1. Training Data:

The following is the data for training and validation.

| Data Item | LAC | SOW | TACA |
|---|---|---|---|
| 1 | 1.98 | 10K | 0 |
| 2 | 1.80 | 10K | 1 |
| 3 | 1.05 | 160K | 2 |
| 4 | 1.45 | 180K | 1 |
| 5 | 1.8 | 80K | 1 |
| 6 | 1.96 | 110K | 1 |
| 7 | 0.4 | 40K | 2 |
| 8 | 2.05 | 130K | 1 |
| 9 | 0.90 | 10K | 1 |
| 10 | 2.5 | 60K | 0 |
| 11 | 1.6 | 105K | 2 |
| 12 | 1.05 | 196K | 1 |
| 13 | 0.52 | 105K | 2 |
| 14 | 1.80 | 32K | 1 |
| 15 | 2.3 | 106K | 0 |
| 16 | 2.4 | 151K | 1 |
| 17 | 2.5 | 170K | 1 |
| 18 | 0.50 | 150K | 2 |
| 19 | 1.1 | 35K | 1 |
| 20 | 0.85 | 70K | 2 |

**1.1 Data processing**

To make the sigmoid function very effective, and also as it is suggested in the problem write up, the input data need to be normalized. Normalization also avoids the activity value from being dominated by components with bigger scale. It also avoids arithmetic/computational error.
I have normalized The LAC and SOW data element linearly as follows:

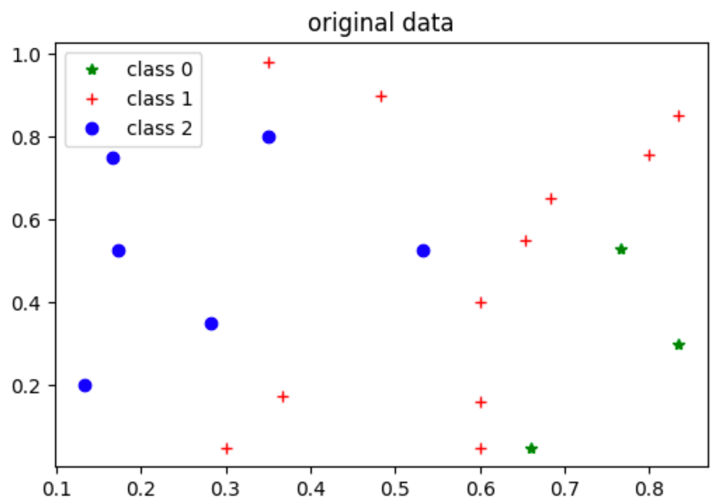
LAC = LAC/3.0 so that LAC will be in [0,1]
SOW = SOW/200000.0 so that SOW will be in [0,1]

Below is data after normalization:

| Data Item | LAC | SOW | TACA |
|---|---|---|---|
| 1 | 0.66 | 0.05 | 0 |
| 2 | 0.6 | 0.05 | 1 |
| 3 | 0.35 | 0.8 | 2 |
| 4 | 0.4833 | 0.9 | 1 |
| 5 | 0.6 | 0.4 | 1 |
| 6 | 0.6533 | 0.55 | 1 |
| 7 | 0.1333 | 0.2 | 2 |
| 8 | 0.6833 | 0.65 | 1 |
| 9 | 0.3 | 0.05 | 1 |

| | | | |
|---|---|---|---|
| 10 | 0.8333 | 0.3 | 0 |
| 11 | 0.5333 | 0.525 | 2 |
| 12 | 0.35 | 0.98 | 1 |
| 13 | 0.1733 | 0.525 | 2 |
| 14 | 0.6 | 0.16 | 1 |
| 15 | 0.7667 | 0.53 | 0 |
| 16 | 0.8 | 0.755 | 1 |
| 17 | 0.8333 | 0.85 | 1 |
| 18 | 0.1667 | 0.75 | 2 |
| 19 | 0.3667 | 0.175 | 1 |
| 20 | 0.2833 | 0.35 | 2 |

To see how the data looks like visually, I have plotted it and obtained the following graph.




As it can be seen from the graph, the classes are not linearly separable. Hence, we need multilayer neural net or higher order perceptron. As it will be discussed later, I have used multilayer neural net.

### 1.2 Training and Testing data: splitting the data set into training and validating sets.

Following the suggestion in the problem write up, the second 10 data items are used as training set and the first 10 is used for testing purpose.

## 2. Building Neural Network:

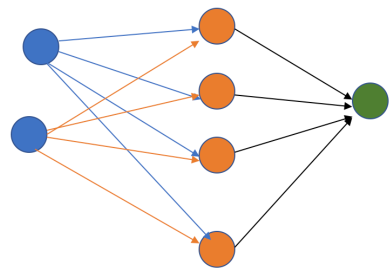
I tried the following two architectures.

### 2.1 Single hidden layer architecture:

The output node uses ramp activation function:

$$y = \ln(1 + e^x)$$

Whereas all hidden nodes use sigmoid activation function. The architecture of the network is depicted below.



