Neural Network and Fuzzy Logic

Assignment-2

Name: Aman Raj Singh
ID No.: 2019B1A31483H

```
from google.colab import drive
drive.mount("/content/gdrive")
```

```
Mounted at /content/gdrive
```

```
%cd /content/gdrive/My Drive/Assignment 2/
```

```
/content/gdrive/My Drive/Assignment 2
```

Question 1

Implement non-linear perceptron algorithm for the classification using Online Learning (Hebbian learning) algorithm. The dataset (data55.xlsx) contains 19 features and the last column is the output (class label). You can use hold-out cross-validation (70, 10, and 20%) for the selection of training, validation and test instances. Evaluate accuracy, sensitivity and specificity measures for the evaluation of test instances (Packages such as Scikitlearn, keras, tensorflow, pytorch etc. are not allowed).

```
import io
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import random
import math
from sklearn.model_selection import train_test_split
```

```
sheet1 = pd.read_excel('data55.xlsx')
data = sheet1.values
x = data[0: , : 19]
x = (x - np.min(x,axis=0))/(np.max(x,axis=0)- np.min(x,axis=0))
yd = data[0: , 19]
```

```
y = []
for i in range(len(yd)):
  if yd[i]==0.:
    y.append(-1.0)
```

```
    else:
      y.append(1.0)

x_train, x_test, y_train, y_test = train_test_split(x,y, train_size=0.7)
x_valid, x_test, y_valid, y_test = train_test_split(x_test,y_test, test_size=0.33)


def perceptron(X,Y,alpha):
  mean = np.zeros(19)
  cov = np.diag(np.ones(19))
  w = np.random.multivariate_normal(mean, cov)
  b = 0
  m = len(X)
  for i in range(500):
    for j in range(m):
      a = np.dot(X[j],w) + b
      hyp = sigmoid(a)
      cl = threshold(hyp)
      if cl != Y[j]:
        w = w + alpha*(np.dot(Y[j],X[j]))
        b = b + alpha*(Y[j])
      else:
        w = w
        b = b
  return w , b


def sigmoid(z):
    z = z.astype(float)
    z_output =1/(1 + np.exp(-z))
    return z_output


def threshold(z):
  r = -1.0
  if z >= 0.5:
    r = 1.0
  return r


def confmat(y_pred,y_ts):
  a, b, c, d = 0 , 0 , 0 , 0
  for i in range(len(y_ts)):
    if y_ts[i] == -1. :
      if y_pred[i] == -1. :
        a = a + 1
      if y_pred[i] == 1. :
        b = b + 1
    if y_ts[i] == 1. :
      if y_pred[i] == -1. :
        c = c + 1
      if y_pred[i] == 1. :
        d = d + 1
  return a, b, c, d


def prediction(x,y,weight_vec,b1):
```

```
  p_outputs = []
  for l in range(len(x)):
    Z = np.dot(x[l],weight_vec)+b1
    pred_op = sigmoid(Z)
    p_outputs.append(threshold(pred_op))
  a, b, c, d = confmat(p_outputs,y)
  acc = (a+d)/(a+b+c+d)
  sens = (a)/(a+b)
  spec = (d)/(d+c)
  return acc,sens,spec
```

```
#Grid search
avals = np.logspace(-3,-1,num=100)
accuracy = []
for a in avals:
  wei, bi = perceptron(x_train,y_train,a)
  ac, sp, sen = prediction(x_valid,y_valid,wei,bi)
  accuracy.append(ac)
acc_opt = max(accuracy)
alloptind = [i for i, j in enumerate(accuracy) if j == acc_opt]
opt_ind = max(alloptind)
a_opt = avals[opt_ind]
#a_opt = avals[mse_opt]
print(accuracy)
print(acc_opt)
print(a_opt)
```

```
[0.6190476190476191, 0.6190476190476191, 0.6190476190476191, 0.5952380952380952, 0.64
0.8095238095238095
0.005590810182512223
```

```
w2,b2 = perceptron(x_train,y_train,a_opt)
a,se,sp = prediction(x_test,y_test,w2,b2)
print("Accuracy:", a*100)
print("Sensitivity:", se*100)
print("Specificity:", sp*100)
```

```
Accuracy: 90.47619047619048
Sensitivity: 90.0
Specificity: 90.9090909090909
```

Question 2

Implement kernel perceptron algorithm for the classification task. The dataset (data55.xlsx) contains 19 features and the last column is the output (class label). You can use hold-out cross-validation (70, 10, and 20%) for the selection of training, validation and test instances. Evaluate accuracy, sensitivity and specificity measures for the evaluation of test instances. Evaluate the classification performance separately using linear, RBF, and polynomial kernels (Packages such

```
import io
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import random
import math
from sklearn.model_selection import train_test_split


sheet1 = pd.read_excel('data55.xlsx')
data = sheet1.values
x = data[0: , : 19]
x = (x - np.min(x,axis=0))/(np.max(x,axis=0)- np.min(x,axis=0))
yd = data[0: , 19]


y = []
for i in range(len(yd)):
  if yd[i]==0.:
    y.append(-1.0)
  else:
    y.append(1.0)


x_train, x_test, y_train, y_test = train_test_split(x,y, train_size=0.7)
x_valid, x_test, y_valid, y_test = train_test_split(x_test,y_test, test_size=0.33)


def sigmoid(z):
    #z = np.dot(X, weight)
    z = z.astype(float)
    z_output =1/(1 + np.exp(-z))
    return z_output


def confmat(y_pred,y_ts):
  a, b, c, d = 0 , 0 , 0 , 0
  for i in range(len(y_ts)):
    if y_ts[i] == -1. :
      if y_pred[i] == -1. :
        a = a + 1
      if y_pred[i] == 1. :
        b = b + 1
    if y_ts[i] == 1. :
      if y_pred[i] == -1. :
        c = c + 1
      if y_pred[i] == 1. :
        d = d + 1
  return a, b, c, d


def threshold(z):
  r = 0
  if z >= 0.5:
    r = 1.0
```

```
    else:
      r = -1.0
    return r


  def lin_predictions(X,Y,Sig):
    h = []
    pred = []
    for j in range(len(X)):
      s = np.sum(np.dot(Sig,np.dot(Y,linear_kernel(X,X[j]))))
      h.append(s)
    hypo = np.array(h)
    for u in range(len(h)):
      if hypo[u] < hypo.mean():
        pred.append(-1.0)
      else:
        pred.append(1.0)
    a, b, c, d = confmat(pred,Y)
    acc = (a+d)/(a+b+c+d)
    sens = (a)/(a+b)
    spec = (d)/(d+c)
    return acc,sens,spec



  def poly_predictions(X,Y,Sig,p):
    h = []
    pred = []
    for j in range(len(X)):
      s = np.sum(np.dot(Sig,np.dot(Y,polynomial_kernel(X,X[j],p))))
      h.append(s)
    hypo = np.array(h)
    for u in range(len(h)):
      if hypo[u] < hypo.mean():
        pred.append(-1.0)
      else:
        pred.append(1.0)
    a, b, c, d = confmat(pred,Y)
    acc = (a+d)/(a+b+c+d)
    sens = (a)/(a+b)
    spec = (d)/(d+c)
    return acc,sens,spec


  def RBF_predictions(X,Y,Sig,gamma):
     h = []
     pred = []
     for j in range(len(X)):
       s = 0
       for l in range(len(X)):
         s = s + Sig[l]*Y[l]*RBF_Kernel(X[l],X[j],gamma)
      #s = np.sum(np.dot(Sig,np.dot(Y,linear_kernel(X,X[j]))))
       h.append(s)
     hypo = np.array(h)
     for u in range(len(h)):
```

```
     if hypo[u] < hypo.mean():
       pred.append(-1.0)
     else:
       pred.append(1.0)
   a, b, c, d = confmat(pred,Y)
   acc = (a+d)/(a+b+c+d)
   sens = (a)/(a+b)
   spec = (d)/(d+c)
   return acc,sens,spec
```

LINEAR KERNEL PERCEPTRON

```
def linear_kernel(x1, x2):
    return np.dot(np.array(x1), np.array(x2))
```

```
def linear_kernel_perceptron(X,Y,iter):
  sig = []
  for y in range(len(X)):
    sig.append(0)
  for i in range(iter):
    for j in range(len(X)):
      s = 0
      for l in range(len(X)):
        s = s + sig[l]*Y[l]*linear_kernel(X[l],X[j])
      #a = np.dot(sig,np.dot(Y,linear_kernel(X,X[j]))) # a =
      #print(a)
      hypo = threshold(s)
      if hypo != Y[j]:
        sig[j] = sig[j] + 1
  return sig
```

```
#gridsearch
accuracy1 = []
list1 = np.linspace(100,500,num=9)
for num in list1:
  #list2.append(num)
  sig1  = linear_kernel_perceptron(x_train,y_train,int(num))
  acc,sp,se = lin_predictions(x_valid,y_valid,sig1)
  accuracy1.append(acc)
acc_max = max(accuracy1)
opt_index = accuracy1.index(acc_max)
p_opt = int(list1[opt_index])
print(p_opt)
```

```
100
```

```
sig_linear = linear_kernel_perceptron(x_train,y_train,p_opt)
accu1,sensi1,speci1 = lin_predictions(x_test,y_test,sig_linear)
print("Accuracy:", accu1*100)
print("Sensitivity:", sensi1*100)
print("Specificity:", speci1*100)
```

```
Accuracy: 85.71428571428571
Sensitivity: 90.0
Specificity: 81.81818181818183
```

