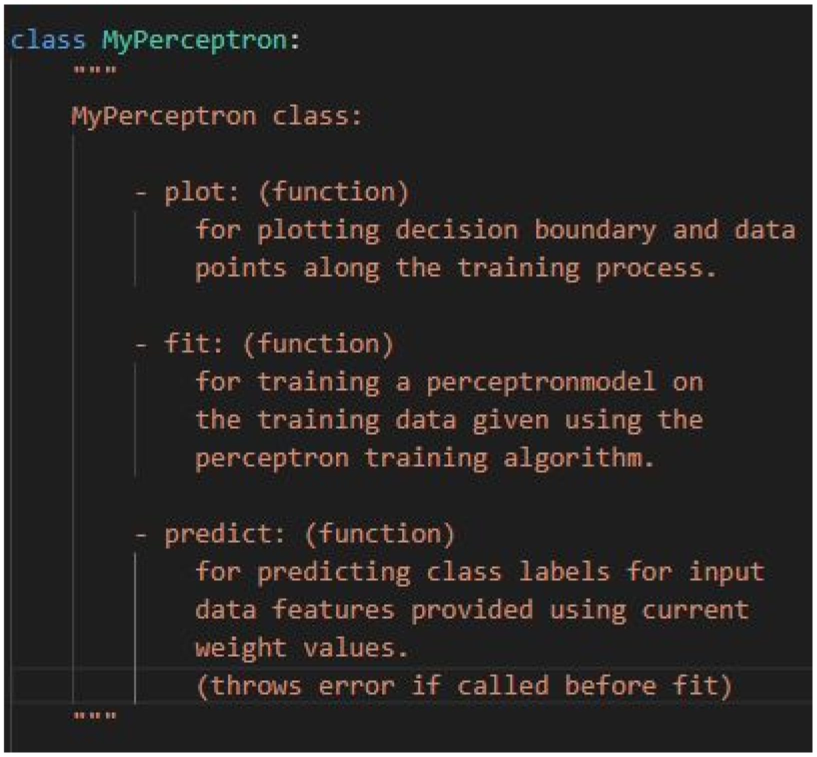
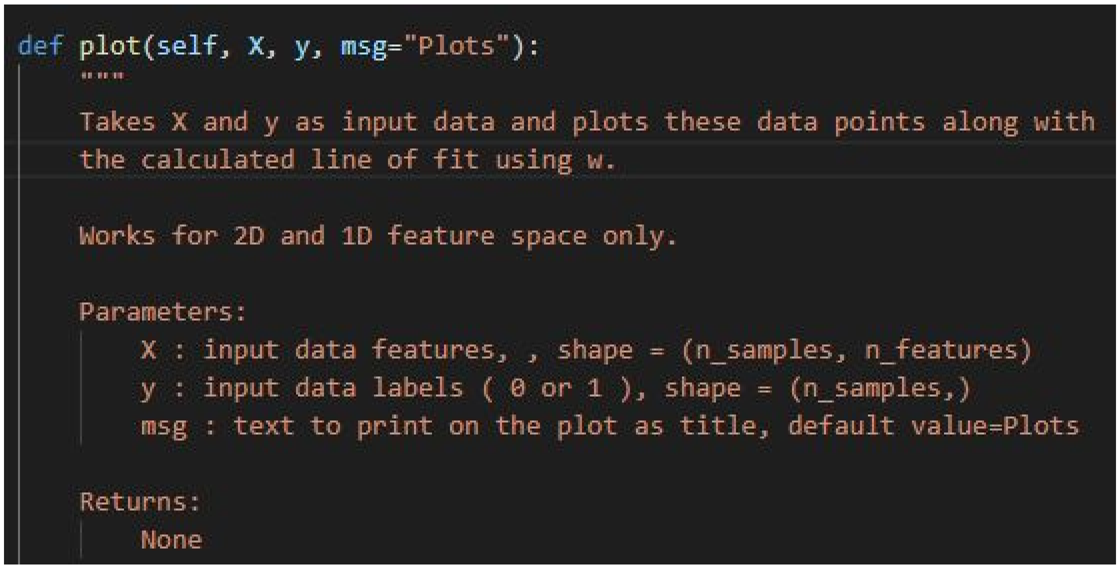
1. Implement a perceptron training algorithm.

- **Methodology**
  - Created a class MyPerceptron defining minimal requirements.



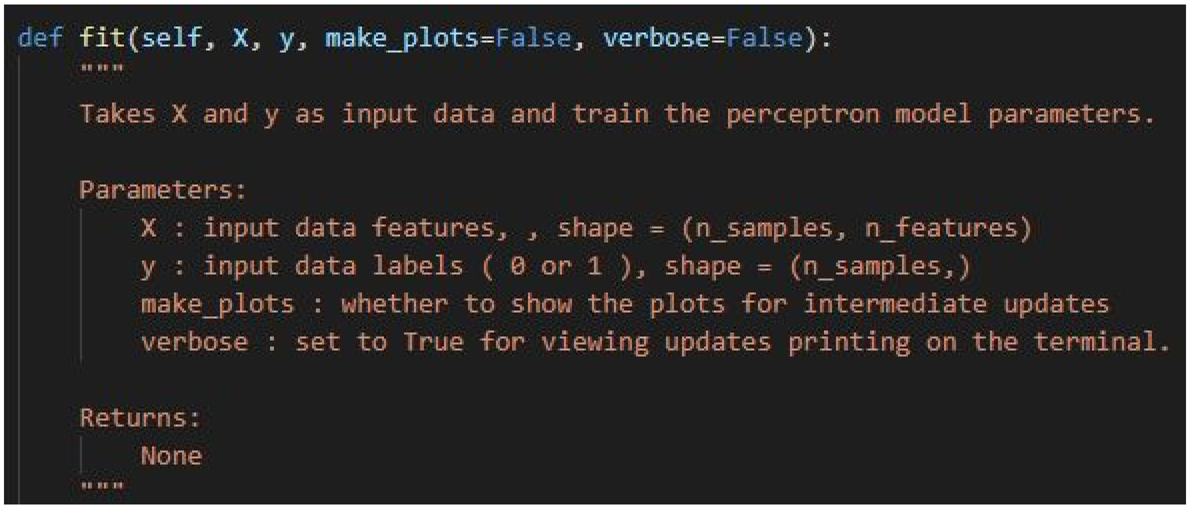
```python
class MyPerceptron:
    """
    MyPerceptron class:

        - plot: (function)
            for plotting decision boundary and data
            points along the training process.

        - fit: (function)
            for training a perceptronmodel on
            the training data given using the
            perceptron training algorithm.

        - predict: (function)
            for predicting class labels for input
            data features provided using current
            weight values.
            (throws error if called before fit)
    """
```

  - The plot function for 1b part.



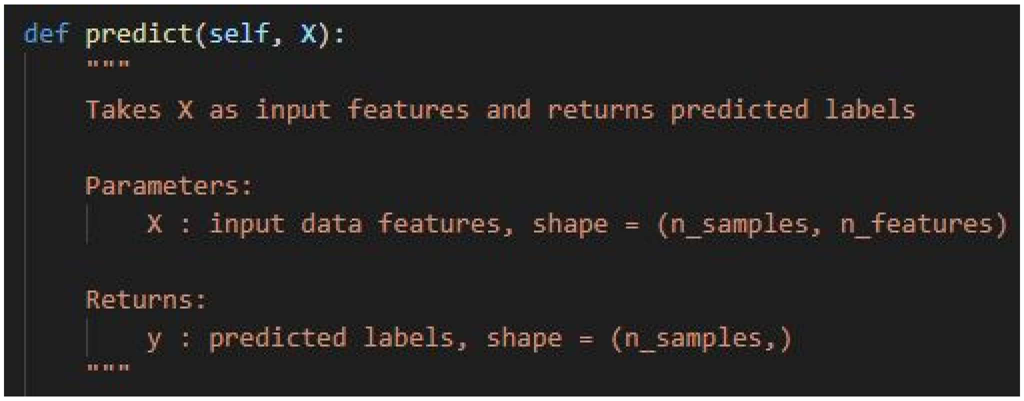
```python
def plot(self, X, y, msg="Plots"):
    """
    Takes X and y as input data and plots these data points along with
    the calculated line of fit using w.

    Works for 2D and 1D feature space only.

    Parameters:
        X : input data features, , shape = (n_samples, n_features)
        y : input data labels ( 0 or 1 ), shape = (n_samples,)
        msg : text to print on the plot as title, default value=Plots

    Returns:
        None
```

  - The fit function for training the data using the perceptron training algorithm.



```
def fit(self, X, y, make_plots=False, verbose=False):
    """
    Takes X and y as input data and train the perceptron model parameters.

    Parameters:
        X : input data features, , shape = (n_samples, n_features)
        y : input data labels ( 0 or 1 ), shape = (n_samples,)
        make_plots : whether to show the plots for intermediate updates
        verbose : set to True for viewing updates printing on the terminal.

    Returns:
        None
    """
```

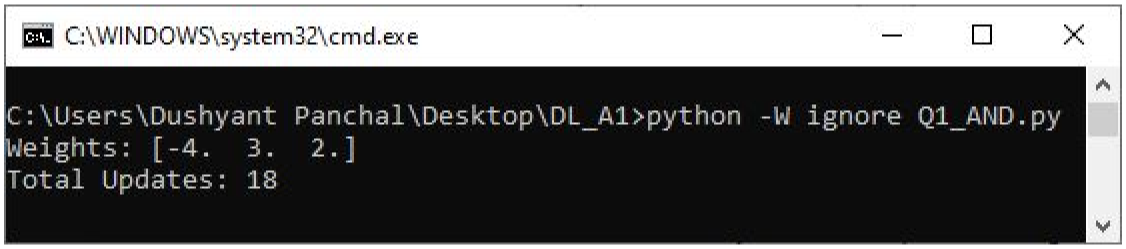
- The predict function for predicting labels for some other data samples.



```
def predict(self, X):
    """
    Takes X as input features and returns predicted labels

    Parameters:
        X : input data features, shape = (n_samples, n_features)

    Returns:
        y : predicted labels, shape = (n_samples,)
    """
```

a. Compute 2-variables AND, OR, and NOT (1-variable) operations and report the number of steps (number of weight-updates) required for the convergence.

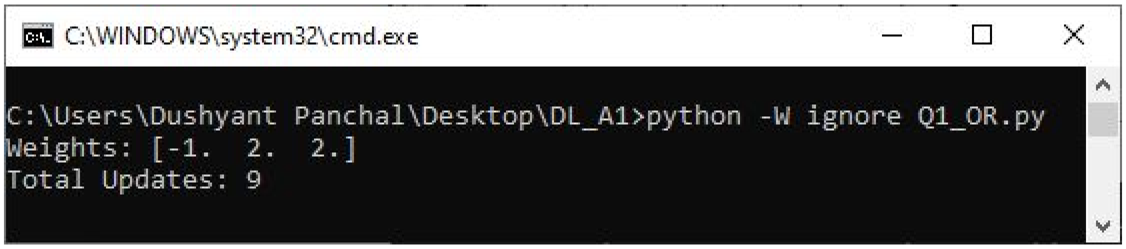
   i. AND



```
C:\Users\Dushyant Panchal\Desktop\DL_A1>python -W ignore Q1_AND.py
Weights: [-4.  3.  2.]
Total Updates: 18
```

Note: The weights are in the order b, w1, w2

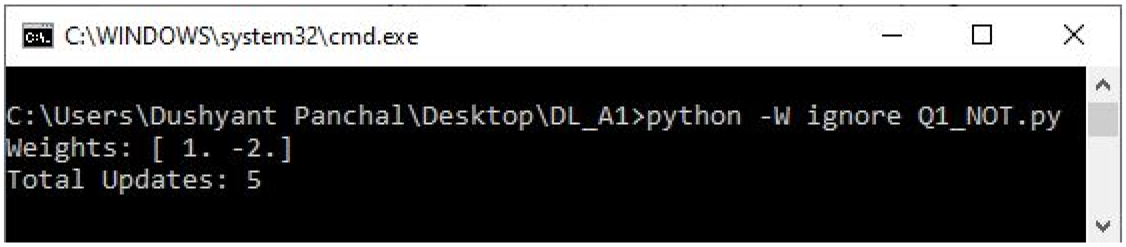
   ii. OR



```
C:\Users\Dushyant Panchal\Desktop\DL_A1>python -W ignore Q1_OR.py
Weights: [-1.  2.  2.]
Total Updates: 9
```

Note: The weights are in the order b, w1, w2

iii. NOT



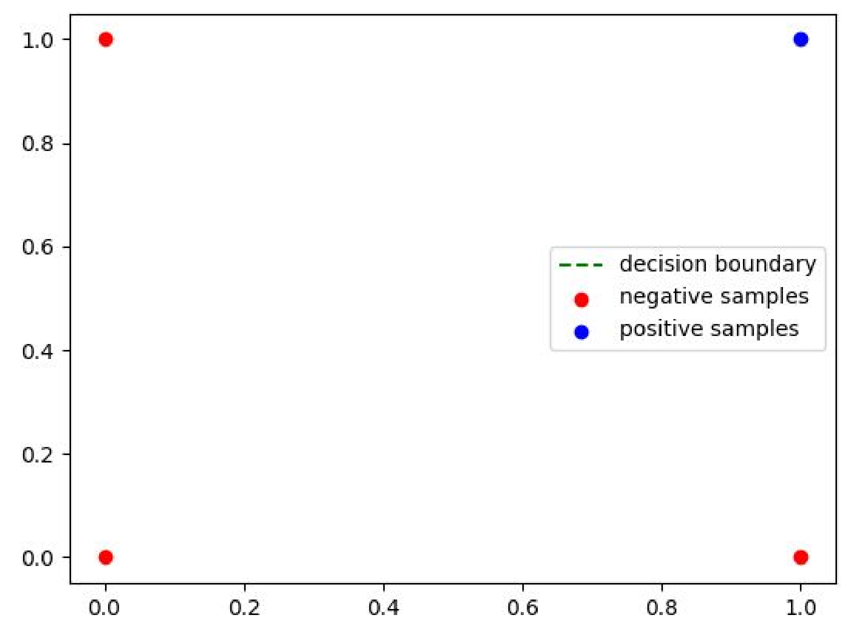
```
C:\Users\Dushyant Panchal\Desktop\DL_A1>python -W ignore Q1_NOT.py
Weights: [ 1. -2.]
Total Updates: 5
```