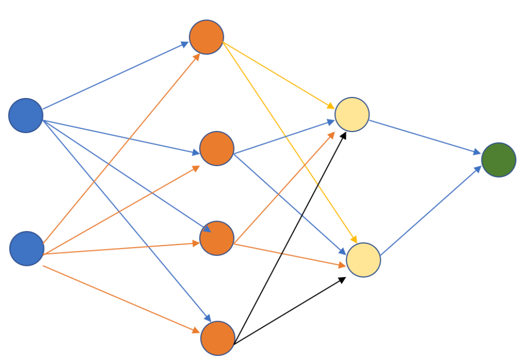
The code used to create this neural network is from line 261-270 in the script found in appendix. I have also printed it below:

```
ntk3 = None
ntk3 = Network()
# add input layer of 2 nodes
ntk3.add_layer(input_layer(2))
#add hidden layer of 3 nodes with default sigmoid act. func
ntk3.add_layer(layer(4,ntk3.layers[0]))
#add hidden layer of 2 nodes with sigmoid function
#ntk3.add_layer(layer(2,ntk3.layers[1]))
#add output layer of 1 node with ramp function
ntk3.add_layer(layer(1,ntk3.layers[1], act_func = 'ramp',intit_weight_range = 5))
```

### 2.2 Two hidden layers architecture:

Here also the output node uses ramp activation function while all hidden nodes use sigmoid function. The architecture is shown in the figure below. The code for creating the network is given below:

```
ntk3 = None
ntk3 = Network()
# add input layer of 2 nodes
ntk3.add_layer(input_layer(2))
#add hidden layer of 3 nodes with default sigmoid act. func
ntk3.add_layer(layer(4,ntk3.layers[0]))
#add hidden layer of 2 nodes with sigmoid function
ntk3.add_layer(layer(2,ntk3.layers[1]))
#add output layer of 1 node with ramp function
ntk3.add_layer(layer(1,ntk3.layers[2], act_func = 'ramp',intit_weight_range = 5))
```



## 3. Training:

### 3.1 Back propagation:

I have used forward/back propagation algorithm to train both architectures. In general, the weight update is given by:

$$\Delta w = -learningRate * sigma * input$$

for output layer, sigma is given as:

$$sigma = \frac{\partial E}{\partial e} * \frac{\partial e}{\partial y} * \frac{\partial y}{\partial A} = (y - desired) * \frac{e^A}{1 + e^A}$$

where:

$$y = 1 + e^A, \frac{\partial y}{\partial A} = \frac{\partial(1 + e^A)}{\partial A} = \frac{e^A}{1 + e^A} \text{ and } A \text{ is activity value}$$

For the rest of the nodes with sigmoid activation function, sigma would have the usual expression which is:

$$sigma = propagtederr * (y * (1 - y))$$

where:

$$y = \frac{1}{1 + e^{-A}}$$

## 4. Experiments and Results

### 4.1 Weight Initialization

Following the discussion in the text book section 8.2.1, the weight associated to hidden nodes with sigmoidal activation function is initialized based on uniform distribution in the range [-1,1]. Whereas, for the output node, as there is no saturation problem associated to the ramp function, I have used relatively bigger range [-5,5]. I have tried many other ranges. However, I obtained better performance with those two ranges.

### 4.2 Learning Rate

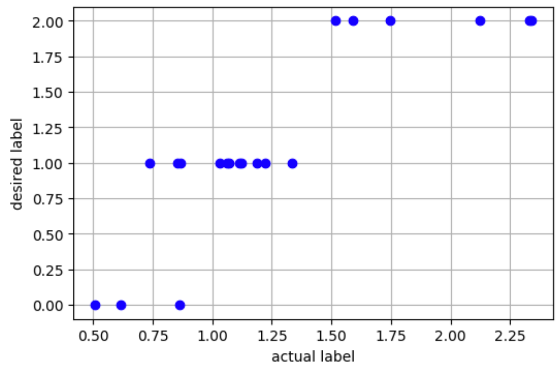
I have tried learning rates 0.01,0.1,1 and compared their results. I found that 0.1 provides better performance for the problem in this homework. As the number of iterations are fixed (set in the problem write up), there is no much flexibility in the choice of learning rate. For example, 0.01 demonstrated poorer performance as compared to 0.1 because it requires relatively higher number of iteration (bigger than 5000 in this case) to converge.

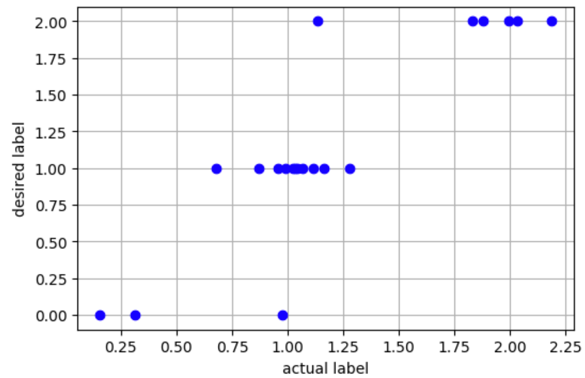
### 4.3 Output Threshold

In order to match the output from NN with the TACA values, I have used the following thresholding logic:

$$f(x) = \begin{cases} 0 & if\ x \leq 0.6 \\ 1 & if\ 0.6 < x \leq 1.5 \\ 2 & otherwise \end{cases}$$

I determined the threshold values through experimenting and identifying the values that give the best performance. Consider the following two figures. They are graph of desired class label versus actual label obtained from trained-NN. It can be seen that, the best way to match the two values in both graphs, resulting in minimum error, would be to follow the above thresholding logic. I have analyzed many of this kind of graph and found out that the above thresholding function works in almost all cases.




