ENPM808A: Introduction to Machine Learning

Assignment 1

Question 1 (Code)

```
 # -*- coding: utf-8 -*-
"""
Created on Sun Sep 18 21:58:33 2022

@author: Bharadwaj Chukkala | 118341705 | bchukkal@umd.edu
"""

import string
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_absolute_error, mean_squared_error

## Loading the Data ##
Sales_Data =
pd.read_csv(r"C:\Users\bhara\OneDrive\Desktop\ENPM808A\Week1_Assignment\mlr05.csv
")
print(Sales_Data)
print('\v')

print(Sales_Data.describe())
print('\v')

## Prepare/Segregate Data
X = Sales_Data[['X2', 'X3', 'X4', 'X5', 'X6']] #Input with 5 dimensions
Y = Sales_Data['X1'] #Corresponding Output

## Splitting DataSet into Testing And Training sets
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=7, train_size
=20, shuffle=False, random_state=0)

## Training the Regression Model ##
regressor = LinearRegression()
```

```
regressor.fit(X_train, Y_train)

## Checking the weights chosen to fit the model
weights = regressor.coef_
weight_data = zip(X.columns, weights)
weights_dataframe = pd.DataFrame(weight_data, columns = ['Feature','Weight'])
print(weights_dataframe)
print('\v')

## Output Predictions for X_Test ##
Y_predict = regressor.predict(X_test)

## Comparing the Predicted Values with Test Values to see closeness
Closeness_Data = zip(Y_predict, Y_test)
Closeness_dataframe = pd.DataFrame(Closeness_Data, columns = ['Predicted Output',
'Actual Output'])
print(Closeness_dataframe)
print('\v')

## Calculating Erro values betweeen Predicted Outputs and actual Test Outputs
print('Mean Absolute Error:', mean_absolute_error(Y_test, Y_predict))
print('Mean Squared Error:', mean_squared_error(Y_test, Y_predict))
print('Root Mean Squared Error:', np.sqrt(mean_squared_error(Y_test, Y_predict)))
```

Output:

```
♂
  Feature     Weight
0      X2  41.952145
1      X3   0.173289
2      X4   6.220788
3      X5   8.280789
4      X6  -6.085925
♂
   Predicted Output  Actual Output
0        554.494481          528.0
1         71.757804           99.0
2         34.238343            0.5
3        351.672272          347.0
4        342.773458          341.0
5        524.835704          507.0
6        548.587847          400.0
♂
Mean Absolute Error: 37.19204289355678
Mean Squared Error: 3571.9723319955046
Root Mean Squared Error: 59.765979720870504
```

Written Responses (Q -2,3,5)

Sunday, September 18, 2022 1:54 PM

Bharadwaj Chukkala
118341705
bchukkal@umd.edu

2) Considering the Perceptron in 2 dimensions:-

$h(x) = \text{sign}(w^T x)$, $w = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix}$ $x = [1, x_1, x_2]$

2D Perceptron because first coordinate is fixed. →

a) i) $h(x) = +1$, $h(x) = -1$ are separated by a line. (Show this)

This line is given by eqn ⇒ $x_2 = ax_1 + b$ $(y = mx + c)$

Basically x_1, x_2 are y and

ii) What are a and b in terms of w_0, w_1, w_2.

Ans

Proof:-

for $h(x) = +1$, from eqn $h(x) = \text{sign}(w^T x)$
we need to have $w^T x > 0$

for $h(x) = -1$, from eqn $h(x) = \text{sign}(w^T x)$
we need to have $w^T x < 0$

for the line dividing $h(x) = +$ & $h(x) = -1$ we need to have $w^T x = 0$

Perceptron in 2Dimensions $h(x) = \text{sign}(w^T x) \Rightarrow \text{sign}\left(\left(\sum_{i=1}^{d} w_i x_i\right)\right)$

$$\sum_{i=1}^{d} w_i x_i \text{ or } w^T x = w_0 \cdot 1 + w_1 x_1 + w_2 x_2$$

so the line that separates $h(x) = 1$ and $h(x) = -1$ can be written as $w_0 + w_1 x_1 + w_2 x_2 = 0$ — ①