Note: The weights are in the order b, w1

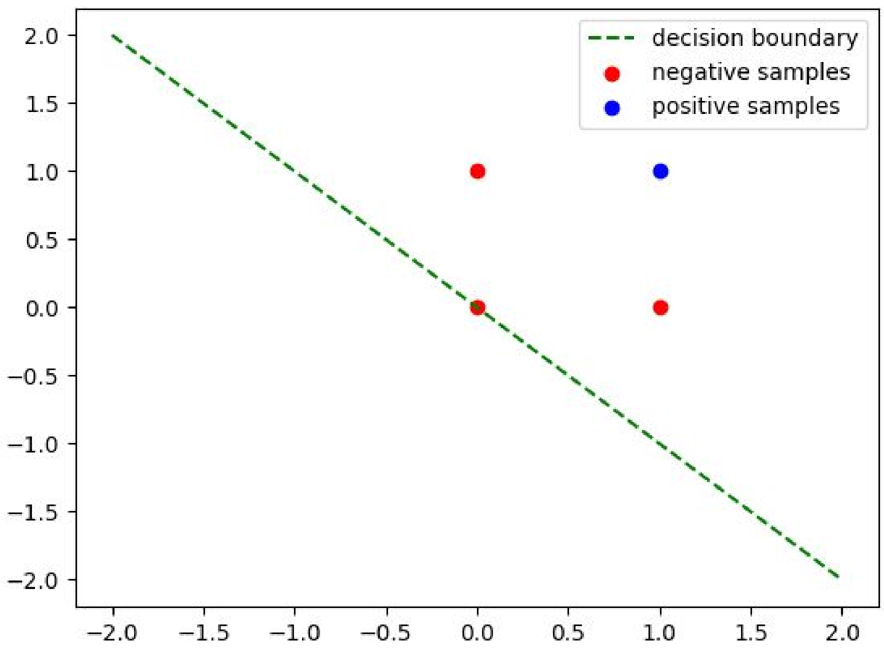
b. Draw the decision boundary at each step of learning. (Note: Decision boundary missing means all the weights are zero at that moment. The bias need not be zero though.)