### 4.4 Experiment 1

First, I used the first neural network architecture and trained the network for 1000 cycles. As the weights are being initialized randomly, I have repeated the experiment many times, with different initial values for the weight, till I get the best result. The result I obtained is summarized as below:

```
 ***** initial weight *********
[ 0.28317035 -0.90001816 -0.3359804 ]
[-0.77819285 -0.09769227  0.96188553]
[-0.1922751   0.4835404  -0.17905833]
[-0.57813947 -0.88787653  0.38141335]
[-2.28992343 -3.43512777 -4.54951377  3.1172399   1.09673992]

 ********** weight after trained **********

[ 2.49061034 -4.43666746 -0.32880595]
[-0.26972754  0.36757136 -0.50548899]
[ 0.56695469 -0.71841775 -2.38299892]
[-1.98032174 -2.41351782  2.26561017]
[-3.58873245 -1.98926224 -3.74786352  3.54144521  1.07495875]
```

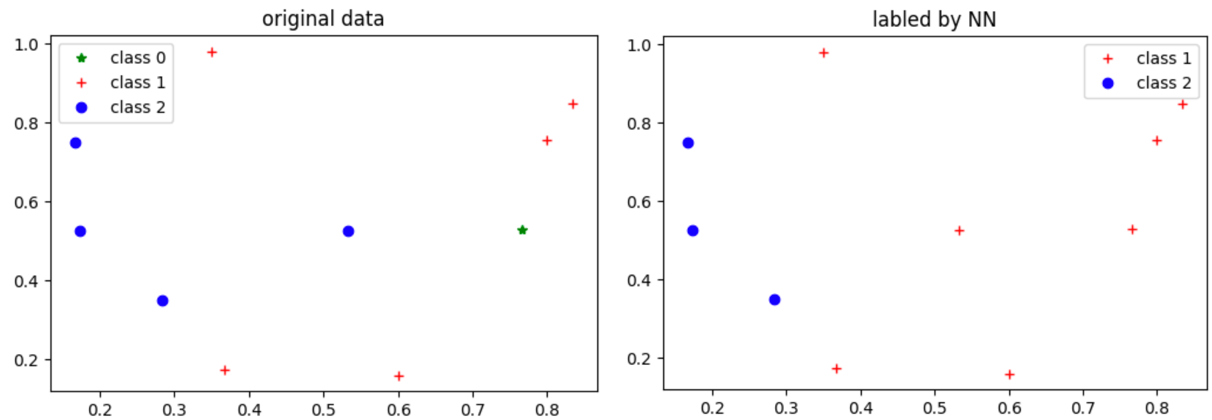
**Training Errors: all based on the training data**

The following summarizes the mean square error, error count and ROC values.

Means Square Error = 0.2
Error count = 2
ROC:

| | | |
|---|---|---|
| class 0 :<br>sensitivity = 0.0<br>specificity = 100.0<br>pos_pred_prob = 0.0<br>neg_pred_prob = 0.9 | class 1 :<br>sensitivity = 100.0<br>specificity = 60.0<br>pos_pred_prob = 0.7143<br>neg_pred_prob = 1.0 | class 2 :<br>sensitivity = 75.0<br>specificity = 100.0<br>pos_pred_prob = 1.0<br>neg_pred_prob = 0.8571 |

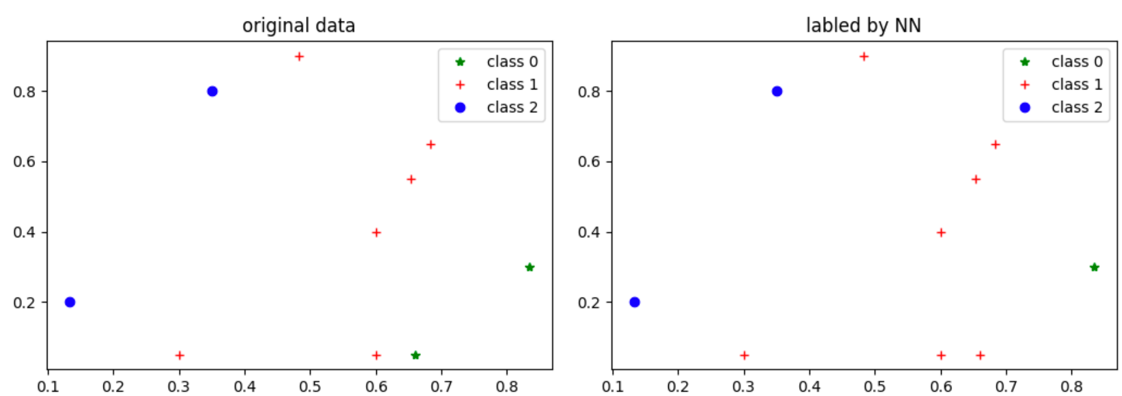
The ROC values agree with the figure below obtained by plotting the original training data (left) and NN labeled data (right).




**Validation error: Based on test Data:**
Performance measures are summarized as follows.
Means Square Error = 0.1
Error count = 1
ROC

| | | |
|---|---|---|
| class 0 :<br>sensitivity = 50.0<br>specificity = 100.0<br>pos_pred_prob = 1.0<br>neg_pred_prob = 0.8889 | class 1 :<br>sensitivity = 100.0<br>specificity = 75.0<br>pos_pred_prob = 0.8571<br>neg_pred_prob = 1.0 | class 2 :<br>sensitivity = 100.0<br>specificity = 100.0<br>pos_pred_prob = 1.0<br>neg_pred_prob = 1.0 |

Again, the ROC values agree with the figure below obtained by plotting the original training data (left) and NN labeled data (right).

