Back Propagation for n bit data

Generating n bit of data

In [1]:

```
n = int(input('Enter Number of bits : '))
count = 0
i = n
string = 'bit_'
total_number = 2 ** n
```

In [2]:

```
value = list()
dictionary = dict()

while i >= 1 :
    key = string + str(i)
    d = 2 ** count

    while len(value) != total_number:
        for j in range(d):
            value.append(0)
        for j in range(d):
            value.append(1)

    dictionary[key] = value
    value = list()
    count = count + 1
    i = i - 1
```

In [3]:

```
dictionary
```

Out[3]:

```
{'bit_3': [0, 1, 0, 1, 0, 1, 0, 1],
 'bit_2': [0, 0, 1, 1, 0, 0, 1, 1],
 'bit_1': [0, 0, 0, 0, 1, 1, 1, 1]}
```

In [4]:

```
list(dictionary.items())
```

Out[4]:

```
[('bit_3', [0, 1, 0, 1, 0, 1, 0, 1]),
 ('bit_2', [0, 0, 1, 1, 0, 0, 1, 1]),
 ('bit_1', [0, 0, 0, 0, 1, 1, 1, 1])]
```

In [5]:

```
l = list(dictionary.items())
l
```

Out[5]:

```
[('bit_3', [0, 1, 0, 1, 0, 1, 0, 1]),
 ('bit_2', [0, 0, 1, 1, 0, 0, 1, 1]),
 ('bit_1', [0, 0, 0, 0, 1, 1, 1, 1])]
```

In [6]:

```
reversed_dictionary = dict()
```

```
i = n-1
while i >= 0:
    reversed_dictionary[l[i][0]] = l[i][1]
    i = i - 1
reversed_dictionary
```

Out[6]:

```
{'bit_1': [0, 0, 0, 0, 1, 1, 1, 1],
 'bit_2': [0, 0, 1, 1, 0, 0, 1, 1],
 'bit_3': [0, 1, 0, 1, 0, 1, 0, 1]}
```

In [7]:

```
dictionary = reversed_dictionary
dictionary
```

Out[7]:

```
{'bit_1': [0, 0, 0, 0, 1, 1, 1, 1],
 'bit_2': [0, 0, 1, 1, 0, 0, 1, 1],
 'bit_3': [0, 1, 0, 1, 0, 1, 0, 1]}
```

In [8]:

```
output = dictionary['bit_1']
output
```

Out[8]:

```
[0, 0, 0, 0, 1, 1, 1, 1]
```

In [9]:

```
import pandas as pd

df = pd.DataFrame(data=dictionary)
df
```

Out[9]:

	bit_1	bit_2	bit_3
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

In [10]:

```
df['Output'] = output
df
```

Out[10]:

	bit_1	bit_2	bit_3	Output
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	0
4	1	0	0	1

	bit_1	bit_2	bit_3	Output
5	1	0	1	1
6	1	1	0	1
7	1	1	1	1

In [11]:

```
df = df.drop('Output',axis=1)
df
```

Out[11]:

	bit_1	bit_2	bit_3
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Train Test Split

In [12]:

```
train_percentage = 60
test_percentage = 100 - train_percentage

print('Train Percentage :',train_percentage)
print('Test Percentage :',test_percentage)
```

```
Train Percentage : 60
Test Percentage : 40
```

In [13]:

```
import math

no_of_train_data = math.ceil(( total_number * train_percentage ) / 100)
no_of_test_data = total_number - no_of_train_data

print('No of Train Data :',no_of_train_data)
print('No of Test Data :',no_of_test_data)
```

```
No of Train Data : 5
No of Test Data : 3
```

Inititalizing Wij , bj , Wjk and bk with random values

In [14]:

```
# n is the number of nodes in input layer and hidden layer
# m is the number of nodes in output layer

unique = dict()

for i in output:
    if i not in unique:
        unique[i] = 1
    else:
```

```
        unique[i] = unique[i] + 1

print('Total Class in Output :',len(unique))
```

```
Total Class in Output : 2
```

In [15]:

```
import numpy as np
import math

# n will be as it is
m = math.ceil(np.log2(len(unique)))

print('Number of nodes in input layer :',n)
print('Number of nodes in hidden layer :',n)
print('Number of nodes in output layer :',m)
```

```
Number of nodes in input layer : 3
Number of nodes in hidden layer : 3
Number of nodes in output layer : 1
```

In [16]:

```
np.random.seed(113)
Wij = np.random.rand(n,n)
bj = np.random.rand(n)
Wjk = np.random.rand(n,m)
bk = np.random.rand(m)
```

In [17]:

```
print('Weights from i to j :\n',Wij)
print('\nBias to j :\n',bj)
print('\nWeights from j to k :\n',Wjk)
print('\nBias to k :\n',bk)
```

```
Weights from i to j :
 [[0.85198549 0.0739036  0.89493176]
 [0.43649355 0.12767773 0.57585787]
 [0.84047092 0.43512055 0.69591056]]

Bias to j :
 [0.6846381  0.70064837 0.77969426]

Weights from j to k :
 [[0.64274937]
 [0.96102617]
 [0.10846489]]

Bias to k :
 [0.79610634]
```

In [18]:

```
Wij[0,0]
```

Out[18]:

```
0.8519854927300882
```

Initializing Oi , netj , Oj and netk with 0

In [19]:

```
Oi = np.zeros(n)

netj = np.zeros(n)
activj = np.zeros(n)
Oj = np.zeros(n)
```

```
netk = np.zeros(m)
activk = np.zeros(m)
Ok = np.zeros(m)
```

In [20]:

```
# learning rate
learning_rate = 0.5
```

initializing delta_Wjk , delta_bk , delta_Wij , delta_bj with 0

In [21]:

```
delta_Wjk = np.zeros((n,m))
delta_bk = np.zeros(m)
delta_Wij = np.zeros((n,n))
delta_bj = np.zeros(n)
```

Forward Propagation and Backward Propagation

In [22]:

```
df
```

Out[22]:

	bit_1	bit_2	bit_3
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

In [23]:

```
output
```

Out[23]:

```
[0, 0, 0, 0, 1, 1, 1, 1]
```

In [24]:

```
df[df.columns[0]][4]
```

Out[24]:

```
1
```

In [25]:

```
np.exp(1)
```

Out[25]:

```
2.718281828459045
```

In [26]:

```
n
```

Out[26]:

3

In [27]:

```
l = list()
count = 0
while count != no_of_train_data :
    for row in range(no_of_train_data):
        # forward propagation starts

        for column in range(len(df.columns)):
            Oi[column] = df.iloc[row,column]

        for i in range(n):
            for j in range(n):
                netj[j] = netj[j] + Oi[i] * Wij[i,j]


        for j in range(n):
            activj[j] = netj[j] + bj[j]

        for j in range(n):
            Oj[j] = 1 / (1 + np.exp(-1*activj[j]))

        for j in range(n):
            for k in range(m):
                netk[k] = netk[k] + Oj[j] * Wjk[j,k]

        for k in range(m):
            activk[k] = netk[k] + bk[k]

        for k in range(m):
            Ok[k] = 1 / (1 + np.exp(-1*activk[k]))

        delta = output[row] - Ok[0] # not generalized
        error = 0.5 * ( delta ** 2)
        if error <= 0.01:
            l.append(Ok[0])
            count = count + 1
            netj = np.zeros(n)
            netk = np.zeros(m)
            continue


        # Backpropagation starts

        for j in range(n):
            for k in range(m):
                delta_Wjk[j,k] = learning_rate * delta * Oj[j] * Ok[k] * (1 - Ok[k])

        for j in range(n):
            for k in range(m):
                Wjk[j,k] = Wjk[j,k] + delta_Wjk[j,k]

        for k in range(m):
            delta_bk[k] = learning_rate * delta * Ok[k] * (1 - Ok[k])

        for k in range(m):
            bk[k] = bk[k] + delta_bk[k]

        summation = 0
        for j in range(n):
            summation = summation + Wjk[j,0] * delta

        for i in range(n):
            for j in range(n):
                delta_Wij[i,j] = learning_rate * Oi[i] * Oj[j] * (1-Oj[j]) * summation
```

```
        for i in range(n):
            for j in range(n):
                Wij[i,j] = Wij[i,j] + delta_Wij[i,j]

        for j in range(n):
            delta_bj[j] = learning_rate * Oj[j] * (1 - Oj[j]) * summation

        for j in range(n):
            bj[j] = bj[j] + delta_bj[j]

        netj = np.zeros(n)
        netk = np.zeros(m)
        count = 0
        l = list()
        break
```

In [28]:

```
count
```

Out[28]:

```
10
```

In [29]:

```
l
```

Out[29]:

```
[0.13860830144318903,
 0.11418434918960044,
 0.10888551745357877,
 0.09765934984803684,
 0.11732680057811581,
 0.10206618514048248,
 0.09910573770542183,
 0.09194846080211491,
 0.8683331736813442,
 0.8601205469920238]
```

In [30]:

```
right = 0
wrong = 0
l = list()
for row in range(no_of_train_data,total_number):
    # forward propagation starts

    for column in range(len(df.columns)):
        Oi[column] = df[df.columns[column]][row]

    for i in range(n):
        for j in range(n):
            netj[j] = netj[j] + Oi[i] * Wij[i,j]


    for j in range(n):
        activj[j] = netj[j] + bj[j]

    for j in range(n):
        Oj[j] = 1 / (1 + np.exp(-1*activj[j]))

    for j in range(n):
        for k in range(m):
            netk[k] = netk[k] + Oj[j] * Wjk[j,k]

    for k in range(m):
        activk[k] = netk[k] + bk[k]

    for k in range(m):
        Ok[k] = 1 / (1 + np.exp(-1*activk[k]))
```

```
    delta = output[row] - Ok[0] # not generalized
    error = 0.5 * ( delta ** 2)
    l.append(error)

    if error <= 0.01:
        right = right + 1
    else:
        wrong = wrong + 1

accuracy = ( right * 100 ) / no_of_test_data

print(l)
print("No of Test Data :",no_of_test_data)
print("Right :",right)
print("Wrong :",wrong)
print("Accuracy :",accuracy)
```

```
[0.010175588270903613, 0.010216402747925865, 0.010216622044655751, 0.01021662425910416, 0
.010216624284780242, 0.010216624285336566]
No of Test Data : 6
Right : 0
Wrong : 6
Accuracy : 0.0
```

Heaven's Light is Our Guide

RAJSHAHI UNIVERSITY OF ENGINEERING & TECHNOLOGY

Course Number : CSE 4204

Course Title : Sessional Based on CSE 4203

<u>Submitted To:</u>

Course Teacher : Dr. Md. Rabiul Islam

Professor

Department of Computer Science and Engineering

Rajshahi University of Engineering & Technology

<u>Submitted By:</u>

Name : Kazi Mehrab Rashid

Section : A

Roll No : 1703003

Department : Computer Science & Engineering

Institution : Rajshahi University of Engineering & Technology

Submission Date : 10/06/2023

Generating n bit of data

In [1]:

```
n = int(input('Enter Number of bits : '))
count = 0
i = n
string = 'bit_'
total_number = 2 ** n
```

In [2]:

```
value = list()
dictionary = dict()

while i >= 1 :
    key = string + str(i)
    d = 2 ** count

    while len(value) != total_number:
        for j in range(d):
            value.append(-1)
        for j in range(d):
            value.append(1)

    dictionary[key] = value
    value = list()
    count = count + 1
    i = i - 1
```