i. AND

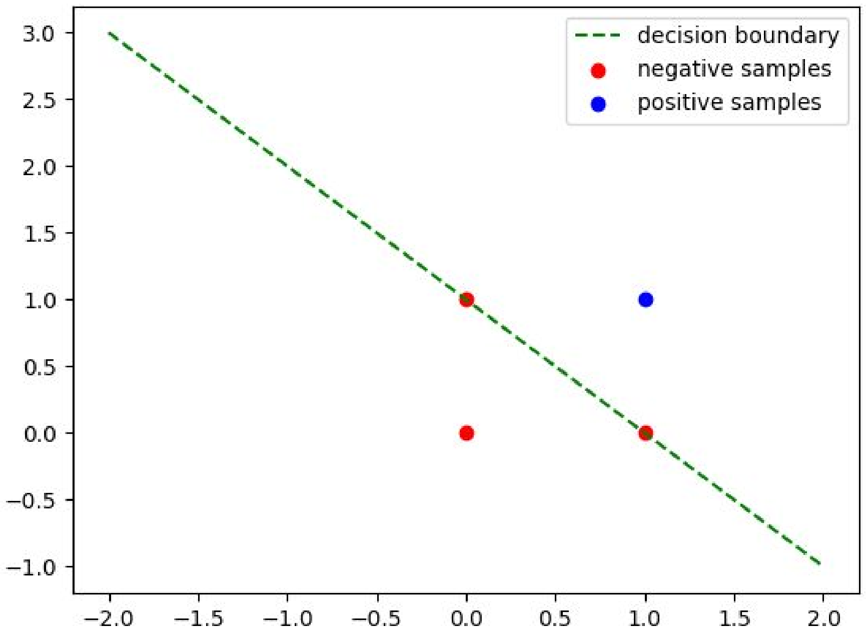
Iteration 1, Sample 1/4



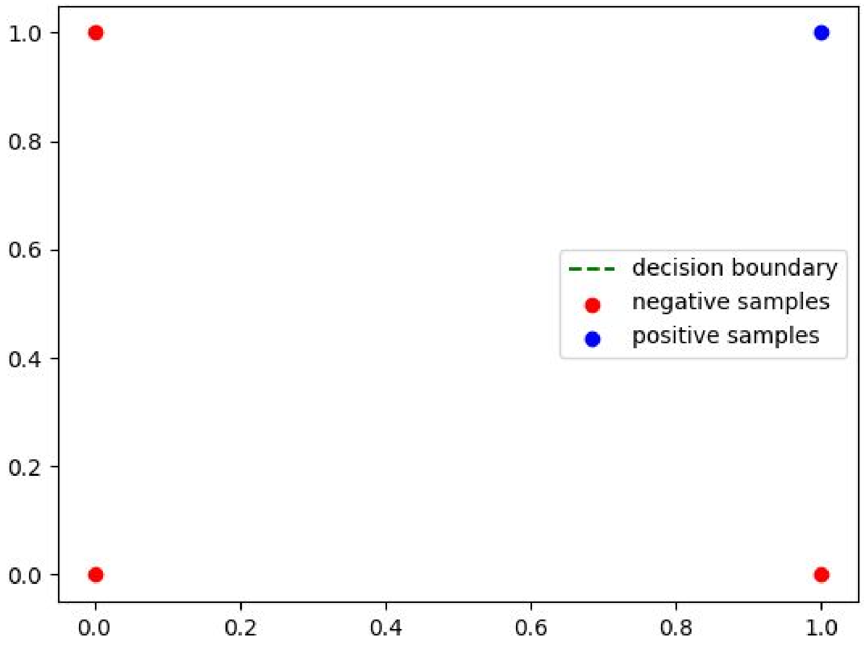
Iteration 1, Sample 4/4



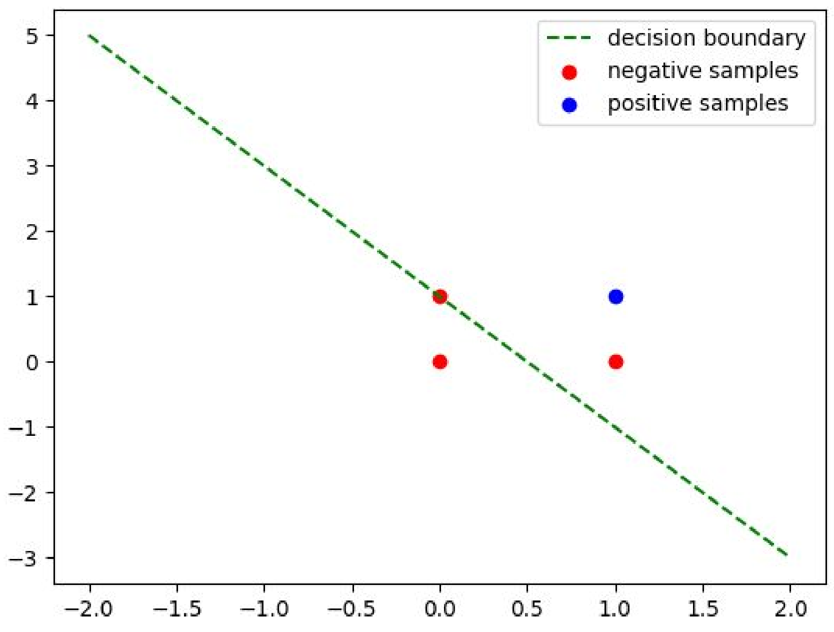
Iteration 2, Sample 1/4



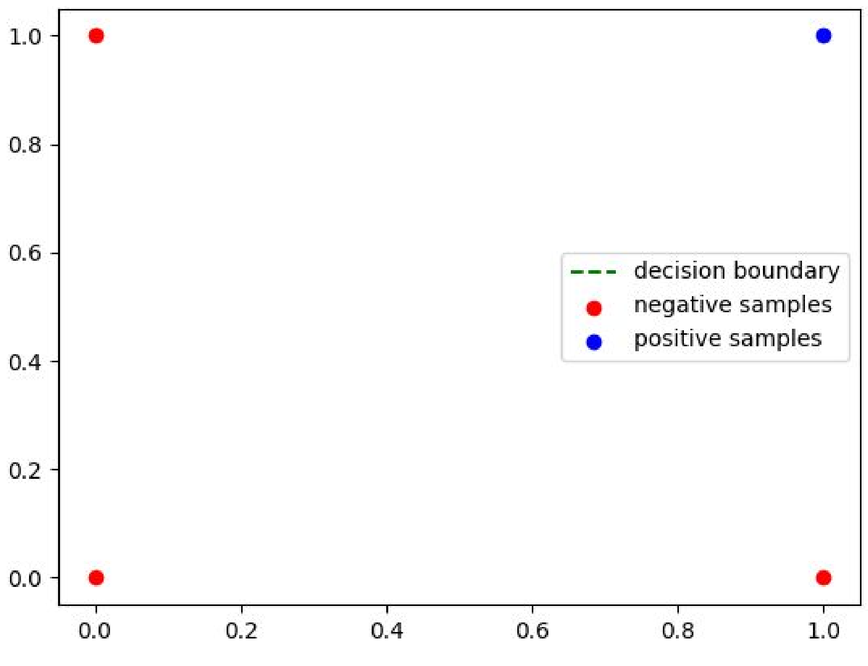
Iteration 2, Sample 2/4



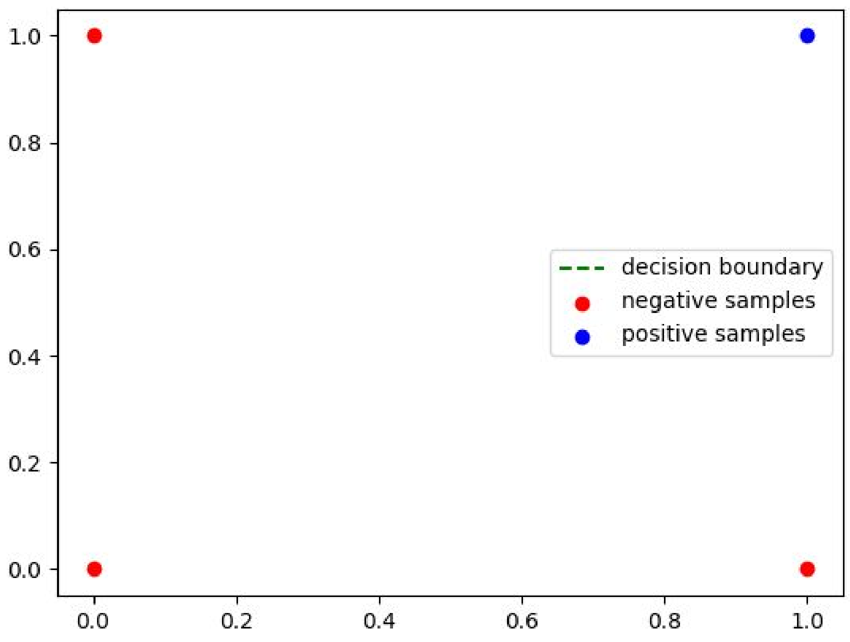
Iteration 2, Sample 4/4



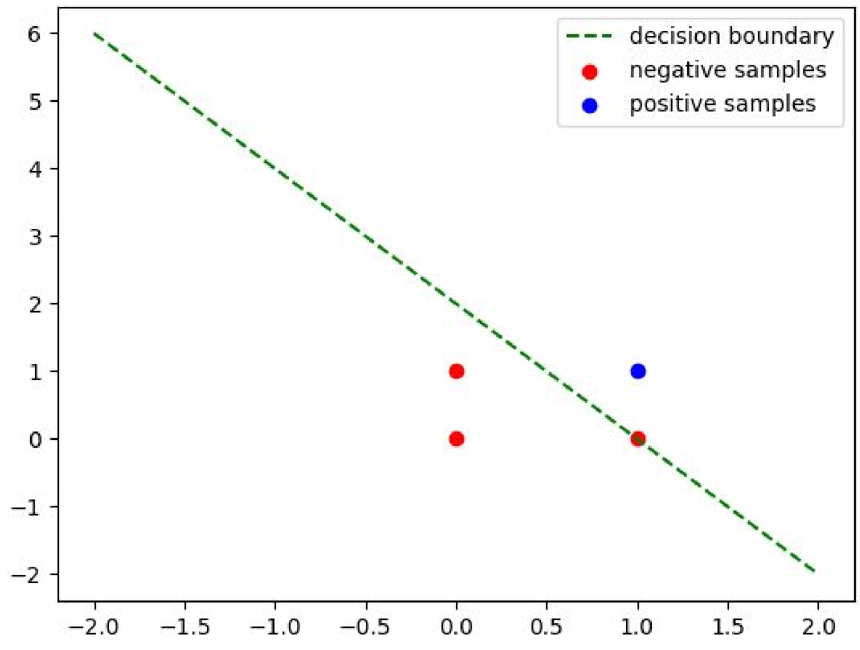
Iteration 3, Sample 2/4



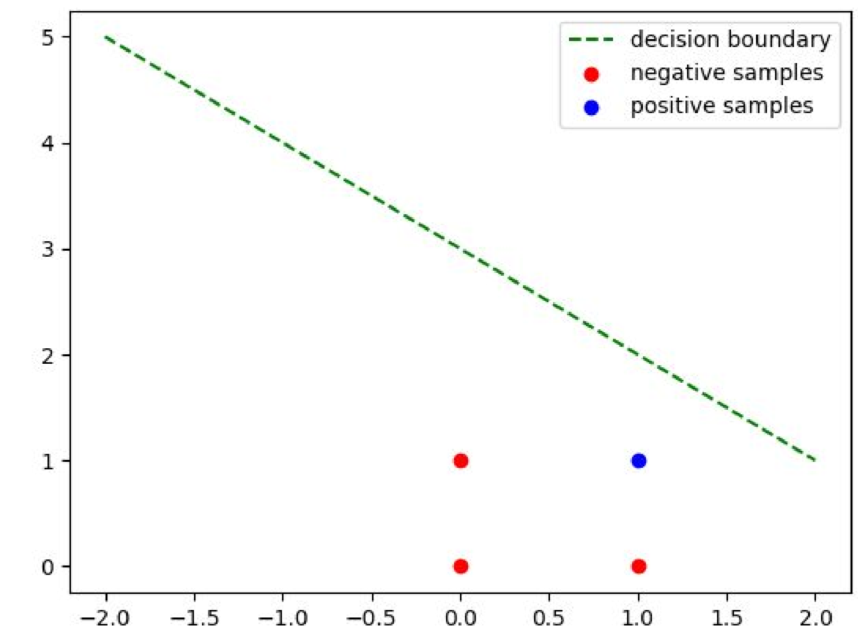
Iteration 3, Sample 3/4



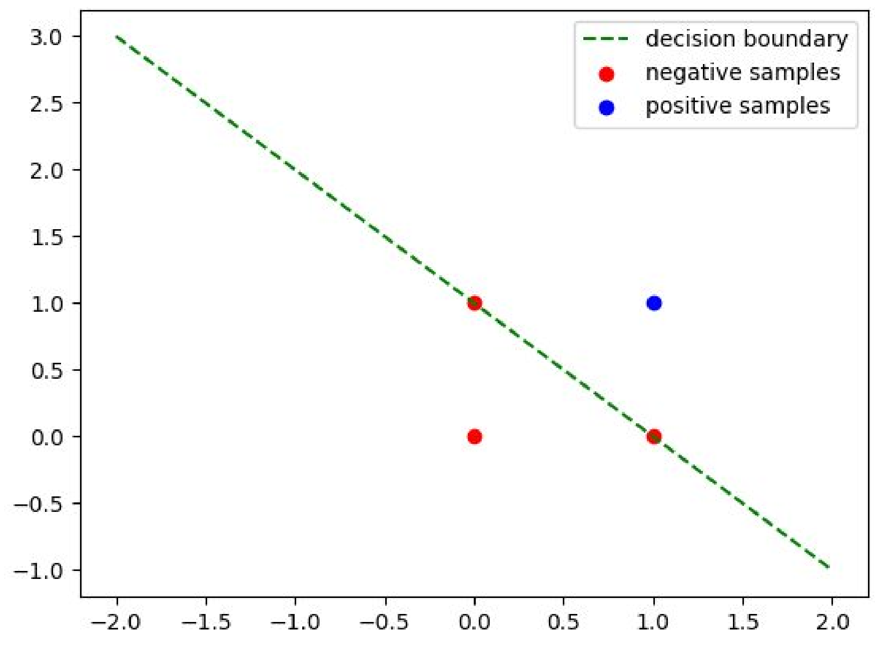
Iteration 3, Sample 4/4



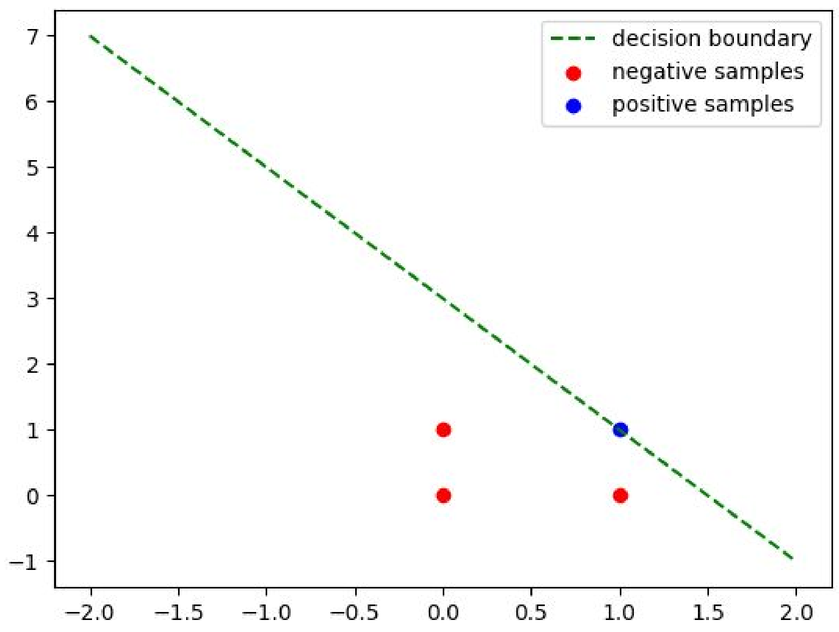
Iteration 4, Sample 3/4



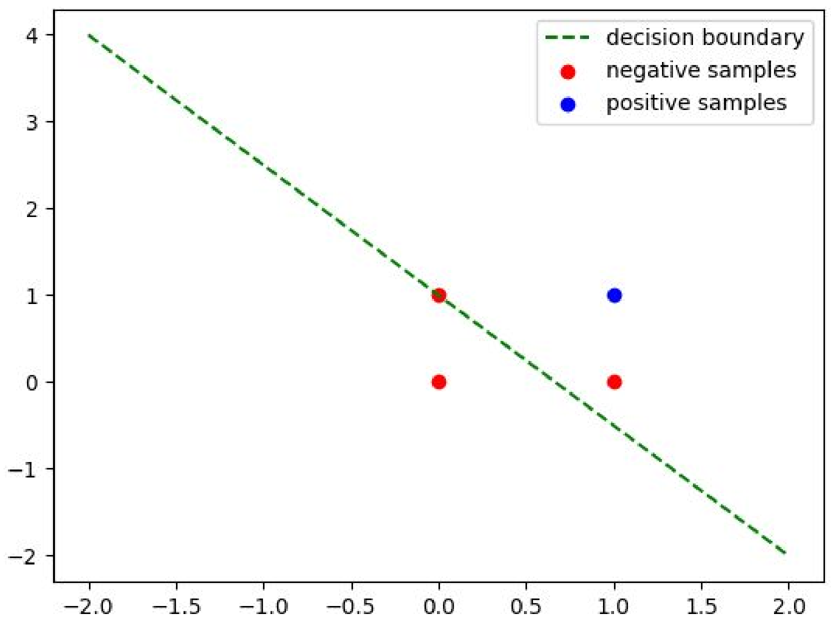
Iteration 4, Sample 4/4



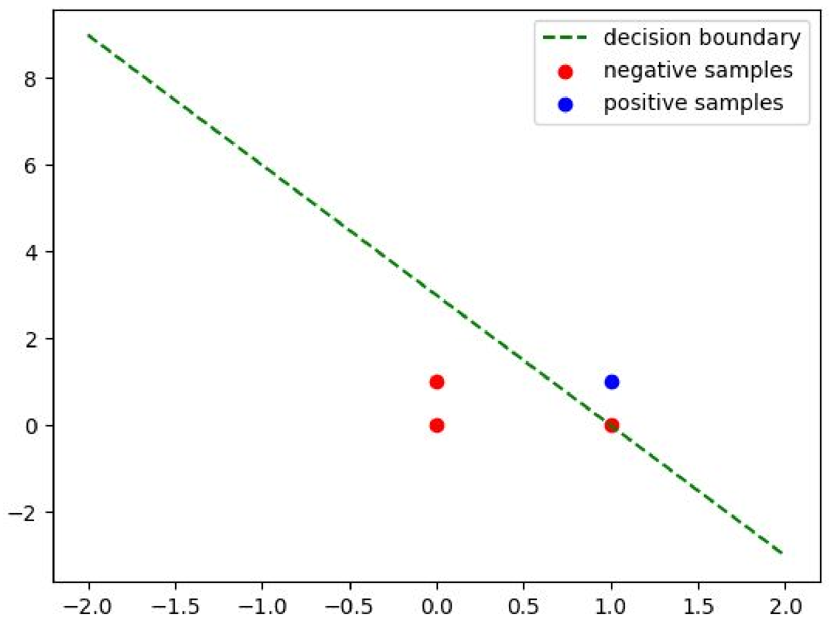
Iteration 5, Sample 2/4



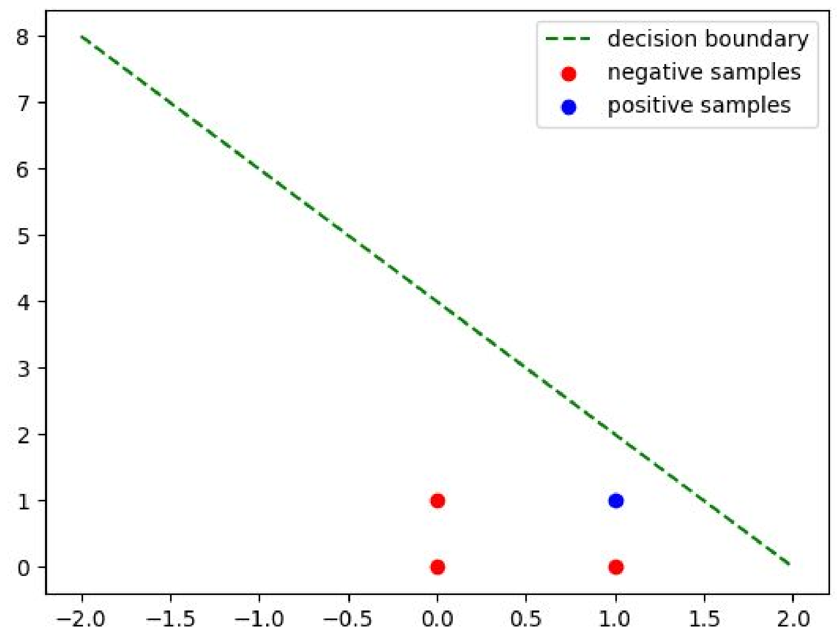
