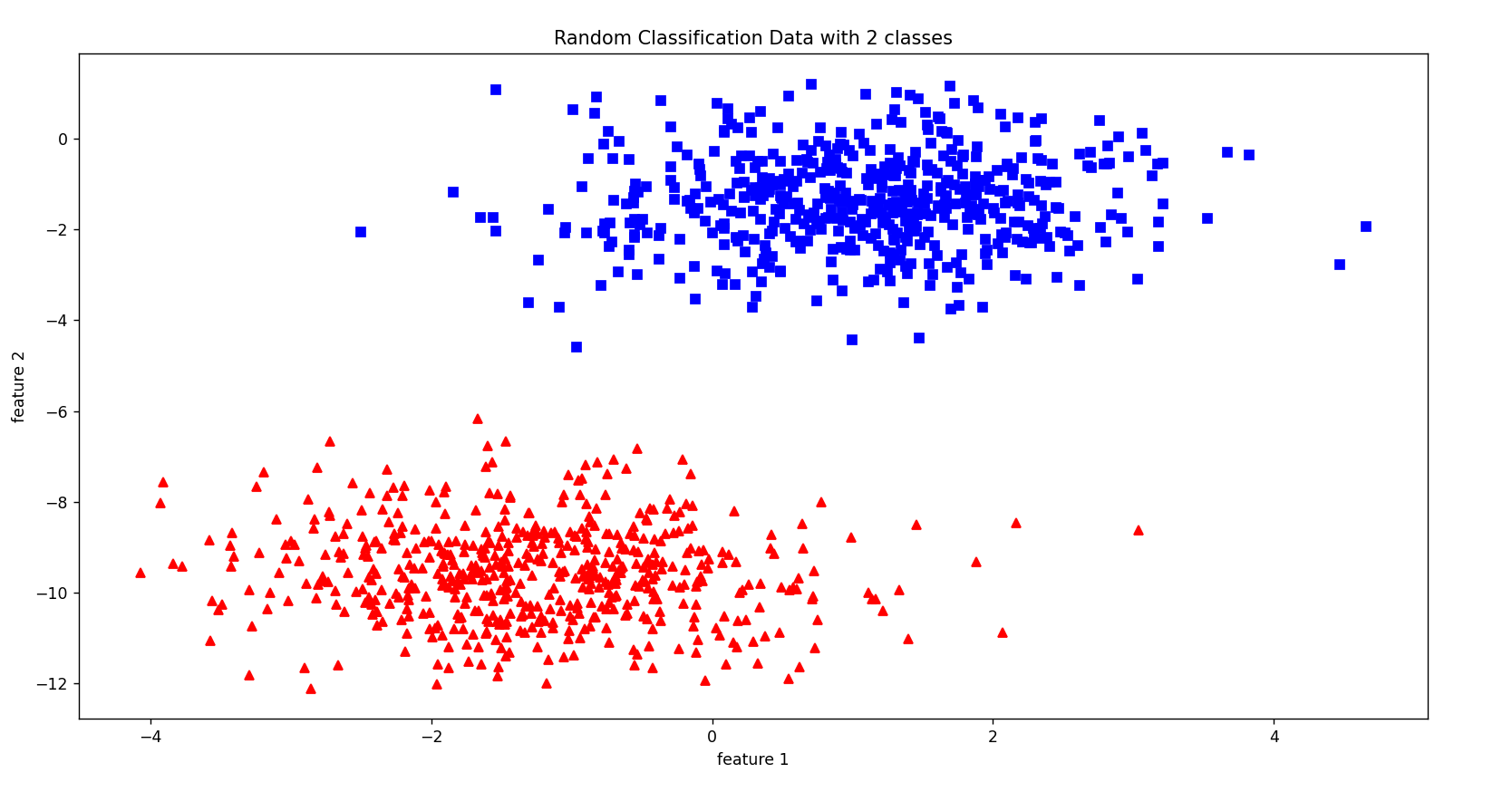
    plt.plot(x1, x2, 'y-')