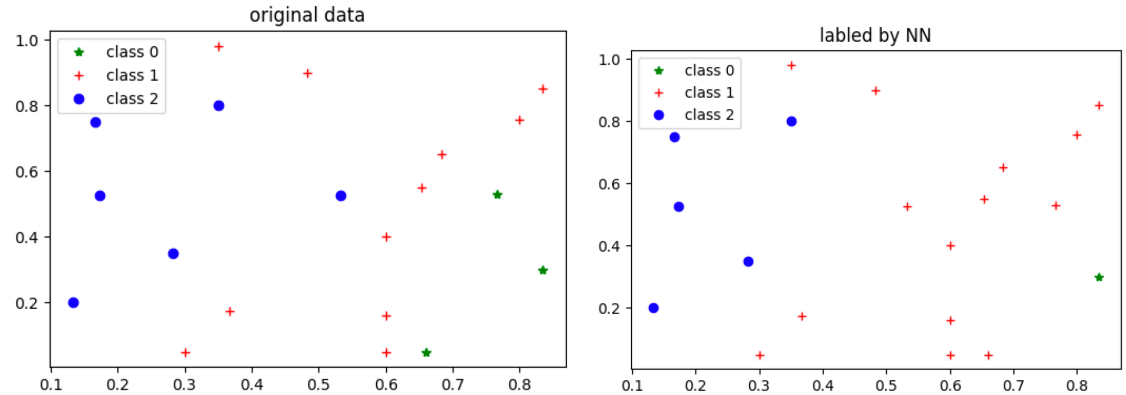



**Performance over all data set:** as total data size is small, I evaluated the performance over all data to get a better feeling.
The following summarizes the result.

Means Square Error = 0.15
Error count = 3
ROC

| | | |
|---|---|---|
| class 0 :<br>sensitivity = 33.3333<br>specificity = 100.0<br>pos_pred_prob = 1.0<br>neg_pred_prob = 0.8947 | class 1 :<br>sensitivity = 100.0<br>specificity = 66.6667<br>pos_pred_prob = 0.7857<br>neg_pred_prob = 1.000000 | class 2 :<br>sensitivity = 83.3333<br>specificity = 100.0<br>pos_pred_prob = 1.0<br>neg_pred_prob = 0.9333 |

One more time, the ROC values agree with the figure below obtained by plotting the original training data (left) and NN labeled data (right).




**4.5 Experiment 2**

I repeated the experiment 1 above keeping all set up the same except changing the number of iteration from 1000 to 5000.

The following summarizes what I obtained.

***** initial weight *********
[ 0.61286626  0.29909642  0.78337307]
[ 0.8247126   0.54906609  0.11827445]
[-0.48027083 -0.46755289 -0.89801093]
[ 0.84232074  0.19351867  0.04550302]
[ 3.87196147  0.21749259 -1.80057858 -4.2035016   2.36193701]

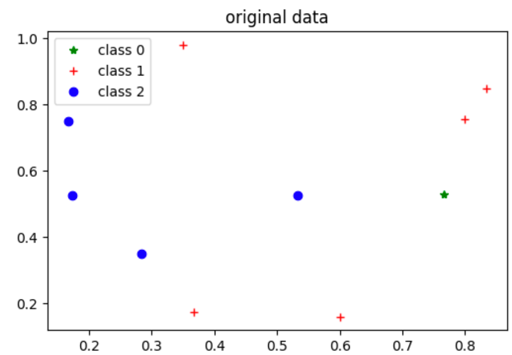
********** weight after trained **********

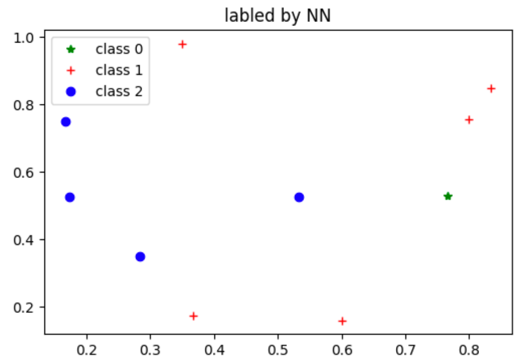
[ -9.69976732  11.49060696   0.12382331]
[-3.74319599  1.41868736  0.50398633]
[-1.5006801  -1.23933412 -1.46788763]
[ 3.43216945  3.06422841 -4.36860169]
[ 3.3372475  -3.23861897 -1.67067898 -4.27282916  1.39367543]

**Training Error:**

Means Square Error = 0.0
Error count = 0
ROC

| | | |
|---|---|---|
| class 0 :<br>sensitivity = 100.0<br>specificity = 100.0<br>pos_pred_prob = 1.0<br>neg_pred_prob = 1.0 | class 1 :<br>sensitivity = 100.0<br>specificity = 100.0<br>pos_pred_prob = 1.0<br>neg_pred_prob = 1.0 | class 2 :<br>sensitivity = 100.0<br>specificity = 100.0<br>pos_pred_prob = 1.0<br>neg_pred_prob = 1.0 |

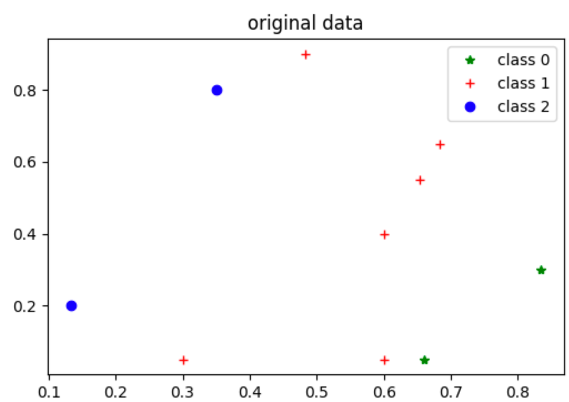


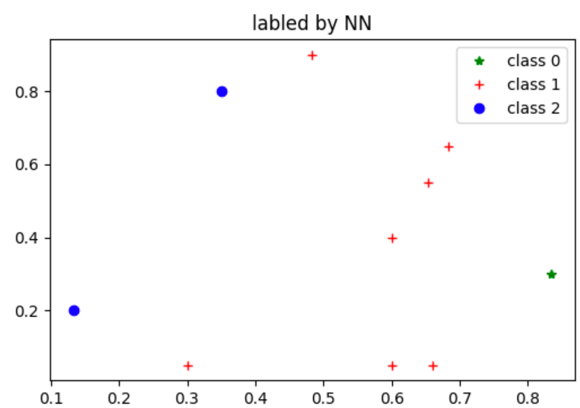




**Validation Error: Using test data**

Means Square Error = 0.1
Error count = 1
ROC

| | | |
|---|---|---|
| class 0 :<br>sensitivity = 50.0<br>specificity = 100.0<br>pos_pred_prob = 1.0<br>neg_pred_prob = 0.8889 | class 1 :<br>sensitivity = 100.0<br>specificity = 75.0<br>pos_pred_prob = 0.8571<br>neg_pred_prob = 1.0 | class 2 :<br>sensitivity = 100.0<br>specificity = 100.0<br>pos_pred_prob = 1.0<br>neg_pred_prob = 1.0 |