Iteration 5, Sample 4/4



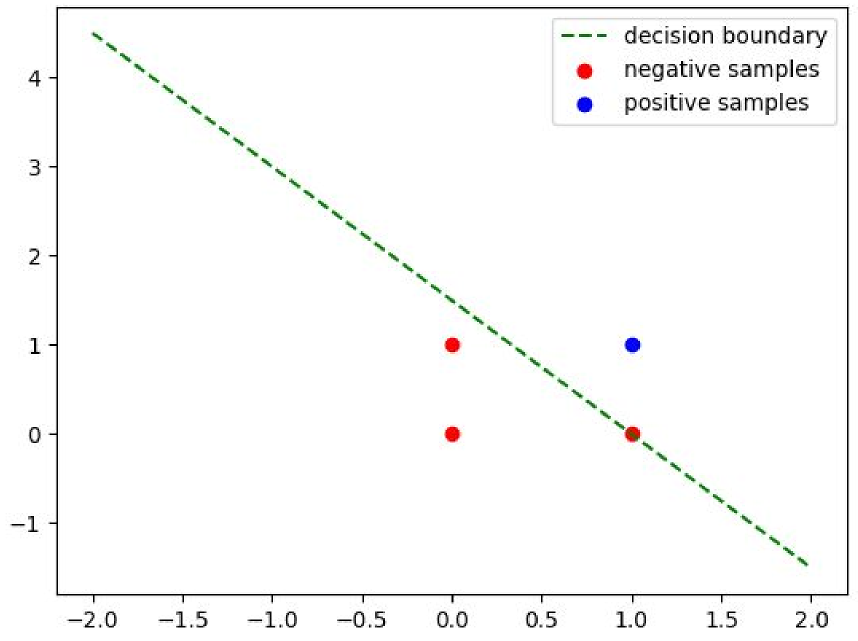
Iteration 6, Sample 2/4



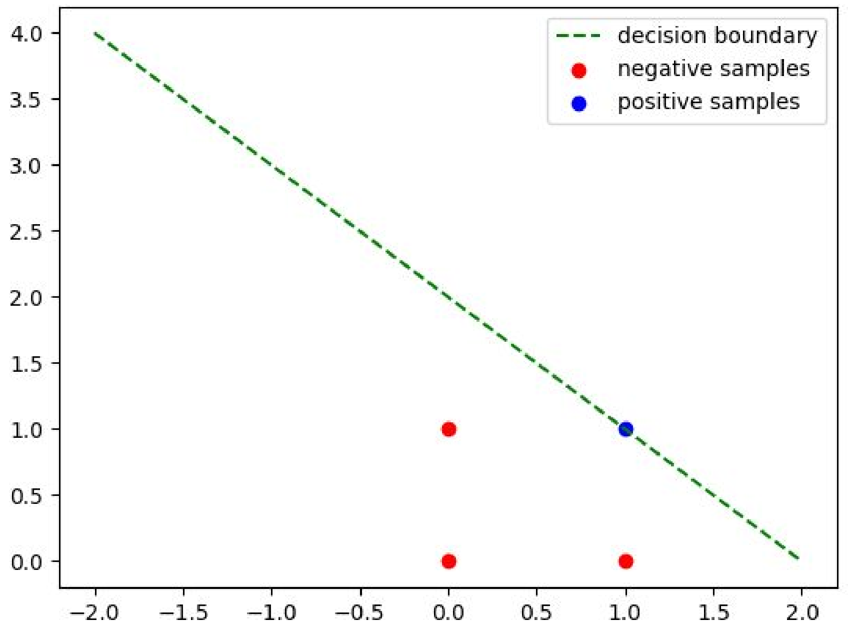
Iteration 6, Sample 3/4



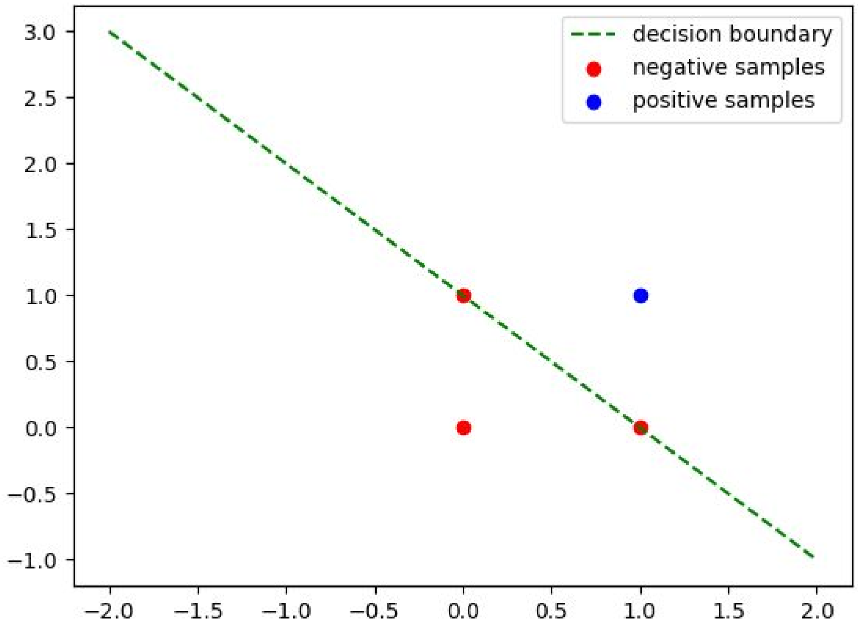
Iteration 6, Sample 4/4



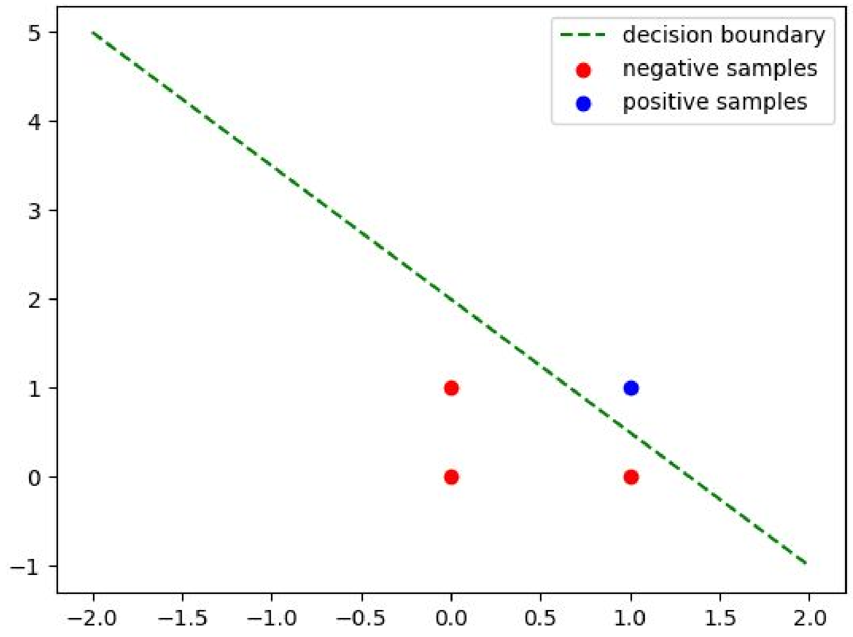
Iteration 7, Sample 3/4



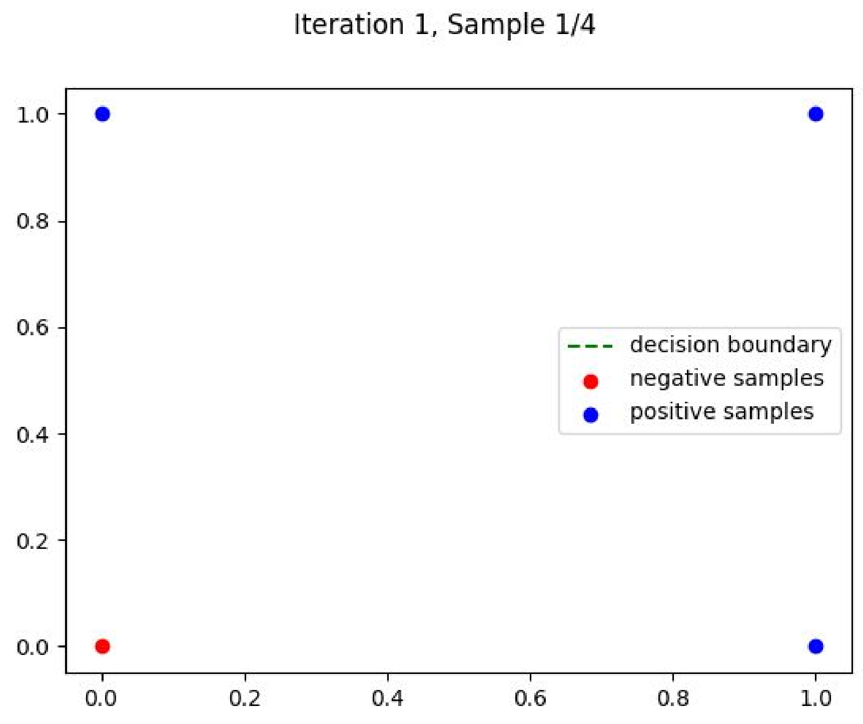
Iteration 7, Sample 4/4

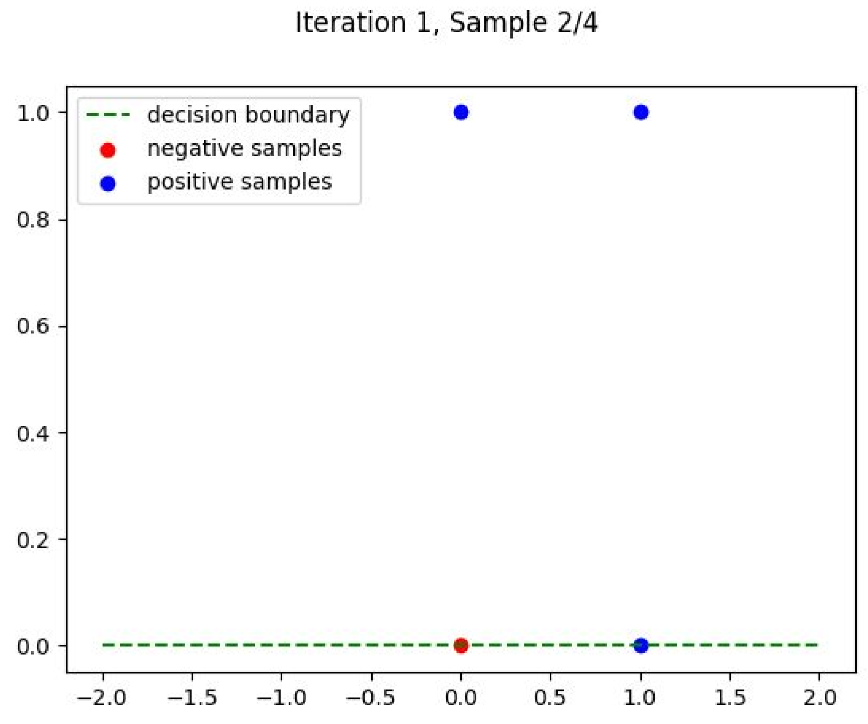


Iteration 8, Sample 2/4

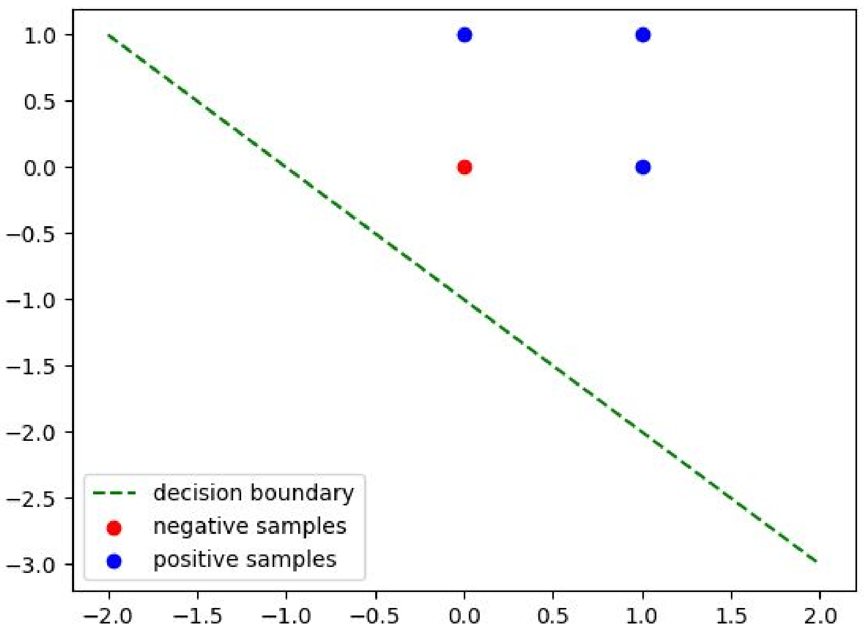


ii. OR

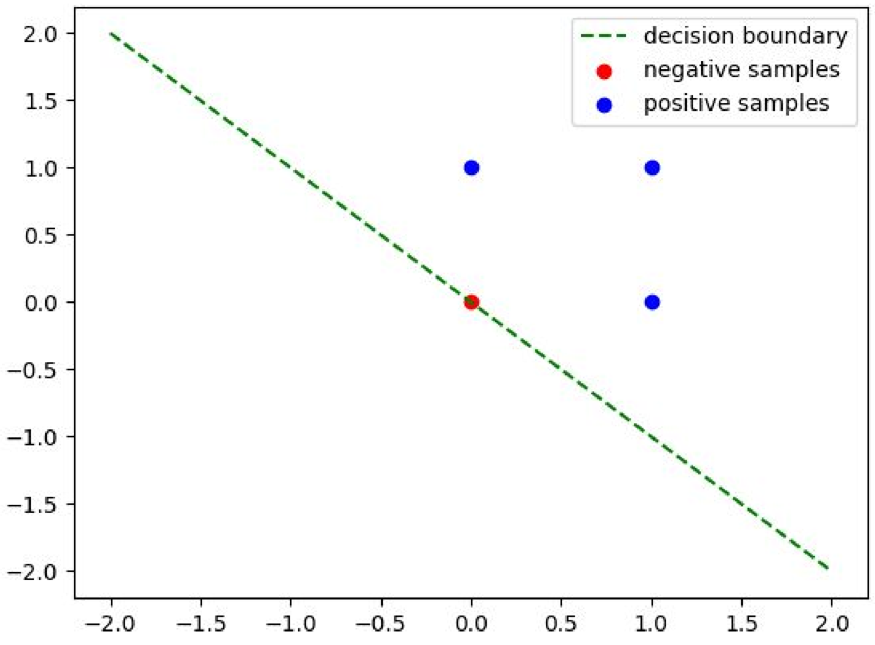

Iteration 1, Sample 1/4
decision boundary
negative samples
positive samples
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0


Iteration 1, Sample 2/4
decision boundary
negative samples
positive samples
1.0
0.8
0.6
0.4
0.2
0.0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