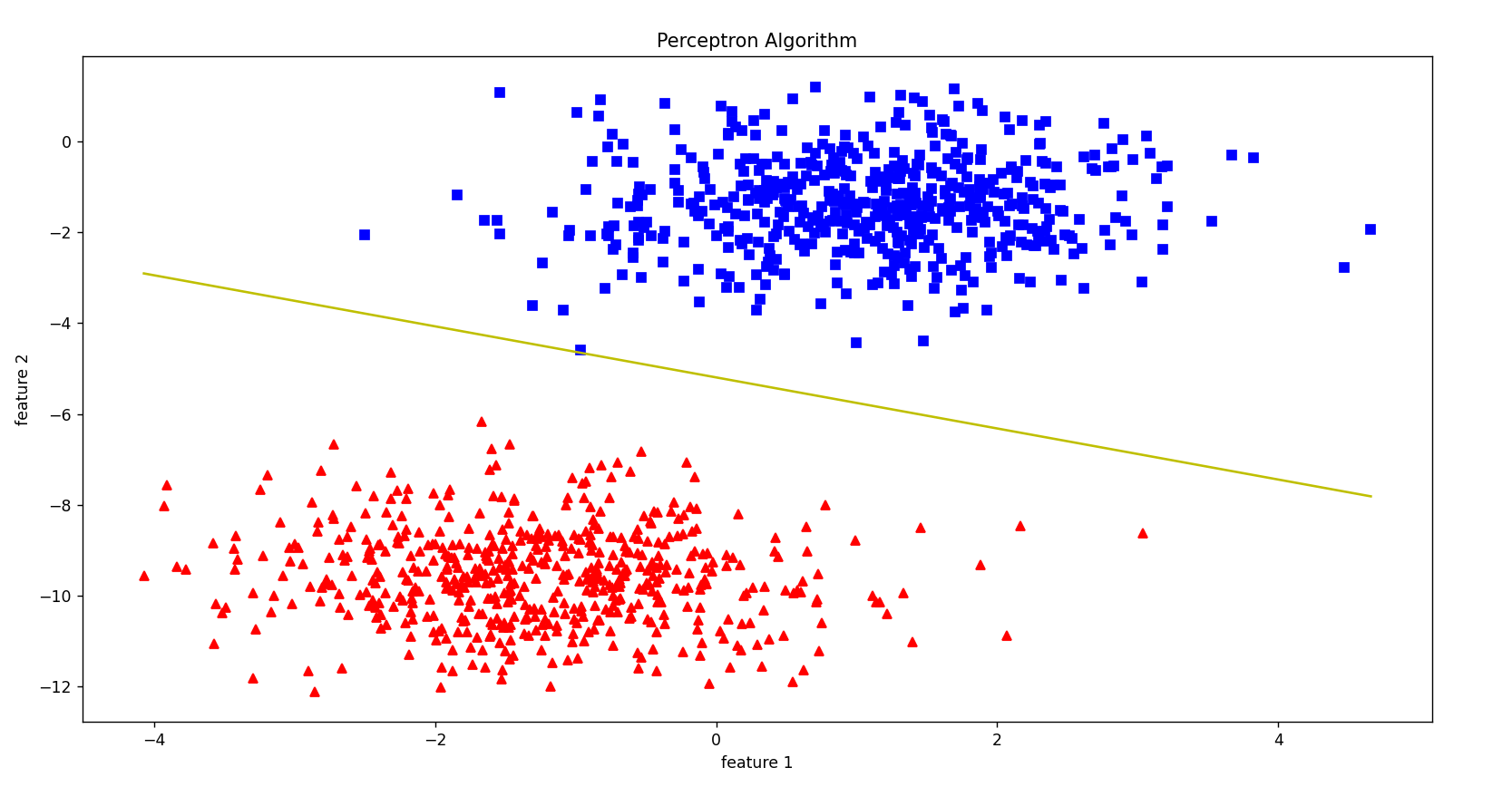
    plt.show()

theta, miss\_l = perceptron(X, y, 0.5, 100)

plot\_decision\_boundary(X, theta)

***Output:***

******

******

***Conclusion:***

*Implemented the Perceptron Algorithm.*