POLYNOMIAL KERNEL PERCEPTRON

```
def polynomial_kernel(x, y, p): # Second degree polynomial is taken here
    return (1 + np.dot(x,y)) ** p
```

```
def polynomial_kernel_perceptron(X,Y,p):
  sig = []
  for y in range(len(X)):
    sig.append(0)
  for i in range(100):
    for j in range(len(X)):
      s = 0
      for l in range(len(X)):
        s = s + sig[l]*Y[l]*polynomial_kernel(X[l],X[j],p)
      hypo = threshold(s)
      if hypo != Y[j]:
        sig[j] = sig[j] + 1
  return sig
```

```
#gridsearch
accuracy = []
list2 = np.linspace(2,10,num=9)
for num in list2:
  #list2.append(num)
  sig2  = polynomial_kernel_perceptron(x_train,y_train,int(num))
  acc,sp,se = poly_predictions(x_valid,y_valid,sig2,int(num))
  accuracy.append(acc)
acc_max = max(accuracy)
alloptind = [i for i, j in enumerate(accuracy) if j == acc_max]
opt_index = max(alloptind)
p_opt = int(list2[opt_index])
print(p_opt)
```

```
2
```

```
sig_poly = polynomial_kernel_perceptron(x_train,y_train,3)
accu,sensi,speci = poly_predictions(x_test,y_test,sig_poly,3)
print("Accuracy:", accu*100)
print("Sensitivity:", sensi*100)
print("Specificity:", speci*100)
```

```
Accuracy: 80.95238095238095
Sensitivity: 90.0
Specificity: 72.72727272727273
```

RBF KERNEL PERCEPTRON

```
variance = np.var(x)
# number of features = 19
gamma = (1/19*variance)



def RBF_Kernel(x1,x2,gam):
  norm_vec = np.subtract(x1,x2)
  return np.exp(-gamma*np.sum((x1-x2)**2))


def RBF_kernel_perceptron(X,Y,gam,iter):
  sig = []
  for y in range(len(X)):
    sig.append(0)
  for i in range(iter):
    for j in range(len(X)):
      s = 0
      for l in range(len(X)):
        s = s + sig[l]*Y[l]*RBF_Kernel(X[l],X[j],gam)
      hypo = threshold(s)
      if hypo != Y[j]:
        sig[j] = sig[j] + 1
  return sig


#gridsearch
accuracy3 = []
list3 = np.linspace(100,500,num=5)
for num in list3:
  #list2.append(num)
  sig3  = RBF_kernel_perceptron(x_train,y_train,gamma/2,int(num))
  acc3,sp3,se3 = RBF_predictions(x_valid,y_valid,sig3,gamma/2)
  accuracy3.append(acc3)
acc_max = max(accuracy3)
opt_index = accuracy3.index(acc_max)
p_opt = int(list3[opt_index])
print(p_opt)
```

```
200
```

```
sig_rbf = RBF_kernel_perceptron(x_train,y_train,gamma/2,500)
accu,sensi,speci = RBF_predictions(x_test,y_test,sig_rbf,gamma/2)
print("Accuracy:", accu*100)
print("Sensitivity:", sensi*100)
print("Specificity:", speci*100)
```

```
Accuracy: 93.46511627906976
Sensitivity: 91.54545454545453
Specificity: 95.66666666666666
```

Question 3

The dataset (data5.xlsx) contains 7 features and the last column is the output (class labels). Design a multilayer perceptron based neural network (two hidden layers) for the classification. You can use both holdout (70, 10, and 20%) and 5-fold cross-validation approaches for evaluating the performance of the classifier (individual accuracy and overall accuracy). You can select the number of hidden neurons of each hidden layer and other MLP parameters using grid-search method. (Packages such as Scikitlearn, keras, tensorflow, pytorch etc. are not allowed).

```
import pandas as pd
import math
import numpy as np
import random
from sklearn.model_selection import train_test_split
```

```
sheet1 = pd.read_excel('data5.xlsx')
data = sheet1.values
np.random.shuffle(data)
x = data[0: , : 7]
x = (x - np.min(x,axis=0))/(np.max(x,axis=0)- np.min(x,axis=0))
y = data[0: , 7]
```

```
def sigmoid(z):
  return 1.0 / ( 1.0 + np.exp(-z))
```

```
def sigmoid_derivative(z):
  derivative = z*(1-z)
  return derivative
```

```
def loss_function(y,y_prime):
  loss = 0
  for i in range(y):
    loss = loss + (y[i]-y_prime[i])**2
  J = loss/(2*len(x))
  return J
```

```
def model(X,hidden_nodes,output_dim=3):

    input_dim = X.shape[1]
    W1 = np.random.randn(input_dim, hidden_nodes) / np.sqrt(input_dim)
    b1 = np.zeros((1, hidden_nodes))
    W2 = np.random.randn(hidden_nodes, hidden_nodes) / np.sqrt(hidden_nodes)
    b2 = np.zeros((1, hidden_nodes))
    W3 = np.random.randn(hidden_nodes, output_dim) / np.sqrt(hidden_nodes)
    b3 = np.zeros((1, output_dim))

    return W1,b1,W2,b2,W3,b3
```

```
def feed_forward(x,W1,b1,W2,b2,W3,b3):
  z1 = np.dot(x,W1) + b1
  a1 = sigmoid(z1)
  z2 = np.dot(a1,W2) + b2
  a2 = sigmoid(z2)
  z3 = np.dot(a2,W3) + b3
  a3 = sigmoid(z3)
  return z1,a1,z2,a2,z3,a3


def back_prop(x,W1,W2,W3,a1,a2,a3):
    delta3 = a3
    dW3 = (a2.T).dot(delta3)
    db3 = np.sum(delta3, axis=0, keepdims=True)
    delta2 = delta3.dot(W3.T) * sigmoid_derivative(a2)
    dW2 = np.dot(a1.T, delta2)
    db2 = np.sum(delta2, axis=0)
    delta1 = delta2.dot(W2.T) * sigmoid_derivative(a1)
    dW1 = np.dot(x.T, delta1)
    db1 = np.sum(delta1, axis=0)
    return dW1, dW2, dW3, db1, db2, db3


def mlp_fn(x,y,hidden_node,alpha):
  W1,b1,W2,b2,W3,b3 = model(x,hidden_node,3)
  for i in range(500):
    z1,a1,z2,a2,z3,a3 = feed_forward(x,W1,b1,W2,b2,W3,b3)
    dW1, dW2, dW3, db1, db2, db3 = back_prop(x,W1,W2,W3,a1,a2,a3)
    W1 -= alpha * dW1
    b1 -= alpha * db1
    W2 -= alpha * dW2
    b2 -= alpha * db2
    W3 -= alpha * dW3
    b3 -= alpha * db3
  return W1,b1,W2,b2,W3,b3


def Predictions(Output):
  y_pred = []
  Output = list(Output)
  for i in range(len(Output)):
    Output[i] = list(Output[i])
    max_val = max(Output[i])
    max_index = Output[i].index(max_val)
    y_pred.append(max_index+1)
  return y_pred


def confusion_matrix(y_pred,y_true):
  conf_mat = np.zeros((3,3))
  for i in range(len(y_true)):
    if y_true[i] == 1.:
      if y_pred[i] == 1.:
        conf_mat [0][0] += 1
      if y_pred[i] == 2.:
```

```
        conf_mat [0][1] += 1
      if y_pred[i] == 3.:
        conf_mat [0][2] += 1
    if y_true[i] == 2.:
      if y_pred[i] == 1.:
        conf_mat [1][0] += 1
      if y_pred[i] == 2.:
        conf_mat [1][1] += 1
      if y_pred[i] == 3.:
        conf_mat [1][2] += 1
    if y_true[i] == 3.:
      if y_pred[i] == 1.:
        conf_mat [2][0] += 1
      if y_pred[i] == 2.:
        conf_mat [2][1] += 1
      if y_pred[i] == 3.:
        conf_mat [2][2] += 1
  return conf_mat

x_tr, x_ts, y_tr, y_ts = train_test_split(x,y, train_size=0.7)
x_valid, x_ts, y_valid, y_ts = train_test_split(x_ts,y_ts, test_size=0.33)


#Grid search
acc_list = []
hidden = []
for i in range(10,101,10):
  hidden.append(i)
  W1,B1,W2,B2,W3,B3 = mlp_fn(x_valid,y_valid,i,0.0001)
  Z1,A1,Z2,A2,Z3,A3 = feed_forward(x_valid,W1,B1,W2,B2,W3,B3)
  predicted_val = Predictions(A3)
  Conf_matrix = confusion_matrix(predicted_val,y_valid)
  overall_accuracy = (Conf_matrix[ 0 ][ 0 ] + Conf_matrix[ 1 ][ 1 ] + Conf_matrix[ 2 ][ 2
  acc_list.append(overall_accuracy)
maximum_value = max(acc_list)
maximum_index = acc_list.index(maximum_value)
optimum_hidden_neurons = hidden[maximum_index]


optimum_hidden_neurons
```

```
50
```

```
w1,b1,w2,b2,w3,b3 = mlp_fn(x_tr,y_tr,optimum_hidden_neurons,0.0001)
z1,a1,z2,a2,z3,out = feed_forward(x_ts,w1,b1,w2,b2,w3,b3)
y_predict = Predictions(out)
confmat = confusion_matrix(y_ts,y_predict)
confmat = np.asarray(confmat)
class_acc = np.zeros(3)
print (confmat)
for i in range(3):
  num = confmat[i][i]
  s =0
  for j in range(3):
    s += confmat[i][j]
```

```
    class_acc[i]= num/s
    if (np.isnan(class_acc[i])):
      class_acc[i] = 0
  acc = np.trace(confmat)/np.sum(confmat)
  for c in range(3):
    print( 'Accuracy of class'+ str(c+1)+' : ' + str(class_acc[c]*100))
  print(' Accuracy : ' + str(acc*100))
```

```
[[13.  0.  7.]
 [ 0. 10.  0.]
 [ 2.  0. 10.]]
Accuracy of class1 : 65.0
Accuracy of class2 : 100.0
Accuracy of class3 : 83.33333333333334
 Accuracy : 78.57142857142857
```

