CHAPTER 1

Title: Implementation and Performance Analysis of K-Nearest Neighbour Algorithm.

1.1 Objectives

KNN is a non-parametric algorithm used for both regression and classification.

- KNN can be used for both regression and classification.
- Output of KNN can easily be interpreted.
- Calculation time is relatively lower.
- But its predictive power is very high.
- It can be useful in the case of high dimensionality data.

1.2 Methodology

- Load the data.
- Initialize the value of k.
- For getting the predicted class, iterate from 1 to total number of training data points
- Calculate the distance between test data and each row of training data. Here we will use Euclidean distance as our distance metric since it's the most popular method
    - Sort the calculated distances in ascending order based on distance values
    - Get top k rows from the sorted array
    - Get the most frequent class of these rows
    - Return the predicted class

1.3 Implementation

I have implemented KKN algorithm according to the above Pseudocode. I used here Euclidian distance as a distance metric. The tools I used here are:

- Python 3.0
- MatPlotlib
- Numpy
- Editor: Sublime Text 3.0

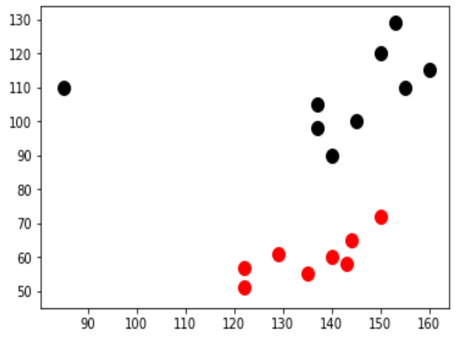
1.3.1 Source Code:

```python
#Source code for k-nearest neighbor algorithm
import numpy as np
from math import sqrt
import matplotlib.pyplot as plt
from matplotlib import style

#Initialize the Dtaset
dataset = {'r':[(135,55),(122,51),(140,60),(143,58),(144,65),(150,72),(129,61),(122,57)],

           'k': [(140,90),(145,100),(155,110),(137,105),(150,120),(153,129),(137,98),(160,
115)]}

#Plotting function
def plot_scat(new_fet):
    [[plt.scatter(ii[0],ii[1] ,s=100, color=i) for ii in dataset[i]] for i in dataset]
    plt.scatter(new_fet[0], new_fet[1], color='b')  #Scatter plot of new feature
    plt.show()

#Adding new members and plot it
def plot_updated_scat(new_fet,result):
    [[plt.scatter(ii[0],ii[1] , s=100, color=i) for ii in dataset[i]] for i in dataset]
    plt.scatter(new_fet[0], new_fet[1], color=result)
    plt.show()

#Distance metric measuring[Euclidian]
def k_nearest_neighbor(data, predict, k=3):  #Define the number of neighbor(k=3)
    distances = []
    for group in data:
        for features in data[group]:
            eucledian_distances = np.linalg.norm( np.array(features) -
            np.array(predict))  #Perform square root operation by Numpy Linear-algebra
            distances.append([eucledian_distances, group])
    votes = [i[1] for i in sorted(distances)[:k]] #Sort the votes count
    print(Counter(votes).most_common(1))

    #To find the most votes to define winner
    vote_result = Counter(votes).most_common(1)[0][0]
    return vote_result

#Decide footballer/wrestler class and append
def dic_add(result, new_features):
    if result == 'r':
        print('He is a footballer..')
        dataset['r'].append(new_features)
    else:
        dataset['k'].append(new_features)
        print('He is a wrestler..')

#Getting user input as 2-D feature vector[Height,weight]
while True:
    print('...Input Test Data...[height,weight]..')
    new_feat = [int(x) for x in input().split()]
    if new_feat==[0,0]:
        break
    else:
        plot_scat(new_feat)  #Drawing a scatter plot of two different color
        result = k_nearest_neighbor(dataset, new_feat, k=3)
        dic_add(result, new_feat)
        print(dataset)
        plot_updated_scat(new_feat, result)

```

1.4 Result

I have used a predefined dataset named [height,weight] for some footballer and wrestler.The dataset are the input as numpy array in the program.

```
[('k', 3)]
He is a wrestler..
{'k': [(140, 90), (145, 100), (155, 110), (137, 105), (150, 120), (153, 129), (137, 98), (160, 115), [85, 110]], 'r': [(135, 55), (122, 51), (140, 60),
(143, 58), (144, 65), (150, 72), (129, 61), (122, 57)]}
```



```
...Input Test Data...[height,weight]..

85 110
```

1.5 Performance Analysis

Table 1 : Measuring Accuracy for the defined dataset.

Number of Train Data	Number of Test Data	Value of K	Correctly Classified	Accuracy (%)	#Rogue Samples Found
17	10	3	8	80%	2
20	10	4	7	70%	2
40	25	3	19	76%	3
40	25	4	17	68%	6

- It produces errors in the classification of a rogue samples.
- Performance highly depends on the optimal selection of k.

- Large value of k reduces noise on classification but may degrade accuracy
- Its main advantage is that require no assumption about data.
- High accuracy.
- But it is computationally slow.
- High memory requirement.

1.6 Conclusion and Observation

The performance of KNN algorithm mostly depends on the distance metric used and the optimal selection of the value of ‘k’. Here I randomly choose ‘k’ as 3 and 4 .And as a distance metric I have selected ‘Euclidian’ distance. And for a defined dataset I have used numpy array which is the best choice for dynamic memory allocation.

As we know that KNN is a lazy learning algorithm,It has high memory requirement as it stores all of the training data. And it is computationally slow.KNN is very sensitive to irrelevant features ,in such case it can not handle data properly.

KNN can produces errors in classification if a rogue sample is selected. So in every single training or testing epoch I have checked for the presence of rogue sample. If a rogue sample is detected then immediately remove it from samples.

For example , when I defined k=3 and the number of train data=17,number of test data=10,it could be able to classify 8 out of 10 test pattern and found 2 rogue pattern. In another case number of train_date=40, number of test_data=25,k=3,then it could classify total 19 samples correctly and found 3 rogue patterns. But as I increased k to 4 on the same dataset, then the accuracy decreased to 68% where previous was 76% and now it found total 6 rogue patterns.

CHAPTER 2

Title: Implementation and Performance Analysis of Single Layer Perceptron Learning Algorithm.

2.1 Objectives

In machine learning single layer perceptron algorithm are the most general degign of neuron.

- Single layer perceptron is a viewing model of single neuron.
- It is an algorithm of supervised learning for binary classes.
- It is used to classify the linearly separable problems.
- For learning binary classifiers.
- It implements a simple function from multi-dimensional real input to binary output.

2.2 Methodology

Variables and parameters

$x(n)$ = input vector = $[+1, x_1(n), x_2(n), \ldots, x_m(n)]^T$

$w(n)$ = weight vector = $[b(n), w_1(n), w_2(n), \ldots, w_m(n)]^T$

$b(n)$ = bias

$y(n)$ = actual response

$d(n)$ = desired response

η = learning rate parameter

- Initialization: set $\mathbf{w}(0) = 0$
- Activation: activate perceptron by applying input example (vector $\mathbf{x}(n)$ and desired response $d(n)$)
- Compute actual response of perceptron :
  - $y(n) = \mathrm{sgn}[\mathbf{w}^T(n)\mathbf{x}(n)]$
- Adapt weight vector: if d(n) and y(n) are different then
  - $\mathbf{w}(n+1) = \mathbf{w}(n) + \eta[d(n)-y(n)]\mathbf{x}(n)$
- Continuation : increment time step n by 1 and go to Activation step

2.3 Implementation

I have implemented single layer perceptron algorithm according to the above Pseudocode. I followed here 'Widrow-Hoff-Delta' rule . The tools I used here are:

- C++ 11
- Codeblocks 13.12

2.3.1 Source Code

```cpp
#include<bits/stdc++.h>
using namespace std;

int b=10,rem;

//vector<double>output;       //Only contains the [num_out_node] number of output
vector<double>target_output;  //Contains all the output for all input pattern
vector<int>binari;

vector<vector<int> > input((int)pow(2,b)); //Dynamic allocation of memory by vector
vector<vector<int> > output((int)pow(2,b));

//Prepare input and output patterns
vector <int>convert_binary_input_bit(int n,int bb)
{
    binari.clear();
    rem=0;
    while(n!=0)
    {
        rem=n%2;
        binari.push_back(rem);
        n/=2;
    }

    reverse(binari.begin(),binari.end());
    int pos=0;
    pos = binari.size();
    for(int i=pos;i<bb;i++)
    {
        binari.push_back(0);
    }

    rotate(binari.begin(),binari.begin()+pos,binari.end());
    return binari;

}
//generate binary input pattern
void generate_input()
{
    int limit = (int)pow(2,b);

    for(int i=0;i<limit;i++)
    {
        vector<int>tmp = convert_binary_input_bit(i,b);
        input.push_back(tmp);
        tmp.clear();
    }
```