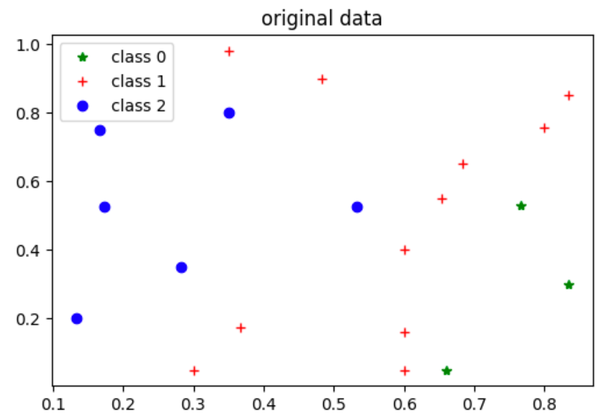


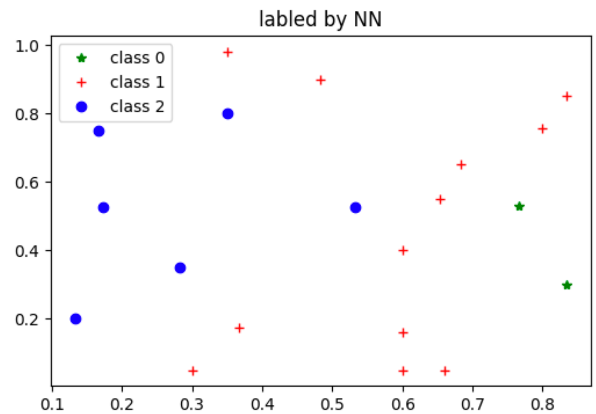




**Overall performance:**

Means Square Error = 0.05
Error count = 1
ROC

| | | |
|---|---|---|
| class 0 :<br>sensitivity = 66.6667<br>specificity = 100.0<br>pos_pred_prob = 1.0<br>neg_pred_prob = 0.9444 | class 1 :<br>sensitivity = 100.0<br>specificity = 88.8889<br>pos_pred_prob = 0.9167<br>neg_pred_prob = 1.0 | class 2 :<br>sensitivity = 100.0<br>specificity = 100.00<br>pos_pred_prob = 1.0<br>neg_pred_prob = 1.0 |



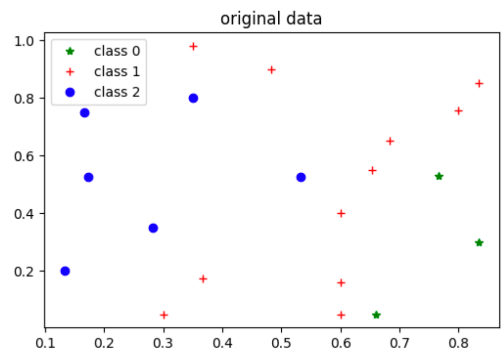

### 4.6 Two hidden Layers architecture

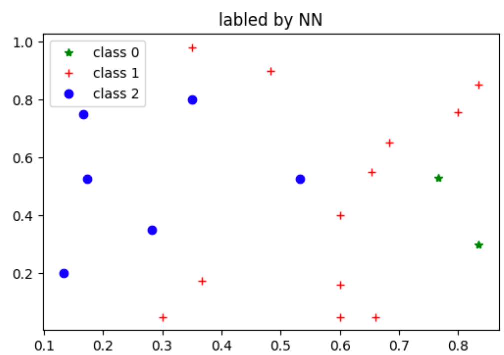
Just for comparison purpose, I have implemented and trained the two-hidden layer neural network and obtained the following details for 5000 iterations.

```
***** initial weight *********
[-0.85859693  0.79671172 -0.87850689]
[-0.03880478 -0.28134747  0.6885554 ]
[-0.53162314 -0.1020432  -0.07639773]
[ 0.81848537  0.79636018 -0.6383813 ]
[ 0.99501021  0.32602608 -0.01573957 -0.83359881 -0.41738459]
[-0.25846612 -0.36060483 -0.68204258 -0.30169257 -0.4613795 ]
[-3.01082331  2.32683441 -4.71940786]

********** weight after trained **********

[ -8.29856307  13.58069191  -2.2219932 ]
[-0.07031238 -2.16301196  0.35729337]
[-0.33535997 -1.52856674 -0.46098029]
[ 0.68671806  5.76750292 -4.08603645]
[-0.18698243 -0.63553597 -0.77231383 -1.3346117  -2.29395211]
[ 7.14234086  1.54222607  0.90843785 -7.80671798  0.39520353]
[-2.12637678  6.02498459 -3.93906436]
```

Means Square Error = 0.1
Error count = 1
ROC

| | | |
|---|---|---|
| class 0 :<br>sensitivity = 50.0<br>specificity = 100.0<br>pos_pred_prob = 1.0<br>neg_pred_prob = 0.8889 | class 1 :<br>sensitivity = 100.0<br>specificity = 75.0<br>pos_pred_prob = 0.8571<br>neg_pred_prob = 1.0 | class 2 :<br>sensitivity = 100.0<br>specificity = 100.0<br>pos_pred_prob = 1.0<br>neg_pred_prob = 1.0 |




**Conclusions:**
The two architectures, single hidden layer and two hidden layers, demonstrated the same performance for 5000 cycles training: i,e they have equal measures of performance . Therefore, following **Occam's razor rule**, I preferred to go with the first simpler architecture. Hence, the following summaries are based the result obtained from using single hidden layer NN.

