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

```
Mounted at /content/drive
```

```
%cd drive/MyDrive/NNFL data/Data_A2/
```

```
/content/drive/MyDrive/NNFL data/Data_A2
```

```
%ls -l
```

```
total 86
-rw------- 1 root root 70617 Apr 22 07:54 data55.xlsx
-rw------- 1 root root 17039 Apr 29 07:25 data5.xlsx
```

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import os
from pprint import pprint
```

```
def sigmoid(x):
  val = 1/(1+ np.exp(-x))
  return val
```

```
def sigmoidDerivative(x):
  val =  x * (1 - x)
  return val
```

```
def perceptron(X_train_data, Y_train_data, bias, W, alpha = 0.001, epochs = 20000):

  for i in range(epochs):

    layer = np.dot(X_train_data, W)
    input = layer + bias
    output = sigmoid(input)

    error = output - Y_train_data
    derivative = sigmoidDerivative(output)
    update = error*derivative
    WNew = np.dot(X_train_data.T, update)
    W = W - alpha*WNew
    update_bias = update
    bias = bias - alpha*update

  return W, bias
def pred_eval(X, W, bias):
  layer = np.dot(X, W)
  input = layer + bias[0]
  output = sigmoid(input)
```

```
  return output


def resultQ1(filename = 'data55.xlsx'):
  dataset = pd.read_excel(filename, header = None)

  row, col = dataset.shape
  feats = col - 1

  # normalization
  dataset.loc[:, dataset.columns != feats] = (dataset.loc[:, dataset.columns != feats]-dat

  # spliting dataset into train test and val
  training_data, validation_data, testing_data = np.split(dataset.sample(frac=1),[int(0.7*

  training_data = np.array(training_data)
  validation_data = np.array(validation_data)
  testing_data = np.array(testing_data)
  training_data_X = training_data[:, :feats]
  training_data_y = training_data[:, feats]
  validation_data_X = validation_data[:, :feats]
  validation_data_y = validation_data[:, feats]
  testing_data_X = testing_data[:, :feats]
  testing_data_y = testing_data[:, feats]

  train_row, train_col = training_data_X.shape

  W = np.random.randn(train_col)
  bias = np.ones(train_row)

  W, bias = perceptron(training_data_X, training_data_y, bias, W)
  print("The Weights after training is as follows: \n")
  pprint(W)
  print("The Bias after training is as follows: ", bias[0])

  train_pred = pred_eval(training_data_X, W, bias)
  train_pred = np.where(train_pred > 0.475, 1,0)
  print("Training Accuracy: ", (np.abs(np.sum(train_pred == training_data_y))/len(training

  test_pred = pred_eval(testing_data_X, W, bias)
  test_pred = np.where(test_pred > 0.475, 1,0)
  print("Testing Accuracy: ", (np.abs(np.sum(test_pred == testing_data_y))/len(testing_dat

  validation_pred = pred_eval(validation_data_X, W, bias)
  validation_pred = np.where(validation_pred > 0.475, 1,0)
  print("Validation Accuracy: ", (np.abs(np.sum(validation_pred == validation_data_y))/len


resultQ1()
```

```
The Weights after training is as follows:

array([ 0.85795909,  1.66906918, -2.09621634,  0.28076942,  0.30669305,
       -0.97514536, -0.81478922, -0.10596437,  2.13440754,  0.84882177,
        1.55784708,  1.07581791,  0.48570685, -1.95426305, -0.06741174,
```

```
       -0.15370371,  0.56661061, -0.76339953,  0.81540497,  1.62162678,
       -0.13283423, -0.57741426,  1.26541463,  1.04988975, -0.62619035,
       -0.82056524,  0.57239927, -0.42799222,  1.42889965,  0.62555476,
       -1.07298089,  0.02928202,  0.56806728, -0.23331988, -0.58169825,
       -2.4625393 , -1.85496705,  0.00650336,  0.95276404, -0.36274683,
        0.48488188,  0.38686099,  1.75117592, -0.04029829,  0.27997382,
        0.0613423 ,  0.85331014,  0.74340836,  0.46711094, -1.88527751,
        1.20728146,  0.84608659, -0.31657515,  2.02640599,  0.33511848,
       -0.01587141,  0.41354977, -0.42870244,  0.19589594, -1.06725365])
The Bias after training is as follows:  1.0000714293080224
Training Accuracy:  0.9655172413793104
Testing Accuracy:  0.7142857142857143
Validation Accuracy:  0.6666666666666666
```

✓ 1s completed at 7:58 PM

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

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.r
```