Question 4

Implement the radial basis function neural network (RBFNN) for the classification problem. You can use Gaussian, multiquadric and linear kernel functions for the implementation. You can use both holdout (70, 10, and 20%) and 5-fold cross-validation approaches for evaluating the performance of the classifier. The classification performance must be evaluated using individual accuracy and overall accuracy measures. The dataset (data5.xlsx) contains 7 features and the last column is the output (class labels). (Packages such as Scikitlearn, keras, tensorflow, pytorch etc. are not allowed).

```
import pandas as pd
import math
import numpy as np
from sklearn.model_selection import train_test_split


sheet1 = pd.read_excel('data5.xlsx')
data = sheet1.values
x = data[0: , : 7]
x = (x - np.min(x,axis=0))/(np.max(x,axis=0)- np.min(x,axis=0))
y = data[0: , 7]
#n = data.shape[0]
x_tr, x_ts, y_tr, y_ts = train_test_split(x, y, test_size= 0.3 )
y_tr1 = []
y_tr2 = []
y_tr3 = []
for i in range(len(y_tr)):
  if y_tr[i] == 1:
    y_tr1.append(1)
    y_tr2.append(0)
    y_tr3.append(0)
  if y_tr[i] == 2:
    y_tr1.append(0)
    y_tr2.append(1)
```

```
      y_tr3.append(0)
    if y_tr[i] == 3:
      y_tr1.append(0)
      y_tr2.append(0)
      y_tr3.append(1)


def labelling(x,p,cluster_index):
  x1 = np.column_stack([x,cluster_index])
  clusters = np.zeros((p,1))
  clusters = clusters.tolist()
  #cluster_index=np.zeros((20,1))
  #cluster_index = cluster_index.tolist()
  l = x1.shape[1]-1
  for k in range(x.shape[0]):
    for j in range(1,p+1):
      if x1[k][l] == j:
        if (type(clusters[j-1][0]) == float):
          clusters[j-1][0]= x1[k, 0:l].tolist()
          #cluster_index[j-1][0]= k
        else:
          clusters[j-1].append(x1[k, 0:l].tolist())
          #cluster_index[j-1].append(k)
  return clusters


def cl_centers(p,clusters):
  cluster_centres = np.zeros((p,7))
  for cl in range(len(clusters)):
    clusters[cl] = np.array(clusters[cl])
    cluster_centres[cl] = (clusters[cl]).mean(0)
  return cluster_centres


def updation(x,p,cluster_centres):
  dev = np.zeros((p,7))
  arg = np.zeros((p,1))
  updated_cluster_index = []
  for i in range(len(x)):
    for j in range(len(cluster_centres)):
      dev = x[i] - cluster_centres[j]
      arg[j] = (np.linalg.norm(dev))**2
    ind = np.argmin(arg)
    updated_cluster_index.append(ind+1)
  return updated_cluster_index


def kmeans(X,p):
  n = X.shape[0]
  randindex = np.random.randint(1,p+1,n)
  randindex = np.array(randindex, copy=False, subok=True, ndmin=2).T
  clus = labelling(X,p,randindex)
  c_mean = cl_centers(p,clus)
  newindex = updation(X,p,c_mean)
  termination = np.zeros((n,1))
```

```
  iter = 1
  while True:
    iter +=1
    clus = labelling(X,p,newindex)
    c_mean = cl_centers(p,clus)
    oldindex = newindex
    newindex = updation(X,p,c_mean)
    ter_cond = np.array(newindex)-np.array(oldindex)
    if all([ v == 0 for v in ter_cond]) or (iter == 50):
      break
    else:
      continue
  return c_mean

def sigmoid(z):
  return 1.0 / ( 1.0 + np.exp(-z))


p=20
centers = kmeans(x_tr,p)
H= np.zeros((x_tr.shape[ 0 ],p))
for i in range (x_tr.shape[ 0 ]):
  for j in range (p):
    H[i][j] = np.linalg.norm(x_tr[i]-centers[j])


H_test = np.empty((x_ts.shape[ 0 ],p), dtype= float )
for i in range (x_ts.shape[ 0 ]):
  for j in range (p):
    H_test[i][j] = np.linalg.norm(x_ts[i]-centers[j])


H = np.matrix(H)
w1= np.dot(H.I,y_tr1)
w2= np.dot(H.I,y_tr2)
w3= np.dot(H.I,y_tr3)
p_outputs = []
for b in range(len(H_test)):
  pred_op = []
  Z1 = np.dot(H_test[b],w1.T)
  pred_op.append(sigmoid(Z1))
  Z2 = np.dot(H_test[b],w2.T)
  pred_op.append(sigmoid(Z2))
  Z3 = np.dot(H_test[b],w3.T)
  pred_op.append(sigmoid(Z3))
  pred_op = np.array(pred_op)
  p_outputs.append(np.argmax(pred_op)+1)
  pred_op = pred_op.tolist()
y_actual = pd.Series(y_ts, name= 'Actual' )
y_pred = pd.Series(p_outputs, name= 'Predicted' )
confmat = pd.crosstab(y_actual,y_pred)
print(confmat)
confmat = np.asarray(confmat)
class_acc = np.zeros(3)
for i in range(3):
```

```
    num = confmat[i][i]
    s =0
    for j in range(3):
      s += confmat[i][j]
    class_acc[i]= num/s
  #print(class_acc)
  acc = np.trace(confmat)/np.sum(confmat)
  for c in range(3):
    print( 'Accuracy of class'+ str(c+1)+' : ' + str(class_acc[c]*100))
  print( 'Overall Accuracy : ' + str(acc*100))
```

```
Predicted   1   2   3
Actual
1.0        17   0   4
2.0         0  20   0
3.0         1   0  21
Accuracy of class1 : 80.95238095238095
Accuracy of class2 : 100.0
Accuracy of class3 : 95.45454545454545
Overall Accuracy : 92.06349206349206
```