we also know that the line is of the form

$x_2 = ax_1 + b$ — ② (given)

upon comparing

$x_2 = ax_1 + b$ and $w_0 + w_1 x_1 + w_2 x_2$

$x_2 = ax_1 + b$

$$x_2 = \frac{-(w_0 + w_1 x_1)}{w_2}$$

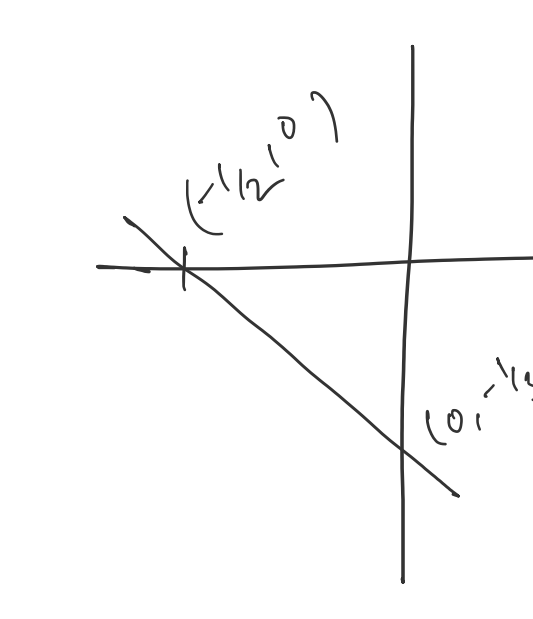
$$\therefore \quad a = \frac{-w_1}{w_2} \qquad b = -\frac{w_0}{w_2}$$

b) 1) $w = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$ $x = [1, x_1, x_2]$

Line eqn: $1 \times 1 + 2x_1 + 3x_2 = 0$

$2x_1 + 3x_2 + 1 = 0$

Draw a line with this Eqn. → $x_2 = -\frac{2}{3}x_1 - \frac{1}{3}$

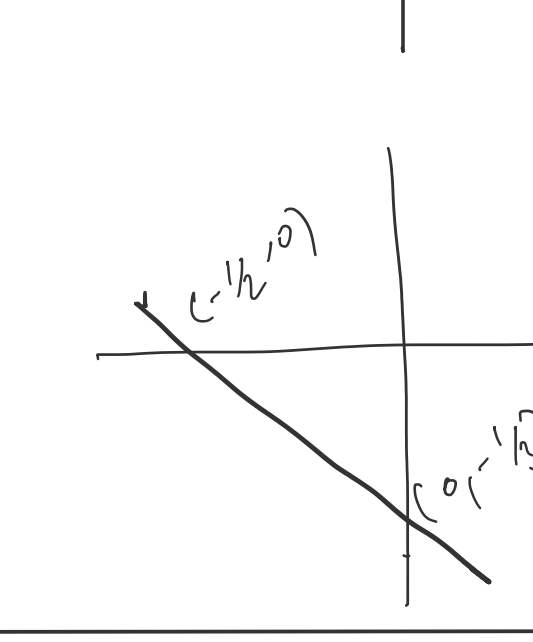


2) $w = -\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$ $x = [1, x_1, x_2]$

Line eqn = $-1 \times 1 - 2x_1 - 3x_2 = 0$

$1 + 2x_1 + 3x_2 = 0$

Draw a line with Eqn. → $x_2 = -\frac{2}{3}x_2 - \frac{1}{3} = 0$



Both have same intercepts

---

3) Consider a linearly separable dataset $(x_1, y_1), \ldots, (x_m, y_m)$

Let $B = \min\{\|w\| : \forall i \in [m], y_i(w^T x_i) \geq 1\}$

and Let $R = \max \|x_i\|$

Show that Perceptron algorithm stops after $(RB)^2$ iterations

Ans)

$B = \min\{\|w\| : \forall i \in [m], y_i(w^T x_i) \geq 1\}$

$R = \max_{i \in [1:n]} \|x_i\|$

No. of updates / iterations = N

Assume that there is a weight

$w^* = \arg\min\{\|w\| : \langle w; x_i \rangle y_i \geq 1, \forall i \in [1:n]\}$

It is clear that the algorithm terminates after at most $(RB)^2$ iterations.

we have to find upper bound and lower bound to decide the max iterations to converge.