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
np.random.seed(0)
```

```
data=pd.read_excel('/content/drive/MyDrive/NNFL data/Data_A2/data55.xlsx')
```

```
data=data.sample(frac=1,random_state=50)
```

```
dataset=np.array(data)
```

```
#splitting data into input and output
X=dataset[:,:-1]
y=dataset[:,-1]

#normalising dataset
m=X.shape[0]
xmin=np.min(X,axis=0)
xmax=np.max(X,axis=0)
X=(X-xmin)/(xmax-xmin)#performing normalization on input features
pp=np.ones([m,1])
X=np.append(pp,X,axis=1)
```

```
#splitting dataset using hold out cross validation of 70 10 and 20
X_train,X_valid,X_test=np.split(X,[int(.7*len(data)),int(.8*len(data))])
y_train,y_valid,y_test=np.split(y,[int(.7*len(data)),int(.8*len(data))])
```

```
def sigmoid(a):
  fa=(1.0/(1+np.exp(-a)))
  return fa
```

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

```
def kernel_Matrix(X, Y):
    k = (1 + np.matmul(X, Y.transpose()))**7
    return k
```

```
def rbf_kernel(x, y, sigma=2.0):
    return np.exp(-np.linalg.norm(x-y)**2 / (2 * (sigma ** 2)))
```

```
def kernel_Perceptron(X, Y, iterations,kernel_name):
    n = len(X)
    lossList = []
    itrList = []
    accuracyList = []
    alphaList = []

    alpha = np.zeros((n,))
    if(kernel_name=="kernel_Matrix"):
      kernel_Mat= kernel_Matrix(X, X)
    elif(kernel_name=="linear_kernel"):
      kernel_Mat=linear_kernel(X,Y)
    else:
      kernel_Mat=rbf_kernel(X,Y,2.0)

    itr = 0

    while (itr < iterations):
        loss=0
        for i in range(n):
            u = np.sign(np.matmul(kernel_Mat[i][:], alpha * Y))
            if (u * Y[i] <= 0):
                alpha[i] = alpha[i] + 1

        for i in range(n):
            u = np.sign(np.matmul(kernel_Mat[i][:], alpha * Y))
            if (u * Y[i] <= 0):

                loss = loss + 1
        loss = loss * 1.0/ n
        accuracy = (1-loss)*100
        itrList.append(itr+1)
        lossList.append(loss)
        accuracyList.append(accuracy)
        alphaList.append(alpha * 1)

        itr = itr + 1
    return alphaList, lossList, itrList, accuracyList, kernel_Mat


def kernel_Perceptron_valid(xtrain, ytrain, xvalid, yvalid, weightList, iterations):
    n=len(xvalid)
    itr=0
    lossList=[]
    itrList=[]
    accuracyList=[]
    kernel_Mat= kernel_Matrix(xvalid, xtrain)

    for alpha in weightList:
        loss=0
        for i in range(0,n):
            u = np.sign(np.matmul(kernel_Mat[i][:], alpha * ytrain))
            if (yvalid[i] * u <=0):
```

```
                loss = loss+1

        loss = loss * 1.0 / n
        accuracy = (1-loss)*100
        lossList.append(loss)
        itrList.append(itr+1)
        accuracyList.append(accuracy)
        itr = itr + 1
    return lossList, itrList, accuracyList

def kernel_Percep(xtrain, ytrain,kernel_name):

    maxAccuracyList = []
    if(kernel_name=="kernel_Matrix"):
      weightList, lossList, itrList, accuracyList, kernel_mat = kernel_Perceptron(xtrain,
    #Validation
    lossList, itrList, accuracyList = kernel_Perceptron_valid(xtrain, ytrain, X_valid, y_v
    maxAccur = max(accuracyList)
    maxAccuracyList.append(maxAccur)