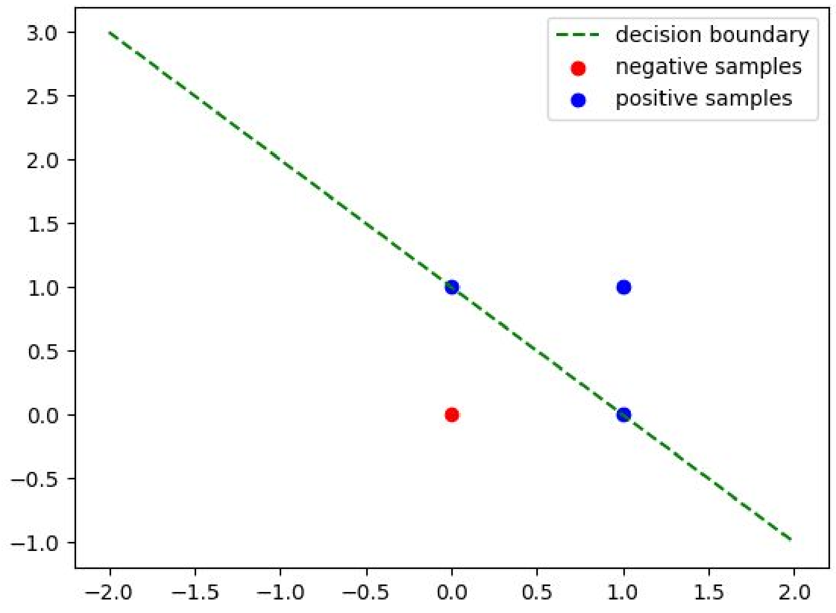
Iteration 1, Sample 3/4



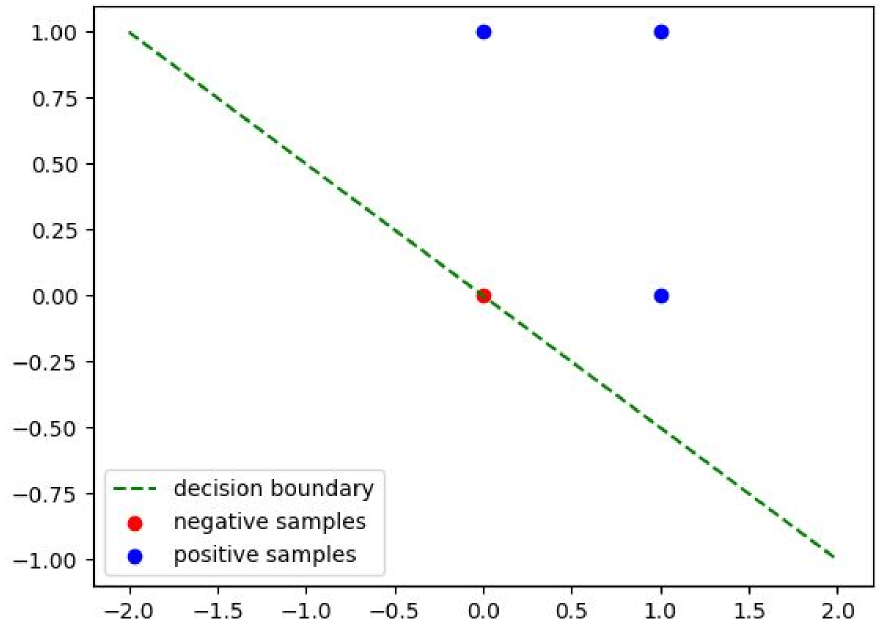
Iteration 2, Sample 1/4



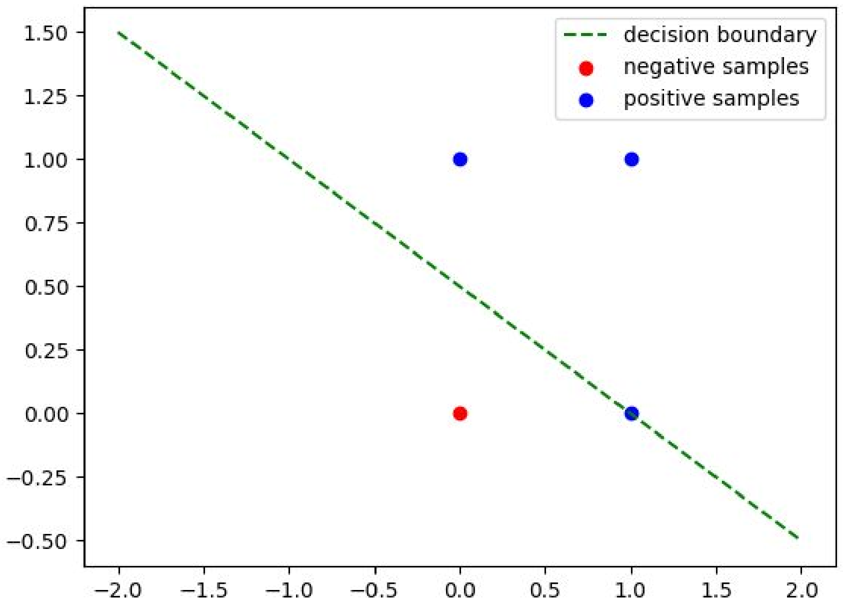
Iteration 3, Sample 1/4



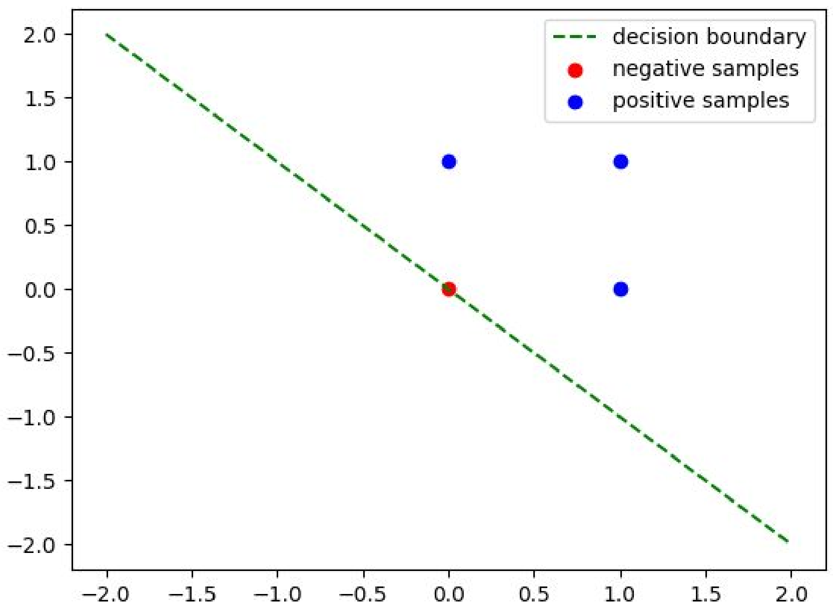
Iteration 3, Sample 2/4



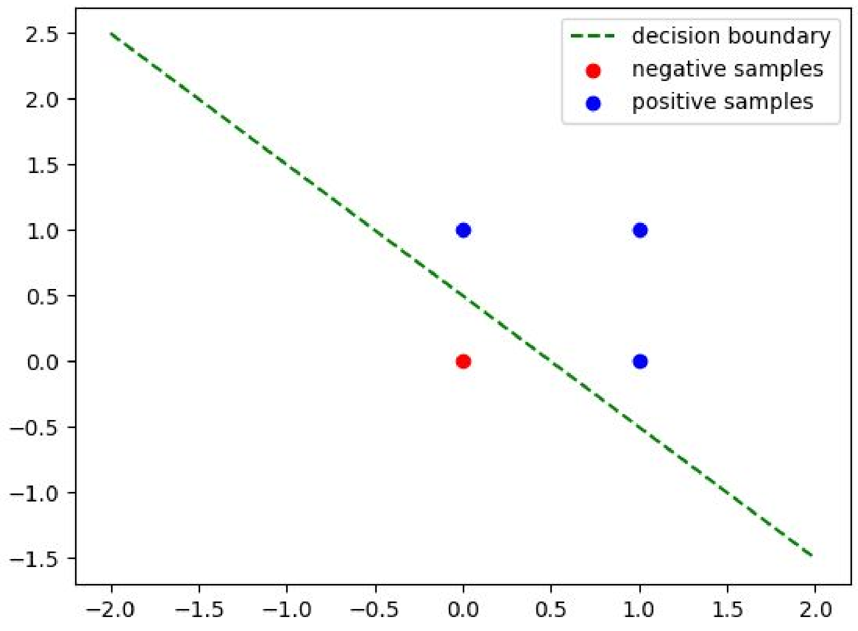
Iteration 4, Sample 1/4



Iteration 4, Sample 3/4

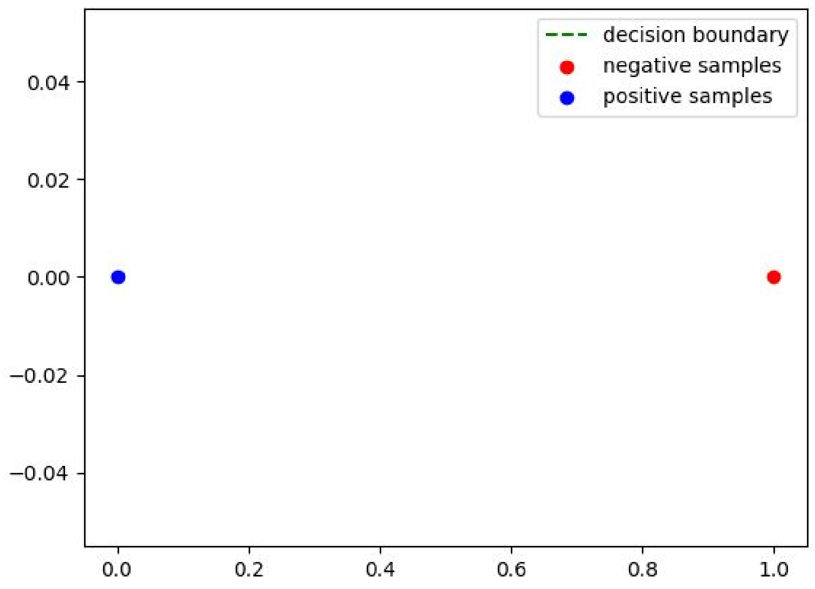


Iteration 5, Sample 1/4

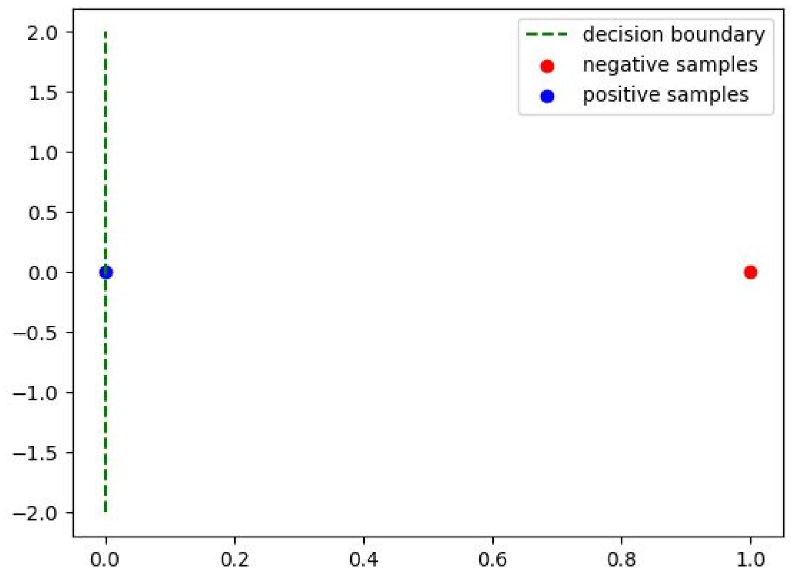


iii. NOT

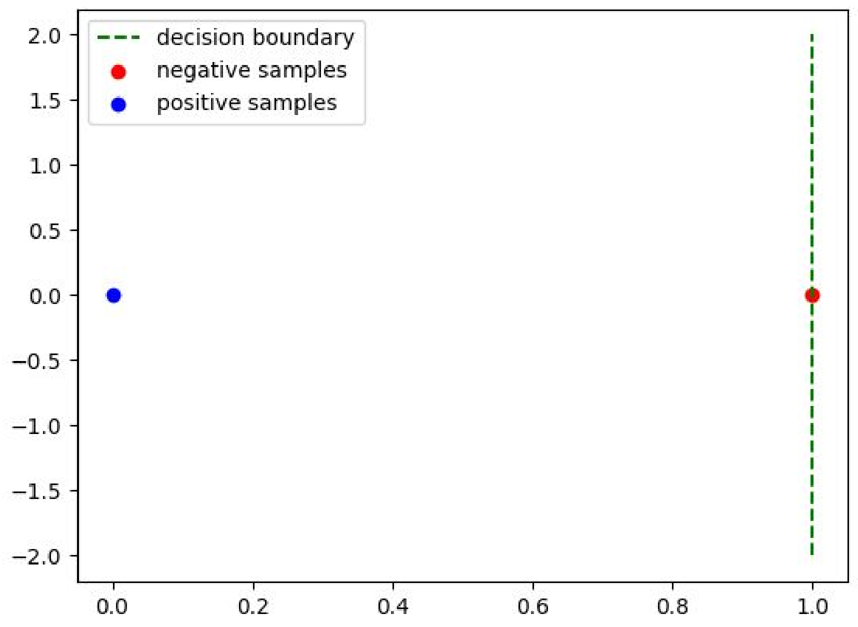
Iteration 1, Sample 1/2



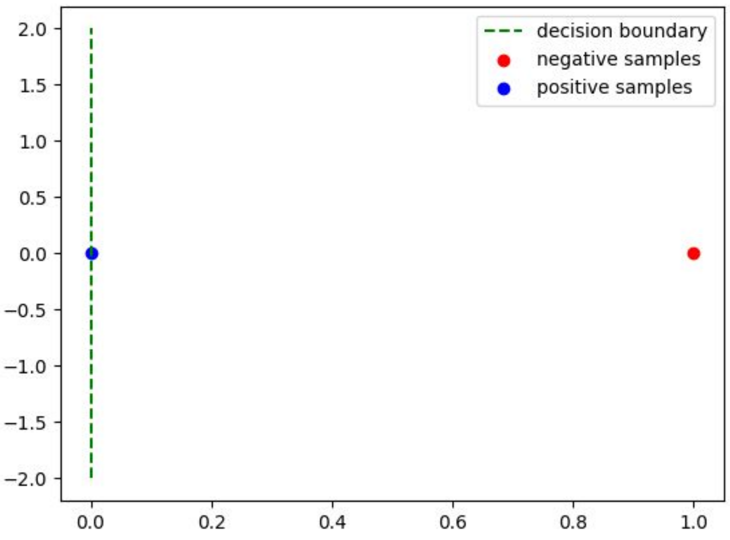
Iteration 1, Sample 2/2



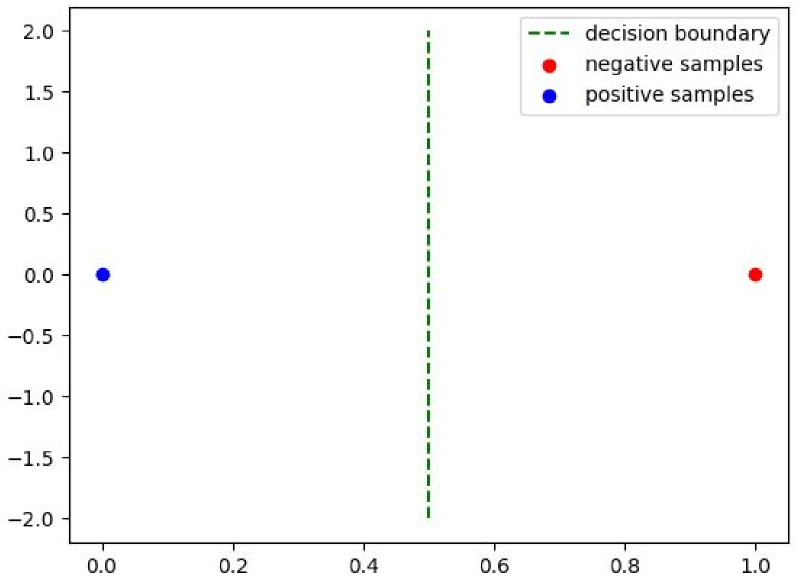
Iteration 2, Sample 1/2



Iteration 2, Sample 2/2

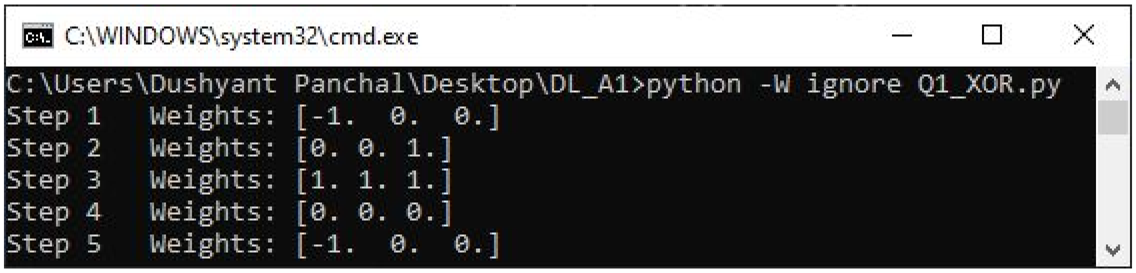


Iteration 3, Sample 1/2

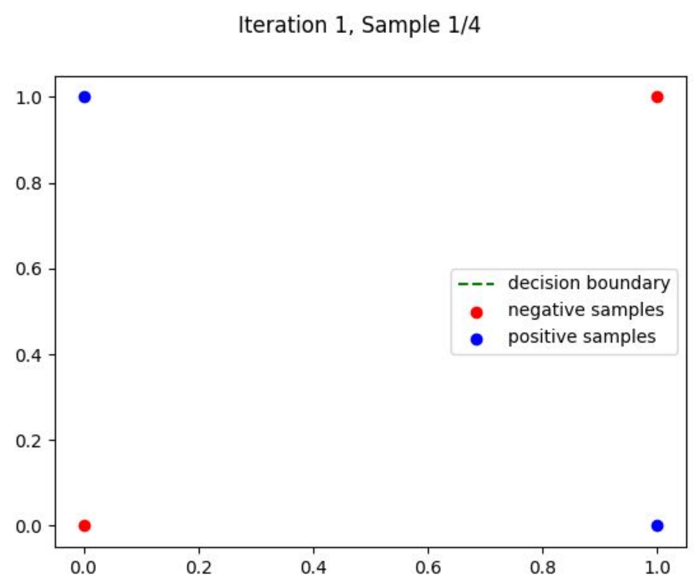


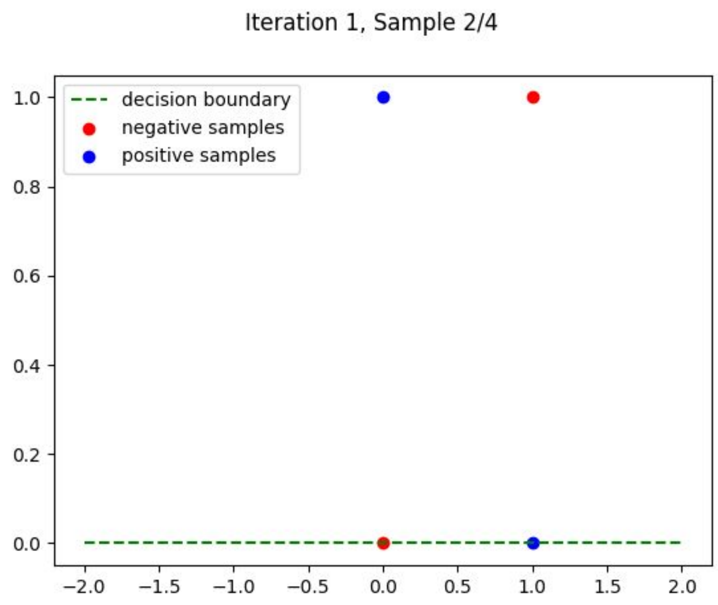
c. **Prove that XOR can't be computed using PTA. How many minimum steps do you need to prove it? (Use PTA convergence theorem to verify it.)**

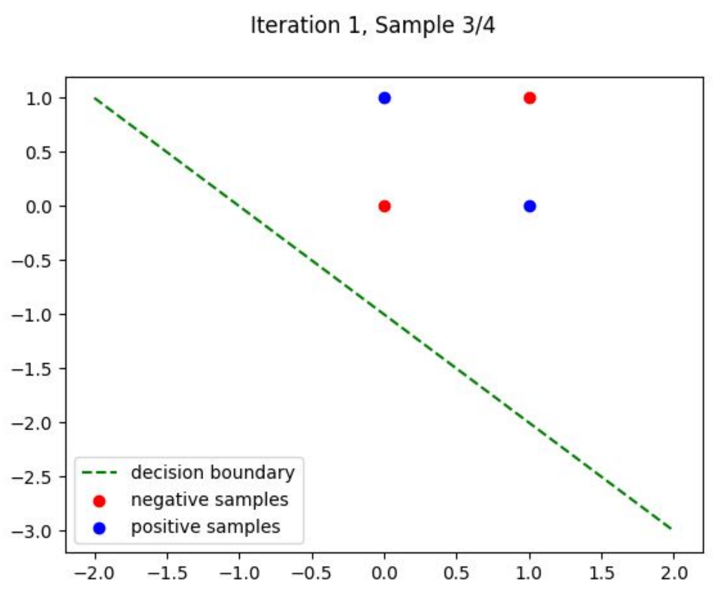
- **4 steps** were enough to identify that after every 4 updates the weights returned to the state they were 4 updates before.
- The following verbose clearly shows that the forth update rendered the weights to be all zeros, which is exactly what they were initialized to.
- As this cycle keeps on repeating itself, the perceptron never converges in a finite number of updates.
- Perceptron Training algorithm states that there is an upper bound on the number of mistakes or updates that the network has to do to achieve convergence. However, we know that in our case, this upper bound can only by +infinity (as the updates will keep oscillating at a cycle of 4 updates and not converge at all within finite number of steps).
- This violates the PTA and hence we can say that the XOR function can't be computed using PTA.
- Also, we know that perceptron can only solve the class of linearly separable problems, which the XOR problem does not belong to.