To prove convergence, we need to show that the upper bound is $(RB)^2$.

Now that $w_1 = 0$, and for $N \geq 1$

Let x_j is the misclassified point during iteration N.

$w_{N+1} = w_N + x_j y_j$ --- (i)

$B = \min\{\|w\| : \forall i \in [m], y_i(w^T x_i) \geq 1\}$ --- (ii)

Let w^* be the line that separates the two classes.

$$(w_{N+1}) \cdot w^* = (w_N + x_j y_j) \cdot w^*$$
$$= w_N \cdot w^* + y_j(x_j \cdot w^*)$$
$$> w_N \cdot w^* + \frac{1}{B}$$

from the rule of induction:-

$w_{N+1} \cdot w^* > \frac{N}{B}$ since $w_{N+1} \cdot w^* \leq \|w_{N+1}\| \|w^*\| = \|w_{N+1}\|$

we get $\|w_{N+1}\| > \frac{N}{B}$ — ①

To obtain upper bound we argue that

$$\|w_{N+1}\|^2 = \|w_N + x_j y_j\|^2$$
$$= \|w_N\|^2 + \|x_j y_j\|^2 + 2(w_N \cdot x_j y_j)$$
$$= \|w_N\|^2 + \|x_j y_j\|^2 + 2(w_N \cdot x_j) y_j$$
$$\leq \|w_N\|^2 + \|x_j y_j\|^2$$
$$\leq \|w_N\|^2 + R^2$$

again from induction we get that

$\|w_{N+1}\|^2 \leq NR^2$ — ②

Together with ① & ②.

$$\frac{N^2}{B^2} < \|w_{N+1}\|^2 \leq NR^2$$

This implies that $N < (RB)^2$

∴ The Algorithm converges at $(RB)^2$ iterations.

---

5) a) Exercise 1.8

Ans) If $\mu = 0.9$, what is the probability that a sample of 10 marbles will have $\nu \leq 0.1$

→ Answer is a Biomial distribution ⟶ ${}^nC_r\, p^r q^{n-r}$
→ It's a very small number.

ν → fraction of red marbles within the sample $\nu \leq 0.1$

$\nu = \frac{1}{10}$ ← fraction of red marbles.

$u = P(\text{Red}) = 0.9$

$P(\text{Red} \leq 1 \mid \text{Marbles} = 10) = P(\text{Red} = 0 \mid \text{Marbles} = 10) + P(\text{Red} = 1 \mid \text{Marbles} = 10)$

$P(\text{Red} = 0 \mid \text{Marbles} = 10) = {}^nC_r\, p^r q^{n-r}$ $\qquad q = (1 - p)$

$= {}^{10}C_0\, p^0 q^{10}$

$= q^{10}$

$= (1 - p)^{10}$ $\qquad$ here p is $\mu = 0.9$.

$= (1 - 0.9)^{10}$

$= (0.1)^{10}$

$P(\text{Red} = 1 \mid \text{Marbles} = 10)$ ${}^nC_r\, p^r q^{n-r}$

$= {}^{10}C_1\, p^1 (1-p)^9$

$= 10 \times (0.9)^1 \times (0.1)^9$

$= 9 \times (0.1)^9$

$P(\text{Red} \leq 1 \mid \text{Marbles} = 10) \Rightarrow (0.1)^{10} + 9 \times (0.1)^9$

$= (0.1)^9 \times 9.1$ very small number

b) Exercise 1.9

If $u = 0.9$, use Hoeffding Inequality to bound the probability that a sample of 10 marbles will have $\nu \leq 0.1$ and compare the answer to exercise 1.8.