    return maxAccuracyList


# weightList, lossList, itrList, accuracyList, kernel_mat = kernel_Perceptron(X_train, y_t
# #Validation
# lossList, itrList, accuracyList = kernel_Perceptron_valid(X_train, y_train, X_valid, y_v


maxAccuracyList = kernel_Percep(X_train, y_train)
best_alpha = max(maxAccuracyList)
print(best_alpha)


best_alpha, lossList, itrList, accuracyList, kernel_mat = kernel_Perceptron(X_train, y_tra


kernel_Mat = kernel_Matrix(X_test, X_train)
alpha_test = best_alpha[5]

def testvalues_y(X, W, k,ytest):
    n = len(X_test)
    predicted_Y= []
    for i in range(n):
        u = np.sign(np.matmul(kernel_Mat[i][:], alpha_test * y_train))
        if u < 0:
            u=0
        predicted_Y.append(u)
        print("Predicted: {0}, Actual: {1}".format(int(u),int(y_test[i])))

    return predicted_Y

#get prediction values for test file
y_pred_test= testvalues_y(X_test, alpha_test, kernel_Mat,y_test)
```

```
Predicted: 1, Actual: 0
Predicted: 1, Actual: 1
```

```
Predicted: 1, Actual: 0
Predicted: 1, Actual: 1
Predicted: 1, Actual: 0
Predicted: 1, Actual: 0
Predicted: 1, Actual: 1
Predicted: 1, Actual: 0
Predicted: 1, Actual: 0
Predicted: 1, Actual: 1
Predicted: 1, Actual: 1
Predicted: 1, Actual: 1
Predicted: 1, Actual: 0
Predicted: 1, Actual: 1
Predicted: 1, Actual: 1
Predicted: 1, Actual: 0
Predicted: 1, Actual: 0
Predicted: 1, Actual: 1
Predicted: 1, Actual: 1
Predicted: 1, Actual: 0
Predicted: 1, Actual: 0
Predicted: 1, Actual: 0
Predicted: 1, Actual: 0
Predicted: 1, Actual: 1
Predicted: 1, Actual: 1
Predicted: 1, Actual: 1
Predicted: 1, Actual: 0
Predicted: 1, Actual: 0
Predicted: 1, Actual: 1
Predicted: 1, Actual: 1
Predicted: 1, Actual: 0
Predicted: 1, Actual: 0
Predicted: 1, Actual: 0
Predicted: 1, Actual: 1
Predicted: 1, Actual: 0
Predicted: 1, Actual: 1
Predicted: 1, Actual: 0
Predicted: 1, Actual: 1
Predicted: 1, Actual: 0
Predicted: 1, Actual: 1
Predicted: 1, Actual: 1
Predicted: 1, Actual: 1
```

```
def sas(ypred, ytest):
    tp = tn = fp = fn = 0 #initialising true positive, true negative, false positive and f
    m = ytest.shape[0]
    for i in range(m):
        if ypred[i] == 1:
            if ytest[i] == 1:
                tp+=1
            else:
                fp+=1
        elif ypred[i] == -1:
            if ytest[i] == 0:
                tn+=1
            else:
                fn+=1
    se = tp/(tp+fn)
    sp = tn/(tn+fp)
    ac = (tn+tp)/m
```

```
    print("Sensitivity: {0}, Specificity: {1} and Accuracy: {2}".format(se,sp,ac))

sas(y_pred_test,y_test)
```

```
Sensitivity: 1.0, Specificity: 0.0 and Accuracy: 0.5
```

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

```
Mounted at /content/drive
```

```
import pandas as pd
import numpy as np
import scipy.io as sio
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
```

```
data=pd.read_excel('/content/drive/MyDrive/NNFL data/Data_A2/data5.xlsx')
```