In [3]:

```
dictionary
```

Out[3]:

```
{'bit_4': [-1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1],
 'bit_3': [-1, -1, 1, 1, -1, -1, 1, 1, -1, -1, 1, 1, -1, -1, 1, 1],
 'bit_2': [-1, -1, -1, -1, 1, 1, 1, 1, -1, -1, -1, -1, 1, 1, 1, 1],
 'bit_1': [-1, -1, -1, -1, -1, -1, -1, -1, 1, 1, 1, 1, 1, 1, 1, 1]}
```

In [4]:

```
list(dictionary.items())
```

Out[4]:

```
[('bit_4', [-1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1]),
 ('bit_3', [-1, -1, 1, 1, -1, -1, 1, 1, -1, -1, 1, 1, -1, -1, 1, 1]),
 ('bit_2', [-1, -1, -1, -1, 1, 1, 1, 1, -1, -1, -1, -1, 1, 1, 1, 1]),
 ('bit_1', [-1, -1, -1, -1, -1, -1, -1, -1, 1, 1, 1, 1, 1, 1, 1, 1])]
```

In [5]:

```
l = list(dictionary.items())
l
```

Out[5]:

```
[('bit_4', [-1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1]),
 ('bit_3', [-1, -1, 1, 1, -1, -1, 1, 1, -1, -1, 1, 1, -1, -1, 1, 1]),
 ('bit_2', [-1, -1, -1, -1, 1, 1, 1, 1, -1, -1, -1, -1, 1, 1, 1, 1]),
 ('bit_1', [-1, -1, -1, -1, -1, -1, -1, -1, 1, 1, 1, 1, 1, 1, 1, 1])]
```

In [6]:

```
reversed_dictionary = dict()
i = n-1
```

```python
while i >= 0:
    reversed_dictionary[l[i][0]] = l[i][1]
    i = i - 1
reversed_dictionary
```

Out[6]:

```
{'bit_1': [-1, -1, -1, -1, -1, -1, -1, -1, 1, 1, 1, 1, 1, 1, 1, 1],
 'bit_2': [-1, -1, -1, -1, 1, 1, 1, 1, -1, -1, -1, -1, 1, 1, 1, 1],
 'bit_3': [-1, -1, 1, 1, -1, -1, 1, 1, -1, -1, 1, 1, -1, -1, 1, 1],
 'bit_4': [-1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1]}
```

In [7]:

```python
dictionary = reversed_dictionary
dictionary
```

Out[7]:

```
{'bit_1': [-1, -1, -1, -1, -1, -1, -1, -1, 1, 1, 1, 1, 1, 1, 1, 1],
 'bit_2': [-1, -1, -1, -1, 1, 1, 1, 1, -1, -1, -1, -1, 1, 1, 1, 1],
 'bit_3': [-1, -1, 1, 1, -1, -1, 1, 1, -1, -1, 1, 1, -1, -1, 1, 1],
 'bit_4': [-1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1]}
```

In [8]:

```python
output = dictionary['bit_1']
output
```

Out[8]:

```
[-1, -1, -1, -1, -1, -1, -1, -1, 1, 1, 1, 1, 1, 1, 1, 1]
```

In [9]:

```python
import pandas as pd

df = pd.DataFrame(data=dictionary)
df
```

Out[9]:

	bit_1	bit_2	bit_3	bit_4
0	-1	-1	-1	-1
1	-1	-1	-1	1
2	-1	-1	1	-1
3	-1	-1	1	1
4	-1	1	-1	-1
5	-1	1	-1	1
6	-1	1	1	-1
7	-1	1	1	1
8	1	-1	-1	-1
9	1	-1	-1	1
10	1	-1	1	-1
11	1	-1	1	1
12	1	1	-1	-1
13	1	1	-1	1
14	1	1	1	-1
15	1	1	1	1

In [27]:

```python
# saving the dataframe
```

```
# saving the dataframe
df.to_csv('file1.csv')
```

Train Test

In [10]:

```
train_percentage = 60
test_percentage = 100 - train_percentage

print('Train Percentage :',train_percentage)
print('Test Percentage :',test_percentage)
```

```
Train Percentage : 60
Test Percentage : 40
```

In [11]:

```
import math

no_of_train_data = math.ceil(( total_number * train_percentage ) / 100)
no_of_test_data = total_number - no_of_train_data

print('No of Train Data :',no_of_train_data)
print('No of Test Data :',no_of_test_data)
```

```
No of Train Data : 10
No of Test Data : 6
```

Weight Adjusting

In [12]:

```
df.shape
```

Out[12]:

```
(16, 4)
```

In [13]:

```
m = df.shape[1]
m
```

Out[13]:

```
4
```

In [14]:

```
import numpy as np
w = np.zeros((m,m))
w
```

Out[14]:

```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

In [15]:

```
summation = 0

for i in range(m):
    for j in range(m):
        if i != j :
            for pattern in range(no_of_train_data):
                summation = summation + df.iloc[pattern,i] * df.iloc[pattern,j]
```

```
            w[i,j] = summation
            summation = 0
```

In [16]:

```
w
```

Out[16]:

```
array([[ 0., -2., -2.,  0.],
       [-2.,  0.,  2.,  0.],
       [-2.,  2.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
```

Pattern matching by taking test data

In [17]:

```
new_pattern = list()
temp = list()
sum = 0
count = 1
flag = True

for pattern in range(no_of_train_data,total_number):
    for j in range(m):
        new_pattern.append(df.iloc[pattern,j])
    print('New Pattern :',new_pattern)

    while flag:
        for row in range(m):
            for j in range(m):
                sum = sum + w[row,j] * new_pattern[j]

            if sum > 0 :
                new_pattern[row] = 1
            elif sum < 0 :
                new_pattern[row] = -1

            print('At Neuron ',row,':',new_pattern)

            if len(temp) == 0 :
                temp = new_pattern.copy()
            else:
                if temp == new_pattern:
                    count = count + 1
                else:
                    count = 1
                temp = new_pattern.copy()

            if count >= 5:
                flag = False
                break

            sum = 0

    print('Converged pattern of the test pattern :',new_pattern)

    for p in range(no_of_train_data):
        if new_pattern == list(df.iloc[p]):
            print('Cluster with',p)
            print('------------------------------------------------------')
            break

    new_pattern = list()
    temp = list()
    sum = 0
    count = 1
    flag = True
```

```
New Pattern : [1, -1, 1, -1]
```

```
At Neuron  0 : [1, -1, 1, -1]
At Neuron  1 : [1, -1, 1, -1]
At Neuron  2 : [1, -1, -1, -1]
At Neuron  3 : [1, -1, -1, -1]
At Neuron  0 : [1, -1, -1, -1]
At Neuron  1 : [1, -1, -1, -1]
At Neuron  2 : [1, -1, -1, -1]
Converged pattern of the test pattern : [1, -1, -1, -1]
Cluster with 8
-----------------------------------------------------
New Pattern : [1, -1, 1, 1]
At Neuron  0 : [1, -1, 1, 1]
At Neuron  1 : [1, -1, 1, 1]
At Neuron  2 : [1, -1, -1, 1]
At Neuron  3 : [1, -1, -1, 1]
At Neuron  0 : [1, -1, -1, 1]
At Neuron  1 : [1, -1, -1, 1]
At Neuron  2 : [1, -1, -1, 1]
Converged pattern of the test pattern : [1, -1, -1, 1]
Cluster with 9
-----------------------------------------------------
New Pattern : [1, 1, -1, -1]
At Neuron  0 : [1, 1, -1, -1]
At Neuron  1 : [1, -1, -1, -1]
At Neuron  2 : [1, -1, -1, -1]
At Neuron  3 : [1, -1, -1, -1]
At Neuron  0 : [1, -1, -1, -1]
At Neuron  1 : [1, -1, -1, -1]
Converged pattern of the test pattern : [1, -1, -1, -1]
Cluster with 8
-----------------------------------------------------
New Pattern : [1, 1, -1, 1]
At Neuron  0 : [1, 1, -1, 1]
At Neuron  1 : [1, -1, -1, 1]
At Neuron  2 : [1, -1, -1, 1]
At Neuron  3 : [1, -1, -1, 1]
At Neuron  0 : [1, -1, -1, 1]
At Neuron  1 : [1, -1, -1, 1]
Converged pattern of the test pattern : [1, -1, -1, 1]
Cluster with 9
-----------------------------------------------------
New Pattern : [1, 1, 1, -1]
At Neuron  0 : [-1, 1, 1, -1]
At Neuron  1 : [-1, 1, 1, -1]
At Neuron  2 : [-1, 1, 1, -1]
At Neuron  3 : [-1, 1, 1, -1]
At Neuron  0 : [-1, 1, 1, -1]
Converged pattern of the test pattern : [-1, 1, 1, -1]
Cluster with 6
-----------------------------------------------------
New Pattern : [1, 1, 1, 1]
At Neuron  0 : [-1, 1, 1, 1]
At Neuron  1 : [-1, 1, 1, 1]
At Neuron  2 : [-1, 1, 1, 1]
At Neuron  3 : [-1, 1, 1, 1]
At Neuron  0 : [-1, 1, 1, 1]
Converged pattern of the test pattern : [-1, 1, 1, 1]
Cluster with 7
-----------------------------------------------------
```

----------------------Just Clearing my doubts-------------------

In [18]:

```
t = [1,2,2]
m = [1,2,3]
t
```

Out[18]:

```
[1, 2, 2]
```

[1, 2, 2]

In [19]:

```
if t == m:
    print('hi')
```

In [20]:

```
df
```

Out[20]:

	bit_1	bit_2	bit_3	bit_4
0	-1	-1	-1	-1
1	-1	-1	-1	1
2	-1	-1	1	-1
3	-1	-1	1	1
4	-1	1	-1	-1
5	-1	1	-1	1
6	-1	1	1	-1
7	-1	1	1	1
8	1	-1	-1	-1
9	1	-1	-1	1
10	1	-1	1	-1
11	1	-1	1	1
12	1	1	-1	-1
13	1	1	-1	1
14	1	1	1	-1
15	1	1	1	1

In [21]:

```
m = [-1,-1,-1]
```

In [22]:

```
df.iloc[0]
```

Out[22]:

```
bit_1    -1
bit_2    -1
bit_3    -1
bit_4    -1
Name: 0, dtype: int64
```

In [23]:

```
list(df.iloc[0])
```

Out[23]:

```
[-1, -1, -1, -1]
```

In [24]:

```
if m == list(df.iloc[0]):
    print('hi')
```

In [25]:

```
m = [1,2,3]
t = []
t = m.copy()
print(m)
print(t)
```

```
[1, 2, 3]
[1, 2, 3]
```

In [26]:

```
m[0] = 10
print(m)
print(t)
```

```
[10, 2, 3]
[1, 2, 3]
```

KNN implementation on Gene Expression Data Set (Generalized)

60% train data , 40% test data

Library

In [2]:

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import math
```

Data Set

In [3]:

```
df = pd.read_csv('gene_expression.csv')
df
```

Out[3]:

	Gene One	Gene Two	Cancer Present
0	4.3	3.9	1
1	2.5	6.3	0
2	5.7	3.9	1
3	6.1	6.2	0
4	7.4	3.4	1
...	...	...	...
2995	5.0	6.5	1
2996	3.4	6.6	0
2997	2.7	6.5	0
2998	3.3	5.6	0
2999	4.6	8.2	0