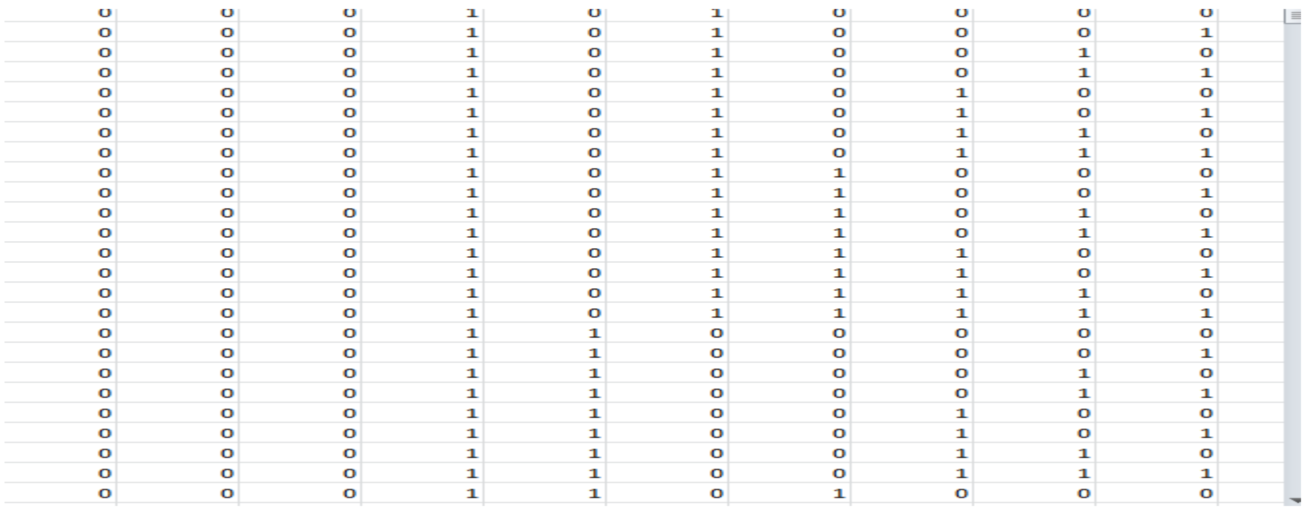
```

}
//Displaying the input pattrn
void display_input_pattern()
{
    for(int i=0;i<input.size();i++)
    {   cout<<"Integer number : "<<i<<endl;;
        for(int j=0;j<input[i].size();j++)
        {
             cout<<input[i][j]<<" ";
        }
       cout<<endl;
    }
}

//generate binary input pattern
void generate_output()
{

   int limit = (int)pow(2,num_out_node);
    for(int i=0;i<limit;i++)
    {
        vector<int>tmp=convert_binary_input_bit(i,num_out_node);
        output.push_back(tmp);
        tmp.clear();
    }

}

void make_output_row_equal_to_input_row()
{
    int j=0;
   for(int i=8;i<(int)pow(2,b);i++)
   {
    if(j>=8)
        j=0;
    output.push_back(output[j]);
    j++;

   }
}
    int output_given[20]={0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1};
    double eta = 0.5,result=0.0;
    double weight[10] = {0.1,-0.5,0.3,0.1,.2,.6,.7,.4,.55,.75};
    int desired,x0,x1,x2,x3,x4,x5,x6,x7,x8,x9,y;
    int xn[10],output,cnt=0;

//Update weight by widrow-hoff delta rule
    void updt_wgt()
    {
        //cout<<"result in update: "<<result<<endl;

               if(result>0.0)
                output=1;
               else
               output=0;

               if(output==desired)
               {
                   cout<<"No Update to weights"<<endl;
               }
```

```

                  else
                  {
                      int diff;
                      double multiply;
                      double tmp_x[10],updated_weight[10];

                      diff=desired-output;
                      cout<<"diff: "<<diff<<endl;
                      multiply = eta*diff;
                      //cout<<"multiply: "<<multiply<<endl;

                      for(int i=0;i<10;i++)
                      {
                          tmp_x[i] = multiply*xn[i];
                      }

                      for(int i=0;i<10;i++)
                      {
                          updated_weight[i] = weight[i]+tmp_x[i];
                      }

                    for(int i=0;i<10;i++)
                    {
                        weight[i]=0;
                        weight[i] = updated_weight[i];
                    }
                  }
              }

            //Calculate the weighted sum
              void oper_sum_w_feature()
              {
                  for(int i=0;i<10;i++)
                  {
                      result = result + xn[i]*weight[i];
                      //cout<<xn[i]<<" "<<weight[i]<<" "<<result<<endl;
                  }
                  cout<<endl<<endl;
              }

              void main_operation()
              {
                  for(int i=0;i<20;i++)
                  {
                       for(int j=0;j<10;j++)
                      {
                         xn[j]=input[i][j];

                       }
                  if(i<10)
                  desired=0;
                  else
                  desired=1;

                  oper_sum_w_feature();
                  //cout<<"result: "<<result<<endl;
                  updt_wgt();

               cout<<"New weights : "<<endl;
               for(int i=0;i<10;i++)
               {
                  cout<<weight[i]<<" ";
               }
               }
              }

          //Main function start here...
          int main()
                  {
                  generate_input();
                  generate_output();
                  make_output_row_equal_to_input_row();
                     main_operation();

                    //Testing Phase Start Here...
                     cout<<"\n\n ....Testing Phase....\n"<<endl;
                     for(int i=0;i<20;i++)         //Get test samples
                      {
                             for(int j=0;j<10;j++)
                           {
                               xn[j]=input[i][j];
                           }
                           cout<<"Input Samples : "<<xn[0]<<" "<<xn[1]<<" "<<xn[2]<<" "<<xn[3
  ]<<" "<<xn[4]<<" "<<xn[5]<<" "<<xn[6]<<" "<<xn[7]<<" "<<xn[8]<<" "<<xn[9]<<endl;

                           result=0;
                           for(int i=0;i<10;i++)
                           result+=xn[i]*weight[i];

                           if(result>0)
                           {
                               cout<<"CLass-B(1)"<<endl;     //Condition to be in Class-B
                               if(i<10)
                               cnt++;
                           }

                           else
                           {
                              cout<<"CLass-A(0)"<<endl;   //Condition to be in Class-A
                              if(i>=10)
                              cnt++;
                           }
                          cout<<endl;
                      }
                      cout<<cnt<<endl;

                      //Accuracy measure as the number of correctly-classification
                      double net_cnt=20.0-cnt;
                      double accuracy = (net_cnt/20.0)*100;
                      cout<<"Accuracy : "<<accuracy<<" %"<<endl;

                  }
```

2.4 Result

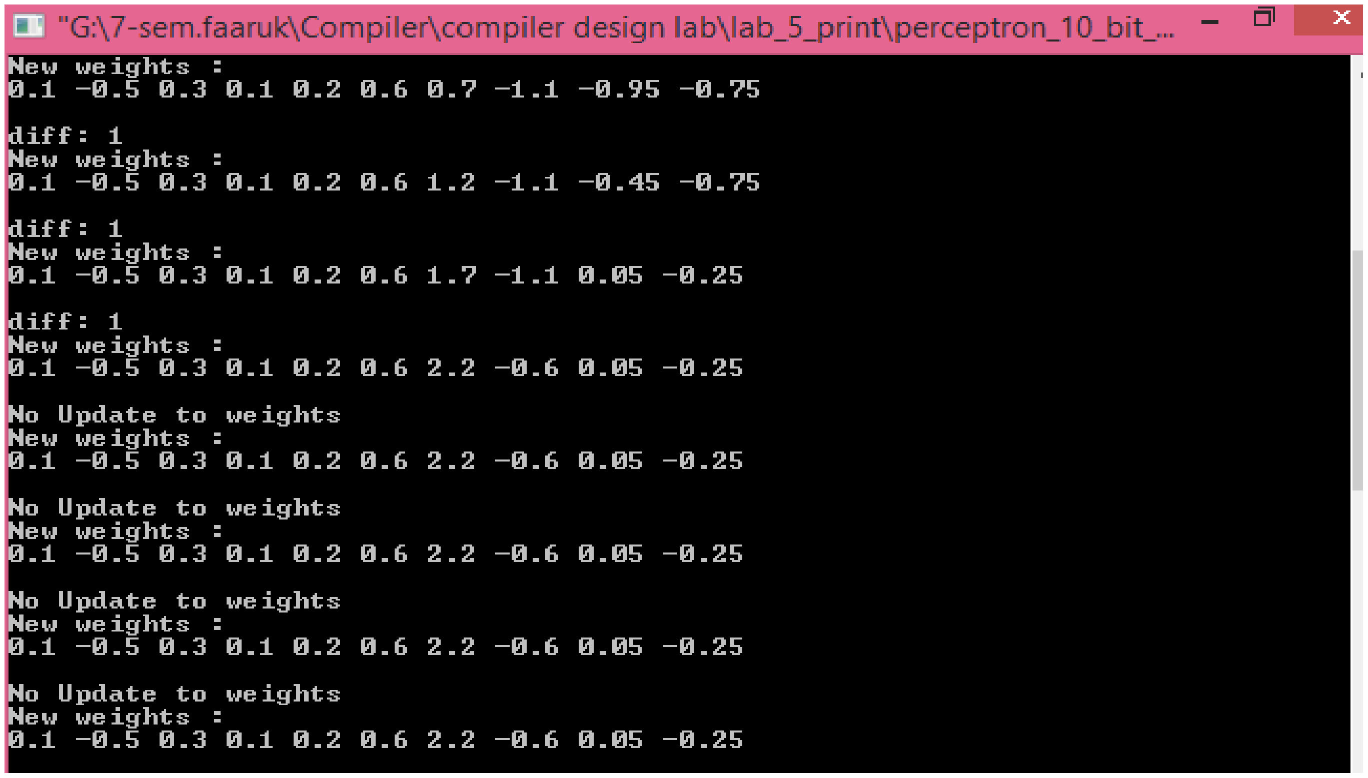
I have used a dataset of 10-bit input pattern .For learning purpose, my program can work with variable number of bit pattern as well as variable number of samples. Here I have used total of 20 samples. The dataset is imported to program as a c++ vector.

2.4.1 Input pattern



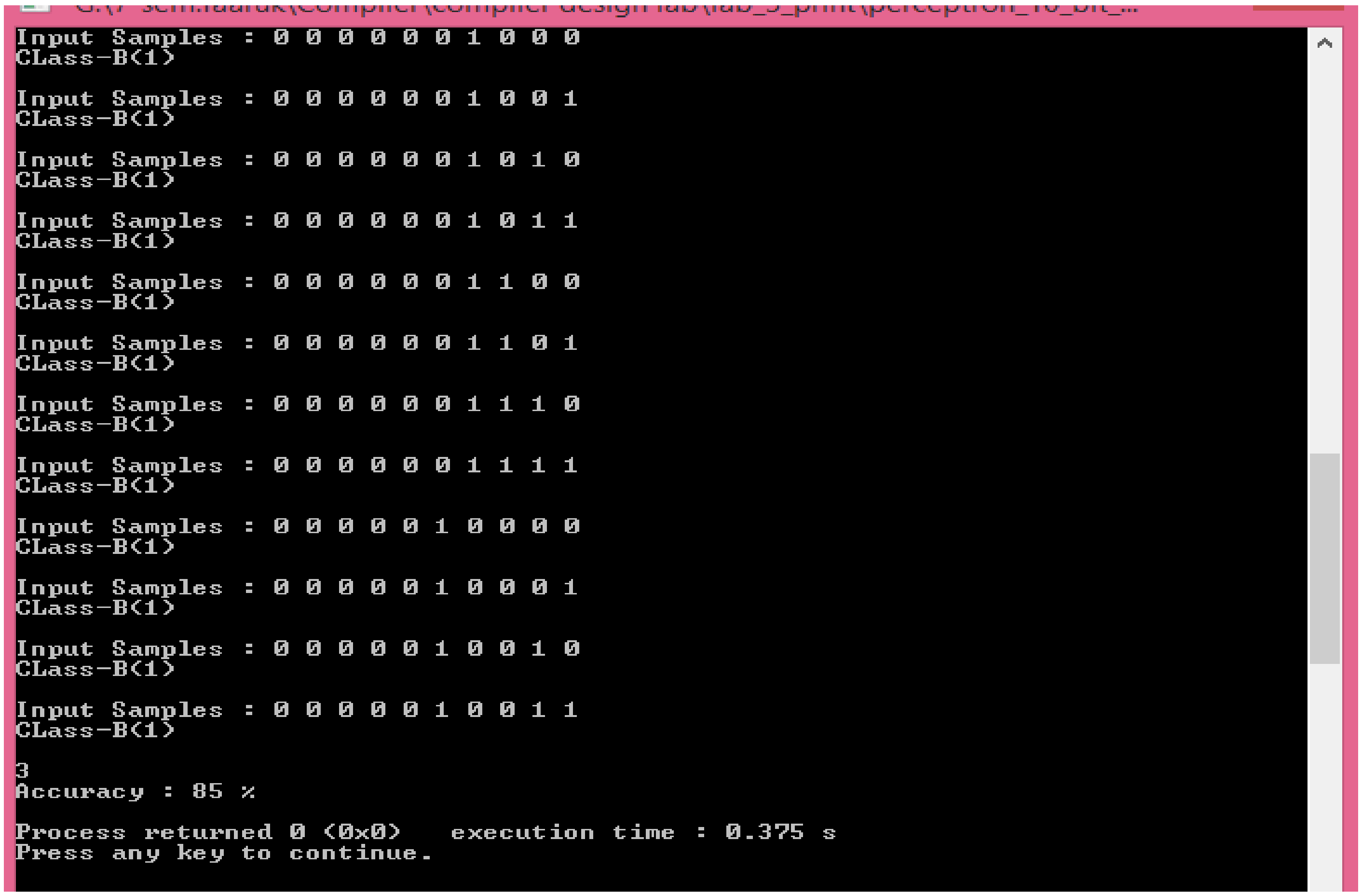
0	0	0	1	0	1	0	0	0	0
0	0	0	1	0	1	0	0	0	1
0	0	0	1	0	1	0	0	1	0
0	0	0	1	0	1	0	0	1	1
0	0	0	1	0	1	0	1	0	0
0	0	0	1	0	1	0	1	0	1
0	0	0	1	0	1	0	1	1	0
0	0	0	1	0	1	0	1	1	1
0	0	0	1	0	1	1	0	0	0
0	0	0	1	0	1	1	0	0	1
0	0	0	1	0	1	1	0	1	0
0	0	0	1	0	1	1	0	1	1
0	0	0	1	0	1	1	1	0	0
0	0	0	1	0	1	1	1	0	1
0	0	0	1	0	1	1	1	1	0
0	0	0	1	0	1	1	1	1	1
0	0	0	1	1	0	0	0	0	0
0	0	0	1	1	0	0	0	0	1
0	0	0	1	1	0	0	0	1	0
0	0	0	1	1	0	0	0	1	1
0	0	0	1	1	0	0	1	0	0
0	0	0	1	1	0	0	1	0	1
0	0	0	1	1	0	0	1	1	0
0	0	0	1	1	0	0	1	1	1
0	0	0	1	1	0	1	0	0	0