Question 5

Implement the stacked autoencoder based deep neural network for the classification problem. The deep neural network must contain 3 hidden layers from three autoencoders. You can use holdout (70, 10, and 20%) cross-validation technique for selecting, training and test instances for the classifier. The dataset (data5.xlsx) contains 7 features and the last column is the output (class labels). For autoencoder implementation, please use back propagation algorithm discussed in the class. Evaluate individual accuracy and overall accuracy. (Packages such as Scikitlearn, keras, tensorflow, pytorch etc. are not allowed).

```
import pandas as pd
import math
import numpy as np
import random
from sklearn.model_selection import train_test_split


sheet1 = pd.read_excel('data5.xlsx')
data = sheet1.values
np.random.shuffle(data)
x = data[0: , : 7]
x = (x - np.min(x,axis=0))/(np.max(x,axis=0)- np.min(x,axis=0))
y = data[0: , 7]


x_train, x_test, y_train, y_test = train_test_split(x,y, train_size=0.7)
x_valid, x_test, y_valid, y_test = train_test_split(x_test,y_test, test_size=0.33)


def sigmoid(z):
    z = z.astype(float)
```

```
      z_output =1/(1 + np.exp(-z))
      return z_output

  def sigmoid_derivative(z):
    derivative = z*(1-z)
    return derivative


  def cost_fn(y,y_p):
    loss = 0
    for i in range(y):
      loss = loss + (y[i]-y_p[i])**2
    J = loss/(2*len(x))
    return J


  def Predictions(Output):
    y_pred = []
    Output = list(Output)
    for i in range(len(Output)):
      Output[i] = list(Output[i])
      max_val = max(Output[i])
      max_index = Output[i].index(max_val)
      y_pred.append(max_index+1)
    y_pred = np.array(y_pred)
    return y_pred


  def confusion_matrix(y_pred,y_true):
    conf_mat = np.zeros((3,3))
    for i in range(len(y_true)):
      if y_true[i] == 1.:
        if y_pred[i] == 1.:
          conf_mat [0][0] += 1
        if y_pred[i] == 2.:
          conf_mat [0][1] += 1
        if y_pred[i] == 3.:
          conf_mat [0][2] += 1
      if y_true[i] == 2.:
        if y_pred[i] == 1.:
          conf_mat [1][0] += 1
        if y_pred[i] == 2.:
          conf_mat [1][1] += 1
        if y_pred[i] == 3.:
          conf_mat [1][2] += 1
      if y_true[i] == 3.:
        if y_pred[i] == 1.:
          conf_mat [2][0] += 1
        if y_pred[i] == 2.:
          conf_mat [2][1] += 1
        if y_pred[i] == 3.:
          conf_mat [2][2] += 1
    return conf_mat


  def model(X,hidden_nodes,output_dim=3):
```

```
    input_dim = X.shape[1]
    W1 = np.random.randn(input_dim, hidden_nodes) / np.sqrt(input_dim)
    b1 = np.zeros((1, hidden_nodes))
    W2 = np.random.randn(hidden_nodes, hidden_nodes) / np.sqrt(hidden_nodes)
    b2 = np.zeros((1, hidden_nodes))
    W3 = np.random.randn(hidden_nodes, hidden_nodes) / np.sqrt(hidden_nodes)
    b3 = np.zeros((1, hidden_nodes))
    W4 = np.random.randn(hidden_nodes, output_dim) / np.sqrt(hidden_nodes)
    b4 = np.zeros((1, output_dim))

    return W1,b1,W2,b2,W3,b3,W4,b4


def sae_ffn(X,W1,b1,W2,b2,W3,b3,W4,b4):
  z1 = np.dot(X,W1) + b1
  a1 = sigmoid(z1)
  z2 = np.dot(a1,W2) + b2
  a2 = sigmoid(z2)
  z3 = np.dot(a2,W3) + b3
  a3 = sigmoid(z3)
  z4 = np.dot(a3,W4) + b4
  a4 = sigmoid(z4)
  return a1,a2,a3,a4


def sae_bp(X,Y,W1,W2,W3,W4,a1,a2,a3,a4,p):
  rho = 1
  rho_o =0.5
  beta = 0.01
  m = X.shape[1]
  KL = beta * (-(rho / rho_o) + ((1 - rho) / (1 - rho_o)))
  delta4 = np.zeros((p,3))
  dW4 = np.zeros((p,3))
  delta_o4 = np.zeros((p,3))
  dW3 = np.zeros((p,p))
  delta_o3 = np.zeros((p,p))
  dW2 = np.zeros((p,p))
  delta_o2 = np.zeros((p,p))
  dW1 = np.zeros((m,p))
  delta_o1 = np.zeros((m,p))
  db4 = np.zeros(3)
  db3 = np.zeros(p)
  db2 = np.zeros(p)
  db1 = np.zeros(m)
  for k in range(3):
    delta4[k] = Y[k] - Predictions(a4[k])
    delta_o4[k] = np.multiply(delta4[k],Predictions(sigmoid_derivative(a4[k])))
    dW4[k] = np.dot(np.transpose(delta_o4[k]),a3)
    db4[k] = delta_o4[k]
  for i in range(p):
    dW3[i],db3[i],delta_o3[i] = delta_derivative(W4[i],delta_o4,KL,a2[i],a3[i])
    dW2[i],db2[i],delta_o2[i] = delta_derivative(W3[i],delta_o3,KL,a1[i],a2[i])
  for j in range(m):
    dW1[j],db1[j],delta_o1[j] = delta_derivative(W2[j],delta_o2,KL,X[j],a1[j])
```

```
  return dW1, dW2, dW3,dW4, db1, db2, db3,db4
```

```
def delta_derivative(w,delta_o,k,a_prev,a):
  delta_n = np.multiply(sum(np.dot(np.transpose(w), delta_o)) + k ,np.multiply(a, 1 - a))
  dW =np.dot(delta_n,np.transpose(a_prev))
  db = np.sum(delta_n,axis=0,keepdims=True)
  return dW,db,delta_n
```

```
def Stacked_autoencoder(X,Y,hidden_node,alpha,lamda):
  W1,b1,W2,b2,W3,b3,W4,b4 = model(X,hidden_node,3)
  for i in range(500):
    a1,a2,a3,a4 = sae_ffn(X,W1,b1,W2,b2,W3,b3,W4,b4)
    dW1, dW2, dW3, dW4 , db1, db2, db3 ,db4 = sae_bp(X,Y,W1,W2,W3,W4,a1,a2,a3,a4,hidden_no
    W1 -= alpha * (dW1 + lamda*W1)
    b1 -= alpha * db1
    W2 -= alpha * (dW2 + lamda*W2)
    b2 -= alpha * db2
    W3 -= alpha * (dW3 + lamda*W3)
    b3 -= alpha * db3
    W4 -= alpha * (dW4 + lamda*W4)
    b4 -= alpha * db4

  return W1,b1,W2,b2,W3,b3,W4,b4
```

```
#Grid search
acc_list = []
hidden = []
for i in range(10,101,10):
  hidden.append(i)
  W1,B1,W2,B2,W3,B3,W4,B4 = Stacked_autoencoder(x_valid,y_valid,i,0.01)
  Z1,A1,Z2,A2,Z3,A3,Z4,A4 = sae_ffn(x_valid,W1,B1,W2,B2,W3,B3,W4,B4)
  predicted_val = Predictions(A4)
  Conf_matrix = confusion_matrix(predicted_val,y_valid)
  ovr_acc = (Conf_matrix[ 0 ][ 0 ] + Conf_matrix[ 1 ][ 1 ] + Conf_matrix[ 2 ][ 2 ])/sum(su
  acc_list.append(ovr_acc)
opt_val = max(acc_list)
opt_ind = acc_list.index(opt_val)
hidden_neurons_opt = hidden[opt_ind]
```

```
hidden_neurons_opt
```

```
20
```

```
w1,b1,w2,b2,w3,b3,w4,b4 = Stacked_autoencoder(x_train,y_train,hidden_neurons_opt,0.01,0.01
z1,a1,z2,a2,z3,a3,z4,out = sae_ffn(x_test,w1,b1,w2,b2,w3,b3,w4,b4)
y_predict = Predictions(out)
```

```
conf_matrix = confusion_matrix(y_predict,y_test)
```

```
accuracy = (conf_matrix[ 0 ][ 0 ] + conf_matrix[ 1 ][ 1 ] + conf_matrix[ 2 ][ 2 ])/sum(sum
class1_acc = conf_matrix[ 0 ][ 0 ]/sum(conf_matrix[ 0 ])
class2_acc = conf_matrix[ 1 ][ 1 ]/sum(conf_matrix[ 1 ])
class3_acc = conf_matrix[ 2 ][ 2 ]/sum(conf_matrix[ 2 ])
print( 'Accuracy of class 1 is ' + str(class1_acc*100))
print( 'Accuracy of class 2 is ' + str(class2_acc*100))
print( 'Accuracy of class 3 is ' + str(class3_acc*100))
print( 'Overall Accuracy is ' + str(accuracy*100))
```

```
Accuracy of class 1 is 80.6895874946
Accuracy of class 2 is 90.0909090909
Accuracy of class 3 is 87.378947884
Overall Accuracy is 86.05314815649999
```