Comparing the results for 1000 cycles and 5000 cycles, given everything the same, the training with 5000 iterations performed better than the 1000 iterations. The main reason for this is that, with 1000 iterations, the NN has not converged yet (under fit). This can be seen from training errors. The training error for 1000 iteration is error count =2 while that of 5000 cycles is 0 error count. This indicate that, The NN learned further more about the training data in the last 3000 iterations.

Learning rate plays a big role in NN training. I have used three different rates 0.01, 0.1 and 1 and found that 0.1 works well for the stated number of iteration. With 0.01, the weight change would be very small even after 5000 iterations and with 1, the weight updates will be too big. Hence, in both cases, I obtained poor result as compared to that of 0.1.

The training data itself has an issue of fairly representing the classes. For example, we have only 3 data points from class 0. Furthermore, the selected training data has only one example from the class 0 and hence it would be 'difficult' for the NN to learn the class. That is why we are getting bigger classification errors associated to this class during validation. Getting good data set for training purpose representing all the classes fairly is essential to get best performance (minimum prediction error).

Moreover, to avoid computational error (register overflow) as well as to avoid one data element dominating the other, it is very important that the data is normalized (usually normalized to 0 mean and 1 standard deviation). For example, if we use SOW values as is, clearly, we would be getting incorrect result. The activity value would totally be determined by SOW and the LAC value would not have effect. On other hand, LAC also needs to be normalized so that it can nicely be approximated by sigmoidal activation functions. In this homework, I have used linear scale normalization to normalize both SOW and LAC as the number of data elements are very small and I have obtained reasonable performance.

Finally, initial weights can be selected randomly. However, the ranges we are using for random sampling need to be defined carefully. For example, for sigmoid activation function, if we select big initial weights, it will saturate the sigmoidal function and the derivative value will be very small. This will result in very small weight update when back propagation algorithm is used and hence requires longer time to converge. On other hand, if the weights are too small, again the error propagation will be hindered (attenuated) and hence it also results in minim weight update. Therefore, we need to choose the range which is not too big and not too small. I have experimented with different weight ranges in this homework and [-1,1] for hidden layer and [-5,5] for output layer are found to work very well.

5. **Appendix:**

The Python code implementing NN. I have used spyder editor.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sat Mar 10 01:23:13 2018

@author: yasfaw
"""
import numpy as np
from matplotlib import pyplot as mp
from random import shuffle
#class for creating neural nodes
class node:
    def __init__(self, inbound_nodes,init_weights =
[],bias=0,intit_weight_range=1,act_func ='sigmoid'):
        #if initial value is not given, initialize randomly
        if init_weights == []:
            self.weights = np.random.uniform(-1*intit_weight_range,
intit_weight_range
                                                   ,len(inbound_nodes) + 1)
        else:
          self.weights = np.append(np.array(init_weights),bias)
        self.bias = bias
        self.activity = 0
        self.activation = 0
        self.sigma = 0
        #kind of activation function used
        self.act_func = act_func
        # a list variable holding inbound nodes from previous layer
connected to
        # this node
        self.inbound_nodes = inbound_nodes
        #variable for holding derivative of activation function
        self.derivative = 0
        #a list variable for holding out bound nodes in the next layer
        #connected to this node
        self.outbound_nodes = []
        #for all inbound nodes, add current node as their out bound
node
        for n in self.inbound_nodes:
            n.outbound_nodes.append(self)
    def calc_activity(self):
        ex_input = []
        #get the activation value of all inbound nodes
        #these are inout for current node
        for v in self.inbound_nodes:
            ex_input.append(v.feedfrwrd())
        #append 1 to create extended input corresponding to bias
```

```
        ex_input = np.append(ex_input,1)
        #caclulate the activity value
        self.activity = np.dot(self.weights,ex_input)
    def calc_activation(self):
        # calculate the activation value based on type of activation
function
        if self.act_func == 'sigmoid':
            self.activation = 1/(1 + np.exp(-1*self.activity))
        if self.act_func == 'ramp':
            self.activation = np.log( 1 + np.exp(self.activity))
    def derivatives(self):
        #compute derivatives based on type of activation function
        if self.act_func == 'sigmoid':
            return self.activation*(1 - self.activation)
        if self.act_func == 'ramp':
            num = np.exp(self.activity)
            return  num/(1 + num )
    # function for carrying out feed forward computation
    def feedfrwrd(self):
        self.calc_activity()
        self.calc_activation()
        self.derivative = self.derivatives()
        return self.activation
# Class for implementing input nodes as they are different from
#the nodes in other layer
class Input:
    def __init__(self):
        self.activation = 0
        self.outbound_nodes =[]
    def feedfrwrd(self):
        return self.activation
# Class that implements a layer in a network
class layer:
    def __init__(self, number_nodes, inbound_layer,weight_matrix =[],
act_func ='sigmoid'
                 , intit_weight_range :"range of uniform dist"=1):
        self.nodes = []
        self.number_nodes = number_nodes
        self.inbound_layer = inbound_layer
        #create nodes in the layer
        if(number_nodes == 1):

self.nodes.append(node(self.inbound_layer.nodes,weight_matrix
                                  ,intit_weight_range =
intit_weight_range, act_func = act_func))
        else:
            indx = 0
            for i in range(self.number_nodes):
                if weight_matrix == []:
                    wt = []
```