2.4.2 Output (Weight Update)



```
"G:\7-sem.faaruk\Compiler\compiler design lab\lab_5_print\perceptron_10_bit_...
New weights :
0.1 -0.5 0.3 0.1 0.2 0.6 0.7 -1.1 -0.95 -0.75

diff: 1
New weights :
0.1 -0.5 0.3 0.1 0.2 0.6 1.2 -1.1 -0.45 -0.75

diff: 1
New weights :
0.1 -0.5 0.3 0.1 0.2 0.6 1.7 -1.1 0.05 -0.25

diff: 1
New weights :
0.1 -0.5 0.3 0.1 0.2 0.6 2.2 -0.6 0.05 -0.25

No Update to weights
New weights :
0.1 -0.5 0.3 0.1 0.2 0.6 2.2 -0.6 0.05 -0.25

No Update to weights
New weights :
0.1 -0.5 0.3 0.1 0.2 0.6 2.2 -0.6 0.05 -0.25

No Update to weights
New weights :
0.1 -0.5 0.3 0.1 0.2 0.6 2.2 -0.6 0.05 -0.25

No Update to weights
New weights :
0.1 -0.5 0.3 0.1 0.2 0.6 2.2 -0.6 0.05 -0.25
```

2.4.2.1 Output (Classification)



```
Input Samples : 0 0 0 0 0 0 1 0 0 0
CLass-B(1)

Input Samples : 0 0 0 0 0 0 1 0 0 1
CLass-B(1)

Input Samples : 0 0 0 0 0 0 1 0 1 0
CLass-B(1)

Input Samples : 0 0 0 0 0 0 1 0 1 1
CLass-B(1)

Input Samples : 0 0 0 0 0 0 1 1 0 0
CLass-B(1)

Input Samples : 0 0 0 0 0 0 1 1 0 1
CLass-B(1)

Input Samples : 0 0 0 0 0 0 1 1 1 0
CLass-B(1)

Input Samples : 0 0 0 0 0 0 1 1 1 1
CLass-B(1)

Input Samples : 0 0 0 0 0 1 0 0 0 0
CLass-B(1)

Input Samples : 0 0 0 0 0 1 0 0 0 1
CLass-B(1)

Input Samples : 0 0 0 0 0 1 0 0 1 0
CLass-B(1)

Input Samples : 0 0 0 0 0 1 0 0 1 1
CLass-B(1)

3
Accuracy : 85 %

Process returned 0 (0x0)   execution time : 0.375 s
Press any key to continue.
```

2.5 Performance Analysis

Table 2 : Measuring Accuracy for the defined dataset.

Number of Train Data	Number of Test Data	eta	Correctly Classified	Accuracy (%)
20	20(same as Train)	0.50	17	85%
20	20(same as train)	0.39	19	95%
20	10(different to train)	0.50	7	35%
20	10(different to train)	0.39	9	45%

- Performance can vary on the selection of gain term(eta).
- Small value of eta slows the convergence but may increase classification accuracy.

- The algorithm is computationally slow.
- Recovery from the damages can be difficult as no backtracking process.
- Performances mostly depend on the behavior of test vs train data.
- It is totally unable to classify the linearly inseparable problems.

2.6 Conclusion and Observation

The performance and training accuracy of single layer perceptron are dependent on the selection of gain term.So an optimal gain term can increase the training accuracy but it increases the computational complexity.We we can introduce a new thresholding function rather than hard-limiting thresholding function ,then the accuracy will dramatically change.The hard limiting thresholding function removes the information that is needed if the network is to successfully learn.For modification we have introduced sigmoid thresholding function to get rid of the 'Credit Assignment' problem.

The positive gain term (0<eta<1) controls the adaptation rate that is multiplied to the error rate to give a correct direction to network. So the random selection of eta may affect the classification result.

For example, if eta = 0.5,number of training and test samples =20 and they are same then it could be able to classify most of the patterns.If eta is decreased to 0.39 then it could classify 19 patterns out of 20 patterns.

The most crucial fact is to train and test with different patterns.If we choose 20 different patterns for train and test and eta as 0.5 then it failed to classify most of the patterns. It could classify only 7 out of 20 patterns.

CHAPTER 3

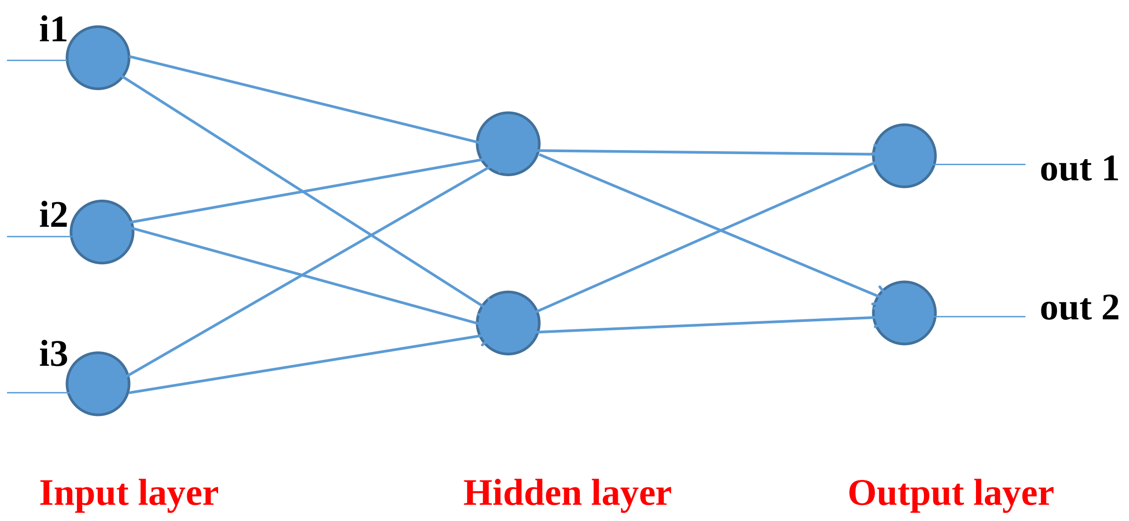
Title: Implementation and Performance Analysis of Multilayer Perceptron –The Back Propagation Algorithm.

3.1 Objectives

The multilayer perceptron learning algorithm is a model of neural network algorithm that is consists of at least three layer, an input layer, a hidden layer and an output layer.

- The multilayer, here we introduced back propagation is able to solve the linearly inseparable problems.
- Here we used sigmoid as a thresholding function that solves the problem occurred by hard-limiting thresholding function.
- Introducing gradient descent algorithm.
- Visualizing the network behavior by energy function.
- Using multilayer perceptron as a classifier.

3.2 Methodology



3.2.1 Pseudocode:

Initialize all weights with small random numbers, typically between [-1 and 1]

```
repeat

   for every pattern in the training set

      Present the pattern to the network

    // Propagated the input forward through the network:
         for each layer in the network
            for every node in the layer
               1. Calculate the weight sum of the inputs to the node
               2. Add the threshold to the sum
               3. Calculate the activation for the node
            end
         end

     // Propagate the errors backward through the network
          for every node in the output layer
            calculate the error signal
         end

         for all hidden layers
            for every node in the layer
               1. Calculate the node's signal error
               2. Update each node's weight in the network
            end
         end

   // Calculate Global Error
         Calculate the Error Function

   end

while ((maximum  number of iterations < than specified) AND
        (Error Function is > than specified))
```

3.3 Implementation

I have implemented the backpropagation algorithm according to the above Pseudocode. I used here sigmoid thresholding function rather than hard-limiting thresholding function.

The tools I used for implementation is :

- Python 3.0
- MatPlotlib
- Numpy
- Pandas data Frame
- Editor: Sublime Text 3.0

3.3.1 Source Code

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

error_threshold = 0.014  #Consider only 1.40% error or less
count=0
#Some global initialization START here
b=10 # Input bit==input node
num_of_hidden_node = 2
number_of_out_node = 3
#b1=.35
#b2=.60
b1 = np.linspace(.3,.9,num_of_hidden_node) #Initialize bias1 vector
b2 = np.linspace(.2,.8,number_of_out_node) #Initialize bias2 vector
k1=.05 #Initialize spread factor1
k2=.057 #Initialize spread factor2
eta1=.478 #Initialize gain term1
eta2=.389 #initialize gain term2
target_output = np.zeros((1024,3))
#Global declaration END here

def convert_DataFrame_To_numpy_Array(df_input,df_output):
    print('\n\nWow Author Faaruk Khan you...from above....')
    #print(df_input)
    input_num_py = df_input.values
    #print('Input numpy is',input_num_py)
    #print('shape of input :',input_num_py.shape)
    output_num_py = df_output.values
    #print('shape of output:',output_num_py.shape)
    #print('Output numpy is:',output_num_py)
    return input_num_py,output_num_py


def read_input_output_file():
    df_input = pd.read_csv('input.csv') #Read input pattern as .csv
    #print(df_input)
    df_output = pd.read_csv('output.csv') #read output pattern
    #print(df_output)
    io=convert_DataFrame_To_numpy_Array(df_input,df_output)
    return io
def generate_weight():  #Generate the weights matrix
    wt_input_to_hidden = np.random.random((b,num_of_hidden_node))
    wt_hidden_to_output = np.random.random((num_of_hidden_node,number_of_out_node))
    return wt_input_to_hidden,wt_hidden_to_output

def sigmoid(x):    #Sigmoid thresholding function
    x = (-1)*x
    x = x*k1
    sig = 1+np.exp(x)
```

```python
    sig = 1/sig
    return sig
def call_again_forward_pass(wt_input_to_hidden,wt_hidden_to_output,input):

    #Start input-hidden layer calculation
    hidden_layer_net_input = np.dot(input,wt_input_to_hidden)
    hidden_layer_net_input += b1 #adding bias b1
    hidden_layer_output = sigmoid(hidden_layer_net_input) # Activation by a sigmoid function
    #End input-hidden layer calculation

    #Start hidden-output layer calculation
    output_layer_net_input = np.dot(hidden_layer_output,wt_hidden_to_output)
    output_layer_net_input += b2 #adding bias b2
    output_layer_output = sigmoid(output_layer_net_input)
    #print(output_layer_output)

    return hidden_layer_output,output_layer_output


def forward_pass(wt,io):
    wt_input_to_hidden = wt[0]
    wt_hidden_to_output =wt[1]

    input  = io[0]
    output = io[1]
    #Start input-hidden layer calculation
    hidden_layer_net_input = np.dot(input,wt_input_to_hidden)
    hidden_layer_net_input += b1 #adding bias b1
    hidden_layer_output = sigmoid(hidden_layer_net_input) # Activation by a sigmoid function
    #End input-hidden layer calculation