Question 6

Implement extreme learning machine (ELM) classifier for the classification. You can use Gaussian and tanh activation functions. Please select the training and test instances using 5-fold cross-validation technique Evaluate individual accuracy and overall accuracy. The dataset (data5.xlsx) contains 7 features and the last column is the output (class labels). (Packages such as Scikitlearn, keras, tensorflow, pytorch etc. are not allowed).

```
import pandas as pd
import math
import numpy as np
import random
from sklearn.model_selection import train_test_split


sheet1 = pd.read_excel('data5.xlsx')
data = sheet1.values
np.random.shuffle(data)
x = data[0: , : 7]
x = (x - np.min(x,axis=0))/(np.max(x,axis=0)- np.min(x,axis=0))
y = data[0: , 7]


m = x.shape[0]
num = math.floor(m/5)
subsets = []
y_subsets = []
for i in range(4):
  subsets.append(x[i*num:(i+1)*num])
  y_subsets.append(y[i*num:(i+1)*num])
subsets.append(x[(i+1)*num:])
y_subsets.append(y[(i+1)*num:])
for j in range(5):
  print(len(subsets[j]))
```

```
41
41
41
```

```
  41
  45
```

```
def sigmoid(z):
  return 1.0 / ( 1.0 + np.exp(-z))


def one_vs_all(y):
  y_model1 = []
  y_model2 = []
  y_model3 = []
  for ele in y:
    if (ele == 1):
      y_model1.append(1)
      y_model2.append(0)
      y_model3.append(0)
    if (ele == 2):
      y_model1.append(0)
      y_model2.append(1)
      y_model3.append(0)
    if (ele == 3):
      y_model1.append(0)
      y_model2.append(0)
      y_model3.append(1)
  return y_model1,y_model2,y_model3


acc_vals = []
pvals = []
for p in range ( 20 , 151 , 10 ):
  pvals.append(p)
  print ( 'Result for hidden neurons : ' + str (p) + ' :' )
  ovr_acc = 0
  acc1 = 0
  acc2 = 0
  acc3 = 0
  for f in range(5):
    x_tr = []
    y_tr = []
    #a_outputs = []
    pred_op = []
    p_outputs = []
    arr1 = np.array(subsets[f])
    x_ts = arr1.tolist()
    y_ts = y_subsets[f]
    for k in range(5):
      if k!=f:
        arr = np.array(subsets[k])
        list1 = arr.tolist()
        x_tr.extend(list1)
        y_tr.extend(y_subsets[k])
    y_tr1,y_tr2,y_tr3 = one_vs_all(y_tr)
    x_tr = np.array(x_tr)
    x_ts = np.array(x_ts)
```

```
      randommat = np.random.randn(x_tr.shape[ 1 ]+ 1 ,p)
      H = np.append(np.ones((x_tr.shape[ 0 ], 1 )), x_tr, axis= 1 )
      H = np.dot(H,randommat)
      H = np.tanh(H)
      H = np.matrix(H)
      w1= np.dot(H.I,np.transpose(y_tr1))
      w2= np.dot(H.I,np.transpose(y_tr2))
      w3= np.dot(H.I,np.transpose(y_tr3))
      p_outputs = []
      H_ts = np.append(np.ones((x_ts.shape[ 0 ], 1 )), x_ts, axis= 1 )
      H_ts = np.dot(H_ts,randommat)
      H_ts = np.tanh(H_ts)
      H_ts = np.matrix(H_ts)
      for b in range(len(H_ts)):
        pred_op = []
        Z1 = np.dot(H_ts[b],w1.T)
        pred_op.append(sigmoid(Z1))
        Z2 = np.dot(H_ts[b],w2.T)
        pred_op.append(sigmoid(Z2))
        Z3 = np.dot(H_ts[b],w3.T)
        pred_op.append(sigmoid(Z3))
        pred_op = np.array(pred_op)
        p_outputs.append(np.argmax(pred_op)+1)
        pred_op = pred_op.tolist()
      y_actual = pd.Series(y_ts, name= 'Actual' )
      y_pred = pd.Series(p_outputs, name= 'Predicted' )
      confmat = pd.crosstab(y_actual,y_pred)
      print ( 'Fold ' + str (f+1) + ' :' )
      print (confmat)
      confmat = np.asarray(confmat)
      class_acc = np.zeros(3)
      for i in range(3):
        num = confmat[i][i]
        s =0
        for j in range(3):
          s += confmat[i][j]
        class_acc[i]= num/s
      acc = np.trace(confmat)/np.sum(confmat)
      for c in range(3):
        print( 'Accuracy of class'+ str(c+1)+' : ' + str(class_acc[c]*100))
      print( 'Fold '+ str(f+1)+ ' Accuracy : ' + str(acc*100))
      ovr_acc += acc
      acc1 += class_acc[0]
      acc2 += class_acc[1]
      acc3 += class_acc[2]
      #print(p_outputs)
      #print(y_ts)
    print( 'Average Accuracy : ' + str(ovr_acc*20))
    print( 'Average Accuracy of class 1 : ' + str(acc1*20))
    print( 'Average Accuracy of class 2 : ' + str(acc2*20))
    print( 'Average Accuracy of class 3 : ' + str(acc3*20))
    acc_vals.append(ovr_acc*20)
  acc_vals =np.array(acc_vals)
  acc_ind = np.argmax(acc_vals)
  print(acc_vals,acc_ind)
```

```
acc_max = pvals[acc_ind]
print('Best Result for '+ str(acc_max)+ ' Hidden Neurons , '+ 'Accuracy - '+ str(acc_vals[
```

```
Fold 1 :
Predicted  1   2  3
Actual
1.0        7   0  3
2.0        5  13  0
3.0        4   0  9
Accuracy of class1 : 70.0
Accuracy of class2 : 72.22222222222221
Accuracy of class3 : 69.23076923076923
Fold 1 Accuracy : 70.73170731707317
Fold 2 :
Predicted   1   2  3
Actual
1.0        13   1  3
2.0         3  11  0
3.0         4   0  6
Accuracy of class1 : 76.47058823529412
Accuracy of class2 : 78.57142857142857
Accuracy of class3 : 60.0
Fold 2 Accuracy : 73.17073170731707
Fold 3 :
Predicted   1   2   3
Actual
1.0        10   1   1
2.0         1  10   2
3.0         1   0  15
Accuracy of class1 : 83.33333333333334
Accuracy of class2 : 76.92307692307693
Accuracy of class3 : 93.75
Fold 3 Accuracy : 85.36585365853658
Fold 4 :
Predicted   1  2   3
Actual
1.0        12  1   3
2.0         1  9   1
3.0         1  3  10
Accuracy of class1 : 75.0
Accuracy of class2 : 81.81818181818183
Accuracy of class3 : 71.42857142857143
Fold 4 Accuracy : 75.60975609756098
Fold 5 :
Predicted   1   2   3
Actual
1.0        11   2   1
2.0         2  11   1
3.0         4   0  13
Accuracy of class1 : 78.57142857142857
Accuracy of class2 : 78.57142857142857
Accuracy of class3 : 76.47058823529412
Fold 5 Accuracy : 77.77777777777779
Average Accuracy : 76.53116531165311
Average Accuracy of class 1 : 76.67507002801119
Average Accuracy of class 2 : 77.62126762126762
Average Accuracy of class 3 : 74.17598577892696
[92.72628726 94.67750678 93.25745257 94.1897019  95.16531165 92.81300813
 94.1897019  92.7696477  92.28184282 89.84281843 88.95392954 83.14363144
 79.28455285 76.53116531] 4
```

```
Best Result for 60 Hidden Neurons , Accuracy - 95.16531165311653
```

Only the best result has been presented here due to the long output

```
# This is formatted as code
Best Result for 50 Hidden Neurons , Accuracy - 95.20867208672087
Result for hidden neurons : 50 :
Fold 1 :
Predicted   1   2   3
Actual
1.0        15   0   1
2.0         0  13   0
3.0         2   0  10
Accuracy of class1 : 93.75
Accuracy of class2 : 100.0
Accuracy of class3 : 83.33333333333334
Fold 1 Accuracy : 92.6829268292683
Fold 2 :
Predicted   1  2   3
Actual
1.0        14  0   2
2.0         0  9   0
3.0         0  0  16
Accuracy of class1 : 87.5
Accuracy of class2 : 100.0
Accuracy of class3 : 100.0
Fold 2 Accuracy : 95.1219512195122
Fold 3 :
Predicted   1   2   3
Actual
1.0        12   0   0
2.0         1  12   0
3.0         1   0  15
Accuracy of class1 : 100.0
Accuracy of class2 : 92.3076923076923
Accuracy of class3 : 93.75
Fold 3 Accuracy : 95.1219512195122
Fold 4 :
Predicted   1   2   3
Actual
1.0        14   0   0
2.0         0  15   0
3.0         1   0  11
Accuracy of class1 : 100.0
Accuracy of class2 : 100.0
Accuracy of class3 : 91.66666666666666
```

```
Fold 4 Accuracy : 97.5609756097561
Fold 5 :
Predicted  1   2   3
Actual
1.0        9   1   1
2.0        0  20   0
3.0        0   0  14
Accuracy of class1 : 81.81818181818183
Accuracy of class2 : 100.0
Accuracy of class3 : 100.0
Fold 5 Accuracy : 95.55555555555556
Average Accuracy : 95.20867208672087
Average Accuracy of class 1 : 92.61363636363637
Average Accuracy of class 2 : 98.46153846153847
Average Accuracy of class 3 : 93.75
```