Ans) To quantify the relationship between ν and μ, we use a simple bound called the Hoeffding inequality.

$$P[|\nu - \mu| > \epsilon] \leq 2e^{-2\epsilon^2 N} \quad \text{for any } \epsilon > 0.$$

$|0.1 - 0.9| = |-0.8|$
$= 0.8$

$0.8 > \epsilon$

will consider $\epsilon = 0.7$ $\qquad$ or (we can consider 0.799 too but i am considering 0.7)

$P[|\nu - \mu| \geq 0.8]$ ← when this is true

$P[|\nu - \mu| > 0.7] \leq 2e^{-2\epsilon^2 N}$

↓

$2e^{-2(0.7)^2 \cdot 10}$

$\simeq 0.0001109$

---

4) Perceptron Learning Algorithm

→ each iteration 't', pick a random $x(t), y(t)$

→ compute the signal $s(t) = w^T(t) x(t)$

→ for initial step assign weight $w = [0, 0, 0]$ (dim+1)

→ If $y(t) \cdot s(t) \leq 0$

update w by $w(t+1) \leftarrow w(t) + y(t) \cdot x(t)$

Adaline:-

↳ Takes closeness between $s(t)$ and $y(t)$ into consideration.

↳ update step differs

if $y(t) \cdot s(t) \leq 1$

update w by

$$w(t+1) \leftarrow w(t) + \eta \cdot (y(t) - s(t)) \cdot x(t)$$

Steps to implement this problem

→ firstly we need to have a target function $(h(x))$
→ All data above $h(x) = +1$ & data below $h(x) = -1$
→ Create a target function for binary classification
→ Generate random numbers
→ use these random numbers to generate datapoints
→ constraint the generation of numbers with bounds
→ constraint the scattering of data such that they fall above and below our target function within the bounds
→ divide the data into two classes and labels (+1, -1)
→ Create a dataframe using those two classes with pandas library
→ generate a test-dataframe for testing purpose
→ Create a function that calculates accuracy by testing the final weights on a test dataframe.
→ Once everything is in place we have our input, output, target function, dimension.
→ we shall implent the adaline algorithm so that the machine can closely / properly approximate the target function we created and classify the data we generated.
→ we have to try a find a hypothesis that will be close to the target function that we created.
→ Now write an algorithm for adaline by changing the update rule as asked in question
→ Iterate the adaline algorithm with different eta values with these input parameters.
→ After running the adaline algorithm, we will get final normalized weights for each eta.
→ use these hypothesised weights on the test dataframe to calculate accuracy.
→ Generate the results (the correctness, accuracy, final weights).
→ compare the results with true weights and true function plot.