3000 rows × 3 columns

In [4]:

```
df.columns
```

Out[4]:

```
Index(['Gene One', 'Gene Two', 'Cancer Present'], dtype='object')
```

In [5]:

```
n = len(df.columns)
n
```

Out[5]:

```
3
```

In [6]:

In [6]:

```
df[df.columns[n-1]].value_counts()
```

Out[6]:

```
1    1500
0    1500
Name: Cancer Present, dtype: int64
```

In [7]:

```
df[df.columns[n-1]].value_counts().index
```

Out[7]:

```
Int64Index([1, 0], dtype='int64')
```

In [8]:

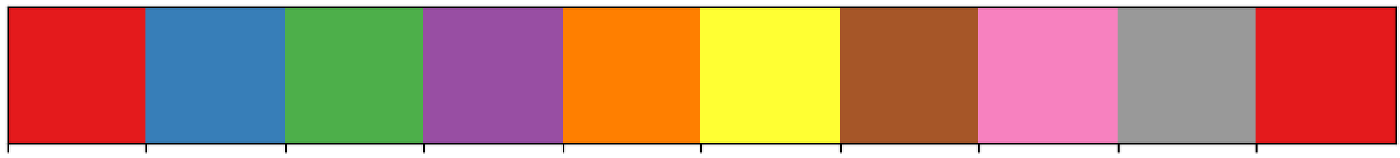
```
df[df.columns[n-1]].nunique()
```

Out[8]:

```
2
```

In [9]:

```
custom_palette = sns.color_palette("Set1",10)
sns.palplot(custom_palette)
```



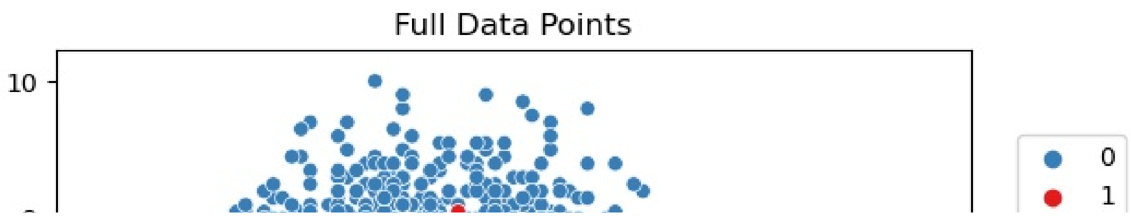
https://www.codecademy.com/article/seaborn-design-ii

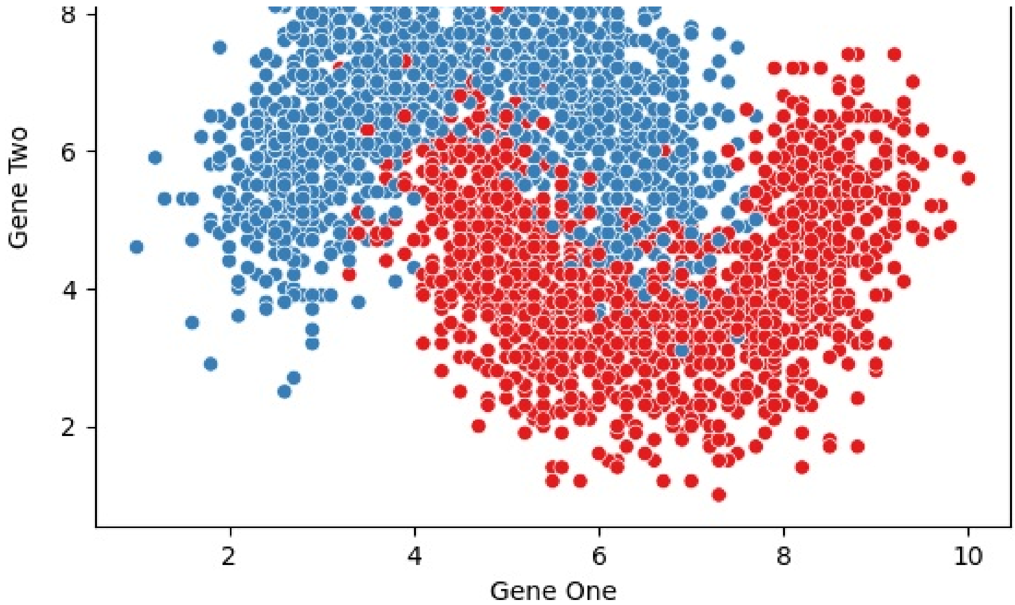
In [10]:

```
custom_palette = sns.color_palette("Set1", df[df.columns[2]].nunique()+1)
color_dict = dict()
markers_dict = dict()
j = 0

for i in df[df.columns[2]].value_counts().index:
    color_dict[i] = custom_palette[j]
    markers_dict[i] = 'o'
    j = j + 1

color_dict['Test Point'] = custom_palette[2]
markers_dict['Test Point'] = 'X'

# print(color_dict)
# print(markers_dict)

sns.scatterplot(x = df[df.columns[0]],y = df[df.columns[1]],hue=df[df.columns[2]],palett
e=color_dict,style=df[df.columns[2]],markers=markers_dict)
plt.title('Full Data Points')
plt.legend(loc=(1.05,0.75))
```

Out[10]:

```
<matplotlib.legend.Legend at 0x1622da71660>
```





Determing the value of K

In [11]:

```
len(df)
```

Out[11]:

```
3000
```

In [12]:

```
k = math.floor(math.sqrt(len(df)))

if k%2==0 :
    k = k + 1

print(k)
```

```
55
```

Train Test split

In [13]:

```
train_percentage = 60
test_percentage = 100 - train_percentage

print('Train Percentage :',train_percentage)
print('Test Percentage :',test_percentage)
```

```
Train Percentage : 60
Test Percentage : 40
```

In [14]:

```
no_of_train_data = math.ceil((train_percentage * len(df)) / 100)
print('No of train data :',no_of_train_data)

no_of_test_data = len(df) - no_of_train_data
print('No of test data',no_of_test_data)
```

```
No of train data : 1800
No of test data 1200
```

In [15]:

```
df.head(no_of_train_data)
```

Out[15]:

	Gene One	Gene Two	Cancer Present
0	4.3	3.9	1
1	2.5	6.3	0
2	5.7	3.9	1
3	6.1	6.2	0
4	7.4	3.4	1
...	...	...	...
1795	2.9	7.3	0
1796	6.1	3.3	1
1797	6.5	8.0	0
1798	2.5	5.5	0
1799	2.6	6.0	0

1800 rows × 3 columns

In [16]:

```
df_train = df.head(no_of_train_data)
df_train
```

Out[16]:

	Gene One	Gene Two	Cancer Present
0	4.3	3.9	1
1	2.5	6.3	0
2	5.7	3.9	1
3	6.1	6.2	0
4	7.4	3.4	1
...	...	...	...
1795	2.9	7.3	0
1796	6.1	3.3	1
1797	6.5	8.0	0
1798	2.5	5.5	0
1799	2.6	6.0	0

1800 rows × 3 columns

In [17]:

```
df.tail(no_of_test_data)
```

Out[17]:

	Gene One	Gene Two	Cancer Present
1800	8.1	3.4	1
1801	7.7	4.1	1
1802	5.1	4.5	1
1803	4.3	9.6	0
1804	7.7	5.4	1
...	...	...	...

	Gene One	Gene Two	Cancer Present
2995	5.0	6.5	1
2996	3.4	6.6	0
2997	2.7	6.5	0
2998	3.3	5.6	0
2999	4.6	8.2	0

1200 rows × 3 columns

In [18]:

```
df_test = df.tail(no_of_test_data)
df_test
```

Out[18]:

	Gene One	Gene Two	Cancer Present
1800	8.1	3.4	1
1801	7.7	4.1	1
1802	5.1	4.5	1
1803	4.3	9.6	0
1804	7.7	5.4	1
...	...	...	...
2995	5.0	6.5	1
2996	3.4	6.6	0
2997	2.7	6.5	0
2998	3.3	5.6	0
2999	4.6	8.2	0

1200 rows × 3 columns

In [19]:

```
df_test = df_test.reset_index()
df_test
```

Out[19]:

	index	Gene One	Gene Two	Cancer Present
0	1800	8.1	3.4	1
1	1801	7.7	4.1	1
2	1802	5.1	4.5	1
3	1803	4.3	9.6	0
4	1804	7.7	5.4	1
...	...	...	...	...
1195	2995	5.0	6.5	1
1196	2996	3.4	6.6	0
1197	2997	2.7	6.5	0
1198	2998	3.3	5.6	0
1199	2999	4.6	8.2	0

1200 rows × 4 columns

In [20]:

```
df_test = df_test.drop('index',axis=1)
df_test
```

Out[20]:

	Gene One	Gene Two	Cancer Present
0	8.1	3.4	1
1	7.7	4.1	1
2	5.1	4.5	1
3	4.3	9.6	0
4	7.7	5.4	1
...	...	...	...
1195	5.0	6.5	1
1196	3.4	6.6	0
1197	2.7	6.5	0
1198	3.3	5.6	0
1199	4.6	8.2	0

1200 rows × 3 columns

In [21]:

```
df_temp = df.copy()
df_temp
```

Out[21]:

	Gene One	Gene Two	Cancer Present
0	4.3	3.9	1
1	2.5	6.3	0
2	5.7	3.9	1
3	6.1	6.2	0
4	7.4	3.4	1
...	...	...	...
2995	5.0	6.5	1
2996	3.4	6.6	0
2997	2.7	6.5	0
2998	3.3	5.6	0
2999	4.6	8.2	0

3000 rows × 3 columns

In [22]:

```
df_temp['Cancer Present'][no_of_train_data:] = ['Test Point'] * no_of_test_data
df_temp
```

```
C:\Users\HP\AppData\Local\Temp\ipykernel_4124\2074753982.py:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_g
uide/indexing.html#returning-a-view-versus-a-copy
  df_temp['Cancer Present'][no_of_train_data:] = ['Test Point'] * no_of_test_data
```

Out[22]:

Gene One Gene Two Cancer Present

	Gene One	Gene Two	Cancer Present
0	4.3	3.9	1
1	2.5	6.3	0
2	5.7	3.9	1
3	6.1	6.2	0
4	7.4	3.4	1
...	...	...	...
2995	5.0	6.5	Test Point
2996	3.4	6.6	Test Point
2997	2.7	6.5	Test Point
2998	3.3	5.6	Test Point
2999	4.6	8.2	Test Point