```
                else:
                    wt = weight_matrix[indx]

self.nodes.append(node(self.inbound_layer.nodes,wt,act_func =
act_func))
                indx+=1
#Class for implemnting input layer. Input layer is different from
other layers
class input_layer:
    def __init__(self,number_nodes):
        self.nodes = []
        self.number_nodes = number_nodes
        for i in range(number_nodes):
            self.nodes.append(Input())
    # a function that sets input layer nodes activation value to input
value
    def addinput(self,input_x):
        assert len(input_x) == self.number_nodes,"Dimension don't
match"
        for i in range(self.number_nodes):
            self.nodes[i].activation = input_x[i]
#class implemnting network
class Network:
    def __init__(self):
        #list variable holding layers in network
        self.layers = []
        #a variable for holding output layer
        self.output_layer = None
    #function for adding fully connected layer to network
    def add_layer(self,layer):
        self.layers.append(layer)
        #the last layer added is the output layer
        self.output_layer = layer
    # function executing feed forward
    def feed_forward(self, inputx):
        self.layers[0].addinput(inputx)
        for n in self.output_layer.nodes:
            n.feedfrwrd()
    # function executing back propagation
    def back_propagation(self, y):
        #output layer needs to be treated differently
        for n in self.output_layer.nodes:
            n.sigma = (n.activation -y)*n.derivative
        # back propagation for the rest of hidden layers
        for l in self.layers[-2:0:-1]:
            idx =0
            for n in l.nodes:
                for ob in n.outbound_nodes:
                    n.sigma = 0
                    n.sigma += ob.sigma*ob.weights[idx] *n.derivative
```

```
                idx +=1
    def weight_update(self, rate):
        for l in self.layers[1:]:
            for n in l.nodes:
                indx =0
                for ib in n.inbound_nodes:
                    n.weights[indx] -= rate*n.sigma*ib.activation
                    indx +=1
                #routine that update bias
                n.weights[-1] -= rate * n.sigma
    def nn_train(self, inputx, outputy,rate):
        #self.layers[0].addinput(inputx)
        self.feed_forward(inputx)
        self.back_propagation(outputy)
        self.weight_update(rate)
    def nn_output(self,inputx):
        #self.layers[0].addinput(inputx)
        self.feed_forward(inputx)
        return self.output_layer.nodes[0].activation
# function that computes mean square error based on desired and actual
#output
def mean_square_error(desired, actual):
    E = 0
    Z = zip(desired,actual)
    for (d,a) in Z:
        E += 1/(len(desired)) *(d - a)**2
    return E
#function that compute error in terms of number of misclassified data
points
def error_count(desired, actual):
    count =0
    Z= zip(desired,actual)
    for (d,a) in Z:
        if (d != a):
            count += 1
    return count
#function that compute the ROC values for a given class based on
# desired and actual output
def ROC (desired, actual, class_label):
    Z = zip(desired, actual)
    #True negative and positive, false negative and positive
    TN,FN,TP,FP = 0,0,0,0
    for (d,a) in Z:
        if(d == class_label):
            if (a == class_label):
                TP +=1
            else:
                FN += 1
        else:
            if (a == class_label):
```

```
                FP +=1
            else:
                TN += 1
    #print(TN,FN,TP,FP)

    sensitivity = TP*100.0/(TP +FN)
    specificity = TN*100.0/(TN + FP)
    neg_pred_prob  = TN/(TN + FN)
    if(TP + FP == 0):
        pos_pred_prob = 0
    else:
        pos_pred_prob = TP/(TP + FP)
    return  sensitivity,specificity , pos_pred_prob,neg_pred_prob
# function for plotting data points.
def visual_display(data, output):
    data_0 = [x for x in data if x[2] == 0]
    data_1 = [x for x in data if x[2]  == 1]
    data_2 = [x for x in data if x[2] == 2]
    np_data_0 = np.array(data_0).T
    np_data_1 = np.array(data_1).T
    np_data_2 = np.array(data_2).T
    #ploting NN output
    error = [[x[0],x[1]] for [x,y] in zip(data,output) if x[2] !=
y[2]]
    ot_0 = [x for x in output if x[2] == 0]
    ot_1 = [x for x in output if x[2]  == 1]
    ot_2 = [x for x in output if x[2] == 2]
    np_ot_0 = np.array(ot_0).T
    np_ot_1 = np.array(ot_1).T
    np_ot_2 = np.array(ot_2).T

    print('\n *************** ploting ******************* \n')
    error = np.array(error).T
    try:
        #mp.subplot(3,1,1)
        #ploting original data
        mp.plot(np_data_0[0,:], np_data_0[1,:],'g*',label = 'class 0')
        mp.legend()
        #mp.show()
        mp.plot( np_data_1[0,:], np_data_1[1,:],'r+',label = 'class
1')
        mp.legend()
        mp.plot(np_data_2[0,:], np_data_2[1,:],'bo', label = 'class
2')
        mp.legend()
        mp.title('original data')
        mp.show()
        #ploting output of NN
        #mp.subplot(3,1,2)
        if(ot_0 != []):
```

```
            mp.plot (np_ot_0[0,:], np_ot_0[1,:],'g*',label = 'class 
0')
            mp.legend()
        mp.plot (np_ot_1[0,:], np_ot_1[1,:],'r+', label = 'class 1')
        mp.legend()
        mp.plot(np_ot_2[0,:], np_ot_2[1,:],'bo',label = 'class 2')
        mp.legend()
        mp.title('labled by NN')
        mp.show()
        #mp.subplot(3,1,3)
        #ploting mis-labeled data
        mp.plot(error[0,:], error[1,:] , 'ro')
        #mp.set_autoscale_on(False)
        mp.ylim(0.0,1.0)
        mp.xlim(0.1,1.0)
        mp.title ('Error')
        mp.show()
    except IndexError:
        if(ot_0 == []):
            print("class 0 is blank")
if __name__ == '__main__':