Question 7

Implement a deep neural network, which contains two hidden layers (the hidden layers are obtained from the ELM-autoencoders). The last layer will be the ELM layer which means the second hidden layer feature vector is used as input to the ELM classifier. The network can be called as deep layer stacked autoencoder based extreme learning machine. You can use holdout approach (70, 10, 20%) for evaluating the performance of the classifier. The dataset (data5.xlsx) contains 7 features and the last column is the output (class labels). Evaluate individual accuracy and overall accuracy. (Packages such as Scikitlearn, keras, tensorflow, pytorch etc. are not allowed)

```
sheet1 = pd.read_excel('data5.xlsx')
data = sheet1.values
np.random.shuffle(data)
X = data[0: , : 7]
X = (X - np.min(X,axis=0))/(np.max(X,axis=0)- np.min(X,axis=0))
Y = data[0: , 7]


x_train, x_test, y_train, y_test = train_test_split(X,Y, train_size=0.7)
x_valid, x_test, y_valid, y_test = train_test_split(x_test,y_test, test_size=0.33)


def sigmoid(z):
    z = z.astype(float)
    z_output =1/(1 + np.exp(-z))
    return z_output


def sigmoid_derivative(z):
  derivative = z*(1-z)
```

```
    return derivative


  def Predictions(Output):
    y_pred = []
    Output = list(Output)
    for i in range(len(Output)):
      Output[i] = list(Output[i])
      max_val = max(Output[i])
      max_index = Output[i].index(max_val)
      y_pred.append(max_index+1)
    y_pred = np.array(y_pred)
    return y_pred


  def one_vs_all(y):
    y_model1 = []
    y_model2 = []
    y_model3 = []
    for ele in y:
      if (ele == 1):
        y_model1.append(1)
        y_model2.append(0)
        y_model3.append(0)
      if (ele == 2):
        y_model1.append(0)
        y_model2.append(1)
        y_model3.append(0)
      if (ele == 3):
        y_model1.append(0)
        y_model2.append(0)
        y_model3.append(1)
    return y_model1,y_model2,y_model3


  def model(X,hidden_nodes,output_dim=3):

      input_dim = X.shape[1]
      W1 = np.random.randn(input_dim, hidden_nodes) / np.sqrt(input_dim)
      b1 = np.zeros((1, hidden_nodes))
      W2 = np.random.randn(hidden_nodes, hidden_nodes) / np.sqrt(hidden_nodes)
      b2 = np.zeros((1, hidden_nodes))
      W3 = np.random.randn(hidden_nodes, output_dim) / np.sqrt(hidden_nodes)
      b3 = np.zeros((1, output_dim))

      return W1,b1,W2,b2,W3,b3


  def sae_ffn(X,W1,b1,W2,b2,W3,b3):
    z1 = np.dot(X,W1) + b1
    a1 = sigmoid(z1)
    z2 = np.dot(a1,W2) + b2
    a2 = sigmoid(z2)
    z3 = np.dot(a2,W3) + b3
    return z1,z2,z3
```

```
def sae_bp(X,Y,W1,W2,W3,a1,a2,a3,p):
  rho = 1
  rho_o =0.5
  beta = 0.01
  m = X.shape[1]
  KL = beta * (-(rho / rho_o) + ((1 - rho) / (1 - rho_o)))
  delta3 = np.zeros((p,3))
  dW3 = np.zeros((p,3))
  delta_o3 = np.zeros((p,3))
  dW2 = np.zeros((p,p))
  delta_o2 = np.zeros((p,p))
  dW1 = np.zeros((m,p))
  delta_o1 = np.zeros((m,p))
  db3 = np.zeros(3)
  db2 = np.zeros(p)
  db1 = np.zeros(m)
  for k in range(3):
    delta3[k] = Y[k] - Predictions(a3[k])
    delta_o3[k] = np.multiply(delta3[k],Predictions(sigmoid_derivative(a3[k])))
    dW3[k] = np.dot(np.transpose(delta_o3[k]),a2)
    db3[k] = delta_o3[k]
  for i in range(p):
    dW2[i],db2[i],delta_o2[i] = delta_derivative(W3[i],delta_o3,KL,a1[i],a2[i])
  for j in range(m):
    dW1[j],db1[j],delta_o1[j] = delta_derivative(W2[j],delta_o2,KL,X[j],a1[j])

  return dW1, dW2, dW3, db1, db2, db3


def delta_derivative(w,delta_o,k,a_prev,a):
  delta_n = np.multiply(sum(np.dot(np.transpose(w), delta_o)) + k ,np.multiply(a, 1 - a))
  dW =np.dot(delta_n,np.transpose(a_prev))
  db = np.sum(delta_n,axis=0,keepdims=True)
  return dW,db,delta_n


def Stacked_autoencoder(X,Y,alpha,lamda,p):
  W1,b1,W2,b2,W3,b3 = model(X,p,3)
  for i in range(500):
    a1,a2,a3 = sae_ffn(X,W1,b1,W2,b2,W3,b3)
    dW1, dW2, dW3, db1, db2, db3 = sae_bp(X,Y,W1,W2,W3,a1,a2,a3,p)
    W1 -= alpha * (dW1 + lamda*W1)
    b1 -= alpha * db1
    W2 -= alpha * (dW2 + lamda*W2)
    b2 -= alpha * db2
    W3 -= alpha * (dW3 + lamda*W3)
    b3 -= alpha * db3
    return W1,b1,W2,b2,W3,b3


def elm_input(x,y,hidden_neurons_opt):
  w1,b1,w2,b2,w3,b3 = Stacked_autoencoder(x,y,0.01,0.01,hidden_neurons_opt)
  z1,z2,out = sae_ffn(x,w1,b1,w2,b2,w3,b3)
  return out
```

```
def elm_fn(x,y,p):
  elm_train_input = elm_input(x_train,y_train,p)
  elm_test_input = elm_input(x,y,p)
  randommat = np.random.randn(elm_train_input.shape[ 1 ]+ 1 ,p)
  H = np.append(np.ones((elm_train_input.shape[ 0 ], 1 )),elm_train_input, axis= 1 )
  H = np.dot(H,randommat)
  H = np.tanh(H)
  H = np.matrix(H)
  y_tr1,y_tr2,y_tr3 = one_vs_all(y_train)
  w1= np.dot(H.I,np.transpose(y_tr1))
  w2= np.dot(H.I,np.transpose(y_tr2))
  w3= np.dot(H.I,np.transpose(y_tr3))
  p_outputs = []
  H_ts = np.append(np.ones((elm_test_input.shape[ 0 ], 1 )),elm_test_input, axis= 1 )
  H_ts = np.dot(H_ts,randommat)
  H_ts = np.tanh(H_ts)
  H_ts = np.matrix(H_ts)
  for b in range(len(H_ts)):
    pred_op = []
    Z1 = np.dot(H_ts[b],w1.T)
    pred_op.append(sigmoid(Z1))
    Z2 = np.dot(H_ts[b],w2.T)
    pred_op.append(sigmoid(Z2))
    Z3 = np.dot(H_ts[b],w3.T)
    pred_op.append(sigmoid(Z3))
    pred_op = np.array(pred_op)
    p_outputs.append(np.argmax(pred_op)+1)
    pred_op = pred_op.tolist()
  y_actual = pd.Series(y, name= 'Actual' )
  y_pred = pd.Series(p_outputs, name= 'Predicted' )
  confmat = pd.crosstab(y_actual,y_pred)
  #print (confmat)
  confmat = np.asarray(confmat)
  class_acc = np.zeros(3)
  for i in range(3):
    num = confmat[i][i]
    s =0
    for j in range(3):
      s += confmat[i][j]
    class_acc[i]= num/s
  acc = np.trace(confmat)/np.sum(confmat)
  return class_acc,acc,confmat


#Grid_search
acc_list = []
hidden = []
for i in range(10,51,5):
  hidden.append(i)
  cl_acc,accu,cm1 = elm_fn(x_valid,y_valid,i)
  acc_list.append(accu)
opt_val = max(acc_list)
opt_ind = acc_list.index(opt_val)
hidden_neurons_opt = hidden[opt_ind]
```

```
hidden_neurons_opt
```

```
20
```

```
class_accuracy,accuracy,conf_mat = elm_fn(x_test,y_test,hidden_neurons_opt)
print(conf_mat)
for c in range(3):
  print( 'Accuracy of class'+ str(c+1)+' : ' + str(class_accuracy[c]*100))
print( 'Overall  Accuracy : ' + str(accuracy*100))
```

```
Predicted   1   2  3
Actual
1.0        10   1  2
2.0         2  10  8
3.0         0   1  8
Accuracy of class1 : 76.92307692307693
Accuracy of class2 : 50.0
Accuracy of class3 : 88.88888888888889
Overall  Accuracy : 66.66666666666666
```

Question 8

Implement support vector machine (SVM) classifier for the multi-class classification task. You can use one vs one and one vs all multiclass coding methods to create binary SVM models. Implement the SMO algorithm for the evaluation of the training parameters of SVM such as Lagrange multipliers. You can use holdout approach (70%, 10%, 20%) for evaluating the performance of the classifier. The dataset (data5.xlsx) contains 7 features and the last column is the output (class labels). Evaluate individual accuracy and overall accuracy. You can use RBF and polynomial kernels. Evaluate the classification performance of multiclass SVM for each kernel function. (Packages such as Scikitlearn, keras, tensorflow, pytorch etc. are not allowed)

```
import pandas as pd
import math
import numpy as np
import random
from sklearn.model_selection import train_test_split
```

```
sheet1 = pd.read_excel('data5.xlsx')
data = sheet1.values
np.random.shuffle(data)
x = data[0: , :-1]
x = (x - np.min(x,axis=0))/(np.max(x,axis=0)- np.min(x,axis=0))
y = data[0: , -1]
x_train,X_test,y_train,y_test = train_test_split(x, y, test_size= 0.2 )
```

```
def one_vs_all(y):
  y_model1 = []
```

```
    y_model2 = []
    y_model3 = []
    for ele in y:
      if (ele == 1):
        y_model1.append(1)
        y_model2.append(-1)
        y_model3.append(-1)
      if (ele == 2):
        y_model1.append(-1)
        y_model2.append(1)
        y_model3.append(-1)
      if (ele == 3):
        y_model1.append(-1)
        y_model2.append(-1)
        y_model3.append(1)
    return y_model1,y_model2,y_model3

  def train_lin_sum (x_tr,y_tr,C,bound,maxiters):
    m = x_tr.shape[0]
    n = x_tr.shape[1]
    b = 0
    #mean = np.zeros(m)
    #cov = np.identity(m)
    #mu = np.random.multivariate_normal(mean, cov)
    mu = np.ones((m,1))
    E = np.zeros((m,1))
    iter = 0
    eta =0
    L =0
    H = 0
    kernel = lambda xi, yi: math.pow((np.dot(xi.T, yi) + 1), 2)
    while iter<maxiters:
      count_mu = 0
      for i in range(m):
        E[i] = f_x(x_tr, y_tr, mu, b, x_tr[i, :], 2) - y_tr[i]
        if (y_tr[i]*E[i]<-bound and mu[i]<C) or (y_tr[i]*E[i]>bound and mu[i]>0):
          j = math.floor(m*np.random.rand())
          while j == i:
            j = math.floor(m*np.random.rand())
          E[j] = f_x(x_tr, y_tr, mu, b, x_tr[j, :], 2) - y_tr[j]
          mu_i_old = mu[i]
          mu_j_old = mu[j]
          if y_tr[i] == y_tr[j]:
            L = max(0, mu[i]+mu[j]-C)
            H = min(C,mu[i]+mu[j])
          else:
            L = max(0,mu[j]-mu[i])
            H = min(C,C+mu[j]-mu[i])
          if (L == H):
            continue
          eta = 2*kernel(x_tr[i, :], x_tr[j, :]) - kernel(x_tr[i, :], x_tr[i, :]) - kernel(x
          if eta>=0:
            continue
          mu[j] = mu[j] - (y_tr[j]*(E[i]-E[j]))/eta
```

```
        mu[j] = min(H,mu[j])
        mu[j] = max(L,mu[j])
        if abs(mu[j]-mu_j_old)<bound:
          mu[j] = mu_j_old
          continue
        mu[i] = mu[i]+y_tr[i]*y_tr[j]*(mu_j_old-mu[j])
        b1 = b - E[i]- (y_tr[i]*(mu[i]-mu_i_old)*kernel(x_tr[i, :], x_tr[i, :]))  - (y_tr[
        b2 = b - E[j]- (y_tr[i]*(mu[i]-mu_i_old)*kernel(x_tr[i, :], x_tr[j, :])) - (y_tr[j
        if (0<mu[i]) and (mu[i]<C):
          b = b1
        elif (0<mu[j]) and (mu[j]<C):
          b = b2
        else:
          b = (b1+b2)/2
        count_mu = count_mu +1
    if (count_mu == 0):
      iter = iter+1
    else:
      iter = 0
  il1 = mu>0
  Xsvm = []
  Ysvm = []
  mus = []
  for v in range(len(il1)):
    if il1[v]:
      Xsvm.append(x_tr[v,:])
      Ysvm.append(y_tr[v])
      mus.append(mu[v])
  Xsvm = np.array(Xsvm)
  Ysvm = np.array(Ysvm)
  mus = np.array(mus)
  #s = mus.shape[1]
  w = np.zeros(7)
  num_sv = Xsvm.shape[1]
  for l in range(7):
    w += mus[l]*Ysvm[l]*((1 + Xsvm[l,:])**2)
  return w,b,Xsvm,Ysvm,mus,num_sv

def f_x(X, y, a, b, x, degree):
    predicted_value = 0.0
    # using polynomial kernel
    for k in range(X.shape[0]):
        predicted_value += (a[k]*y[k]*((X[k, :].T@x + 1)**degree))
    return predicted_value + b


def sigmoid(z):
  return 1.0 / ( 1.0 + np.exp(-z))


def prediction(xs,ys,x_ts,mean,bias,n_svm):
  yp = 0
  for s in range(n_svm):
    yp += (mean[s]*ys[s]*np.dot(x_ts,xs[s]))
  return np.sign(yp+bias)
```

```
C = 100
# kernelFunc = linear
iters =100
p_outputs = []
y_tr1,y_tr2,y_tr3 = one_vs_all(y_train)
y_ts1,y_ts2,y_ts3 = one_vs_all(y_test)
w1,b1,x1,y1,m1,n1= train_lin_sum(x_train,y_tr1,C,0.001,iters)
w2,b2,x2,y2,m2,n2= train_lin_sum(x_train,y_tr2,C,0.001,iters)
w3,b3,x3,y3,m3,n3= train_lin_sum(x_train,y_tr3,C,0.001,iters)


yp1 = []
yp2 = []
yp3 = []
lis = [-1,1]
for b in range(len(X_test)):
  if f_x(x1, y1, m1, b1, X_test[b, :], 4) >= 0:
    yp1.append(1.0)
  else:
    yp1.append(-1.0)
  if f_x(x2, y2, m2, b2, X_test[b, :], 4) >= 0:
    yp2.append(1.0)
  else:
    yp2.append(-1.0)
  if f_x(x3, y3, m3, b3, X_test[b, :], 4) >= 0:
    yp3.append(1.0)
  else:
    yp3.append(-1.0)


y_actual1 = pd.Series(y_ts1, name= 'Actual' )
y_pred1 = pd.Series(yp1, name= 'Predicted' )
confmat1 = pd.crosstab(y_actual1,y_pred1)
y_actual2 = pd.Series(y_ts2, name= 'Actual' )
y_pred2 = pd.Series(yp2, name= 'Predicted' )
confmat2 = pd.crosstab(y_actual2,y_pred2)
y_actual3 = pd.Series(y_ts3, name= 'Actual' )
y_pred3 = pd.Series(yp3, name= 'Predicted' )
confmat3 = pd.crosstab(y_actual3,y_pred3)
print("Class 1:\n", confmat1)
print("Class 2:\n", confmat2)
print("Class 3:\n", confmat3)
```

```
Class 1:
 Predicted  -1.0   1.0
Actual
-1           15    12
 1            8     7
Class 2:
 Predicted  -1.0   1.0
Actual
-1           16    13
 1            0    13
Class 3:
 Predicted  -1.0   1.0
```

```
Actual
-1          28    0
 1          12    2
```

```
def accuracy_val(confmat):
  class_acc = np.zeros(3)
  confmat = np.asarray(confmat)
  for i in range(2):
    num = confmat[i][i]
    s =0
    for j in range(2):
      s += confmat[i][j]
    class_acc[i]= num/s
  acc = np.trace(confmat)/np.sum(confmat)
  return acc*100
```

```
acc1 = accuracy_val(confmat1)
print( 'Class 1 Accuracy : ' + str(acc1))
acc2 = accuracy_val(confmat2)
print( 'Class 2 Accuracy : ' + str(acc2))
acc3 = accuracy_val(confmat3)
print( 'Class 3 Accuracy : ' + str(acc3))
ovr_acc = (acc1+acc2+acc3)/3
print("Overall Accuracy :", ovr_acc)
```

```
Class 1 Accuracy : 52.38095238095239
Class 2 Accuracy : 69.04761904761905
Class 3 Accuracy : 71.42857142857143
Overall Accuracy : 64.28571428571429
```