Question 4 (Code)

```
# -*- coding: utf-8 -*-
"""
Created on Sat Sep 16 16:25:42 2022

@author: Bharadwaj Chukkala | 118341705 | bchukkal@umd.edu
"""

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from functools import partial

def adaline(points, dim, max_iterations, use_adaline=False,
                eta = 1, randomize=False, print_out = True):
    '''
    Def: Adapative Linear Neuron, a variant of perceptron used in this case for
#classification
    Param 1: points = Dataframe values for performing algorithm
    Param 2: dim = dimensions of the input data
    '''
    w = np.zeros(dim+1) #introducing initial weights
    xs, ys = points[:,:dim+1], points[:,dim+1] #Inputs and Corresponding outputs
extracted from dataframe
    num_points = points.shape[0] #Number of Inputs
    # Iterating through the number of iteration by selecting random points
    for iteration in range(max_iterations):
        correctly_predicted_ids=  []
        idxs = np.arange(num_points)
        if randomize:
            idxs = np.random.choice(np.arange(num_points), num_points,
replace=False)
        for idx in idxs:
            x, y = xs[idx], ys[idx] #inputs and corresponding outputs
            st = np.dot(w.T, x) #signal_function
            prod = st*y #np.dot(w.T, x)*y
            if prod < -100: #avoid out of bound error we bound the signal at -100
                st = -100
            threshold = 1 if use_adaline else 0 #adaline can learn without the
algorithm making errors
            st = st if use_adaline else 0
            if prod <= threshold:
                w = w + eta *(y-st)*x
```

```
                break #PLA picks one example at each iteration
            else:
                correctly_predicted_ids.append(idx)
        if len(correctly_predicted_ids) == num_points:
            break

    c = 0 #correctness
    zipped_matrix = zip(xs, ys) #Input and Output clubbed for brevity
    for x, y in zipped_matrix:
        prod = np.dot(w.T, x)*y
        if prod > 0:
            c +=1
    w = w/w[-1]
    if print_out:
        print('Final Correctness of Hypothesis: ', c)
        print('Total iterations: ', iteration)
        print('Final Normalized Weights:', w)
    return w, iteration


def generate_random_numbers(N, dim, num_grid_points, lb, ub):
    '''
    Def: The function generates random numbers to help generate a dataframe
    Param 1: N = Number of Data points
    Param 2: dim = dimensions of the input data
    Param 3: num_grid_points = number of grid points in the plane
    Param 4: lb = Lower Bound for the input Data Points
    Param 5: ub = Upper Bound for the input Data Points
    '''
    rand_ints = np.random.randint(num_grid_points, size =(N, dim))
    init_lb = 0
    zero_to_one_points = (rand_ints - init_lb)/(num_grid_points -1 -init_lb)
    res = lb + (ub - lb)*zero_to_one_points
    return res

def generate_random_coeffs(dim):
    '''
    Def: Generates random weights given the dimensions
    Param 1: dim = dimension of the input data
    '''
    rn = generate_random_numbers(1, dim, 1000, -10, 10)
    return rn

def true_f(x, coeffs):
```

```
    '''
    Def: A helper function to generate the true function
    Param 1: x = input
    Param 2: coeffs = weights for each input
    '''
    return coeffs.flatten()[0] + np.dot(coeffs.flatten()[1:], x.flatten())

def generate_two_classes(N, dim, true_function, rn_func):
    '''
    Def: Function that generates two classes to be classified
    Param 1: N = Number of Datapoints for each class
    Param 2: D = Dimensions of the input data
    Param 3: true_function = Target Function that classifies the data
    Param 4: rn_func = Random Function that works according to use case with
constrained params
    '''
    cls1, cls2 = [], []
    while True:
        rn = rn_func(1, dim).flatten()
        if true_function(rn) > 0 and len(cls1) < N:
            cls1.append(rn)
        elif true_function(rn) < 0 and len(cls2) < N:
            cls2.append(rn)
        if len(cls1) == N and len(cls2) == N:
            break
    return np.asarray(cls1), np.asarray(cls2)

def generate_dataframe(N, dim,true_function, rn_func):
    '''
    Def: A Fucntion that generates the dataframe using the two classes of data
    Param 1: N = Number of Datapoints
    Param 2: dim = Dimension of the input data
    Param 3: true_function = A Target Functon that classifies the data
    Param 4: rn_func = Random Function that works according to use case with
constrained params
    '''
    cls1, cls2 = generate_two_classes(N/2, dim, true_function, rn_func)
    cols = ['x' + str(i) for i in range(1, dim+1)]
    df1 = pd.DataFrame(cls1, columns=cols)
    df1['y'] = 1
    df2 = pd.DataFrame(cls2, columns=cols)
    df2['y'] = -1
    df = pd.concat([df1, df2])
    df['x0'] = 1
```