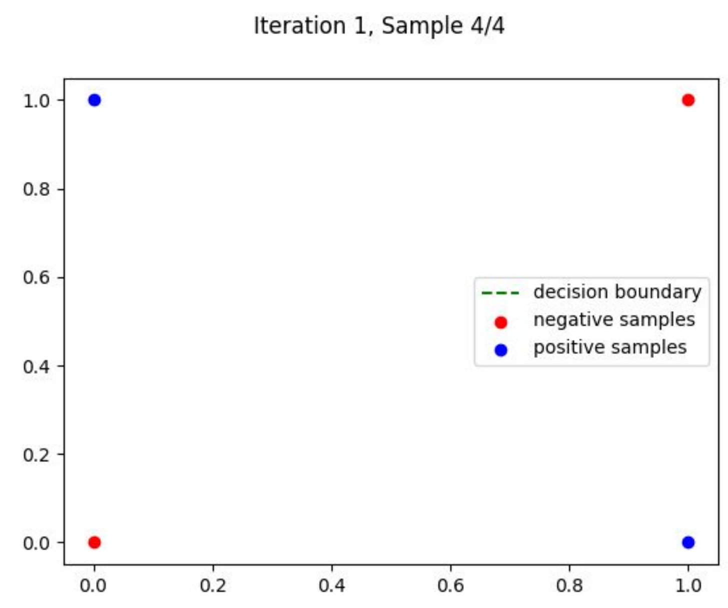


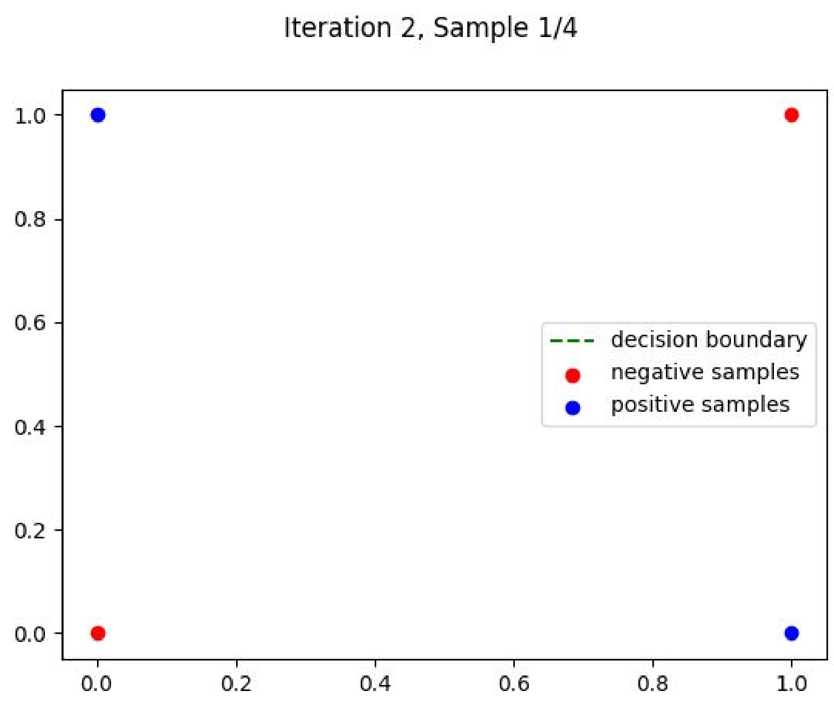
```
C:\Users\Dushyant Panchal\Desktop\DL_A1>python -W ignore Q1_XOR.py
Step 1   Weights: [-1.  0.  0.]
Step 2   Weights: [0. 0. 1.]
Step 3   Weights: [1. 1. 1.]
Step 4   Weights: [0. 0. 0.]
Step 5   Weights: [-1.  0.  0.]
```


Iteration 1, Sample 1/4
decision boundary
negative samples
positive samples
0.0
0.2
0.4
0.6
0.8
1.0


Iteration 1, Sample 2/4
decision boundary
negative samples
positive samples
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0


Iteration 1, Sample 3/4
decision boundary
negative samples
positive samples
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

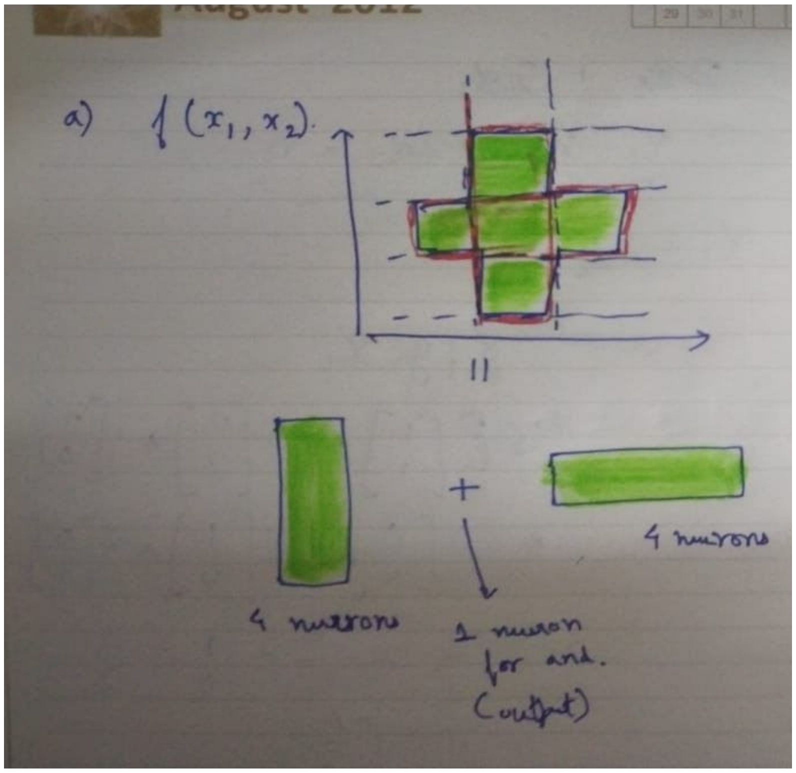


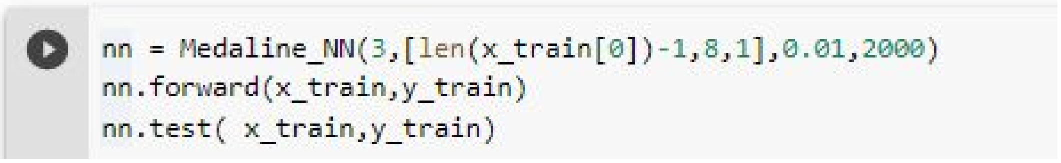


2. Using the Madeleine learning algorithm compute the following two functions. Shaded regions are 1, rest are 0. Report the number of neurons for each case.

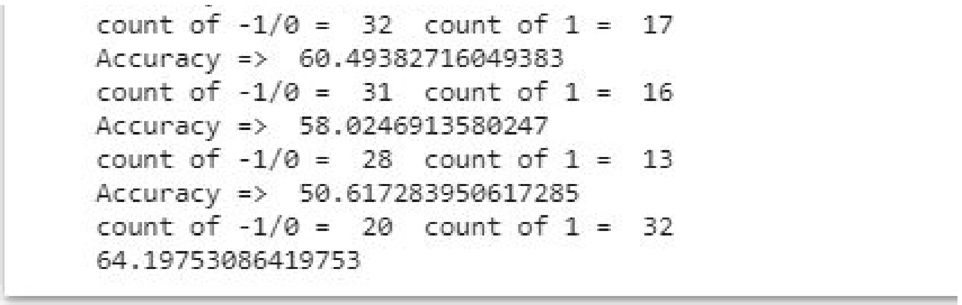
a. f1 (x1, x2)

The first function was implemented by using only 8 hidden units so the neural network contains **2,8,1** neuron units as input ,hidden and outputs respectively. The following requirement of only 8 hidden unit can be viewed through the following visualization:

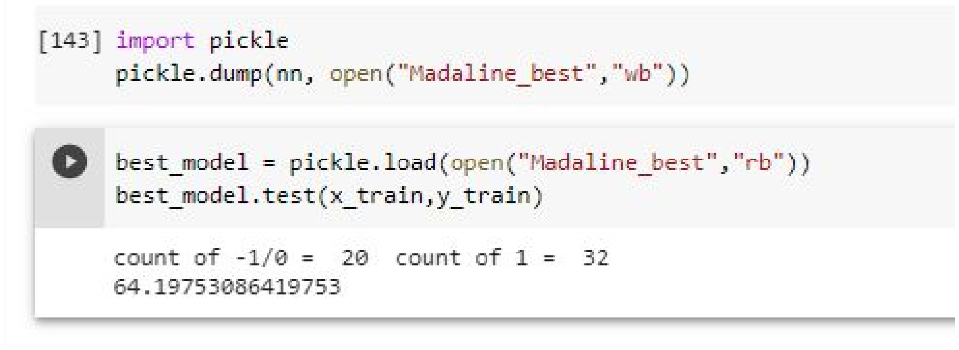




```
nn = Medaline_NN(3,[len(x_train[0])-1,8,1],0.01,2000)
nn.forward(x_train,y_train)
nn.test( x_train,y_train)
```



```
count of -1/0 =  32  count of 1 =  17
Accuracy =>  60.49382716049383
count of -1/0 =  31  count of 1 =  16
Accuracy =>  58.0246913580247
count of -1/0 =  28  count of 1 =  13
Accuracy =>  50.617283950617285
count of -1/0 =  20  count of 1 =  32
64.19753086419753
```


```
[143] import pickle
      pickle.dump(nn, open("Madaline_best","wb"))
```

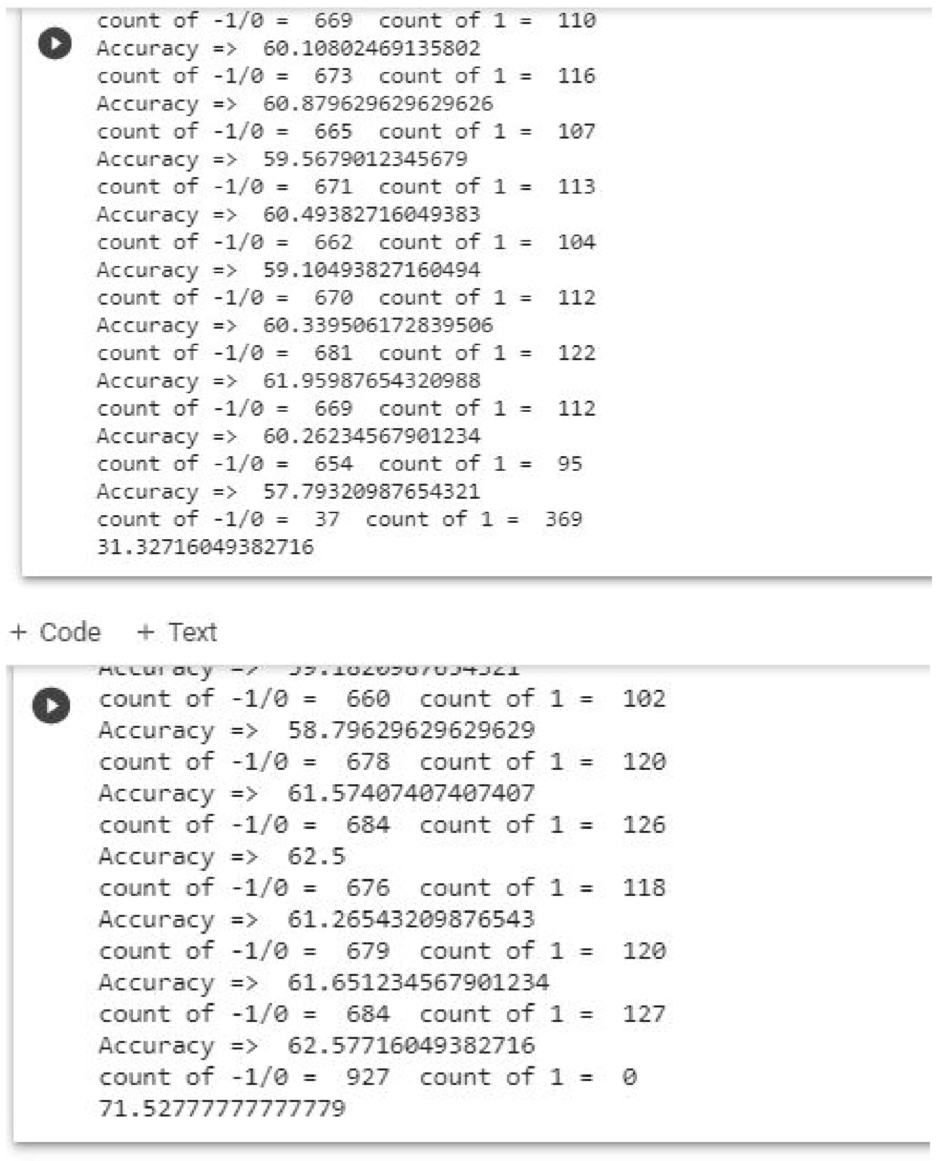
```
best_model = pickle.load(open("Madaline_best","rb"))
best_model.test(x_train,y_train)
```

```
count of -1/0 =  20  count of 1 =  32
64.19753086419753
```

And the best we achieved were at lr =0.01 and epochs = 2000 given below, Though there are many complications and thus we do not receive this type of results every time we run the code with the same parameters. It is due to the fact that

1) We enter in an exhausting cycle: The weights of the neurons responsible for changing the output of the final neuron are very small or very close to zero. So, when we receive 3 or 4 +1 input rows in a row, the weights completely shift from favouring -1 to +1 and vice versa for the opposite case. This limitation is also mentioned in the MR-II paper [https://www-isl.stanford.edu/people/widrow/papers/c1988madalinerule.pdf] and a solution was suggested to shuffle the input data rows which we tried but didn't receive promising results. (Mentioned in paper → Unfortunately, a cycle can develop where the same first-layer ADALINEs are adapted back and forth to accommodate different subsets of the training set. A way to break such cycles is to weight the confidence level of the ADALINEs by the number of times they have participated in successful trial adaptations. This will cause the order of consideration for trial adaptation to change. This forces other first-layer ADALINEs to assume responsibility for correct pattern mapping. It is also important to present the patterns acyclically. Random presentation order of the training patterns helps to prevent cycles of adaptation from occurring.)