Question 9

Implement a multi-channel 1D deep CNN architecture (as shown in Fig. 1) for the seven-class classification task. The input and the class labels are given in. mat file format. There is a total of 17160 number of instances present in both input and class-label data files. The input data for each instance is a multichannel time series (12-channel) with size as (12 ×800). The class label for each multichannel time series instance is given in the class_label.mat file. You can select the training and test instances using

hold-out cross-validation (70% training, 10% validation, and 20% testing). The architecture of the multi- channel deep CNN is given as follows. The number of filters, length of each filter, and number of neurons

in the fully connected layers are shown in the following figure. Evaluate individual accuracy and overall accuracy. (Packages such as Scikitlearn, keras, tensorflow, pytorch etc. are allowed)

```
!pip install mat4py
from mat4py import loadmat
import keras
```

```
import tensorflow as tf
import sklearn
import numpy as np
import pandas as pd
import scipy.io as sio
import matplotlib.pyplot as plt

from sklearn import preprocessing
from sklearn.metrics import classification_report,confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn import preprocessing
from keras.models import Sequential,Model
from keras.layers import Input, Dense, Activation, Flatten, Conv1D, Dropout,MaxPooling1D,M
from tensorflow.keras.optimizers import Adam
#from keras.optimizers import SGD,Adam
from keras.utils import np_utils
from sklearn.utils import shuffle
from mat4py import loadmat
```

```
Collecting mat4py
  Downloading mat4py-0.5.0-py2.py3-none-any.whl (13 kB)
Installing collected packages: mat4py
Successfully installed mat4py-0.5.0
```

```
ftr = sio.loadmat("./input.mat")
ftr_vec = pd.DataFrame(ftr["x"])
ftr_vec=(np.asarray(ftr_vec)).T

class_label = sio.loadmat("./class_label.mat")
Y=np.asarray(class_label["y"])
```

```
X=[]
for i in range(len(ftr_vec)):
  X.append(ftr_vec[i][0])
X=np.asarray(X)
X=X.transpose(0,2,1)
for i in range(len(X)):
  X[i]=preprocessing.normalize(X[i])
```

```
y=[]
for i in range(len(Y)):
  y.append(Y[i][0]-1)
y=np.asarray(y)
```

```
X_train,X_test,Y_train,Y_test=train_test_split(X,y,test_size=0.2,train_size=0.8,random_sta
X_train,X_valid,Y_train,Y_valid=train_test_split(X_train,Y_train,train_size=7/8,random_sta
```

```
def CNN_model():
```

```
  model = Sequential()
  #model.add(tf.keras.layers.Flatten())
  model.add(Conv1D(kernel_size=7,filters=20,input_shape = (800,12))) #activation = 'relu')
  model.add(MaxPool1D(pool_size=3, strides=3))   #, padding='valid'))
  model.add(Activation("relu"))
  model.add(Conv1D(kernel_size=7,filters=60))#activation = 'relu'))
  model.add(MaxPool1D(pool_size=3, strides=3))#, padding='valid'))
  model.add(Activation("relu"))
  model.add(Dropout(0.7))
  model.add(Conv1D(kernel_size=7,filters=120))#,activation = 'relu'))
  model.add(Conv1D(kernel_size=7,filters=120))#,activation = 'relu'))
  model.add(Flatten())
  #model.add(Dense(3000,activation = 'relu'))
  model.add(Dense(2000,activation = 'relu'))
  model.add(Dense(700))#,activation = 'relu'))
  model.add(Dense(50))#,activation = 'relu'))
  model.add(Dense(7,activation = 'sigmoid'))
  #model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
  return model

model1 = CNN_model()
```

```
model1.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=['accurac
execution_history=model1.fit(X_train, Y_train, epochs=7, batch_size=1000,validation_data=(
print(execution_history)
```

```
Epoch 1/7
13/13 [==============================] - 66s 5s/step - loss: 2.2163 - accuracy: 0.38
Epoch 2/7
13/13 [==============================] - 65s 5s/step - loss: 0.6316 - accuracy: 0.77
Epoch 3/7
13/13 [==============================] - 63s 5s/step - loss: 0.1946 - accuracy: 0.94
Epoch 4/7
13/13 [==============================] - 74s 6s/step - loss: 0.0398 - accuracy: 0.99
Epoch 5/7
13/13 [==============================] - 64s 5s/step - loss: 0.0124 - accuracy: 0.99
Epoch 6/7
13/13 [==============================] - 64s 5s/step - loss: 0.0046 - accuracy: 0.99
Epoch 7/7
13/13 [==============================] - 63s 5s/step - loss: 0.0024 - accuracy: 0.99
<keras.callbacks.History object at 0x7f799c2cdd90>
```

```
model1.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=['accurac
execution_history=model1.fit(X_train, Y_train, epochs=5, batch_size=1000,validation_data=(
print(execution_history)
```

```
Epoch 1/5
13/13 [==============================] - 66s 5s/step - loss: 0.2408 - accuracy: 0.94
Epoch 2/5
13/13 [==============================] - 63s 5s/step - loss: 0.0138 - accuracy: 0.99
Epoch 3/5
13/13 [==============================] - 64s 5s/step - loss: 0.0026 - accuracy: 0.99
Epoch 4/5
```

```
13/13 [==============================] - 64s 5s/step - loss: 9.0851e-04 - accuracy: (
Epoch 5/5
13/13 [==============================] - 64s 5s/step - loss: 2.8307e-04 - accuracy: 1
<keras.callbacks.History object at 0x7f799d0832d0>
```

```
y_pred = model1.predict(X_test)
p_outputs = []
for i in range(len(y_pred)):
  p_outputs.append(np.argmax(y_pred[i]))
```

```
def conf_mat(y_actual,y_predict):
  y_actual = pd.Series(y_actual, name= 'Actual' )
  y_p = pd.Series(y_predict, name= 'Predicted' )
  confmat = pd.crosstab(y_actual,y_p)
  return confmat
```

```
confusion_mat = conf_mat(Y_test,p_outputs)
```

```
confusion_mat
```