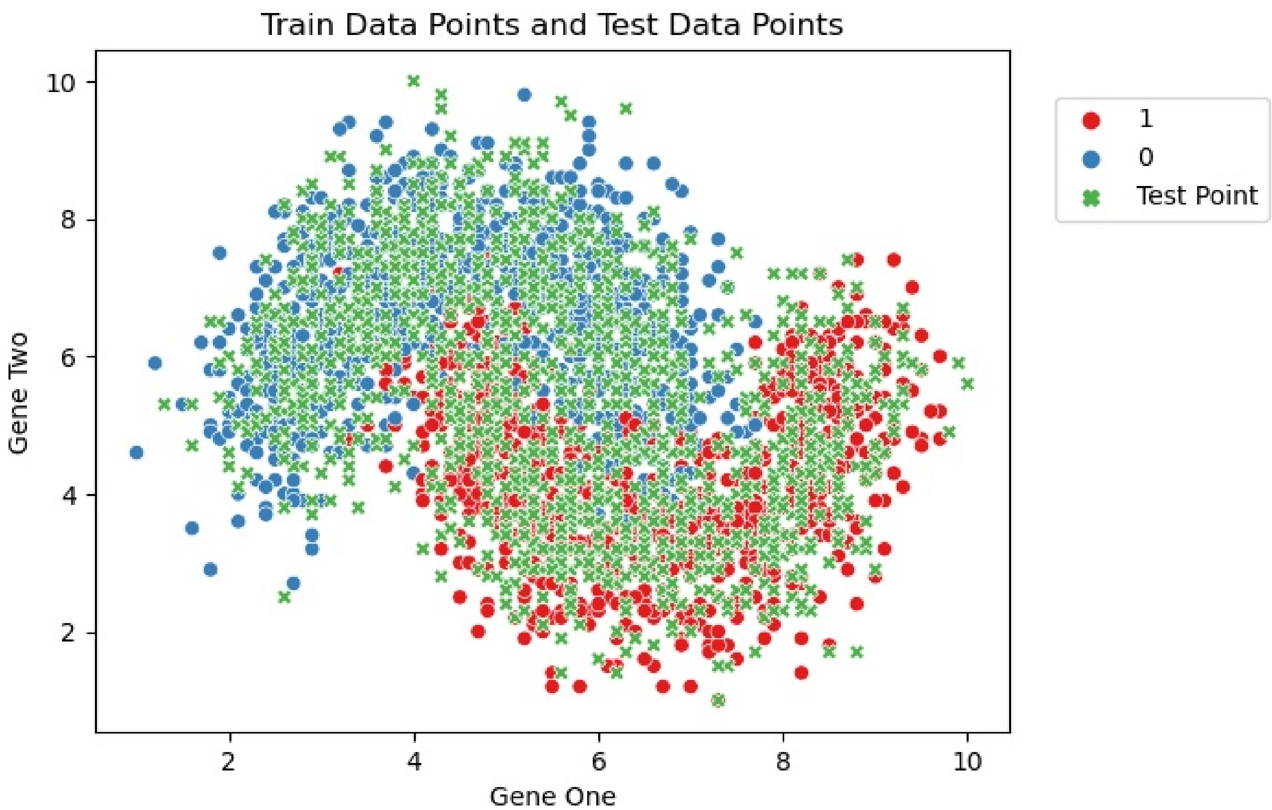
3000 rows × 3 columns

In [23]:

```
sns.scatterplot(x = df_temp[df_temp.columns[0]],y = df_temp[df_temp.columns[1]],hue=df_t
emp[df_temp.columns[2]],palette=color_dict,style=df_temp[df_temp.columns[2]],markers=mar
kers_dict)
plt.title('Train Data Points and Test Data Points')
plt.legend(loc=(1.05,0.75))

# hue without palette : sns will provide default color for each group or class in df_temp
['Cancer Present]
# huw with palette : sns will provide color we want for each group or class in df_temp['C
ancer Present]
# style without markers : sns will provide default shape for each group or class in df_te
mp['Cancer Present]
# style with markers : sns will provide shape we want for each group or class in df_temp[
'Cancer Present]
```

Out[23]:

```
<matplotlib.legend.Legend at 0x1623352bfd0>
```



Calculating Euclean Distance from Test point to Train point , sorting it

Calculating Euclean Distance from Test point to Train point , sorting it ascending order and then finding the nearest neighbor

In [24]:

```
df_train
```

Out[24]:

	Gene One	Gene Two	Cancer Present
0	4.3	3.9	1
1	2.5	6.3	0
2	5.7	3.9	1
3	6.1	6.2	0
4	7.4	3.4	1
...	...	...	...
1795	2.9	7.3	0
1796	6.1	3.3	1
1797	6.5	8.0	0
1798	2.5	5.5	0
1799	2.6	6.0	0

1800 rows × 3 columns

In [25]:

```
df_test
```

Out[25]:

	Gene One	Gene Two	Cancer Present
0	8.1	3.4	1
1	7.7	4.1	1
2	5.1	4.5	1
3	4.3	9.6	0
4	7.7	5.4	1
...	...	...	...
1195	5.0	6.5	1
1196	3.4	6.6	0
1197	2.7	6.5	0
1198	3.3	5.6	0
1199	4.6	8.2	0

1200 rows × 3 columns

In [26]:

```
df_test.iloc[0]
```

Out[26]:

```
Gene One          8.1
Gene Two          3.4
Cancer Present    1.0
Name: 0, dtype: float64
```

In [27]:

```
df_train.iloc[0][0]
```

Out[27]:

```
4.3
```

In [28]:

```
df_train[df.columns[n-1]][0]
```

Out[28]:

```
1
```

In [29]:

```
df_train.columns
```

Out[29]:

```
Index(['Gene One', 'Gene Two', 'Cancer Present'], dtype='object')
```

In [30]:

```
# distance_list = list()
# class_name_list = list()
# total_distance = 0

# # calculating euclidean distance from test data to train data

# for i in range(len(df_test)):
#     for j in range(len(df_train)):
#         for c in range(n-1):
#             distance = (df_test.iloc[i][c] - df_train.iloc[j][c])**2
#             total_distance = total_distance + distance
#         total_distance = math.sqrt(total_distance)
#         distance_list.append((df_train[df_train.columns[n-1]][j],total_distance))
#         total_distance = 0


#     # sorting all those distances

#     for ii in range(len(distance_list)):
#         for jj in range(ii+1,len(distance_list)):
#             if distance_list[jj][1] < distance_list[ii][1]:
#                 temp = distance_list[ii]
#                 distance_list[ii] = distance_list[jj]
#                 distance_list[jj] = temp


#     # selecting first 'k' points and then counting the number of classes

#     count = dict()
#     for ii in range(k):
#         if distance_list[ii][0] not in count:
#             count[distance_list[ii][0]] = 1
#         else:
#             count[distance_list[ii][0]] = count[distance_list[ii][0]] + 1


#     # finding out the most nearest class

#     min = 0
#     for ii in count:
#         if count[ii] > min:
#             class_name = ii
#             min = count[ii]

#     class_name_list.append(class_name)
#     distance_list = list()

# print(class_name_list)
```

This code takes a lot of time for giving the output since we are using nested loop here and also used bubble sort inside the first loop which consumed too much time. and then again we had to select first k points with smallest distance using one loop and find number of classes with another loop.

so too much use of loop made this code too much slower.

So here i have optimized this code with eliminating the bubble sort.

here , after i got the distance list , i only took the first k points with smallest distance without sorting them and everytime i took k(i) smallest point , i counted the class number, the point belongs to , too in dicitonary.after that I just found the most nearest neghbor

In [31]:

```
class_name_list = list()

# calculating euclidean distance from test data to train data

for i in range(len(df_test)):
    distance_list = list()
    for j in range(len(df_train)):
        distance = np.linalg.norm(df_test.iloc[i]-df_train.iloc[j])
        distance_list.append((df_train[df_train.columns[n-1]][j],distance))
    # print(distance_list)



    # selecting first 'k' points with smallest distance without sorting and then counting the number of classes

    minimum = None
    checked = dict()
    count = dict()
    min_list = list()

    for ii in range(k):
        for jj in range(len(distance_list)):
            if jj not in checked :
                if minimum is None :
                    minimum = distance_list[jj]
                elif distance_list[jj][1] < minimum[1]:
                    minimum = distance_list[jj]
                    index = jj
        min_list.append(minimum)
        if minimum[0] not in count:
            count[minimum[0]] = 1
        else:
            count[minimum[0]] = count[minimum[0]] + 1
        checked[index] = 1
        minimum = None

    # print(min_list)
    # print(count)

    # finding out the most nearest class

    min = 0
    for ii in count:
        if count[ii] > min:
            class_name = ii
            min = count[ii]
    class_name_list.append(class_name)
print(class_name_list)
```

```
[1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0,
1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1,
0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0,
1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1,
1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0,
1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1,
```

```
 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0,
 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1,
 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0,
 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0,
 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0,
 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1,
 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0,
 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1,
 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1,
 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0,
 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0,
 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0,
 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0,
 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1,
 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1,
 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0,
 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1,
 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0,
 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1,
 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1,
 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1,
 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0,
 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1,
 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0,
 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0,
 0, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1,
 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0,
 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1,
 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0,
 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1,
 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0,
 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1,
 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0]
```

In [32]:

```
print(len(class_name_list))
```

```
1200
```

In [33]:

```
df_temp2 = df.copy()
df_temp2
```

Out[33]:

	Gene One	Gene Two	Cancer Present
0	4.3	3.9	1
1	2.5	6.3	0
2	5.7	3.9	1
3	6.1	6.2	0
4	7.4	3.4	1
...	...	...	...
2995	5.0	6.5	1
2996	3.4	6.6	0
2997	2.7	6.5	0
2998	3.3	5.6	0
2999	4.6	8.2	0

3000 rows × 3 columns

In [34]:

```
df_temp2[df_temp2.columns[n-1]][no_of_train_data:] = class_name_list
df_temp2
```

```
C:\Users\HP\AppData\Local\Temp\ipykernel_4124\2165292902.py:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_g
uide/indexing.html#returning-a-view-versus-a-copy
  df_temp2[df_temp2.columns[n-1]][no_of_train_data:] = class_name_list
```

Out[34]:

	Gene One	Gene Two	Cancer Present
0	4.3	3.9	1
1	2.5	6.3	0
2	5.7	3.9	1
3	6.1	6.2	0
4	7.4	3.4	1
...	...	...	...
2995	5.0	6.5	1
2996	3.4	6.6	0
2997	2.7	6.5	0
2998	3.3	5.6	0
2999	4.6	8.2	0

3000 rows × 3 columns

In [35]:

```
color_dict
```

Out[35]:

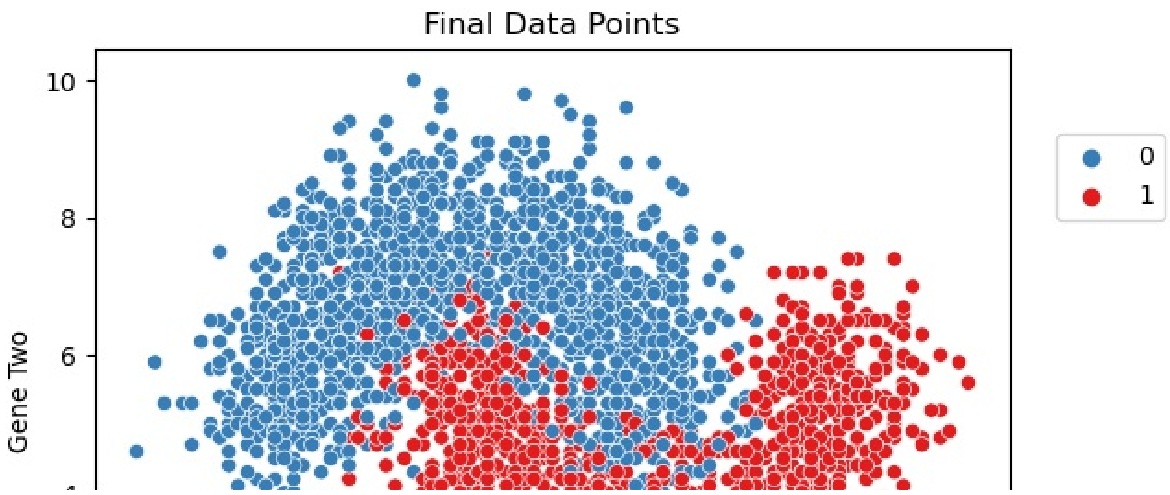
```
{1: (0.8941176470588236, 0.10196078431372549, 0.10980392156862745),
 0: (0.21568627450980393, 0.49411764705882355, 0.7215686274509804),
 'Test Point': (0.30196078431372547, 0.6862745098039216, 0.2901960784313726)}
```