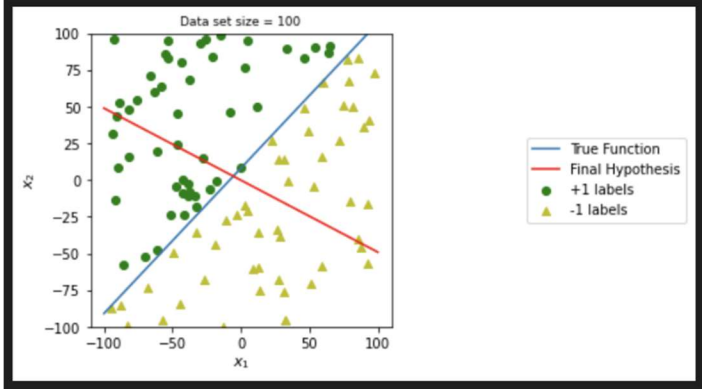
```
    df = df[['x0'] + cols + ['y']]
    return df

def fitness_test(test_df, norm_g):
    '''
    Def: A Utility Function that tests the final weights on a data frame and
measures the accuracy of model
    Param 1: test_df = A test data frame to calculate accuracy
    Param 2: norm_g = Final normalized weights
    '''
    xs = test_df[['x0', 'x1', 'x2']].values
    ys = test_df['y'].values
    accuracy = 0
    for x, y in zip(xs, ys):
        prod = np.dot(norm_g.T, x)*y
        if prod > 0:
            accuracy +=1
    accuracy = accuracy/100
    return accuracy

def plot_data(x1, df, norm_coeffs, norm_g, lb, ub):
    '''
    Def:A Function to plot the Dataframe(Classes), True Function, Hypothesis etc
    Param 1: A Range of numbers between lowerbound and upper bound
    Param 2: Dataframe that was generated
    Param 3: norm_coeffs = Weights for the input data
    Param 4: norm_g = Weights that were normalized with the first value
    Param 5: lb = lower bound of data
    Param 6: ub = upper bound of data
    '''
    figsize = plt.figaspect(1)
    f, ax = plt.subplots(1, 1, figsize=figsize)
    cls1_df = df.loc[df['y']==1]
    cls2_df = df.loc[df['y']==-1]
    line = ax.plot(x1, -(norm_coeffs[0]+norm_coeffs[1]*x1), label='True
Function')
    pluses = ax.scatter(cls1_df[['x1']].values, cls1_df[['x2']].values,
marker="o", c= 'g', label='+1 labels')
    minuses = ax.scatter(cls2_df[['x1']].values, cls2_df[['x2']].values,
marker="^", c= 'y', label='-1 labels')
    if norm_g is not None:
        hypothesis = ax.plot(x1, -(norm_g[0]+norm_g[1]*x1), c = 'r', label='Final
Hypothesis')
```

```
    ax.set_ylabel(r"$x_2$", fontsize=11)
    ax.set_xlabel(r"$x_1$", fontsize=11)
    ax.set_title('Data set size = %s'%N, fontsize=9)
    ax.axis('tight')
    legend_x = 2.0
    legend_y = 0.5
    ax.legend(['True Function', 'Final Hypothesis',
               '+1 labels', '-1 labels', ],
               loc='center right', bbox_to_anchor=(legend_x, legend_y))
    #ax.legend(handles=[pluses, minuses], fontsize=9)
    ax.set_ylim(bottom=lb, top=ub)
    plt.show()
    print('True weights of Target Function: ', norm_coeffs)


lb,ub = -100, 100 #Bounds
N = 100 #Number of Data points
dim = 2 #Dimensions
num_grid_points = 2000
test_N = 10000 #Test_data_points
coeff_lb, coeff_ub = -10, 10 #Coefficients of bounds
use_adaline, randomize = True, True #Setting parameters for functions
show_plot = True #Setting parameter to display plots
max_iterations=100

random_numbers = generate_random_numbers(N, dim, num_grid_points, lb, ub)
#Generating random numbers
rn_func = partial(generate_random_numbers, num_grid_points = num_grid_points, lb
= lb, ub = ub)
coeffs = generate_random_numbers(1, dim+1, num_grid_points, coeff_lb, coeff_ub)
#Weights
norm_coeffs = coeffs.flatten()/coeffs.flatten()[-1] #Normalized Weights
true_function = partial(true_f, coeffs = norm_coeffs) #True function with
normalzed weights

df = generate_dataframe(N, dim, true_function, rn_func)
test_df = generate_dataframe(test_N, dim, true_function, rn_func)

x1 = np.arange(lb, ub, 0.01)

for eta in [100, 1, 0.01, 0.0001]:
    norm_g, num_its = adaline(df.values, dim, max_iterations, use_adaline,
                              eta, randomize, show_plot)
```

```
    if show_plot:
        plot_data(x1, df, norm_coeffs, norm_g, lb, ub)

    accuracy = fitness_test(test_df, norm_g)
    print('Eta = ', eta, ' Accuracy = ', accuracy)
```