    #Start hidden-output layer calculation
    output_layer_net_input = np.dot(hidden_layer_output,wt_hidden_to_output)
    output_layer_net_input += b2 #adding bias b2
    output_layer_output = sigmoid(output_layer_net_input)
    #print(output_layer_output)

    return hidden_layer_output,output_layer_output

def backward_pass(h_and_o_output,wt,input):
    global target_output
    #print('target_output in back pass:',target_output)
    #print('shape of target_output in back pass:',target_output.shape)
    global count
    wt_input_to_hidden = wt[0]
    wt_hidden_to_output = wt[1]

    hidden_layer_output  = h_and_o_output[0]
    output_layer_output = h_and_o_output[1]
    flag=0
        while True:
            #start updating hidden-to-output layer weight..

            delta_of_error =  target_output-output_layer_output
            print('Error : ',np.abs(delta_of_error))
            avg = np.sum(np.abs(delta_of_error))
            dim = delta_of_error.shape
            r = dim[0]
            c = dim[1]
            avg = avg/(r*c)
            print('avg=',avg)
            if avg<=error_threshold:
                break

            delta_hidden_output_layer = delta_of_error*output_layer_output*(1-output_layer_output)
            delta_b2 = delta_hidden_output_layer*eta2*k2
            delta_hidden_output_layer = hidden_layer_output.T.dot(delta_hidden_output_layer)
            delta_hidden_output_layer = delta_hidden_output_layer*eta2*k2
            wt_hidden_to_output = wt_hidden_to_output + delta_hidden_output_layer

            #End updating hidden-output layer weight

            #Start updating bias 2
            global b2
            d_b = b2 + delta_b2
            b2  = d_b
            #End updating bias 2

            #Start updating input-to-hidden layer weights
            delta_hidden_input_layer = wt_hidden_to_output.dot(delta_of_error.T)
            derivative = hidden_layer_output*(1-hidden_layer_output)
            delta_hidden_input_layer =delta_hidden_input_layer*(derivative.T)
            delta_b1 = delta_hidden_input_layer
            delta_b1 = delta_b1*eta1*k1
            delta_hidden_input_layer = input.T.dot(delta_hidden_input_layer.T)
            delta_hidden_input_layer = delta_hidden_input_layer*eta1*k1
            wt_input_to_hidden = wt_input_to_hidden + delta_hidden_input_layer
            #print('shape wt_hidden_to_output:',wt_input_to_hidden.shape)
            #End updating input-to-hidden layer weights


            #Start updating bias 1
            global b1
            d_b = b1 + delta_b1.T
            b1  = d_b

            #End updating bias 1

            res=call_again_forward_pass(wt_input_to_hidden,wt_hidden_to_output,input)
            hidden_layer_output = res[0]
            output_layer_output = res[1]

            count+=1
            print('Count :',count)
        return wt_input_to_hidden,wt_hidden_to_output

    def main_function():
        io=read_input_output_file() #Read input/out pattern as pandas dataFrame
        input = io[0]
        global target_output
        target_output = io[1]
        #print(input)
        #print(input.shape)

        wt = generate_weight()
        wt_input_to_hidden = wt[0]
        wt_hidden_to_output = wt[1]

```

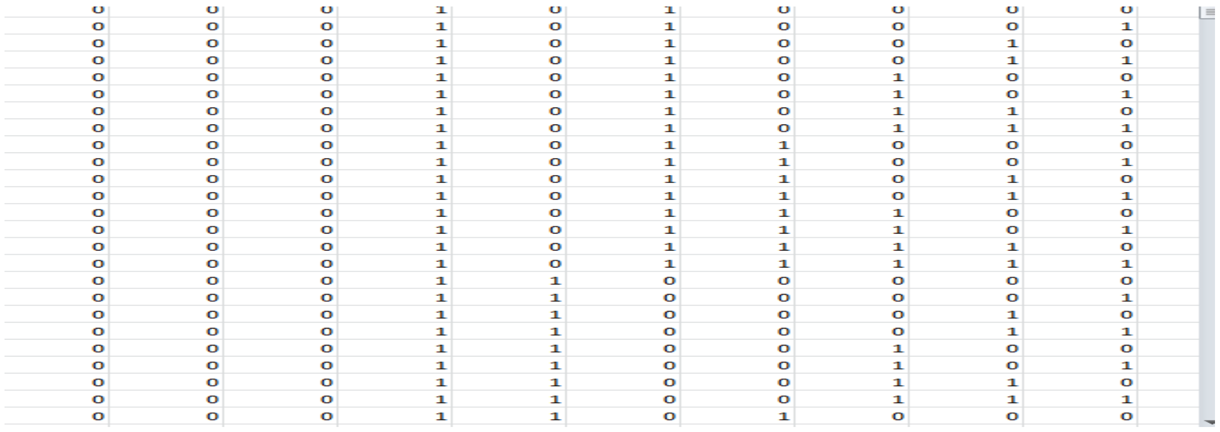
```
            #print('bias : b1',b1)
            #print('weight before update:\ni-h:',wt_input_to_hidden,'\nh-o:',wt_hidden_to_output)
            h_and_o_output = forward_pass(wt,io)
            wt=backward_pass(h_and_o_output,wt,input)
            print('weight after update:\ni-h:',wt[0],'\nh-o:',wt[1])
            #print('bias updated : b2',b2.shape)
            test_result = forward_pass(wt,io)
            print('Result set[Output set] :',test_result[1])
            print('result after [Error Rete ]:',count,' times back propagation',np.abs(test_result[1]-target_output))

            #START MEASURE ACCURACY
            #START 100%
            val = np.abs(test_result[1]-target_output)

            sum_of_error_case1 = np.sum(val)
            print('VAL = ',sum_of_error_case1)
            sum_of_error_case1 = sum_of_error_case1/1024.0
            sum_of_error_case1 = np.abs(1-sum_of_error_case1)
            print('VAL = ',sum_of_error_case1)
            print('Accuracy at 40% Train and 60% Test ::',sum_of_error_case1*100,'%')
            val_case1_up = sum_of_error_case1*100
            val_case1_up_random=val_case1_up*0.6
            #END [Train & Test 100%]

            #sTART 40 AND 60
            #train_data_case2 = pd.read_csv('')
            df_input = pd.read_csv('input.csv')
            df_output = pd.read_csv('output.csv')
            train_data_in_case2 = df_input.head(420)
            train_data_out_case2 = df_output.tail(1024-420)


            #start pass
            h_and_o_output = forward_pass(wt,io)
            wt=backward_pass(h_and_o_output,wt,input)
            print('weight after update:\ni-h:',wt[0],'\nh-o:',wt[1])
            #print('bias updated : b2',b2.shape)
            test_result = forward_pass(wt,io)
            print('Result set[Output set] :',test_result[1])
            print('result after [Error Rete ]:',count,' times back propagation',np.abs(test_result[1]-target_output))
            #end pass
            #Test
            val = np.abs(test_result[1]-target_output)

            sum_of_error_case1 = np.sum(np.square(val))
            print('VAL = ',sum_of_error_case1)
            sum_of_error_case1 = sum_of_error_case1/1024.0
            sum_of_error_case1 = np.abs(1-sum_of_error_case1)
            #print('VAL = ',sum_of_error_case1)
            val_case1 = sum_of_error_case1*100
            print('Accuracy at 100% Train & 100% Test ::',sum_of_error_case1*100,'%')
            #end test

            train_data_in_case2_random  =  df_input.head(420)
            train_data_out_case2_random =  df_output.tail(1024-420)

            #START 10 AND 90 for train and test
            #train_data_case2 = pd.read_csv('')
            df_input = pd.read_csv('input.csv')
            df_output = pd.read_csv('output.csv')
            train_data_in_case2 = df_input.head(100)
            train_data_out_case2 = df_output.tail(924)

            val = np.abs(test_result[1]-target_output)

            sum_of_error_case3 = np.sum(np.square(val))
            #print('VAL = ',sum_of_error_case3)
            sum_of_error_case3 = sum_of_error_case3/1024.0
            sum_of_error_case3 = np.abs(1-sum_of_error_case3)
            sum_of_error_case3 = 1-sum_of_error_case3

            #print('VAL = ',sum_of_error_case3)
            print('Accuracy at 40% Train & 60% Test[1st 40%] ::',val_case1_up_random,'%')
            print('Accuracy at 40% Train & 60% Test[random 40%] ::',val_case1_up,'%')
            print('Accuracy at 10% Train & 90% Test ::',sum_of_error_case3*100,'%')