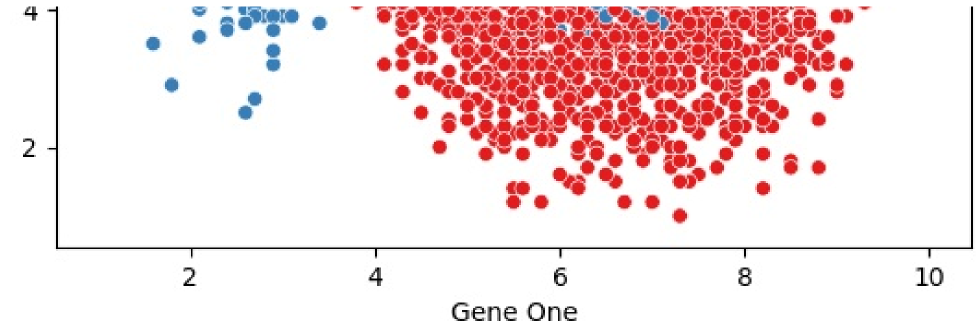
In [36]:

```
sns.scatterplot(x = df_temp2[df_temp2.columns[0]],y = df_temp2[df_temp2.columns[1]],hue=d
f_temp2[df_temp2.columns[2]],palette=color_dict,style=df_temp2[df_temp2.columns[2]],marke
rs=markers_dict)
plt.title('Final Data Points')
plt.legend(loc=(1.05,0.75))
```

Out[36]:

```
<matplotlib.legend.Legend at 0x162335714b0>
```



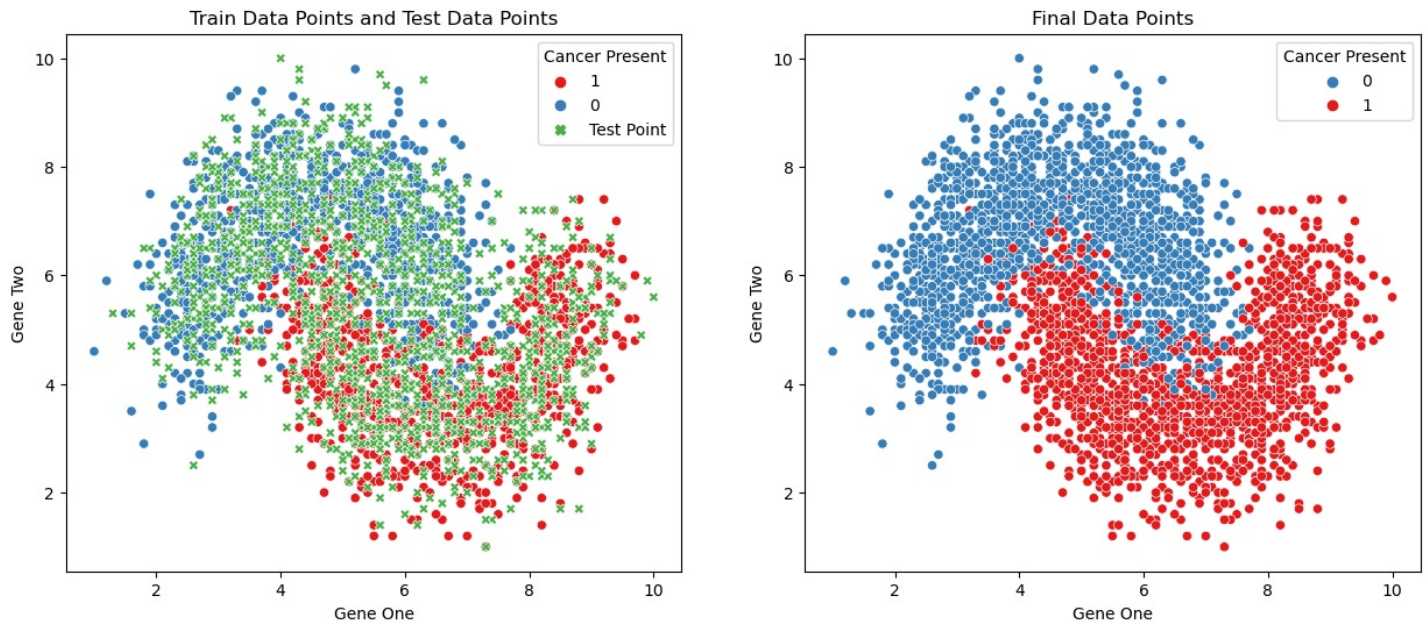


In [37]:

```
plt.figure(figsize=(15,6))

plt.subplot(1, 2, 1) # row 1, col 2 index 1

sns.scatterplot(x = df_temp[df_temp.columns[0]],y = df_temp[df_temp.columns[1]],hue=df_t
emp[df_temp.columns[2]],palette=color_dict,style=df_temp[df_temp.columns[2]],markers=mar
kers_dict)
plt.title('Train Data Points and Test Data Points')

# hue without palette : sns will provide default color for each group or class in df_temp
['Cancer Present]
# huw with palette : sns will provide color we want for each group or class in df_temp['C
ancer Present]
# style without markers : sns will provide default shape for each group or class in df_te
mp['Cancer Present]
# style with markers : sns will provide shape we want for each group or class in df_temp[
'Cancer Present]




plt.subplot(1, 2, 2) # index 2


sns.scatterplot(x = df_temp2[df_temp2.columns[0]],y = df_temp2[df_temp2.columns[1]],hue=d
f_temp2[df_temp2.columns[2]],palette=color_dict,style=df_temp2[df_temp2.columns[2]],marke
rs=markers_dict)
plt.title('Final Data Points')
```

Out[37]:

```
Text(0.5, 1.0, 'Final Data Points')
```



In [41]:

```
right = 0
```

```
wrong = 0

for i in range(len(class_name_list)):
    if class_name_list[i] == df_test[df_test.columns[n-1]][i]:
        right = right + 1
    else:
        wrong = wrong + 1

print(len(df_test))
print(right,wrong)
```

```
1200
1189 11
```

In [39]:

```
accuarcy = (right * 100) / len(class_name_list)
print(accuarcy)
```

```
99.08333333333333
```

In [40]:

```
from sklearn.metrics import accuracy_score

accuracy_score(df_test[df_test.columns[n-1]],class_name_list) * 100
```

Out[40]:

```
99.08333333333333
```

Generating n bit of data

In [2]:

```
n = int(input('Enter Number of bits : '))
count = 0
i = n
string = 'bit_'
total_number = 2 ** n
```

In [3]:

```
value = list()
dictionary = dict()

while i >= 1 :
    key = string + str(i)
    d = 2 ** count

    while len(value) != total_number:
        for j in range(d):
            value.append(0)
        for j in range(d):
            value.append(1)

    dictionary[key] = value
    value = list()
    count = count + 1
    i = i - 1
```

In [4]:

```
#dictionary
```

In [5]:

```
#list(dictionary.items())
```

In [6]:

```
l = list(dictionary.items())
#l
```

In [7]:

```
reversed_dictionary = dict()
i = n-1
while i >= 0:
    reversed_dictionary[l[i][0]] = l[i][1]
    i = i - 1
#reversed_dictionary
```

In [8]:

```
dictionary = reversed_dictionary
#dictionary
```

In [9]:

```
output = dictionary['bit_1']
#output
```

In [100]:

```
import pandas as pd
```

```
df = pd.DataFrame(data=dictionary)
df
```

Out[100]:

	bit_1	bit_2	bit_3	bit_4	bit_5
0	0	0	0	0	0
1	0	0	0	0	1
2	0	0	0	1	0
3	0	0	0	1	1
4	0	0	1	0	0
5	0	0	1	0	1
6	0	0	1	1	0
7	0	0	1	1	1
8	0	1	0	0	0
9	0	1	0	0	1
10	0	1	0	1	0
11	0	1	0	1	1
12	0	1	1	0	0
13	0	1	1	0	1
14	0	1	1	1	0
15	0	1	1	1	1
16	1	0	0	0	0
17	1	0	0	0	1
18	1	0	0	1	0
19	1	0	0	1	1
20	1	0	1	0	0
21	1	0	1	0	1
22	1	0	1	1	0
23	1	0	1	1	1
24	1	1	0	0	0
25	1	1	0	0	1
26	1	1	0	1	0
27	1	1	0	1	1
28	1	1	1	0	0
29	1	1	1	0	1
30	1	1	1	1	0
31	1	1	1	1	1

In [101]:

```
df['Output'] = output
df
```

Out[101]:

	bit_1	bit_2	bit_3	bit_4	bit_5	Output
0	0	0	0	0	0	0
1	0	0	0	0	1	0

2	bit_1	bit_2	bit_3	bit_4	bit_5	Output
3	0	0	0	1	1	0
4	0	0	1	0	0	0
5	0	0	1	0	1	0
6	0	0	1	1	0	0
7	0	0	1	1	1	0
8	0	1	0	0	0	0
9	0	1	0	0	1	0
10	0	1	0	1	0	0
11	0	1	0	1	1	0
12	0	1	1	0	0	0
13	0	1	1	0	1	0
14	0	1	1	1	0	0
15	0	1	1	1	1	0
16	1	0	0	0	0	1
17	1	0	0	0	1	1
18	1	0	0	1	0	1
19	1	0	0	1	1	1
20	1	0	1	0	0	1
21	1	0	1	0	1	1
22	1	0	1	1	0	1
23	1	0	1	1	1	1
24	1	1	0	0	0	1
25	1	1	0	0	1	1
26	1	1	0	1	0	1
27	1	1	0	1	1	1
28	1	1	1	0	0	1
29	1	1	1	0	1	1
30	1	1	1	1	0	1
31	1	1	1	1	1	1

In [102]:

```
df = df.drop('Output',axis=1)
df
```

Out[102]:

	bit_1	bit_2	bit_3	bit_4	bit_5
0	0	0	0	0	0
1	0	0	0	0	1
2	0	0	0	1	0
3	0	0	0	1	1
4	0	0	1	0	0
5	0	0	1	0	1
6	0	0	1	1	0
7	0	0	1	1	1
8	0	1	0	0	0

	bit_1	bit_2	bit_3	bit_4	bit_5
9	0	1	0	0	1
10	0	1	0	1	0
11	0	1	0	1	1
12	0	1	1	0	0
13	0	1	1	0	1
14	0	1	1	1	0
15	0	1	1	1	1
16	1	0	0	0	0
17	1	0	0	0	1
18	1	0	0	1	0
19	1	0	0	1	1
20	1	0	1	0	0
21	1	0	1	0	1
22	1	0	1	1	0
23	1	0	1	1	1
24	1	1	0	0	0
25	1	1	0	0	1
26	1	1	0	1	0
27	1	1	0	1	1
28	1	1	1	0	0
29	1	1	1	0	1
30	1	1	1	1	0
31	1	1	1	1	1