    #training data
    training_data = [[1.6, 105000, 2],[1.05, 196000, 1],[0.52, 105000, 
2],\
                      [1.80, 32000, 1],[2.3, 106000, 0],[2.4, 151000, 
1],\
                      [2.5, 170000, 1],[0.50, 150000, 2],[1.1, 35000, 
1],[0.85, 70000, 2]]
    testing_data = [[1.98, 10000, 0],[1.80, 10000, 1],[1.05, 160000, 
2],\
                    [1.45, 180000, 1],[1.8, 80000, 1],[1.96, 110000, 
1],[0.4, 40000, 2],\
                    [2.05, 130000, 1],[0.90, 10000, 1],[2.5, 60000, 
0]]
    #Data Processing -- Normalizing LAC and SOW
    training_data = [[x[0]/3.0,x[1]/200000,x[2]] for x in 
training_data]
    y_desired_training =[y[2] for y in training_data]
    testing_data = [[x[0]/3.0,x[1]/200000,x[2]] for x in testing_data]
    y_desired_testing = [y[2] for y in testing_data]
    #Builiding the network
    ntk3 = None
    ntk3 = Network()
    # add input layer of 2 nodes
    ntk3.add_layer(input_layer(2))
    #add hidden layer of 4 nodes with default sigmoid act. func
    ntk3.add_layer(layer(4,ntk3.layers[0]))
    #add second hidden layer of 2 nodes with sigmoid function
    #ntk3.add_layer(layer(2,ntk3.layers[1]))
```

```
    #add output layer of 1 node with ramp function
    ntk3.add_layer(layer(1,ntk3.layers[1], act_func = 
'ramp',intit_weight_range = 5))

    # training the network
    print('**** training ......')
    #learning rate
    lrate =0.1
    # priniting initial weight set randomly
    print('***** initial weight *********')
    for l in ntk3.layers[1:]:
        for n in l.nodes:
            print(n.weights)
    #training many number of epochs
    for itr in range(5000):
        trn = training_data[:]
        shuffle(trn)
        for data in trn:
            ntk3.nn_train(data[:2], data[2],lrate)
    #weights after training
    print('\n ********** weight after trained ********** \n')
    for l in ntk3.layers[1:]:
            for n in l.nodes:
                print(n.weights)
    #testing the network
    print('\n **** testing based on training data**** \n')
    #get the actual label of training data labeled by NN.
    output_training = []
    for inp in training_data:
        output_training.append([inp,ntk3.nn_output(inp[:2])])
    print(output_training)
    #Thresholding the output
    output_training = [[x[0],x[1],0 if y <=0.6 else 1 if 0.6 <y <= 1.5 
else 2] for [x,y] in output_training]
    y_output_training = [y[2] for y in output_training]
    # training error
    ms_err = mean_square_error (y_desired_training,y_output_training)
    err_count = error_count(y_desired_training,y_output_training)
    print("\nMeans Square Error = %f " % ms_err)
    print("Error count = %d \n" % err_count)
    # computing ROC for each class
    for c in [0,1,2]:
        s,sp,npr,ppr = ROC(y_desired_training,y_output_training,c)
        print("\n class %d : \n sensitivity = %f  \n specificity = %f 
\n pos_pred_prob = %f \n neg_pred_prob = %f"
          % (c,s,sp,npr,ppr))
    visual_display(training_data,output_training)
    print('\n **** testing based on test data**** \n')
    #get the actual label of testing data labeled by NN.
    output_testing = []
```

```
    for inp in testing_data:
        output_testing.append([inp,ntk3.nn_output(inp[:2])])
    print(output_testing)
    #Thresholding the output
    output_testing = [[x[0],x[1],0 if y <=0.6 else 1 if 0.6 <y <= 1.5
else 2] for [x,y] in output_testing]
    y_output_testing = [y[2] for y in output_testing]
    #testing error
    ms_err = mean_square_error (y_desired_testing,y_output_testing)
    err_count = error_count(y_desired_testing,y_output_testing)
    print("\nMeans Square Error = %f " % ms_err)
    print("Error count = %d \n" % err_count)
    # computing ROC for each class
    for c in [0,1,2]:
        s,sp,npr,ppr = ROC(y_desired_testing,y_output_testing,c)
        print("\n class %d : \n sensitivity = %f  \n specificity = %f
\n pos_pred_prob = %f \n neg_pred_prob = %f"
           % (c,s,sp,npr,ppr))
    visual_display(testing_data,output_testing)


    #Over all performance and visual display
    print('\n*************** over all performance **************\n')
    data_all = training_data
    data_all.extend(testing_data)
    y_desired_all = [y[2] for y in data_all]
    output_all = output_training
    output_all.extend(output_testing)
    y_output_all = [y[2] for y in output_all]

    ms_err = mean_square_error (y_desired_all,y_output_all)
    err_count = error_count(y_desired_all,y_output_all)
    print("\nMeans Square Error = %f " % ms_err)
    print("Error count = %d \n" % err_count)
    # computing ROC for each class
    for c in [0,1,2]:
        s,sp,npr,ppr = ROC(y_desired_all,y_output_all,c)
        print("\n class %d : \n sensitivity = %f  \n specificity = %f
\n pos_pred_prob = %f \n neg_pred_prob = %f"
           % (c,s,sp,npr,ppr))
    visual_display(data_all,output_all)
```