            #END measuring accuracy

        '''
        Call Main function
        '''
        main_function()
```

3.4 Result

I have used a predefined dataset of 10-bit input pattern, total of 1024 training samples. There are of total three cases I have observed with my implementation.

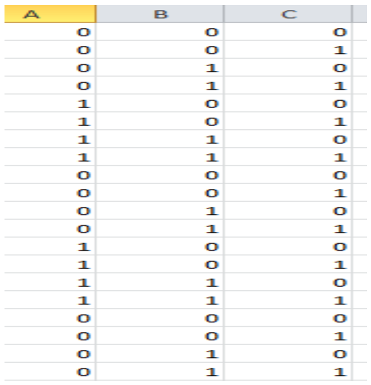
- Case1 :
    - Training Samples-100%
    - Test samples -100%
- Case2 :
    - Training Samples-40%
    - Test Samples -60%
- Case3 :
    - Training samples -10%
    - Test Samples- 90%

Input Pattern :



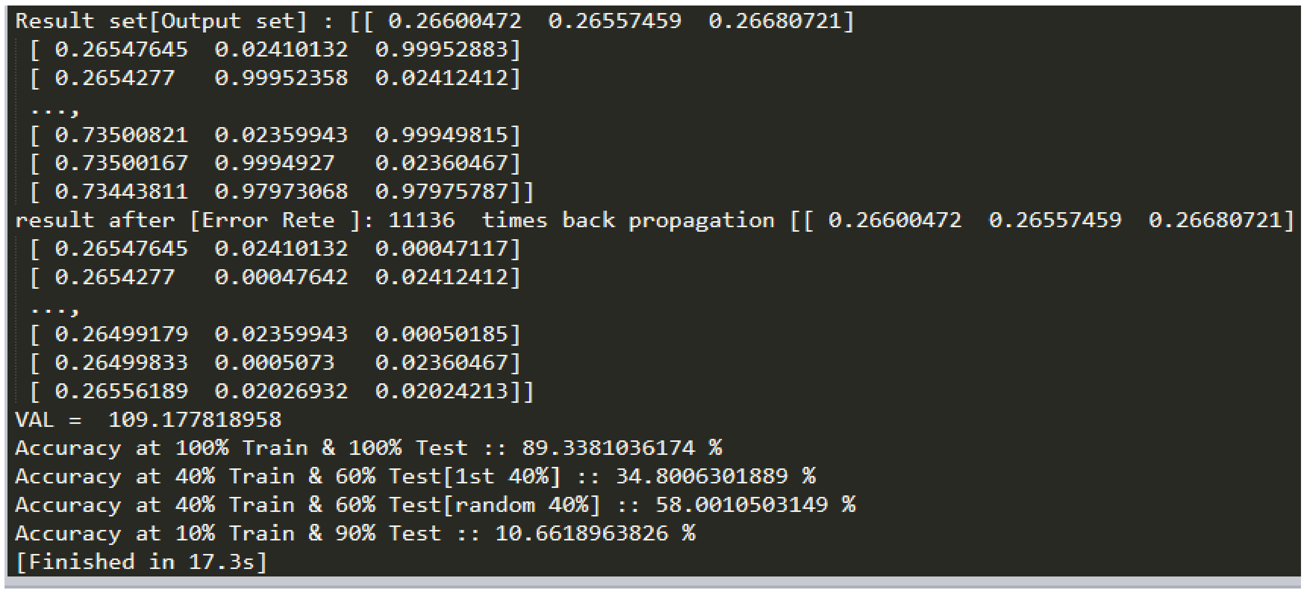
0	0	0	1	0	1	0	0	0	0
0	0	0	1	0	1	0	0	0	1
0	0	0	1	0	1	0	0	1	0
0	0	0	1	0	1	0	0	1	1
0	0	0	1	0	1	0	1	0	0
0	0	0	1	0	1	0	1	0	1
0	0	0	1	0	1	0	1	1	0
0	0	0	1	0	1	0	1	1	1
0	0	0	1	0	1	1	0	0	0
0	0	0	1	0	1	1	0	0	1
0	0	0	1	0	1	1	0	1	0
0	0	0	1	0	1	1	0	1	1
0	0	0	1	0	1	1	1	0	0
0	0	0	1	0	1	1	1	0	1
0	0	0	1	0	1	1	1	1	0
0	0	0	1	0	1	1	1	1	1
0	0	0	1	1	0	0	0	0	0
0	0	0	1	1	0	0	0	0	1
0	0	0	1	1	0	0	0	1	0
0	0	0	1	1	0	0	0	1	1
0	0	0	1	1	0	0	1	0	0
0	0	0	1	1	0	0	1	0	1
0	0	0	1	1	0	0	1	1	0
0	0	0	1	1	0	0	1	1	1
0	0	0	1	1	0	1	0	0	0

Output Pattern (3-bit) :



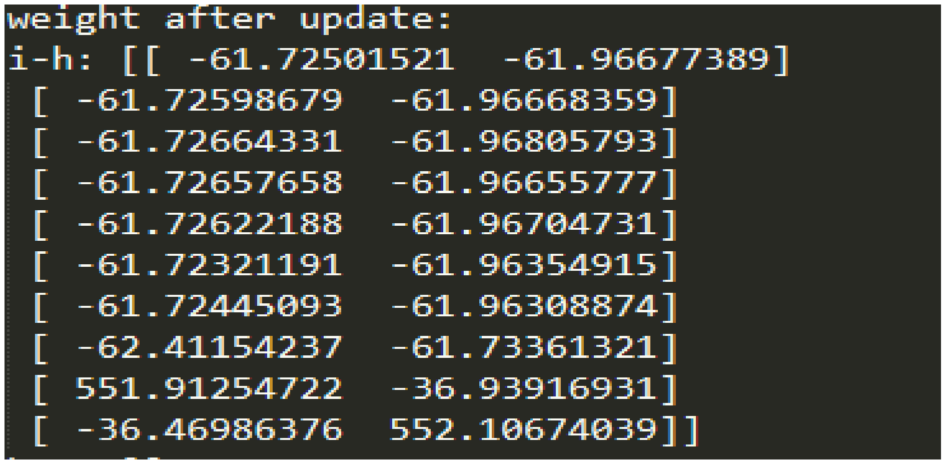
A	B	C
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1
0	0	0
0	0	1
0	1	0
0	1	1

Output(Result and Accuracy) :

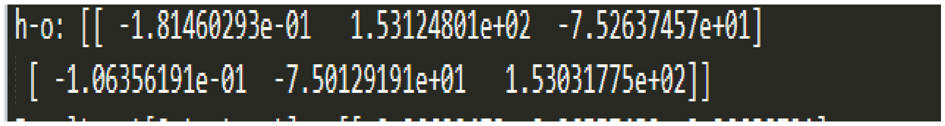


```
Result set[Output set] : [[ 0.26600472  0.26557459  0.26680721]
 [ 0.26547645  0.02410132  0.99952883]
 [ 0.2654277   0.99952358  0.02412412]
 ...,
 [ 0.73500821  0.02359943  0.99949815]
 [ 0.73500167  0.9994927   0.02360467]
 [ 0.73443811  0.97973068  0.97975787]]
result after [Error Rete ]: 11136  times back propagation [[ 0.26600472  0.26557459  0.26680721]
 [ 0.26547645  0.02410132  0.00047117]
 [ 0.2654277   0.00047642  0.02412412]
 ...,
 [ 0.26499179  0.02359943  0.00050185]
 [ 0.26499833  0.0005073   0.02360467]
 [ 0.26556189  0.02026932  0.02024213]]
VAL =  109.177818958
Accuracy at 100% Train & 100% Test :: 89.3381036174 %
Accuracy at 40% Train & 60% Test[1st 40%] :: 34.8006301889 %
Accuracy at 40% Train & 60% Test[random 40%] :: 58.0010503149 %
Accuracy at 10% Train & 90% Test :: 10.6618963826 %
[Finished in 17.3s]
```

Output (Updated weight vector) :



```
weight after update:
i-h: [[ -61.72501521  -61.96677389]
 [ -61.72598679  -61.96668359]
 [ -61.72664331  -61.96805793]
 [ -61.72657658  -61.96655777]
 [ -61.72622188  -61.96704731]
 [ -61.72321191  -61.96354915]
 [ -61.72445093  -61.96308874]
 [ -62.41154237  -61.73361321]
 [ 551.91254722  -36.93916931]
 [ -36.46986376  552.10674039]]
```



```
h-o: [[ -1.81460293e-01   1.53124801e+02  -7.52637457e+01]
 [ -1.06356191e-01  -7.50129191e+01   1.53031775e+02]]
```

3.5 Performance Analysis

Table 3 : Measuring Accuracy of back propagation for the defined dataset.

Total Samples	Number of Hidden Nodes	Training Percentage	Testing Percentage	Accuracy(%)
1024	3	100%	100%	89.338%
1024	3	40%	60%	58.001%
1024	3	10%	90%	10.661%
1024	10	100%	100%	90.223%
1024	10	40%	60%	60.114%
1024	10	10%	90%	16.06%
1024	30	40%	60%	75.0%

- Back propagation has given a high accuracy with the same training and test samples.
- But accuracy degrades a large quantity with the increasing ratio of test samples to train.
- For a dataset with 1024 samples with 10% train and 90% test, it gives only of 10.66% accuracy. But with the increase of hidden nodes accuracy was 16.06%.

3.6 Conclusion and Observation

The multilayer perceptron algorithm is useful for solving linearly non-separable problems that has been proved with our predefined dataset with 1024 samples. In our implementation we introduced sigmoid thresholding function for graceful degradation of back propagation algorithm.

Though it has given a better accuracy with only three hidden layer nodes and 100% training and test samples, accuracy was degraded a large scale with the change of patterns in traing and test percentage. For examples with 40% train and 60% test with different patterns it has given only 58.001% accuracy with three hidden nodes. If we increased the hidden layer nodes to 10,then the accuracy has increased to 60.114% with the same train and test samples. Finally a dramatically change has been occurred with 30 hidden nodes and with the 40% train and 60% test samples. This time accuracy was 75%.

It is observed that with the increasing number of internal nodes accuracy is highly related.Besides the number of internal nodes, the successful classification is highly related to the random choice of two gain factors, the spread factor of sigmoid and the nonlinear thresholding function.

CHAPTER 4

Title: Implementation and Performance Analysis of Kohenen Self-Organizing Networks.

4.1 Objectives

A self-organizing map is a type of artificial neural network that is trained using unsupervised learning to produce a low dimensional discretized representation of the input space of the training samples called a map.

- Kohenen-SOM is a method to do dimensionality reduction.
- It is useful for data visualization in low dimensional space.
- It is one of the most useful unsupervised learning algorithm.
- It is also common use to U-matrix.
- It is possible to perform cluster operations on the map itself.
- In Kohenen-SOM, spatial or topological relationships in training data are maintained and represented in a meaningful way.
- It allows the network to store data via vector quantization.

4.2 Methodology

4.2.1 Pseudocode

```
1) Initialize the network.Define $w_{ij}(t)$ (0<i<n-1) to be the weight from input i to node j at time t.
2) Set the initial radius of the neighbourhood around the node j, $N_j(0)$ to be very large.
3) Present input as $x_0(t), x_1(t), x_2(t), \ldots\ldots\ldots x_{n-1}(t)$ where $x_i(t)$ is the input to node i at time t.
4) Calculate the distances
        Compute the distance $d_j$ between the input and each output mode j given by,
        $d_j = \sum_{i=0}^{n-1}(x(t) - w(t))^2$
5) Select the minimum distances ,Designate the output node with minimum $d_j$
6) Update weights for node j* and its neighbours defined by the neighbourhood size $N_{j*}(t)$ New weights are
    i.  $w_{ij}(t+1) = w_{ij}(t) + \mu(t) ( x_i(t) - w_{ij}(t) )$
    ii. The term $\mu$ is gain term ($0< \mu <1$ )
7) Repeat by going (2)
```

4.3 Implementation

I have implemented the Kohenen_SOM algorithm according to the above Pseudocode.
The tools I used for implementation is :

- C++ 11
- CodeBlocks 13.12

4.3.1 Source Code (train_kohenen_som.cpp)

```
#include<bits/stdc++.h>
using namespace std;
int b=10,rem;
const double decayRate = 0.96;
const double minAlpha = 0.01;
double alpha = 0.6;
double d[maxClusters];
double w[maxClusters][vecLen];
int pattern[vectors][vecLen];
//vector<double>output; //Only contains the [num_out_node] number of output
vector<double>target_output;  //Contains all the output for all input pattern
vector<int>binari;
int num_hidden_node=2,num_out_node=3;//user defined
vector<vector<int> > input((int)pow(2,b));
int tests[vectors][vecLen];
void training();
void testing();
void computeInput(int vectorNumber);
int minimum(double valueA, double valueB);
const int maxClusters =2; //number of class
const int vectors = 10;  //Number of input pattern
const int vecLen = 10;   //number of input bits

vector <int>convert_binary_input_bit(int n,int bb)
{
    binari.clear();
    rem=0;
    while(n!=0)
    {
        rem=n%2;
        binari.push_back(rem);
        n/=2;
    }
    reverse(binari.begin(),binari.end());
    int pos=0;
    pos = binari.size();
    for(int i=pos;i<bb;i++)
    {
        binari.push_back(0);
    }
    rotate(binari.begin(),binari.begin()+pos,binari.end());
    return binari;
}
void generate_input()
{
    int limit = (int)pow(2,b);

    for(int i=0;i<limit;i++)
    {
            vector<int>tmp = convert_binary_input_bit(i,b);
            input.push_back(tmp);
          tmp.clear();
        }
}
void generate_weight()
{
    double wt=0.0;
    for(int i=0;i<maxClusters;i++)
    {
        for(int j=0;j<vecLen;j++)
        {
            wt = (rand()%6 )/10.0;
            w[i][j]=wt;
        }
    }
}
void generate_in_out()
{
    input.clear();
    generate_input();

}
void get_input_pattern_for_train_test()
{
    for(int i=0;i<10;i++)
    {
        for(int j=0;j<10;j++)
        {
            pattern[i][j]=input[i][j];
            //cout<<input[i][j]<<" ";
            cout<<pattern[i][j]<<" ";
        }
        cout<<endl;
    }
    int len = vectors+vectors;
    for(int i=10;i<20;i++)
    {
        for(int j=0;j<10;j++)
        {
            tests[i][j]=input[i][j];
        }
    }
}
int main(){
    cout << fixed << setprecision(3) << endl;  //Format all the output.
    generate_weight();
    generate_in_out();
    get_input_pattern_for_train_test();
    training();
    testing();
    return 0;
}
void training(){
    int iterations = 0;
    int dMin = 0;
    do {
        iterations += 1;
        for(int vecNum = 0; vecNum <= (vectors - 1); vecNum++){
            //Compute input.
            computeInput(vecNum);