Predicted	0	1	2	3	4	5	6
Actual							
0	568	0	0	0	0	0	0
1	0	353	0	0	0	0	0
2	0	0	600	0	0	0	0
3	0	0	0	298	0	0	0
4	0	0	0	0	615	0	0
5	0	0	0	0	0	637	0
6	0	0	0	0	0	35	326

```
confusion_mat = np.asarray(confusion_mat)
class_acc = np.zeros(7)
for i in range(7):
  num = confusion_mat[i][i]
  s =0
  for j in range(7):
    s += confusion_mat[i][j]
  class_acc[i]= num/s

ovr_acc = np.trace(confusion_mat)/np.sum(confusion_mat)
for d in range(7):
  print( 'Accuracy of class'+ str(d)+' : ' + str(class_acc[d]*100))
print( 'Overall Accuracy : ' + str(ovr_acc*100))
```

```
Accuracy of class0 : 100.0
```

```
Accuracy of class1 : 100.0
Accuracy of class2 : 100.0
Accuracy of class3 : 100.0
Accuracy of class4 : 100.0
Accuracy of class5 : 100.0
Accuracy of class6 : 90.30470914127424
Overall Accuracy : 98.98018648018649
```

```
Loss , Accuracy = model1.evaluate(X_test,Y_test)
print("Accuracy is",Accuracy*100)
```

```
108/108 [==============================] - 8s 73ms/step - loss: 1.4068e-04 - accuracy
Accuracy is 100.0
```

Question 10

Implement the hybrid fuzzy deep neural network (HFDNN) for the three-class classification task. The input and output instances for the HFDNN are given in data5.xlsx file (first seven columns input and last column is the output). For a single instance, the input size is 7. There is a total of 210 instances given in the input and label datasets. You can select the training and test instances using hold-out cross-validation (70%training, 10% validation, and 20% testing). The HFDNN architecture shown in Fig. 2 has neural network hidden layers, fuzzy membership and rule layers, and a fusion layer. The descriptions of the

HFDNN architecture are given in reference. Evaluate individual accuracy and overall accuracy. (Packages such as Scikitlearn, keras, tensorflow, pytorch etc. are not allowed)

```
import keras
import tensorflow as tf
import numpy as np
import pandas as pd
import time
from sklearn.model_selection import train_test_split
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
```

```
WARNING:tensorflow:From /usr/local/lib/python3.7/dist-packages/tensorflow/python/comp
Instructions for updating:
non-resource variables are not supported in the long term
```

```
def set(y):
    for i in range(len(y)):
        if(0.0<y[i]<=1.5):
            y[i] = 1.0
        if(1.5<y[i]<=2.5):
            y[i] = 2.0
        if(y[i]>=2.5):
```

```
            y[i] = 3.0
        return y


    def norm(x):
        return (x - x.mean(axis=0))/x.std(axis=0)


    class ANFIS:

        def __init__(self, n_inputs, n_rules, learning_rate=1e-2):
            self.n = n_inputs
            self.m = n_rules
            self.inputs = tf.placeholder(tf.float32, shape=(None, n_inputs))  # Input
            self.targets = tf.placeholder(tf.float32, shape=None)  # Desired output
            mu = tf.get_variable("mu", [n_rules * n_inputs],
                                 initializer=tf.random_normal_initializer(0, 1))  # Means of G
            sigma = tf.get_variable("sigma", [n_rules * n_inputs],
                                    initializer=tf.random_normal_initializer(0, 1))  # Standar
            y = tf.get_variable("y", [1, n_rules], initializer=tf.random_normal_initializer(0,

            self.params = tf.trainable_variables()

            self.rul = tf.reduce_prod(
                tf.reshape(tf.exp(-0.5 * tf.square(tf.subtract(tf.tile(self.inputs, (1, n_rule
                           (-1, n_rules, n_inputs)), axis=2)  # Rule activations
            # Fuzzy base expansion function:
            num = tf.reduce_sum(tf.multiply(self.rul, y), axis=1)
            den = tf.clip_by_value(tf.reduce_sum(self.rul, axis=1), 1e-12, 1e12)
            self.out = tf.divide(num, den)

            self.loss = tf.losses.huber_loss(self.targets, self.out)  # Loss function computat
            self.optimize = tf.train.AdamOptimizer(learning_rate=alpha).minimize(self.loss)  #
            self.init_variables = tf.global_variables_initializer()  # Variable initializer

        # Function to get predictions from test samples
        def infer(self, sess, x, targets=None):
            if targets is None:
                return sess.run(self.out, feed_dict={self.inputs: x})
            else:
                return sess.run([self.out, self.loss], feed_dict={self.inputs: x, self.targets

        # Function to initiate and train the graph
        def train(self, sess, x, targets):
            yp, l, _ = sess.run([self.out, self.loss, self.optimize], feed_dict={self.inputs:
            return l, yp


    data = pd.read_excel('data5.xlsx')
    data = pd.DataFrame(data)
    data = np.asarray(data)
    y = data[:,-1]
    x = data[:,:-1]
    x = norm(x)
```

```
x_tr, x_ts, y_tr, y_ts = train_test_split(x, y, test_size=0.3)
m = x_tr.shape[0]
n = x_tr.shape[1]


m = 16  # number of rules
alpha = 0.01  # learning rate
epochs= 2000
fis = ANFIS(n_inputs=7, n_rules=m, learning_rate=alpha)


with tf.Session() as sess:
    # Initialize model parameters
    sess.run(fis.init_variables)
    trn_costs = []
    val_costs = []
    time_start = time.time()
    for epoch in range(epochs):
        # Train the model
        trn_loss, train_pred = fis.train(sess, x_tr, y_tr)
        # Evaluate on test set
        test_pred, val_loss = fis.infer(sess, x_ts, y_ts)
        # Print the training cost
        if epoch % 10 == 0:
            print("Train cost after epoch %i: %f" % (epoch, trn_loss))
        if epoch == epochs - 1:
            time_end = time.time()
```

```
Train cost after epoch 1420: 0.003376
Train cost after epoch 1430: 0.003376
Train cost after epoch 1440: 0.003376
Train cost after epoch 1450: 0.003375
Train cost after epoch 1460: 0.003375
Train cost after epoch 1470: 0.003374
Train cost after epoch 1480: 0.003374
Train cost after epoch 1490: 0.003374
Train cost after epoch 1500: 0.003373
Train cost after epoch 1510: 0.003373
Train cost after epoch 1520: 0.003373
Train cost after epoch 1530: 0.003372
Train cost after epoch 1540: 0.003388
Train cost after epoch 1550: 0.003373
Train cost after epoch 1560: 0.003373
Train cost after epoch 1570: 0.003372
Train cost after epoch 1580: 0.003372
Train cost after epoch 1590: 0.003371
Train cost after epoch 1600: 0.003371
Train cost after epoch 1610: 0.003371
Train cost after epoch 1620: 0.003371
Train cost after epoch 1630: 0.003370
Train cost after epoch 1640: 0.003370
Train cost after epoch 1650: 0.003371
Train cost after epoch 1660: 0.003370
Train cost after epoch 1670: 0.003370
Train cost after epoch 1680: 0.003370
Train cost after epoch 1690: 0.003370
Train cost after epoch 1700: 0.003369
Train cost after epoch 1710: 0.003370
```

```
Train cost after epoch 1720: 0.003372
Train cost after epoch 1730: 0.003370
Train cost after epoch 1740: 0.003369
Train cost after epoch 1750: 0.003369
Train cost after epoch 1760: 0.003369
Train cost after epoch 1770: 0.003369
Train cost after epoch 1780: 0.003373
Train cost after epoch 1790: 0.003372
Train cost after epoch 1800: 0.003370
Train cost after epoch 1810: 0.003369
Train cost after epoch 1820: 0.003369
Train cost after epoch 1830: 0.003369
Train cost after epoch 1840: 0.003372
Train cost after epoch 1850: 0.003368
Train cost after epoch 1860: 0.003368
Train cost after epoch 1870: 0.003368
Train cost after epoch 1880: 0.003368
Train cost after epoch 1890: 0.003369
Train cost after epoch 1900: 0.003369
Train cost after epoch 1910: 0.003368
Train cost after epoch 1920: 0.003368
Train cost after epoch 1930: 0.003368
Train cost after epoch 1940: 0.003368
Train cost after epoch 1950: 0.003372
Train cost after epoch 1960: 0.003368
Train cost after epoch 1970: 0.003367
Train cost after epoch 1980: 0.003367
Train cost after epoch 1990: 0.003367
```

```
yp = test_pred # Get the predictions
yp = set(yp)

# Confusion matrix and accuracy
y_actual = pd.Series(y_ts, name='Actual')
y_pred = pd.Series(yp, name='Predicted')
confmat = pd.crosstab(y_actual, y_pred)
print(confmat)

confmat = np.asarray(confmat)
Accuracy = float(confmat[0][0]+confmat[1][1]+confmat[2][2])/float(yp.shape[0])
print('Accuracy :' + ' ' + str(Accuracy))
```

```
Predicted  1.0  2.0  3.0
Actual
1.0         17    3    0
2.0          2   13    0
3.0          2    2   24
Accuracy : 0.8571428571428571
```

2s completed at 19:38