In [103]:

```
df.iloc[0,0]
```

Out[103]:

```
0
```

In [104]:

```
df.shape
```

Out[104]:

```
(32, 5)
```

In [105]:

```
n = df.shape[0]
m = df.shape[1]

print('number of total rows :',n)
print('number of features :',m)
```

```
number of total rows : 32
number of features : 5
```

Train Test

In [106]:

```
train_percentage = 60
```

```
test_percentage = 100 - train_percentage

print('Train Percentage :',train_percentage)
print('Test Percentage :',test_percentage)
```

```
Train Percentage : 60
Test Percentage : 40
```

In [107]:

```
import math

no_of_train_data = math.ceil(( total_number * train_percentage ) / 100)
no_of_test_data = total_number - no_of_train_data

print('No of Train Data :',no_of_train_data)
print('No of Test Data :',no_of_test_data)
```

```
No of Train Data : 20
No of Test Data : 12
```

In [108]:

```
n = no_of_train_data # row
m = df.shape[1] # column

print('number of train rows :',n)
print('number of features :',m)
```

```
number of train rows : 20
number of features : 5
```

Initializing weight with random number

In [109]:

```
import numpy as np

np.random.seed(113)
w = np.random.rand(n,m)
print(w)
```

```
[[0.85198549 0.0739036  0.89493176 0.43649355 0.12767773]
 [0.57585787 0.84047092 0.43512055 0.69591056 0.6846381 ]
 [0.70064837 0.77969426 0.64274937 0.96102617 0.10846489]
 [0.79610634 0.83258008 0.26600836 0.83668539 0.53212691]
 [0.51690756 0.09858771 0.91886899 0.66665849 0.17477948]
 [0.21769151 0.46787528 0.43589124 0.88935448 0.22259927]
 [0.58901937 0.27720157 0.52572218 0.25935711 0.52894863]
 [0.31214075 0.54416225 0.2420565  0.09423802 0.18946638]
 [0.15028533 0.89444684 0.3007521  0.27286447 0.00647975]
 [0.59801345 0.79435088 0.59862107 0.61498669 0.87010577]
 [0.72948669 0.76516178 0.98117598 0.52135838 0.53482608]
 [0.08298453 0.01905823 0.26417891 0.77072226 0.96964001]
 [0.3147297  0.49847345 0.2032428  0.68422641 0.8370478 ]
 [0.77362072 0.33219103 0.13979055 0.18148193 0.77215136]
 [0.12510639 0.81139091 0.69946877 0.69293721 0.659838  ]
 [0.93853729 0.84554181 0.28678798 0.72945576 0.40825844]
 [0.70259877 0.2926497  0.70089211 0.09127827 0.36000804]
 [0.08585043 0.48548256 0.24627121 0.67633576 0.82430543]
 [0.17854142 0.0199978  0.73323042 0.6815786  0.79907516]
 [0.21139877 0.982588   0.45313877 0.64182862 0.33975144]]
```

In [110]:

```
w.shape
```

Out[110]:

```
(20, 5)
```

In [111]:

```
w[0,0]
```

Out[111]:

```
0.8519854927300882
```

Initializing distance with 0

In [112]:

```
closest_node = np.zeros(total_number,dtype='int')
closest_node
```

Out[112]:

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Adjusting weight in Training

In [113]:

```
d_list = list()
learning_rate = 0.5
distance = 0
neighbor_int = 3
neighbor_float = 3

while neighbor_float >= 0.0000001 :

    # calculating distance from input node to every output nodes
    for i in range(n):
        for j in range(n):
            for k in range(m):
                distance = distance + ((df.iloc[i,k]-w[j,k]) ** 2)
            d_list.append((j,distance))
            distance = 0

        # sorting the distances in ascending order
        for ii in range(len(d_list)):
            for jj in range(ii+1,len(d_list)):
                if d_list[jj][1] < d_list[ii][1]:
                    temp = d_list[ii]
                    d_list[ii] = d_list[jj]
                    d_list[jj] = temp

        # saving the closest node
        closest_node[i] = d_list[0][0]

        # updating weights of closest node and it's neighbor nodes
        for ii in range(neighbor_int+1):
            node = d_list[ii][0]
            for k in range(m):
                w[node][k] = w[node][k] + learning_rate * (df.iloc[i,k] - w[node][k])

        d_list = list()

    neighbor_float = neighbor_float - learning_rate * neighbor_float
    neighbor_int = int(np.ceil(neighbor_float))
```

In [114]:

```
closest_node[:no_of_train_data]
```

Out[114]:

```
array([ 7, 13,  8, 11,  4,  4,  5, 18,  1, 19,  3,  9, 15, 10,  2, 14, 12,
        6, 12, 17])
```

```
       0, 12, 17])
```

In [115]:

```
print('Training Input No.','\t','   Output node')
for i in range(no_of_train_data):
    print('\t',i,'\t\t\t',closest_node[i])
```

```
Training Input No.      Output node
  0      7
  1      13
  2      8
  3      11
  4      4
  5      4
  6      5
  7      18
  8      1
  9      19
  10      3
  11      9
  12      15
  13      10
  14      2
  15      14
  16      12
  17      6
  18      12
  19      17
```

Clustering testing nodes

In [116]:

```
# calculating distance from input node to every output nodes
for i in range(no_of_train_data,total_number):
    for j in range(n):
        for k in range(m):
            distance = distance + ((df.iloc[i,k]-w[j,k]) ** 2)
        d_list.append((j,distance))
        distance = 0

    # sorting the distances in ascending order
    for ii in range(len(d_list)):
        for jj in range(ii+1,len(d_list)):
            if d_list[jj][1] < d_list[ii][1]:
                temp = d_list[ii]
                d_list[ii] = d_list[jj]
                d_list[jj] = temp

    # saving the closest node
    closest_node[i] = d_list[0][0]
    d_list = list()
```

In [117]:

```
closest_node[no_of_train_data:]
```

Out[117]:

```
array([12,  4,  0, 18,  1, 19,  8,  3, 15, 10,  2, 14])
```

In [118]:

```
print('Testing Input No.','\t','   Output node')
for i in range(no_of_train_data,total_number):
    print('\t',i,'\t\t\t',closest_node[i])
```

```
Testing Input No.      Output node
  20      12
  21      4
```

```
   22      0
   23      18
   24      1
   25      19
   26      8
   27      3
   28      15
   29      10
   30      2
   31      14
```

In [119]:

```
print('All Input Node','\t','   Output node')
for i in range(total_number):
    print('\t',i,'\t\t\t',closest_node[i])
```

```
All Input Node       Output node
  0      7
  1      13
  2      8
  3      11
  4      4
  5      4
  6      5
  7      18
  8      1
  9      19
  10      3
  11      9
  12      15
  13      10
  14      2
  15      14
  16      12
  17      6
  18      12
  19      17
  20      12
  21      4
  22      0
  23      18
  24      1
  25      19
  26      8
  27      3
  28      15
  29      10
  30      2
  31      14
```

In [10]:

```
# n = 4
# for i in range(40):
#     n = n - 0.5 * n
#     print(n)
```

Experiment No : 01

Program Title : Implementing K Nearest Neighbor Algorithm

Objective :

1. Take a dataset and divide it into training and testing dataset
2. Implement KNN algorithm on testing dataset
3. Calculate accuracy
4. Do the same process for 60-40 , 70-30 and 80-20 training testing dataset and compare the accuracy

Methodology :

1. Select the value of k by taking the square root of total data points.
2. Take an unknown data point or testing data point as input.
3. Calculate Euclidean distance from testing data points to all training points
4. Sort the distance in ascending order
5. Pick the first 'K' points from the sorted list
6. Calculate the number of classes
7. The class with the highest number of occurrence will be the nearest neighbor or the class for that unknown or testing data points

Implementation in Code :

The implementation of KNN algorithm on a particular dataset is given next page.

Performance Analysis :

Training Percentage	Testing Percentage	Accuracy Percentage
60	40	99.0833
70	30	99.3333
80	20	99.3333

Conclusion and Observation :

1. **Instance-based learning:** KNN is an instance based learning , meaning that it doesn't exactly build a model for training dataset , instead it uses training dataset to for test dataset.
2. **Choice of K:** The accuracy mostly depends on the value of k. So it's wiser to check the accuracy for different values of k
3. **Curse of dimensionality :** The more the number of dimensions increases , the time taken for knn will also be higher.
4. **Feature Scaling :** For distance based algorithm like KNN , it's necessary to keep all the data in same scaling. If data are in different scaling , then it will result in low accuracy.

Experiment No : 02

Program Title : Implementing Single Layer Perceptron

Objective :

1. Create a dataset of n bit where values will be 0 and 1 only.
2. Output of First half row will be 0 and the rest half will be 1.
3. Divide it into train and test dataset.
4. Apply Single Layer Perceptron algorithm on test dataset
5. Calculate accuracy
6. Do the same process for 60-40 , 70-30 and 80-20 training testing dataset and compare the accuracy

Methodology :

For Training Phase :

1. Generate 'n' number of random weights where 'n' is the number of feature in dataset
2. Get a fixed value for threshold value and learning rate
3. Divide the dataset into train and test part.
4. Take the first row in training data as input and multiply all input data with corresponding weights and take the summation of it
5. If summation >= Threshold value , then the predicted output will be 1
6. If summation < Threshold value , then the predicted output will be 0.
7. If predicted output for that input and actual output for that input are same , then no weight updation is required and go to 10 no step.
8. If predicted output and actual output for that input are not same , then update the weight using the formula :

W_New = W_Old + learning_rate * (Actual Output – Predicted Output) * input

9. Break the loop and go back to 4 no step with the updated weights
10. Go to next input row.
11. Repeat the process from 4 to 10 until all the training input gets 100% classified with a certain set of weights.

For Testing Phase :

1. Take the first row in testing data as input and multiply all input data with corresponding weights and take the summation of it
2. If summation >= Threshold value , then the predicted output will be 1
3. If summation < Threshold value , then the predicted output will be 0.
4. Take the first row in training data as input and multiply all input data with corresponding weights and take the summation of it
5. Calculate the accuracy after checking how many of them are classified correctly.

Implementation in Code :

The implementation of Single Layer Perceptron algorithm on a particular dataset is given next page.

Performance Analysis :

Training Percentage	Testing Percentage	Accuracy Percentage
60	40	92
70	30	100
80	20	100

Conclusion and Observation :

1. The more the testing percentage increases , the higher the accuracy becomes.
2. If learning rate is higher then , it will reach to local minima quickly but when it's small , it reaches to local minima slowly.
3. This algorithm is time consuming for large dataset , since it is learning step by step.

Implementation in Code :

This implementation of Single layer perceptron has been done for groupwise learning.