Images showing oscillations between 1 and -1 as we can see that 1 is being predicted all times correctly in first image while second image shows all prediction as -1



```
count of -1/0 =  669  count of 1 =  110
Accuracy =>  60.10802469135802
count of -1/0 =  673  count of 1 =  116
Accuracy =>  60.879629629629626
count of -1/0 =  665  count of 1 =  107
Accuracy =>  59.5679012345679
count of -1/0 =  671  count of 1 =  113
Accuracy =>  60.4938271604938З
count of -1/0 =  662  count of 1 =  104
Accuracy =>  59.10493827160494
count of -1/0 =  670  count of 1 =  112
Accuracy =>  60.339506172839506
count of -1/0 =  681  count of 1 =  122
Accuracy =>  61.95987654320988
count of -1/0 =  669  count of 1 =  112
Accuracy =>  60.26234567901234
count of -1/0 =  654  count of 1 =  95
Accuracy =>  57.79320987654321
count of -1/0 =  37  count of 1 =  369
31.32716049382716
```

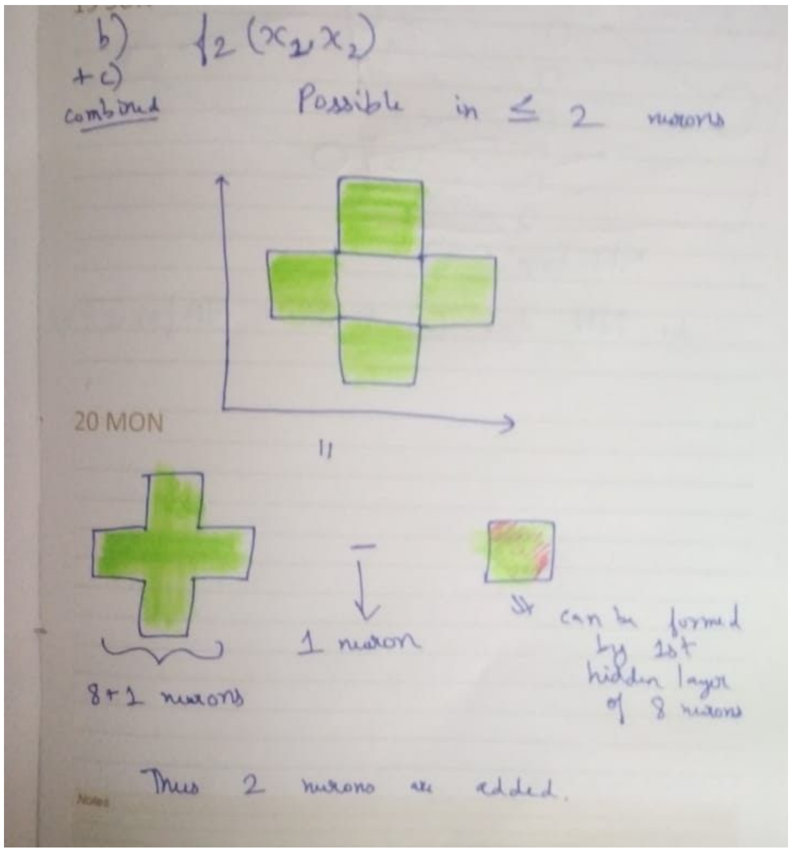
```
Accuracy =>  59.1820987654321
count of -1/0 =  660  count of 1 =  102
Accuracy =>  58.79629629629629
count of -1/0 =  678  count of 1 =  120
Accuracy =>  61.57407407407407
count of -1/0 =  684  count of 1 =  126
Accuracy =>  62.5
count of -1/0 =  676  count of 1 =  118
Accuracy =>  61.26543209876543
count of -1/0 =  679  count of 1 =  120
Accuracy =>  61.651234567901234
count of -1/0 =  684  count of 1 =  127
Accuracy =>  62.57716049382716
count of -1/0 =  927  count of 1 =  0
71.52777777777779
```

2) Increasing of Dataset doesn’t give any difference / better results- We tried to increase the dataset from 81 rows to 1296 rows but the problem still exists, Although this time weights became a little bit more sensitive or very high adaptive as mentioned in the paper but this time it is being converted into a limitation.
3) Non - classification of close points: We tried to include points (or outliers) like (2,-0.01) and (2,6.001). This madeline doesn’t adapt to the point. Due to MSE as an error function to calculate losses and less Z value for these points, they are classified as +1 in the best case which should not be the case. IT is because these points are very close to the decision boundary of the neurons, thus the model classify them wrong in order to generalize itself and this is the reason why this model doesn’t give high training accuracy as written clearly in the research paper as (generalization performance tracks training performance very well. This suggests that one can “catch” the network at a point of good training performance and be confident that its generalization performance will be nearly

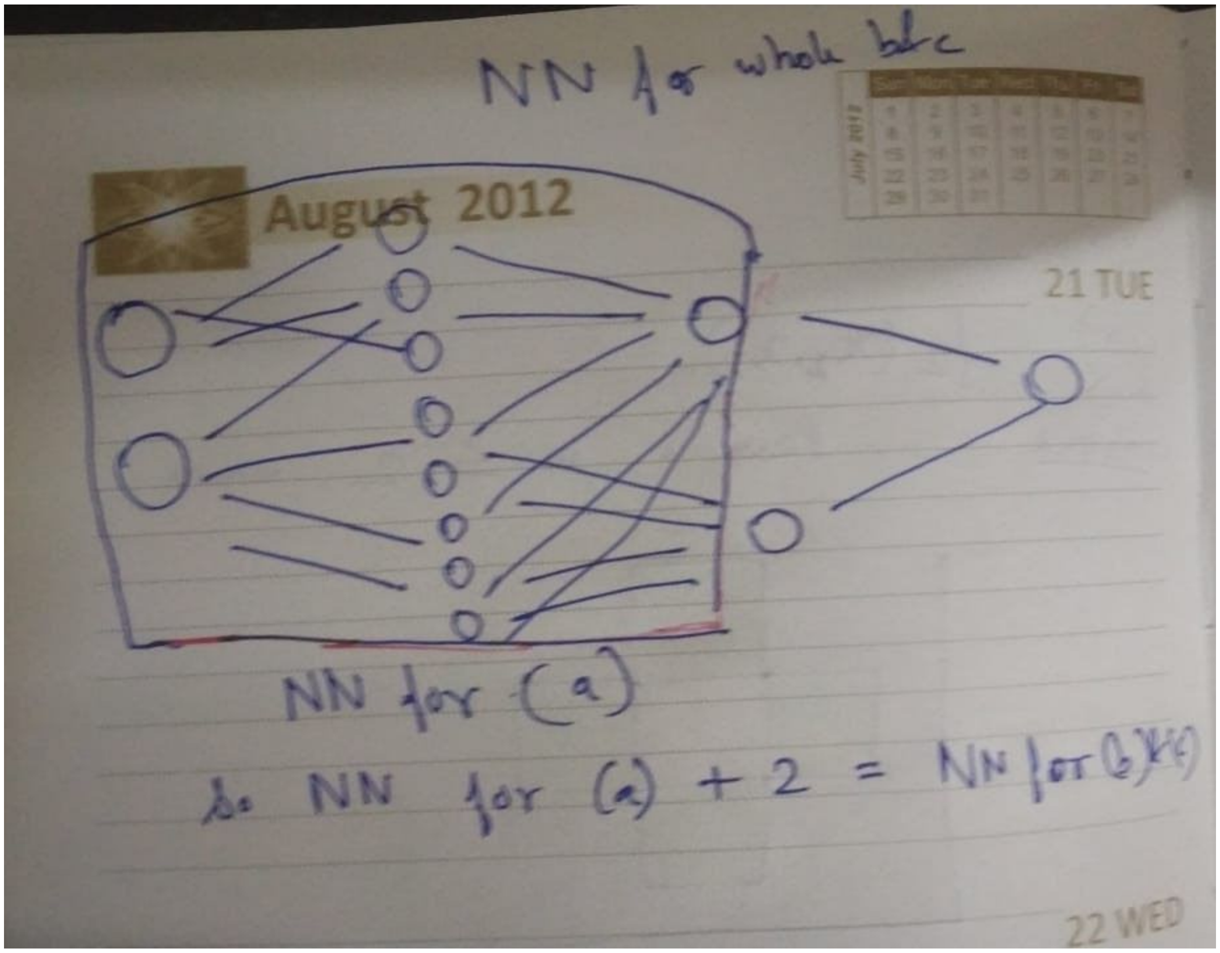
as good. The algorithm rarely achieved perfect performance on the training set, but occasionally came quite close.)

b. f2 (x1, x2) and
c. In comparison with f1, can you compute f2 in <= 2 more neurons? Justify and implement it.

Ans: Theoretically, using analogy of MLP, the function f2 is possible to make by adding 2 neurons to f1 in the following manner.
There will be 2 hidden layers with 8 and 2 neurons respectively thus layers are **2,8,2,1** where 2 is input, 8,2 are hidden and 1 is output.
The f1 gives a following plus sign but when the middle square is subtracted from the above part, we get f2 thus we need one neuron to make the square from 8 hidden units of layer 1 and one neuron to make the expression A-B or (A AND (B')) thus 8 neurons in hidden layers will be able to make the following expression.



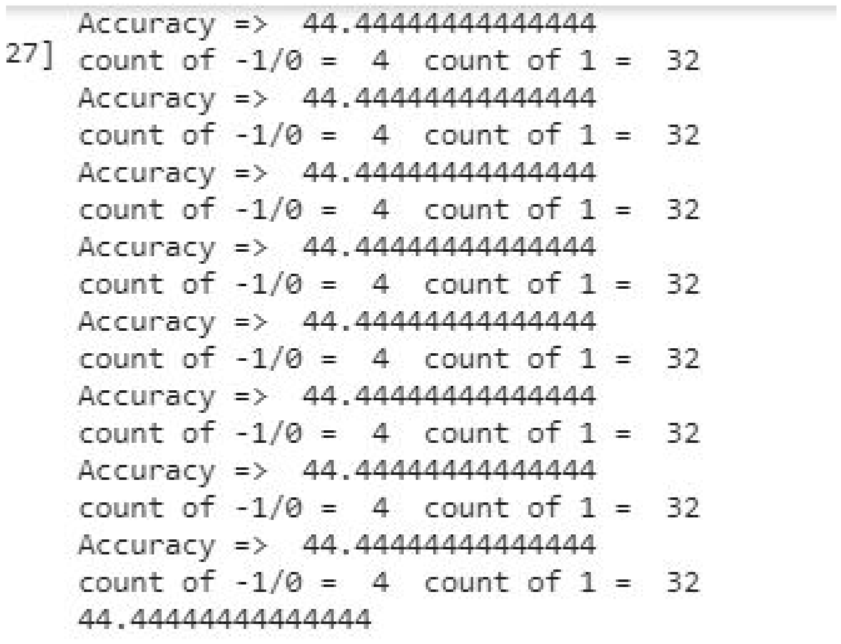
NN representation:



But it was not possible in the case Madaline due to the following reasons:

1) Multiple Adaline networks have a problem in weight updation when hidden layers are increased. There was no mention of increasing the number of layers in the paper while there were mention of increasing hidden units in the particular layer or increasing of output neurons. We tried to experiment it to see what happens and the weights don't get updates. I think the reason is because while you calculate the neuron with minimum disturbance but in the after layers, it can be a possible case that that particular neuron has huge weights or bias in upcoming layers thus it will no longer be neuron which does minimum disturbance.

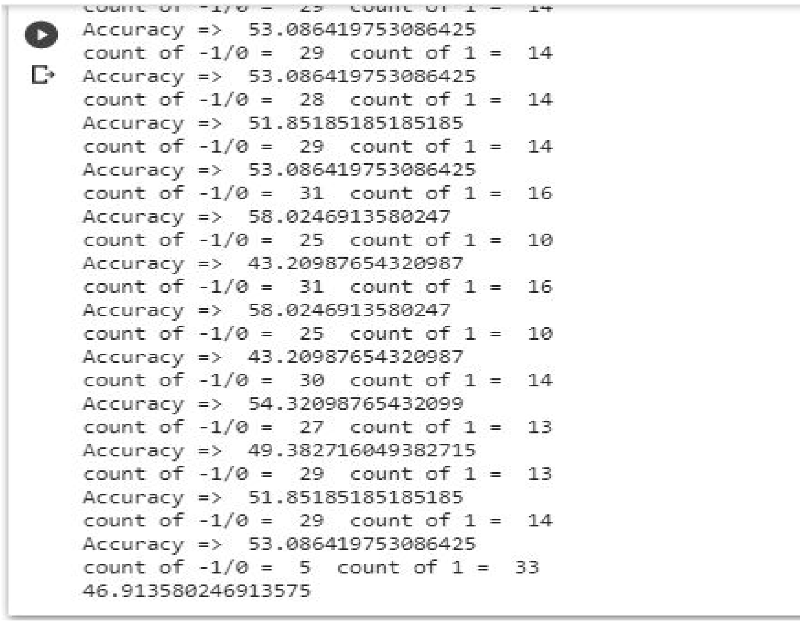
   Output:



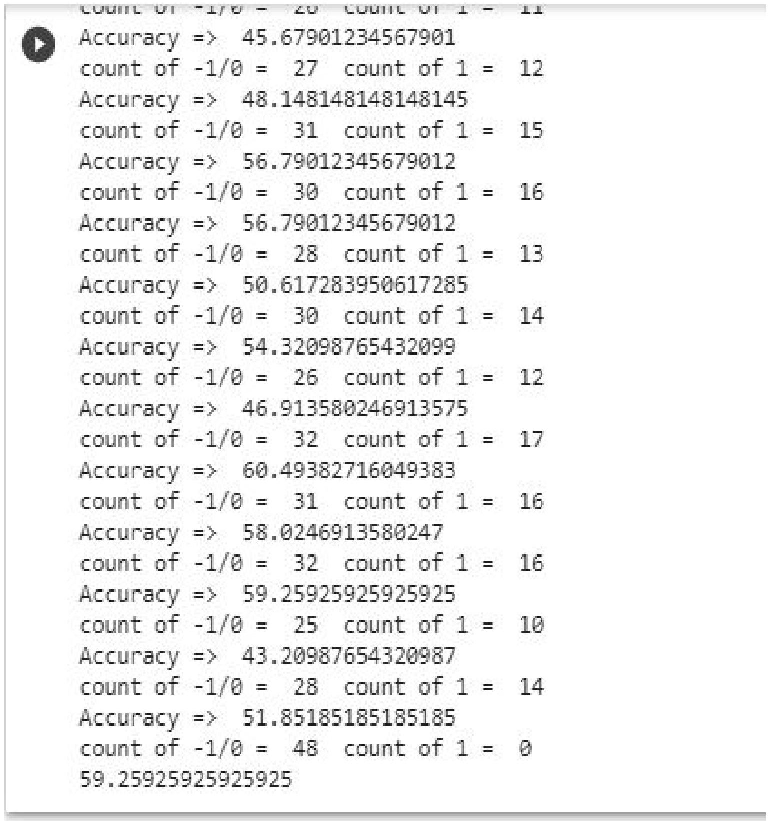
```
    Accuracy =>  44.4444444444444
27] count of -1/0 =  4  count of 1 =  32
    Accuracy =>  44.4444444444444
    count of -1/0 =  4  count of 1 =  32
    Accuracy =>  44.4444444444444
    count of -1/0 =  4  count of 1 =  32
    Accuracy =>  44.4444444444444
    count of -1/0 =  4  count of 1 =  32
    Accuracy =>  44.4444444444444
    count of -1/0 =  4  count of 1 =  32
    Accuracy =>  44.4444444444444
    count of -1/0 =  4  count of 1 =  32
    Accuracy =>  44.4444444444444
    count of -1/0 =  4  count of 1 =  32
    Accuracy =>  44.4444444444444
    count of -1/0 =  4  count of 1 =  32
    44.4444444444444
```

2) Exhausting case of no more updates in single changes: It is possible that all the single change updates are exhausted thus there is no possible neuron with minimum affinity value that i can choose now. While the paper suggests a method to counter these cases by choosing 2 or 3 or 4 instead of 1, it is not computationally feasible as mentioned: (It is possible that the procedure will exhaust all trials involving a single ADALINE without reducing the output errors to zero. In such a case, try pairwise trial adaptations. That is, reverse the outputs of the two ADALINEs with confidence levels closest to zero. If this trial is rejected, return the output of the lowest confidence ADALINE to its previous value and reverse the output of the third least confident ADALINE, etc. Again, if an adaptation trial is accepted, restart the procedure starting with single trials. If pairwise trials do not correct all the output errors, then 3-wise, 4-wise, etc., trials are made.).
3) The above limitations as suggested in a) part is also applied.
4) For doing only (b) part, i tried increasing number of hidden units in the hidden layers and increasing it till 20 but it is not feasible as we require one more layer to take AND or OR operations unlike 1).