            //See which is smaller, d[0] or d[1]?
            int smaller=999999;
            for(int i=0;i<maxClusters;i++)
            {
                if(d[i]<smaller)
                    smaller=d[i];
            }
            dMin=smaller;
            cout<<"dmin = "<<dMin<<endl;
            //Update the weights on the winning unit.
            for(int i = 0; i <= (vectors - 1); i++){
            w[dMin][i] = w[dMin][i] + (alpha * (pattern[vecNum][i] - w[dMin][i]));
            }
        }
        //Reduce the learning rate.
     alpha = decayRate * alpha;
    } while(alpha > minAlpha);
    cout << "Iterations: " << iterations << "\n\n";
}

//Start test Kohenen Map
void testing(){
    int dMin;
//Print clusters created.
    cout << "Clusters for training input:" << endl;
    for(int vecNum = 0; vecNum <= (vectors - 1); vecNum++){
        //Compute input.
        computeInput(vecNum);
        //See which is smaller, d[0] or d[1]?
        int smaller=999999;
            for(int i=0;i<maxClusters;i++)
            {
                if(d[i]<smaller)
                    smaller=d[i];
            }
            dMin=smaller;
        cout << "\nVector (";
        for(int i = 0; i <= (vectors - 1); i++){
            cout << pattern[vecNum][i] << ", ";
        } // i
        cout << ") fits into category " << dMin << endl;
    } // vecNum
//Print weight matrix.
    cout << "\n";
    for(int i = 0; i <= (maxClusters - 1); i++){
        cout << "Weights for Node " << i << " connections:" << endl;
        for(int j = 0; j <= (vecLen - 1); j++){
            cout << w[i][j] << ", ";
        } // j
        cout << "\n\n";
    } // i
//Print post-training tests.
    cout << "Categorized test input:" << endl;
    for(int vecNum = 0; vecNum <= (vectors - 1); vecNum++){
        //Compute input.
        computeInput(vecNum);
        //See which is smaller, d[0] or d[1]?
        int smaller=999999;
            for(int i=0;i<maxClusters;i++)
            {
```

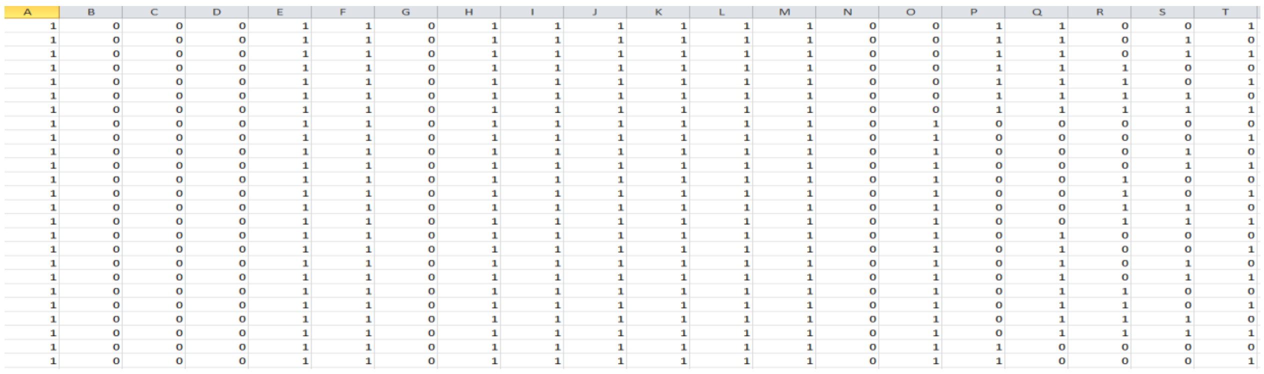
```
        if(d[i]<smaller)
        smaller=d[i];
      }
      dMin=smaller;
    cout << "\nVector (";
    for(int i = 0; i <= (vectors - 1); i++){
      cout << tests[vecNum][i] << ", ";
    } // i
    cout << ") fits into category " << dMin << endl;
  } // vecNum
}
void computeInput(int vectorNumber){

      for(int i=0;i<maxClusters;i++)
      {
        d[i]=0;
      }
  for(int i = 0; i <= (maxClusters - 1); i++){
    for(int j = 0; j <= (vectors - 1); j++){
      d[i] += pow((w[i][j] - tests[vectorNumber][j]), 2);
      cout << "Distance= " << d[i] << "\n";
    }
  }
}
```

4.4 Result

I have used a predefined dataset of 20-bit input pattern, total of 100 training samples. There are of total two cases I have observed with my implementation.

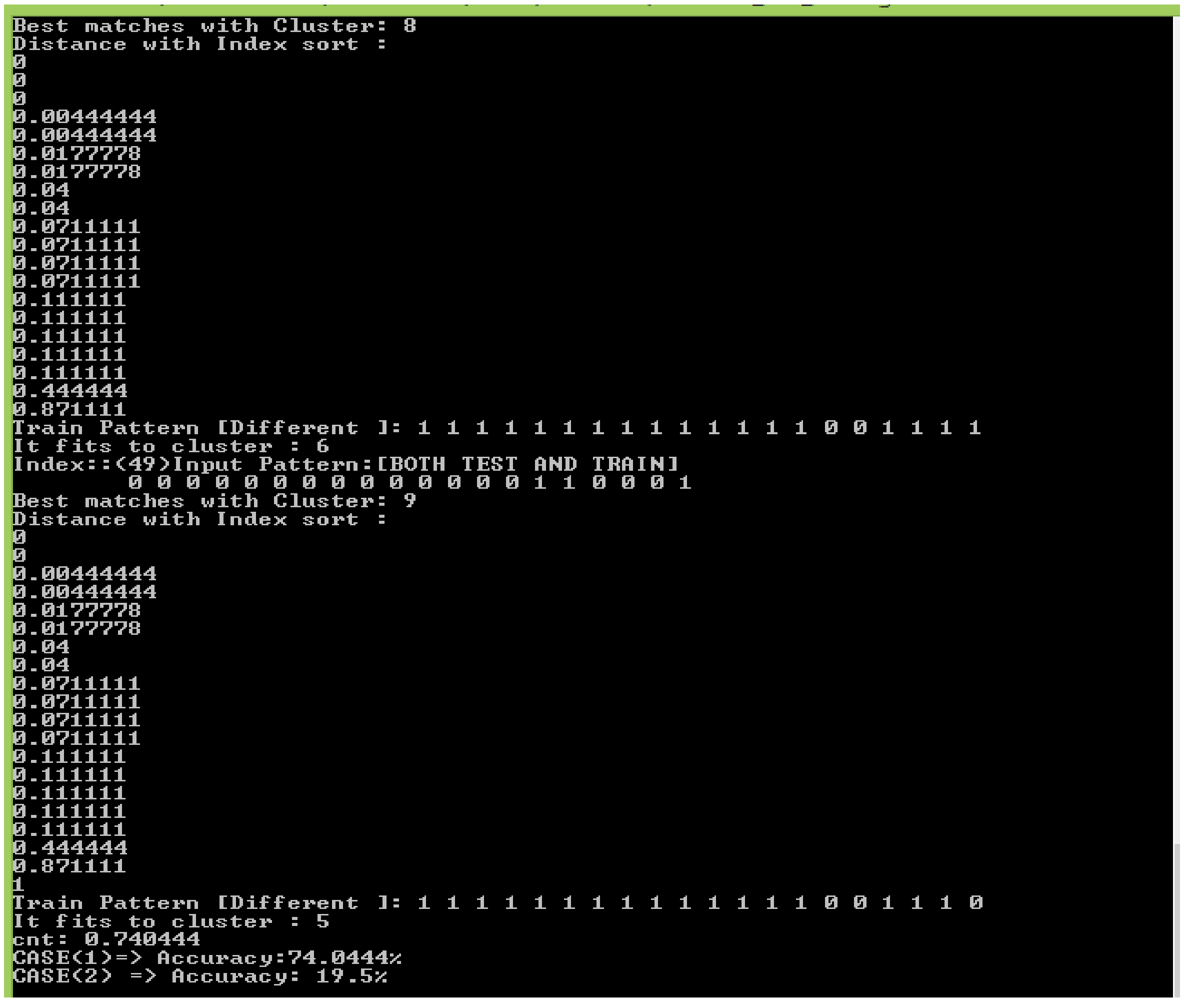
- **Case1 :**
  - Training Samples-Combination of last 5 bit
  - Test samples–Same group as training
- **Case2 :**
  - Training Samples-Combination of last 5 bit
  - Test Samples–Different group of testing sample

Input Pattern (20-bit) :



A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
1	0	0	0	1	1	0	1	1	1	1	1	1	0	0	1	1	0	0	1
1	0	0	0	1	1	0	1	1	1	1	1	1	0	0	1	1	0	1	0
1	0	0	0	1	1	0	1	1	1	1	1	1	0	0	1	1	0	1	1
1	0	0	0	1	1	0	1	1	1	1	1	1	0	0	1	1	1	0	0
1	0	0	0	1	1	0	1	1	1	1	1	1	0	0	1	1	1	0	1
1	0	0	0	1	1	0	1	1	1	1	1	1	0	0	1	1	1	1	0
1	0	0	0	1	1	0	1	1	1	1	1	1	0	0	1	1	1	1	1
1	0	0	0	1	1	0	1	1	1	1	1	1	0	1	0	0	0	0	0
1	0	0	0	1	1	0	1	1	1	1	1	1	0	1	0	0	0	0	1
1	0	0	0	1	1	0	1	1	1	1	1	1	0	1	0	0	0	1	0
1	0	0	0	1	1	0	1	1	1	1	1	1	0	1	0	0	0	1	1
1	0	0	0	1	1	0	1	1	1	1	1	1	0	1	0	0	1	0	0
1	0	0	0	1	1	0	1	1	1	1	1	1	0	1	0	0	1	0	1
1	0	0	0	1	1	0	1	1	1	1	1	1	0	1	0	0	1	1	0
1	0	0	0	1	1	0	1	1	1	1	1	1	0	1	0	0	1	1	1
1	0	0	0	1	1	0	1	1	1	1	1	1	0	1	0	1	0	0	0
1	0	0	0	1	1	0	1	1	1	1	1	1	0	1	0	1	0	0	1
1	0	0	0	1	1	0	1	1	1	1	1	1	0	1	0	1	0	1	0
1	0	0	0	1	1	0	1	1	1	1	1	1	0	1	0	1	0	1	1
1	0	0	0	1	1	0	1	1	1	1	1	1	0	1	0	1	1	0	0
1	0	0	0	1	1	0	1	1	1	1	1	1	0	1	0	1	1	0	1
1	0	0	0	1	1	0	1	1	1	1	1	1	0	1	0	1	1	1	0
1	0	0	0	1	1	0	1	1	1	1	1	1	0	1	0	1	1	1	1
1	0	0	0	1	1	0	1	1	1	1	1	1	0	1	1	0	0	0	0
1	0	0	0	1	1	0	1	1	1	1	1	1	0	1	1	0	0	0	1

Program's Output(From test_kohenen.cpp) :