Performance Analysis :

Training Percentage	Testing Percentage	Accuracy Percentage
60	40	92.1568
70	30	94.7882
80	20	96.0880

Conclusion and Observation :

1. The more the testing percentage increases , the higher the accuracy becomes.
2. If learning rate is higher then , it will reach to local minima quickly but when it's small , it reaches to local minima slowly.
3. This algorithm is time consuming for large dataset , since it is learning groupwise

Experiment No : 03

Program Title : Implementing Backpropagation Algorithm

Objective :

1. Create a dataset of n bit where values will be 0 and 1 only.
2. Output of First half row will be 0 and the rest half will be 1.
3. Divide it into train and test dataset.
4. Apply backpropagation algorithm on test dataset
5. Calculate accuracy
6. Do the same process for 60-40 , 70-30 and 80-20 training testing dataset and compare the accuracy

Methodology :

For Training Phase :

1. Take ‘n’ as no of nodes in input layer which is equal to no of feature. Here for the convenience of code , no of hides layers is also considered to be ‘n’. The number of output layer is m = 1 , since it’s a binary classification.
2. Initialize Wij , bj , Wjk and bk with random values.
   Here ,
   Wij = n X n order of weight matrix between input and hidden layer
   bj = Bias to ‘n’ no nodes of hidden layer
   Wjk = (n X m = n X 1) order of weight matrix between hidden and output layer
   bk = Bias to m = 1 ‘no’ of nodes of output layer.
3. Initialize Oi[n] , netj[n] , activj[n], Oj[n] and netk[m],activk[m], Ok[m] with 0.
4. Take first input and start Forward Propagation
5. If error <= 0.01 , then go to 8.

6. If error > 0.01 , then backpropagation starts.
7. Break the loop and go back to 4 no step with the updated weights
8. Go to next input row.
9. Repeat the process from 4 to 8 until all the training input gets 100% classified with a certain set of weights.

For Testing Phase :

1. Take first input and start Forward Propagation
2. If error <= 0.01 , then right = right + 1
3. If error > 0.01 , then wrong = wrong + 1.
4. Calculate the accuracy after checking how many of them are classified correctly.

Implementation in code :

The implementation of Multi layer perceptron or backpropagation has been done in the next page.

Performance Analysis :

Training Percentage	Testing Percentage	Accuracy Percentage
60	40	0.0
70	30	100.0
80	20	100.0

Conclusion and Observation :

In conclusion, backpropagation is a powerful and widely used algorithm for training neural networks. It addresses the challenge of optimizing the network's weights to minimize the error between predicted and target outputs. Backpropagation, combined with gradient descent optimization, enables deep neural networks to learn complex patterns and make accurate predictions in various domains. Here are observations :

1. The more the testing percentage increases , the higher the accuracy becomes.
2. If learning rate is higher then , it will reach to local minima quickly but when it’s small , it reaches to local minima slowly.
3. This algorithm is time consuming for large dataset.

Experiment No : 04

Program Title : Implementing kohonen neural network

Objective :

1. Create a dataset of n bit where values will be 0 and 1 only.
2. Divide it into train and test dataset.
3. Apply kohonen neural network on test dataset
4. Determine which test data clusters with which output node

Methodology :

1. Take 2 layer , one of which is output layer where the number of nodes will be as same as the number of training data and other one is input layer where number of node is as same as number of feature.
2. Check if neighbor != 0 , If true then proceed and if not , go to 10.
3. Take first input and calculate distance from that input to every output node
4. Sort the distance in ascending order
5. Select the output node with minimum distance from that input
6. Update the node with minimum distance and it's neighbors
7. Continue 3 to 6 until the all training input are used
8. Update no of neighbor using this formula :

   Neighbor_new = Neighbor_old – learning_rate * Neighbor_old
9. Go to 2 again
10. End

Implemention in Code : The implementation of Kohonen neural network is at next page

Performance Analysis :

For Training Data :

Training Input No.	Output node
0	7
1	13
2	8
3	11
4	4
5	4
6	5
7	18
8	1
9	19
10	3
11	9
12	15
13	10
14	2
15	14
16	12
17	6
18	12
19	17

For Testing Data :

Testing Input No.	Output node
20	12
21	4
22	0
23	18
24	1
25	19
26	8
27	3
28	15
29	10
30	2
31	14

Conclusion and Observations : In conclusion, the Kohonen neural network, also known as the Self-Organizing Map (SOM), is a powerful unsupervised learning algorithm that can discover and represent the underlying structure of complex data. It offers a unique approach to clustering and visualizing high-dimensional data in a lower-dimensional space.

Experiment No : 05

Program Title : Implementing hopfield network algorithm

Objective :

1. Create a dataset of n bit where values will be 1 and -1 only.
2. Divide it into train and test dataset.
3. Apply hopfield network algorithm on test dataset
4. Determine which test data clusters with which training input.

Methodology :

1. Divide the dataset into training and testing part.
2. Take m = number of features
3. Number of neurons will also be m = number of features
4. Order of weight matrix will be (m X m)
5. If i == j , Wij = 0
6. else , Wij = Summation(feature_i * feature_j)
7. Take the first test input as new pattern
8. Take this new pattern at the first neuron
9. Summation = new pattern * W[neuron][:]
10. If summation > 0 , then new_pattern[neuron] = +1
11. elif summation < 0 , then new_pattern[neuron] = -1
12. else , no change
13. Take the updated_pattern to second neuron and then next neuron and keep updating until convergence happens

Implementation in Code : The implementation of hopfield network algorithm is at next page

Performance Analysis :

New Pattern : [1, -1, 1, -1]

At Neuron 0 : [1, -1, 1, -1]

At Neuron 1 : [1, -1, 1, -1]

At Neuron 2 : [1, -1, -1, -1]

At Neuron 3 : [1, -1, -1, -1]

At Neuron 0 : [1, -1, -1, -1]

At Neuron 1 : [1, -1, -1, -1]

At Neuron 2 : [1, -1, -1, -1]

Converged pattern of the test pattern : [1, -1, -1, -1]

Cluster with 8

New Pattern : [1, -1, 1, 1]

At Neuron 0 : [1, -1, 1, 1]

At Neuron 1 : [1, -1, 1, 1]

At Neuron 2 : [1, -1, -1, 1]

At Neuron 3 : [1, -1, -1, 1]

At Neuron 0 : [1, -1, -1, 1]

At Neuron 1 : [1, -1, -1, 1]

At Neuron 2 : [1, -1, -1, 1]

Converged pattern of the test pattern : [1, -1, -1, 1]

Cluster with 9

New Pattern : [1, 1, -1, -1]

At Neuron 0 : [1, 1, -1, -1]

At Neuron 1 : [1, -1, -1, -1]

At Neuron 2 : [1, -1, -1, -1]

At Neuron 3 : [1, -1, -1, -1]

At Neuron 0 : [1, -1, -1, -1]

At Neuron 1 : [1, -1, -1, -1]

Converged pattern of the test pattern : [1, -1, -1, -1]

Cluster with 8

New Pattern : [1, 1, -1, 1]

At Neuron 0 : [1, 1, -1, 1]

At Neuron 1 : [1, -1, -1, 1]

At Neuron 2 : [1, -1, -1, 1]

At Neuron 3 : [1, -1, -1, 1]

At Neuron 0 : [1, -1, -1, 1]

At Neuron 1 : [1, -1, -1, 1]

Converged pattern of the test pattern : [1, -1, -1, 1]

Cluster with 9

New Pattern : [1, 1, 1, -1]

At Neuron 0 : [-1, 1, 1, -1]

At Neuron 1 : [-1, 1, 1, -1]

At Neuron 2 : [-1, 1, 1, -1]

At Neuron 3 : [-1, 1, 1, -1]

At Neuron 0 : [-1, 1, 1, -1]

Converged pattern of the test pattern : [-1, 1, 1, -1]

Cluster with 6

New Pattern : [1, 1, 1, 1]

At Neuron 0 : [-1, 1, 1, 1]

At Neuron 1 : [-1, 1, 1, 1]

At Neuron 2 : [-1, 1, 1, 1]

At Neuron 3 : [-1, 1, 1, 1]

At Neuron 0 : [-1, 1, 1, 1]

Converged pattern of the test pattern : [-1, 1, 1, 1]

Cluster with 7

Conclusion and Observation :

he Hopfield neural network is a type of recurrent neural network (RNN) that is capable of storing and retrieving patterns or memories. It provides a unique approach to associative memory and pattern recognition tasks. The Hopfield network is designed to converge to stable states or attractors. Each stored pattern in the network corresponds to an attractor state, and the network dynamics converge to one of these states during recall. This convergence property allows the network to recognize and complete partial or noisy input patterns.

Single Perceptron for n bit data where 60% is train data and 40% is test data

Generating n bit of data

In [1]:

```
n = int(input('Enter Number of bits : '))
count = 0
i = n
string = 'bit_'
total_number = 2 ** n
```

In [2]:

```
value = list()
dictionary = dict()

while i >= 1 :
    key = string + str(i)
    d = 2 ** count

    while len(value) != total_number:
        for j in range(d):
            value.append(0)
        for j in range(d):
            value.append(1)

    dictionary[key] = value
    value = list()
    count = count + 1
    i = i - 1
```

In [3]:

```
# dictionary
```

In [4]:

```
#list(dictionary.items())
```

In [5]:

```
l = list(dictionary.items())
# l
```

In [6]:

```
reversed_dictionary = dict()
i = n-1
while i >= 0:
    reversed_dictionary[l[i][0]] = l[i][1]
    i = i - 1
# reversed_dictionary
```

In [7]:

```
dictionary = reversed_dictionary
# dictionary
```

In [8]:

```
output = dictionary['bit_1']
```

```
# output
```

In [9]:

```
import pandas as pd

df = pd.DataFrame(data=dictionary)
df
```

Out[9]:

	bit_1	bit_2	bit_3	bit_4	bit_5	bit_6
0	0	0	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	0	0	1	0	0
...	...	...	...	...	...	...
59	1	1	1	0	1	1
60	1	1	1	1	0	0
61	1	1	1	1	0	1
62	1	1	1	1	1	0
63	1	1	1	1	1	1

64 rows × 6 columns

In [81]:

```
df['Output'] = output
df
```

Out[81]:

	bit_1	bit_2	bit_3	bit_4	bit_5	bit_6	Output
0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	0
2	0	0	0	0	1	0	0
3	0	0	0	0	1	1	0
4	0	0	0	1	0	0	0
...	...	...	...	...	...	...	...
59	1	1	1	0	1	1	1
60	1	1	1	1	0	0	1
61	1	1	1	1	0	1	1
62	1	1	1	1	1	0	1
63	1	1	1	1	1	1	1

64 rows × 7 columns

In [82]:

```
df = df.drop('Output',axis=1)
df
```

Out[82]:

	bit_1	bit_2	bit_3	bit_4	bit_5	bit_6

	bit_1	bit_2	bit_3	bit_4	bit_5	bit_6
0	0	0	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	0	0	1	0	0
...	...	...	...	...	...	...
59	1	1	1	0	1	1
60	1	1	1	1	0	0
61	1	1	1	1	0	1
62	1	1	1	1	1	0
63	1	1	1	1	1	1

64 rows × 6 columns

Generating n number of random weights , a fixed threshold value and a fixed learning rate

In [83]:

```
n
```

Out[83]:

```
6
```

In [84]:

```
import numpy as np
```

In [85]:

```
np.random.seed(42)
weights = np.random.rand(n)
weights
```

Out[85]:

```
array([0.37454012, 0.95071431, 0.73199394, 0.59865848, 0.15601864,
       0.15599452])
```

In [86]:

```
threshold = 0.5
threshold
```

Out[86]:

```
0.5
```

In [87]:

```
learning_rate = 0.4
learning_rate
```

Out[87]:

```
0.4
```

Train Test

In [88]:

```
train_percentage = 60
```

```
test_percentage = 100 - train_percentage

print('Train Percentage :',train_percentage)
print('Test Percentage :',test_percentage)
```

```
Train Percentage : 60
Test Percentage : 40
```