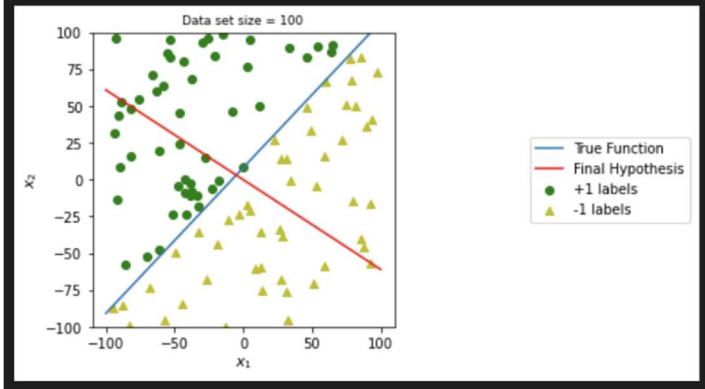
Outputs:

```
Final Correctness of Hypothesis:  58
Total iterations:  99
Final Normalized Weights: [0.08069481 0.49001002 1.        ]
```



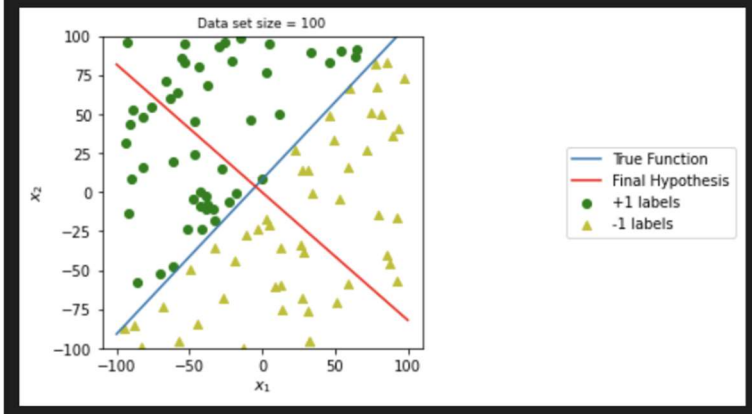

```
True weights of Target Function:  [-8.09392265 -0.98895028  1.        ]
Eta =  100  Accuracy =  62.83
```

```
Final Correctness of Hypothesis:  53
Total iterations:  99
Final Normalized Weights: [0.05366481 0.6086189  1.        ]
```



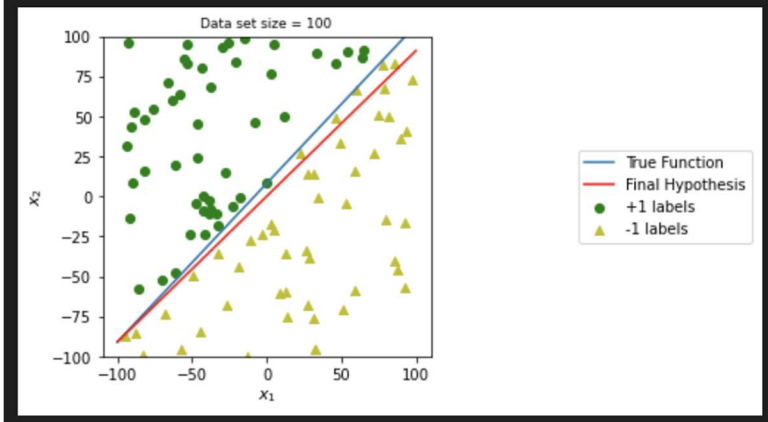

```
True weights of Target Function:  [-8.09392265 -0.98895028  1.        ]
Eta =  1  Accuracy =  59.82
```

```
Final Correctness of Hypothesis:  51
Total iterations:  99
Final Normalized Weights: [0.08837265 0.81833896 1.        ]
```




```
True weights of Target Function:  [-8.09392265 -0.98895028  1.        ]
Eta =  0.01  Accuracy =  55.02
```

```
Final Correctness of Hypothesis:  95
Total iterations:  99
Final Normalized Weights: [-0.03127311 -0.90899602  1.        ]
```




```
True weights of Target Function:  [-8.09392265 -0.98895028  1.        ]
Eta =  0.0001  Accuracy =  96.26
```