```
Best matches with Cluster: 8
Distance with Index sort :
0
0
0
0.00444444
0.00444444
0.0177778
0.0177778
0.04
0.04
0.0711111
0.0711111
0.0711111
0.0711111
0.111111
0.111111
0.111111
0.111111
0.111111
0.444444
0.871111
Train Pattern [Different ]: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1
It fits to cluster : 6
Index::(49)Input Pattern:[BOTH TEST AND TRAIN]
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1
Best matches with Cluster: 9
Distance with Index sort :
0
0
0.00444444
0.00444444
0.0177778
0.0177778
0.04
0.04
0.0711111
0.0711111
0.0711111
0.0711111
0.111111
0.111111
0.111111
0.111111
0.111111
0.444444
0.871111
1
Train Pattern [Different ]: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1 0
It fits to cluster : 5
cnt: 0.740444
CASE(1)=> Accuracy:74.0444%
CASE(2) => Accuracy: 19.5%
```

4.5 Performance Analysis

Table 4 : Measuring Performance of Kohenen-SOM for the defined dataset.

Total samples	Train/Test Property	Number of Predefined Clusters	Decay rate	Accuracy(%)
100	Same group	6	.93	74.004%
100	Different group	6	.93	19.58%
200	Same group	10	.85	65%
200	Different group	10	.93	25.28%

4.6 Conclusion and Observation

I have implemented Kohenen Self-organizing algorithm by c++ and for input I used c++11 dynamic vector. But the difficulty is to store high dimensional feature vector,here I used 20bit.By applying SOM the 20-bit feature vector can be mapped to a low dimensional vector space.

The performance of SOM can not be directly measured like other supervised learning algorithm.As it's a clustering approach I predefined the number of cluster to calculate the accuracy.

The performance of Kohenen depends on many factor such as the train/test property,decay-rate,neighbourhood radius and so on. I considered total four cases.For 100 samples of same train and test data and with .93 decay rate I found 74% of accuracy.But if the train/test samples are selected from different group then the accuracy degrades to a large scale.And in this case I found only 19.58% accuracy.

It can be easy to define a value for the number of cluster,k.And the programs then tries to build at most k cluster from the dataset.

The clustering quality of our Kohenen map largely depends on how we deign the neighborhood decrease function and the initial radius of the neighbourhood.

CHAPTER 5

Title: Implementation and Performance Analysis of Hopfield Networks.

5.1 Objectives

A **Hopfield network** is a form of recurrent artificial neural network popularized by John Hopfield. Hopfield nets serve as content-addressable memory systems with binary/bipolar threshold nodes. They are guaranteed to converge to a local minimum, but convergence to a false pattern (wrong local minimum) rather than the stored pattern (expected local minimum). Accurate recognition even if the noise level is greater than 50%, and even a man is hard to recognize. It has been proved that Hopfield network is resistant. In general, it can be more than one fixed point.

It is a symmetrically weighted network as the weights on the link from one node to another are the same in both directon.It is a fully connected network.

5.2 Methodology

Pseudocode:

```
1) Calculate the weight matrix W using the formula

   $W_{ij} = \sum_{s=0}^{m-1} x_i^s x_j^s \quad ; i \neq j$
   $\quad\;\; = 0 \quad ; i = j$

2) Calculate the output vector components, j = 1,2, .., n, using the formula below:

   $y_j = T\left(\sum_{i=1}^{n} w_{ij} x_i\right)$
   $T(x) = \begin{cases} -1, & x < 0 \\ 1, & x > 0 \end{cases}$

3) Perform the asynchronous correction.
   Start with the input vector $(x_1, x_2, \ldots, x_n)$.
   Find $\mathbf{y}_j$ according to formula (2).

4) Replace the $(x_1, x_2, \ldots, x_n)$ on the $(y_1, x_2, x_3, \ldots, x_n) = Y$ and feed Y on the input X.
```

5) Repeat steps 2-3 for as long as the vector $Y = (y_1, y_2, \ldots, y_n)$ is no longer changed. Each step reduces the amount of energy ties:

$$E = \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} w_{ij} x_i x_j$$

5.3 Implementation

I have implemented the Hopfield Network according to the above Pseudocode.
The tools I used for implementation is :

- Python 3.0
- Pandas DataFrame
- Numpy
- MatPlotlib

5.3.1 Source Code (hopField.py)

```
#Its Faaruk's Python Shell...Lets ENJOY
#HopFiled network python 3.0
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import math

def convert_binary_to_bipolar(input,test):

    rows_input = input.shape[0] #rows of train patterns
    cols_input = input.shape[1] #coloumnss of train patterns

    for i in range(rows_input):
        for j in range(cols_input):
            if input[i][j]==0:
                input[i][j]=-1  #Replace 0 by -1 as bipolar input


    rows_test = test.shape[0]  #rows of test patterns
    cols_test = test.shape[1]  #Colomns of test patterns

    for i in range(rows_test):
        for j in range(cols_test):
            if test[i][j]==0:
                test[i][j]=-1 #Replace 0 by -1


    return input,test


def get_input():
    df = pd.read_csv('input.csv')  #Read .csv input file
    #ufo.sample(n=3, random_state=2)
    df1 = df.sample(n=20,random_state=2) #Random sample of Train patterns
    df2 = df.sample(n=5,random_state=2)  #Random sample of Test patterns
```

```python
    convert_to_numpy_pattern = df1.values #Convert Pandas dataframe to Numpy Array
    convert_to_numpy_test_pattern = df2.values

    #Convert to bipoloar train nand test
    input_test=convert_binary_to_bipolar(convert_to_numpy_pattern,convert_to_numpy_test_pattern)
    return input_test

def generate_weight_matrix(input,n):

    weight = np.full((n,n),0) #Create a NxN empty Weight matrix for 'N' bit input pattern
    #print('weight matrix looks like:',weight)

    total_pattern = input.shape[0]

    for i in range(n):
        for j in range(n):

            if i==j:     #Hebbian condition for i=j,Wij(t+1)=0
                continue #As already assign '0'

            else:
                sum=0.0
                for ii in range(total_pattern):
                        sum = sum + input[ii][i] * input[ii][j] #Calculate the Sum of dot product  i and j train columns
                weight[i][j] = sum  #Assign weight for Wij(t+1)


    #print('weight matrix :',weight)
    return weight

def send_to_test_iteration(pattern,weight):

    n = weight.shape[0]

    for i in range(n):
        dot_product = pattern*weight[i]
        sum=np.sum(dot_product)
        sum+=pattern[i]
        if sum >= 0:
            pattern[i] = 1
        else:
            pattern[i] = -1


    return pattern


def testing_HopField(test_pattern,weight):
    rows = test_pattern.shape[0]
    cols = test_pattern.shape[1]

    for i in range(rows):
        pattern = test_pattern[i]
        print('[',i+1,']. The Test Pattern:',pattern)
        modified_pattern = send_to_test_iteration(pattern,weight)

        print('is Symmetrical To     ::',modified_pattern)
        print('\n')

```

```python

def main_function():

    input_test_pattern = get_input() # Get Train & Test pattern as .csv file
    input_pattern = input_test_pattern[0] #Spilit into train
    test_pattern = input_test_pattern[1]
    print('\n The input Patterns of ',input_pattern.shape[1],' Bit \n',input_pattern)
    number_of_input_bit = input_pattern.shape[1] #The number of column represents number of bits

    weight = generate_weight_matrix(input_pattern,number_of_input_bit) #External function to generate weight matrix
    print('\n The weight matrix are : \n',weight)
    print('\n')

    testing_HopField(test_pattern,weight) #Call a HopField Test function to recovery of Hopfield Network