In [89]:

```
import math

no_of_train_data = math.ceil(( total_number * train_percentage ) / 100)
no_of_test_data = total_number - no_of_train_data

print('No of Train Data :',no_of_train_data)
print('No of Test Data :',no_of_test_data)
```

```
No of Train Data : 39
No of Test Data : 25
```

Adjusting Weights

In [90]:

```
total_number
```

Out[90]:

```
64
```

In [91]:

```
# df.columns
# df.columns[0]
# df[df.columns[0]]
df[df.columns[0]]
```

Out[91]:

```
0      0
1      0
2      0
3      0
4      0
      ..
59     1
60     1
61     1
62     1
63     1
Name: bit_1, Length: 64, dtype: int64
```

In [92]:

```
counter = 0

while counter!=no_of_train_data :

    for i in range(no_of_train_data):
        summation = 0

        # Take the first row in training data as input and
        # multiply all input data with corresponding weights and take the summation of it
        for j in range(n):
            summation = summation + df.iloc[i,j] * weights[j]

        if summation >= threshold :
            predicted_output = 1

        else:
```

```
            predicted_output = 0

        if predicted_output == output[i]:
            counter = counter + 1

        # weight updation happens if doesn't match
        else:
            difference = output[i] - predicted_output
            for j in range(n):
                weights[j] = weights[j] + learning_rate * difference * df.iloc[i,j]
            counter = 0
            break
```

In [93]:

```
weights
```

Out[93]:

```
array([ 1.17454012,  0.15071431, -0.06800606, -0.20134152, -0.24398136,
       -0.24400548])
```

In [94]:

```
# proof that these weights are valid for train data

for i in range(no_of_train_data) :
    summation = 0

    for j in range(n):
        summation = summation + df[df.columns[j]][i] * weights[j]

    if summation >= threshold :
        predicted_output = 1
    else:
        predicted_output = 0

    print(predicted_output)
```

```
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
1
```

```
1
1
1
1
1
1
```

In [95]:

```
right = 0
wrong = 0

for i in range(no_of_train_data,total_number) :
    summation = 0

    for j in range(n):
        summation = summation + df[df.columns[j]][i] * weights[j]

    if summation >= threshold :
        predicted_output = 1

    else:
        predicted_output = 0

    if predicted_output == output[i]:
        right = right + 1

    else:
        wrong = wrong + 1

accuracy = ( right * 100 ) / no_of_test_data

print('No of Test Data :',no_of_test_data)
print('Right :',right)
print('Wrong :',wrong)
print('Accuracy :',accuracy)
```

```
No of Test Data : 25
Right : 23
Wrong : 2
Accuracy : 92.0
```

Single Perceptron for n bit data where 80% is train data and 20% is test data and Groupwise learning

Generating n bit of data

In [2]:

```
n = int(input('Enter Number of bits : '))
count = 0
i = n
string = 'bit_'
total_number = 2 ** n
```

In [3]:

```
value = list()
dictionary = dict()

while i >= 1 :
    key = string + str(i)
    d = 2 ** count

    while len(value) != total_number:
        for j in range(d):
            value.append(0)
        for j in range(d):
            value.append(1)

    dictionary[key] = value
    value = list()
    count = count + 1
    i = i - 1
```

In [4]:

```
# dictionary
```

In [5]:

```
# list(dictionary.items())
```

In [6]:

```
l = list(dictionary.items())
#l
```

In [7]:

```
reversed_dictionary = dict()
i = n-1
while i >= 0:
    reversed_dictionary[l[i][0]] = l[i][1]
    i = i - 1
#reversed_dictionary
```

In [8]:

```
dictionary = reversed_dictionary
#dictionary
```

In [9]:

```
output = dictionary['bit_1']
```

```
#output
```

In [10]:

```
import pandas as pd

df = pd.DataFrame(data=dictionary)
df
```

Out[10]:

	bit_1	bit_2	bit_3	bit_4	bit_5	bit_6	bit_7	bit_8	bit_9	bit_10
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	0	0	1	0
3	0	0	0	0	0	0	0	0	1	1
4	0	0	0	0	0	0	0	1	0	0
...	...	...	...	...	...	...	...	...	...	...
1019	1	1	1	1	1	1	1	0	1	1
1020	1	1	1	1	1	1	1	1	0	0
1021	1	1	1	1	1	1	1	1	0	1
1022	1	1	1	1	1	1	1	1	1	0
1023	1	1	1	1	1	1	1	1	1	1

1024 rows × 10 columns

In [229]:

```
df['Output'] = output
df
```

Out[229]:

	bit_1	bit_2	bit_3	bit_4	bit_5	bit_6	bit_7	bit_8	bit_9	bit_10	Output
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	1	0
2	0	0	0	0	0	0	0	0	1	0	0
3	0	0	0	0	0	0	0	0	1	1	0
4	0	0	0	0	0	0	0	1	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...
1019	1	1	1	1	1	1	1	0	1	1	1
1020	1	1	1	1	1	1	1	1	0	0	1
1021	1	1	1	1	1	1	1	1	0	1	1
1022	1	1	1	1	1	1	1	1	1	0	1
1023	1	1	1	1	1	1	1	1	1	1	1

1024 rows × 11 columns

In [230]:

```
df = df.drop('Output',axis=1)
df
```

Out[230]:

	bit_1	bit_2	bit_3	bit_4	bit_5	bit_6	bit_7	bit_8	bit_9	bit_10

0	bit_1	bit_2	bit_3	bit_4	bit_5	bit_6	bit_7	bit_8	bit_9	bit_10
1	0	0	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	0	0	1	0
3	0	0	0	0	0	0	0	0	1	1
4	0	0	0	0	0	0	0	1	0	0
...	...	...	...	...	...	...	...	...	...	...
1019	1	1	1	1	1	1	1	0	1	1
1020	1	1	1	1	1	1	1	1	0	0
1021	1	1	1	1	1	1	1	1	0	1
1022	1	1	1	1	1	1	1	1	1	0
1023	1	1	1	1	1	1	1	1	1	1

1024 rows × 10 columns

Generating n number of random weights , a fixed threshold value and a fixed learning rate

In [231]:

```
n
```

Out[231]:

```
10
```

In [232]:

```
import numpy as np
```

In [233]:

```
np.random.seed(42)
weights = np.random.rand(n)
weights
```

Out[233]:

```
array([0.37454012, 0.95071431, 0.73199394, 0.59865848, 0.15601864,
       0.15599452, 0.05808361, 0.86617615, 0.60111501, 0.70807258])
```

In [234]:

```
threshold = 0.5
threshold
```

Out[234]:

```
0.5
```

In [235]:

```
learning_rate = 0.4
learning_rate
```

Out[235]:

```
0.4
```

Train Test

In [236]:

```
train_percentage = 80
```

```
test_percentage = 100 - train_percentage

print('Train Percentage :',train_percentage)
print('Test Percentage :',test_percentage)
```

```
Train Percentage : 80
Test Percentage : 20
```

In [237]:

```
import math

no_of_train_data = math.ceil(( total_number * train_percentage ) / 100)
no_of_test_data = total_number - no_of_train_data

print('No of Train Data :',no_of_train_data)
print('No of Test Data :',no_of_test_data)
```

```
No of Train Data : 820
No of Test Data : 204
```

Adjusting Weights

In [238]:

```
total_number
```

Out[238]:

```
1024
```

In [11]:

```
# df.columns
# df.columns[0]
# df[df.columns[0]]
#df[df.columns[0]]
```

In [240]:

```
# training 0 class and getting weights

counter = 0

while counter!=total_number/2 :

    for i in range(int(total_number/2)):
        summation = 0

        for j in range(n):
            summation = summation + df[df.columns[j]][i] * weights[j]

        if summation >= threshold :
            predicted_output = 1

        else:
            predicted_output = 0

        if predicted_output == output[i]:
            counter = counter + 1

        else:
            difference = output[i] - predicted_output
            for j in range(n):
                weights[j] = weights[j] + learning_rate * difference * df[df.columns[j]]
[i]
            counter = 0
            break
```

In [241]:

```
weights
```

Out[241]:

```
array([ 0.37454012, -0.24928569, -0.06800606, -0.20134152, -0.24398136,
       -0.24400548, -0.34191639, -0.33382385, -0.19888499, -0.09192742])
```

In [242]:

```
while True:
    # training 1 class and getting weights
    counter = 0

    while counter!=no_of_train_data - total_number/2 :

        for i in range(int(total_number/2),no_of_train_data):
            summation = 0

            for j in range(n):
                summation = summation + df[df.columns[j]][i] * weights[j]

            if summation >= threshold :
                predicted_output = 1

            else:
                predicted_output = 0

            if predicted_output == output[i]:
                counter = counter + 1

            else:
                difference = output[i] - predicted_output
                for j in range(n):
                    weights[j] = weights[j] + learning_rate * difference * df[df.columns
[j]][i]
                counter = 0
                break


    # weights gained after training class 1 are applied on the 0 class again
    right = 0
    wrong = 0

    for i in range(int(total_number/2)) :
        summation = 0

        for j in range(n):
            summation = summation + df[df.columns[j]][i] * weights[j]

        if summation >= threshold :
            predicted_output = 1
        else:
            predicted_output = 0

        if predicted_output == output[i]:
            right = right + 1
        else:
            wrong = wrong + 1

    # if weights gained after training class 1 works on all 0 class then break the loop
    if wrong == 0:
        break
    # otherwise train the 0 class with new weights
    else:
        counter = 0

        while counter!=total_number/2 :

            for i in range(int(total_number/2)):
                summation = 0
```

```
            for j in range(n):
                summation = summation + df[df.columns[j]][i] * weights[j]

            if summation >= threshold :
                predicted_output = 1

            else:
                predicted_output = 0

            if predicted_output == output[i]:
                counter = counter + 1

            else:
                difference = output[i] - predicted_output
                for j in range(n):
                    weights[j] = weights[j] + learning_rate * difference * df[df.col
umns[j]][i]
                counter = 0
                break
```

In [243]:

```
weights
```

Out[243]:

```
array([ 1.57454012, -0.24928569, -0.06800606, -0.20134152, -0.24398136,
       -0.24400548,  0.05808361,  0.06617615, -0.19888499, -0.09192742])
```

In [1]:

```
# # proof that these weights are valid for train data

# for i in range(no_of_train_data) :
#     summation = 0

#     for j in range(n):
#         summation = summation + df[df.columns[j]][i] * weights[j]

#     if summation >= threshold :
#         predicted_output = 1
#     else:
#         predicted_output = 0

#     print(predicted_output)
```

In [245]:

```
right = 0
wrong = 0

for i in range(no_of_train_data,total_number) :
    summation = 0

    for j in range(n):
        summation = summation + df[df.columns[j]][i] * weights[j]

    if summation >= threshold :
        predicted_output = 1

    else:
        predicted_output = 0

    if predicted_output == output[i]:
        right = right + 1

    else:
        wrong = wrong + 1

accuracy = ( right * 100 ) / no_of_test_data
```

```python
print('No of Test Data :',no_of_test_data)
print('Right :',right)
print('Wrong :',wrong)
print('Accuracy :',accuracy)
```

```
No of Test Data : 204
Right : 188
Wrong : 16
Accuracy : 92.15686274509804
```