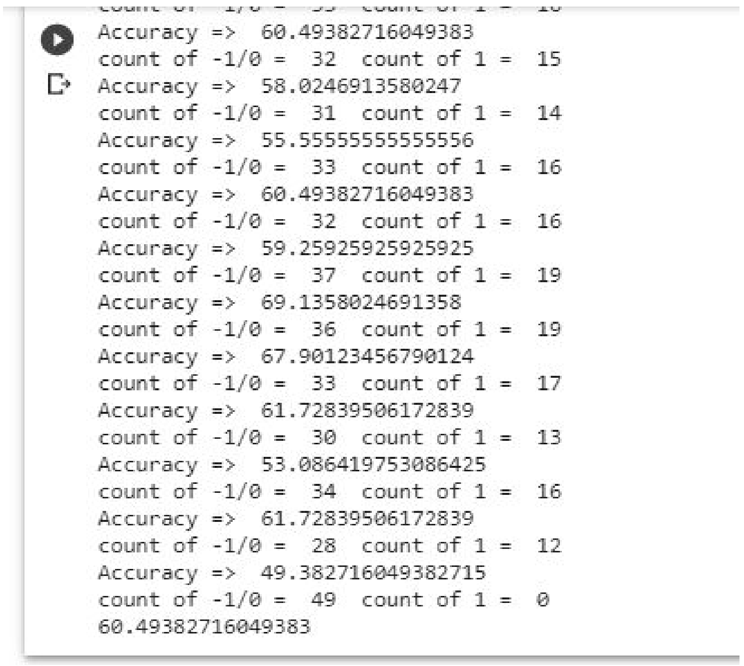
b) part at different hidden units 10, 16 and 20.



```
count of -1/0 =  29  count of 1 =  14
Accuracy =>  53.086419753086425
count of -1/0 =  29  count of 1 =  14
Accuracy =>  53.086419753086425
count of -1/0 =  28  count of 1 =  14
Accuracy =>  51.85185185185185
count of -1/0 =  29  count of 1 =  14
Accuracy =>  53.086419753086425
count of -1/0 =  31  count of 1 =  16
Accuracy =>  58.0246913580247
count of -1/0 =  25  count of 1 =  10
Accuracy =>  43.20987654320987
count of -1/0 =  31  count of 1 =  16
Accuracy =>  58.0246913580247
count of -1/0 =  25  count of 1 =  10
Accuracy =>  43.20987654320987
count of -1/0 =  30  count of 1 =  14
Accuracy =>  54.32098765432099
count of -1/0 =  27  count of 1 =  13
Accuracy =>  49.382716049382715
count of -1/0 =  29  count of 1 =  13
Accuracy =>  51.85185185185185
count of -1/0 =  29  count of 1 =  14
Accuracy =>  53.086419753086425
count of -1/0 =  5  count of 1 =  33
46.913580246913575
```



```
count of -1/0 =  26  count of 1 =  11
Accuracy =>  45.67901234567901
count of -1/0 =  27  count of 1 =  12
Accuracy =>  48.148148148148145
count of -1/0 =  31  count of 1 =  15
Accuracy =>  56.79012345679012
count of -1/0 =  30  count of 1 =  16
Accuracy =>  56.79012345679012
count of -1/0 =  28  count of 1 =  13
Accuracy =>  50.617283950617285
count of -1/0 =  30  count of 1 =  14
Accuracy =>  54.32098765432099
count of -1/0 =  26  count of 1 =  12
Accuracy =>  46.913580246913575
count of -1/0 =  32  count of 1 =  17
Accuracy =>  60.49382716049383
count of -1/0 =  31  count of 1 =  16
Accuracy =>  58.0246913580247
count of -1/0 =  32  count of 1 =  16
Accuracy =>  59.25925925925925
count of -1/0 =  25  count of 1 =  10
Accuracy =>  43.20987654320987
count of -1/0 =  28  count of 1 =  14
Accuracy =>  51.85185185185185
count of -1/0 =  48  count of 1 =  0
59.25925925925925
```

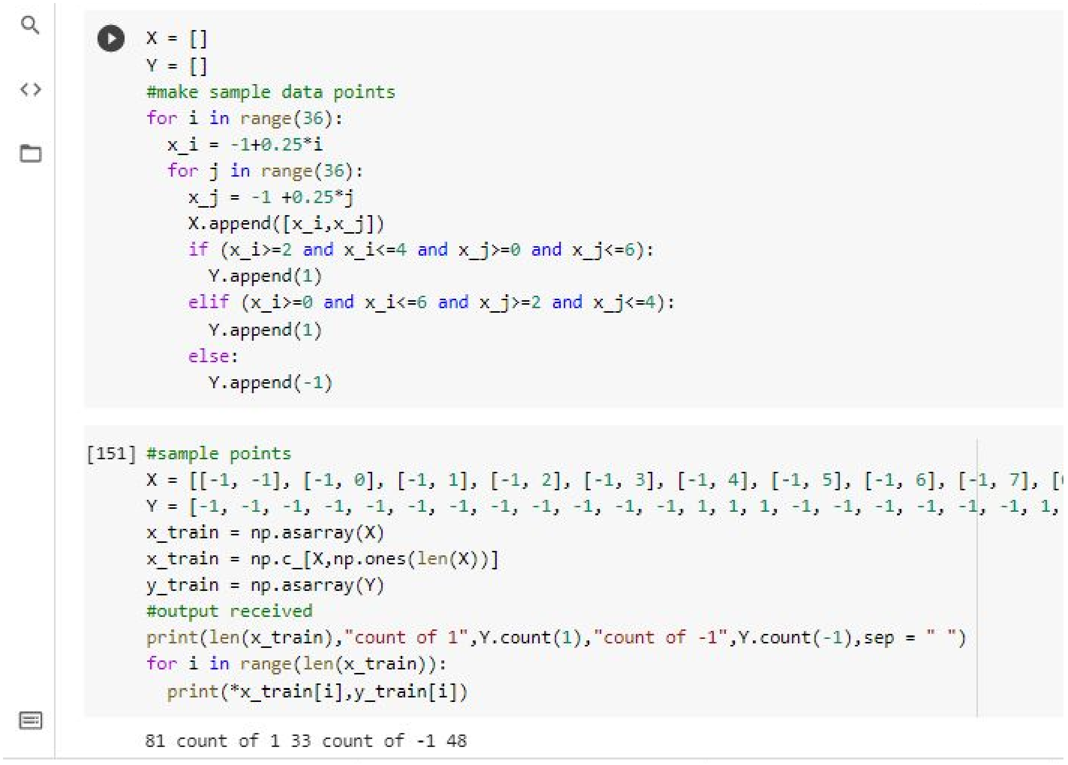


```
Accuracy =>  60.49382716049383
count of -1/0 =  32  count of 1 =  15
Accuracy =>  58.0246913580247
count of -1/0 =  31  count of 1 =  14
Accuracy =>  55.55555555555556
count of -1/0 =  33  count of 1 =  16
Accuracy =>  60.49382716049383
count of -1/0 =  32  count of 1 =  16
Accuracy =>  59.25925925925925
count of -1/0 =  37  count of 1 =  19
Accuracy =>  69.1358024691358
count of -1/0 =  36  count of 1 =  19
Accuracy =>  67.90123456790124
count of -1/0 =  33  count of 1 =  17
Accuracy =>  61.72839506172839
count of -1/0 =  30  count of 1 =  13
Accuracy =>  53.086419753086425
count of -1/0 =  34  count of 1 =  16
Accuracy =>  61.72839506172839
count of -1/0 =  28  count of 1 =  12
Accuracy =>  49.382716049382715
count of -1/0 =  49  count of 1 =  0
60.49382716049383
```

Thus, though it's theoretically possible to make b) part with using 2 more neurons with a) it is not technically feasible due to limitations by these algorithms.

Preprocessing steps:

1) Created Dataset by for loops
2) Adding column of bias to the data



```
X = []
Y = []
#make sample data points
for i in range(36):
  x_i = -1+0.25*i
  for j in range(36):
    x_j = -1 +0.25*j
    X.append([x_i,x_j])
    if (x_i>=2 and x_i<=4 and x_j>=0 and x_j<=6):
      Y.append(1)
    elif (x_i>=0 and x_i<=6 and x_j>=2 and x_j<=4):
      Y.append(1)
    else:
      Y.append(-1)
```

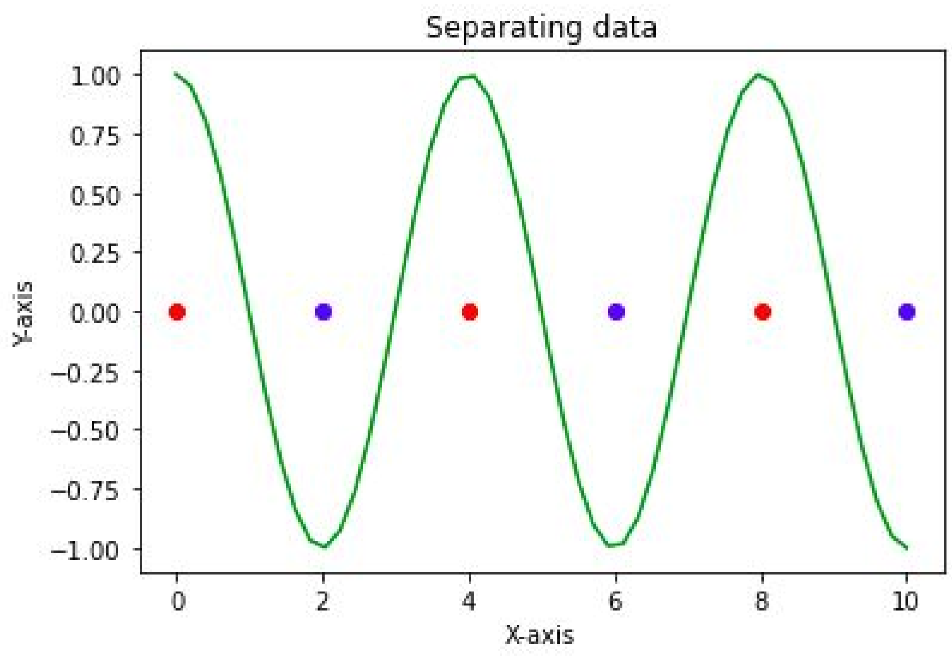
```
[151] #sample points
      X = [[-1, -1], [-1, 0], [-1, 1], [-1, 2], [-1, 3], [-1, 4], [-1, 5], [-1, 6], [-1, 7], [
      Y = [-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 1, 1, 1, -1, -1, -1, -1, -1, -1, 1,
      x_train = np.asarray(X)
      x_train = np.c_[X,np.ones(len(X))]
      y_train = np.asarray(Y)
      #output received
      print(len(x_train),"count of 1",Y.count(1),"count of -1",Y.count(-1),sep = " ")
      for i in range(len(x_train)):
        print(*x_train[i],y_train[i])

      81 count of 1 33 count of -1 48
```

Helper functions: Find_min , Adaline_update, threshold, random_init, forward, test. These all functions are part of class Madaline_NN which is a library in which you can use forward as a training function and test as prediction function to print accuracy.
The code is heavily documented as information about these functions can be found there in great detail.

Methodology:

1) Implemented Madaline algorithm with pseudo code given in the lectures
2) Preprocessed/ created dataset with 81 points and 1296 points respectively using for loops
3) Tried different Learning rates as mentioned in the paper: 1<=Alpha<=1/n (n is no. of samples as mentioned in the paper)
4) Threshold function as suggested by the algorithm.
5) Read the paper for Mp-II madaline algorithm to see why the code doesn't work unlike back propagation.

3. Implement a single-neuron neural network to compute the following function y = f(x). You can use the generalized delta rule for learning. If you think it's not possible, justify your claims properly.



Red Points- Class 0
Blue Points- Class 1
We use Sin(wx +b) as activation function to separate the given input.
The activation function used:

```
def activation(x):
 '''
   input: a 1-D array x

   output: sin of all values of array
 '''
 return np.sin(x)
```

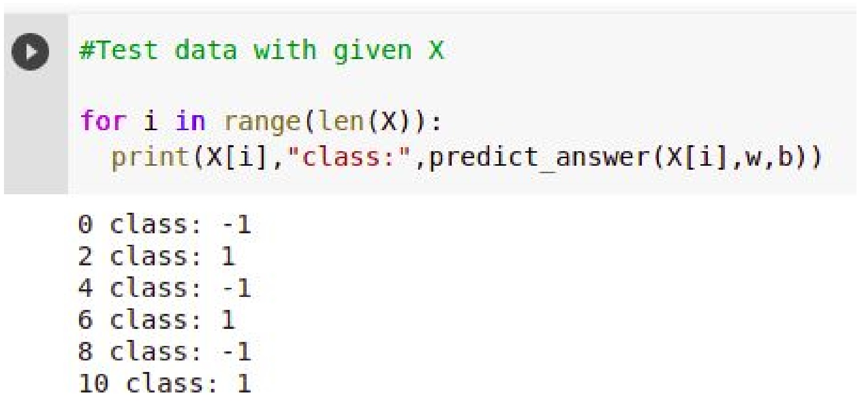
No preprocessing done

<u>**Helper function**</u>: <u>activation, predict, loss, predict_answer, update, train</u>
The code is heavily documented as information about these functions can be found there in great detail.

<u>Methodology:</u>

- First we see the data and see a certain pattern or frequency
- This leads us to think of a sinusoidal wave like pattern
- Thus we take sin(x) as our activation function
- Using delta-learning weight update we get optimal values of w,b both as pi/2
- The Graph and points are plotted above.

Output:



```
#Test data with given X

for i in range(len(X)):
  print(X[i],"class:",predict_answer(X[i],w,b))
```

```
0 class: -1
2 class: 1
4 class: -1
6 class: 1
8 class: -1
10 class: 1
```

*Pickle dump not uploaded for Q3 as it takes less than a minute to run. Thus the results can be produced during the demo easily.