main_function()
```

5.4 Result

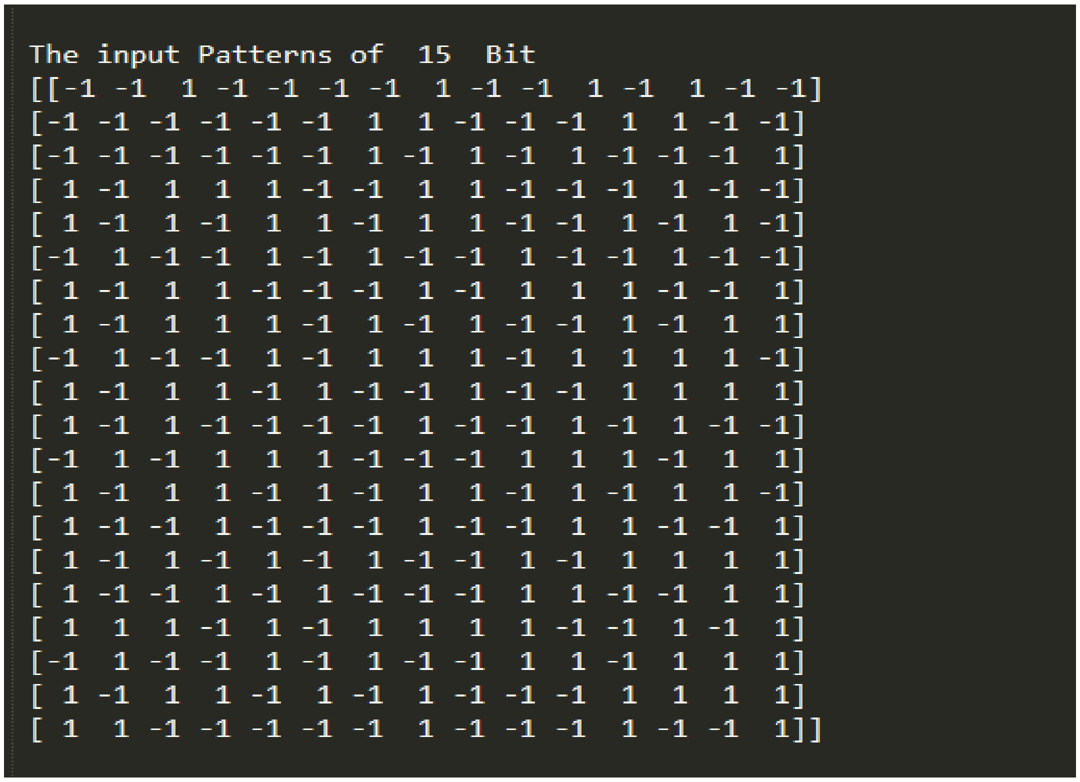
I have used a predefined dataset of variable-bit input pattern.There are the following cases I have observed with my implementation according to the rules : M = 0.15N where N is the number of input nodes and M is the total training pattern.I have described all of the following cases but in this report I showed only the 15 bit pattern result.

- **Case 1 :**
  - N = 15 and M = 2 ; 0.15*15 = 2.25
  - Convergence epoch 1100
- **Case 2 :**
  - N = 15 and M = 20 ; 0.15*15 <<20
  - Convergence epoch 6000

- **Case 3 :**
  - N = 20 and M = 3 ; 0.15*20 = 3
  - Convergence epoch 100

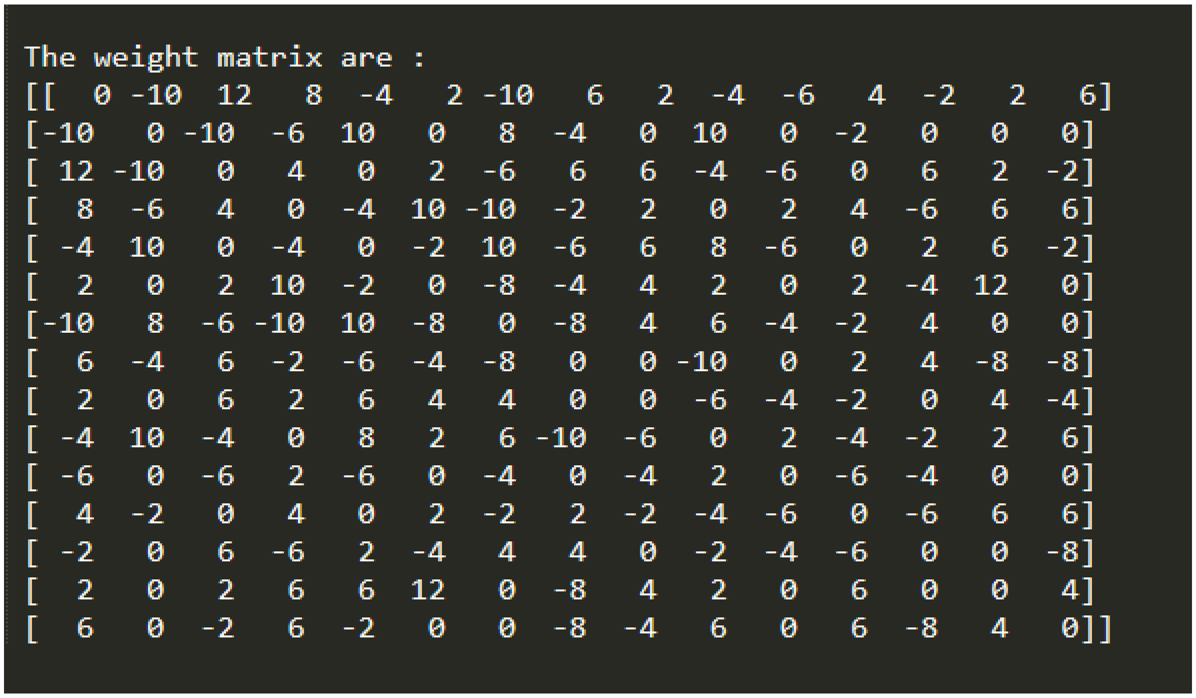
- **Case 4 :**
  - N = 20 and M = 20 ; 0.15*20 <<20
  - Convergence epoch 9780

Input Pattern (15-bit samples) :



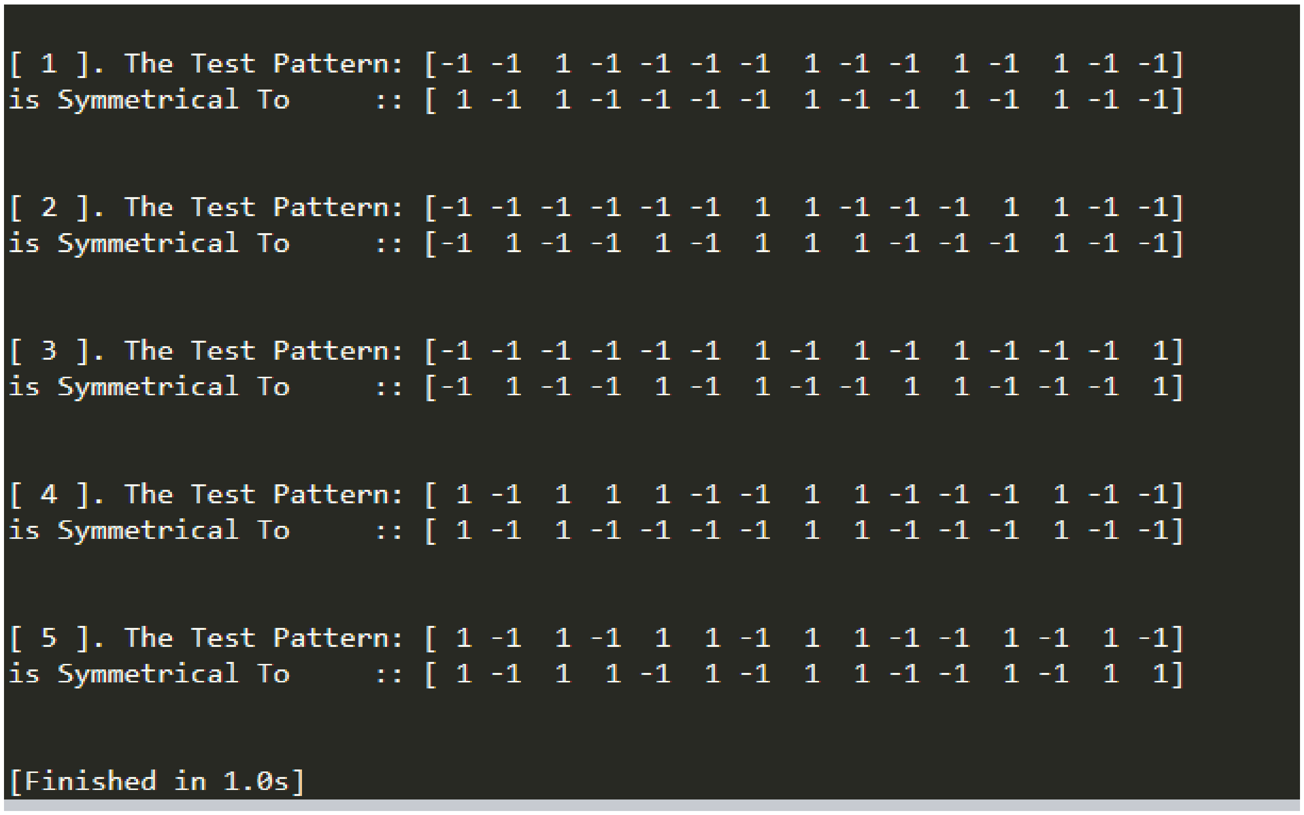
```
The input Patterns of  15  Bit
[[-1 -1  1 -1 -1 -1 -1  1 -1 -1  1 -1  1 -1 -1]
 [-1 -1 -1 -1 -1 -1  1  1 -1 -1 -1  1  1 -1 -1]
 [-1 -1 -1 -1 -1 -1  1 -1  1 -1  1 -1 -1 -1  1]
 [ 1 -1  1  1  1 -1 -1  1  1 -1 -1 -1  1 -1 -1]
 [ 1 -1  1 -1  1  1 -1  1  1 -1 -1  1 -1  1 -1]
 [-1  1 -1 -1  1 -1  1 -1 -1  1 -1 -1  1 -1 -1]
 [ 1 -1  1  1 -1 -1 -1  1 -1  1  1  1 -1 -1  1]
 [ 1 -1  1  1  1 -1  1 -1  1 -1 -1  1 -1  1  1]
 [-1  1 -1 -1  1 -1  1  1  1 -1  1  1  1  1 -1]
 [ 1 -1  1  1 -1  1 -1 -1  1 -1 -1  1  1  1  1]
 [ 1 -1  1 -1 -1 -1 -1  1 -1 -1  1 -1  1 -1 -1]
 [-1  1 -1  1  1  1 -1 -1 -1  1  1  1 -1  1  1]
 [ 1 -1  1  1 -1  1 -1  1  1 -1  1 -1  1  1 -1]
 [ 1 -1 -1  1 -1 -1 -1  1 -1 -1  1  1 -1 -1  1]
 [ 1 -1  1 -1  1 -1  1 -1 -1  1 -1  1  1  1  1]
 [ 1 -1 -1  1 -1  1 -1 -1 -1  1  1 -1 -1  1  1]
 [ 1  1  1 -1  1 -1  1  1  1  1 -1 -1  1 -1  1]
 [-1  1 -1 -1  1 -1  1 -1 -1  1  1 -1  1  1  1]
 [ 1 -1  1  1 -1  1 -1  1 -1 -1 -1  1  1  1  1]
 [ 1  1 -1 -1 -1 -1 -1  1 -1 -1 -1  1 -1 -1  1]]
```

Program's Output (Weight Matrix) :



```
The weight matrix are :
[[  0 -10  12   8  -4   2 -10   6   2  -4  -6   4  -2   2   6]
 [-10   0 -10  -6  10   0   8  -4   0  10   0  -2   0   0   0]
 [ 12 -10   0   4   0   2  -6   6   6  -4  -6   0   6   2  -2]
 [  8  -6   4   0  -4  10 -10  -2   2   0   2   4  -6   6   6]
 [ -4  10   0  -4   0  -2  10  -6   6   8  -6   0   2   6  -2]
 [  2   0   2  10  -2   0  -8  -4   4   2   0   2  -4  12   0]
 [-10   8  -6 -10  10  -8   0  -8   4   6  -4  -2   4   0   0]
 [  6  -4   6  -2  -6  -4  -8   0   0 -10   0   2   4  -8  -8]
 [  2   0   6   2   6   4   4   0   0  -6  -4  -2   0   4  -4]
 [ -4  10  -4   0   8   2   6 -10  -6   0   2  -4  -2   2   6]
 [ -6   0  -6   2  -6   0  -4   0  -4   2   0  -6  -4   0   0]
 [  4  -2   0   4   0   2  -2   2  -2  -4  -6   0  -6   6   6]
 [ -2   0   6  -6   2  -4   4   4   0  -2  -4  -6   0   0  -8]
 [  2   0   2   6   6  12   0  -8   4   2   0   6   0   0   4]
 [  6   0  -2   6  -2   0   0  -8  -4   6   0   6  -8   4   0]]
```

Program's Output (Test result) :



```
[ 1 ]. The Test Pattern: [-1 -1  1 -1 -1 -1 -1  1 -1 -1  1 -1  1 -1 -1]
is Symmetrical To     :: [ 1 -1  1 -1 -1 -1 -1  1 -1 -1  1 -1  1 -1 -1]


[ 2 ]. The Test Pattern: [-1 -1 -1 -1 -1 -1  1  1 -1 -1 -1  1  1 -1 -1]
is Symmetrical To     :: [-1  1 -1 -1  1 -1  1  1  1 -1 -1 -1  1 -1 -1]


[ 3 ]. The Test Pattern: [-1 -1 -1 -1 -1 -1  1 -1  1 -1  1 -1 -1 -1  1]
is Symmetrical To     :: [-1  1 -1 -1  1 -1  1 -1 -1  1  1 -1 -1 -1  1]


[ 4 ]. The Test Pattern: [ 1 -1  1  1  1 -1 -1  1  1 -1 -1 -1  1 -1 -1]
is Symmetrical To     :: [ 1 -1  1 -1 -1 -1 -1  1  1 -1 -1 -1  1 -1 -1]


[ 5 ]. The Test Pattern: [ 1 -1  1 -1  1  1 -1  1  1 -1 -1  1 -1  1 -1]
is Symmetrical To     :: [ 1 -1  1  1 -1  1 -1  1  1 -1 -1  1 -1  1  1]


[Finished in 1.0s]
```

5.5 Performance Analysis

Table 5: Measuring performance of HopField Network

Cases	Number of Internal Nodes (N)	Number of Train Patterns (M)	Number of Test Patterns	Number of Pattern Recovery	Accuracy(%)
Case 1	15	2	5	2	100%
Case 2	15	20	5	2	40%
Case 3	20	3	6	6	100%
Case 4	20	20	6	1	16.67%

5.6 Conclusion and Observation

The assumption made in a Hopfield network of a symmetric,zero-diagonal weight matrix are the central to its operation.Even slight deviation from this symmetry can give rise to networks that are unstable and do not settle into any final state.

The performance of Hopfield network greatly depends on the ratio of input nodes and number of train patterns.The main task in training Hopfield network is to generate a weight matrix from the input patterns.If the weight matrix can be generated accurately according to the rules ,M=0.15N,then it can be able to recover the testing pattern successfully.

For example in case 1 I have considered N=15 and M=2 here 0.15*15=2.25 which has given 100% accuracy as it could be able to recover all 5 patterns out of 5 patterns.If case 2 ,N = 15 and M = 20 that violates the rules as 0.15*15 <<20,and it could not recover maximum patterns.If case 3, N = 20 and M = 3 which totally fulfills the above condition, and it could be able to recover all of the 6 testing patterns.

So it is proved that if the above rules are not maintained, the hopfield network may generate errors in the recovery of new patterns.

Besides this the performance of the network also greatly depends on the convergence epoch and convergence condition.