```
data[data.columns[-1]].unique()
```

```
array([1., 2., 3.])
```

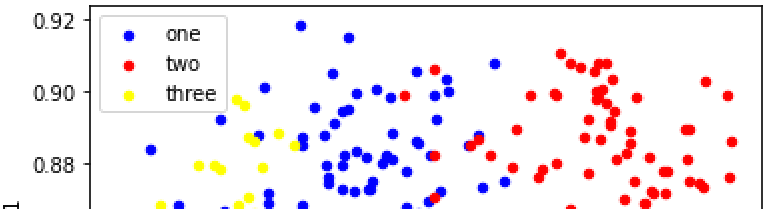
```
data=data.sample(frac=1)
data.head()
```

	15.260	14.840	0.871	5.763	3.312	2.221	5.220	1.000
51	14.49	14.61	0.8538	5.715	3.113	4.116	5.396	1.0
83	19.51	16.71	0.8780	6.366	3.801	2.962	6.185	2.0
96	18.98	16.57	0.8687	6.449	3.552	2.144	6.453	2.0
3	16.14	14.99	0.9034	5.658	3.562	1.355	5.175	1.0
173	10.83	12.96	0.8099	5.278	2.641	5.182	5.185	3.0

```
class_one = data[data.iloc[:,-1]==1][0:210]
class_two = data[data.iloc[:,-1]==2][0:210]
class_three = data[data.iloc[:,-1]==3][0:210]
```

```
axes = class_one.plot(kind='scatter', x=1, y=2, color='blue', label='one')
class_two.plot(kind='scatter', x=1, y=2, color='red', label='two', ax=axes)
class_three.plot(kind='scatter', x=1, y=2, color='yellow', label='three', ax=axes)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fd3c9dd1950>
```



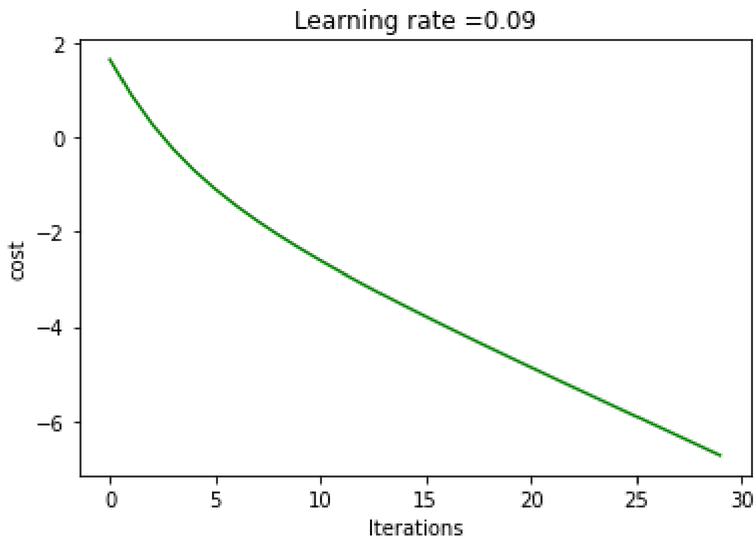

```
def sigmoid(Z):
    return 1/(1+np.exp(-Z)),Z
```

```
def relu(Z):
    A = np.maximum(0,Z)
    return Z,A


def relu_backward(dA, cache):
    Z = cache
    dZ = np.array(dA, copy=True)
    dZ[Z <= 0] = 0
    return dZ


def sigmoid_backward(dA, cache):
    Z = cache
    s = 1/(1+np.exp(-Z))
    dZ = dA * s * (1-s)
    return dZ


def initialize_parameters_deep(layer_dims):
    parameters = {}
    L = len(layer_dims)
    for l in range(1, L):
        parameters['W' + str(l)] = np.random.randn(layer_dims[l],layer_dims[l-1])
        parameters['b' + str(l)] = np.zeros((layer_dims[l],1))
    return parameters


def linear_forward(A,W,b):
    Z = np.dot(W,A)+b
    cache = (A, W, b)
    return Z,cache


def linear_activation_forward(A_prev,W,b,activation):

    if activation == "sigmoid":
        Z, linear_cache = linear_forward(A_prev,W,b)
        A, activation_cache = sigmoid(Z)

    elif activation == "relu":
        Z, linear_cache = linear_forward(A_prev,W,b)
        A, activation_cache = relu(Z)

    cache = (linear_cache, activation_cache)
```

```
    return A,cache



def L_model_forward(X,parameters):
    caches = []
    A = X
    L = len(parameters) //2
    for l in range(1, L):
        A_prev = A
        A, cache = linear_activation_forward(A,parameters['W'+str(l)],parameters['b'+str(l
        caches.append(cache)

    AL, cache = linear_activation_forward(A,parameters['W'+str(L)],parameters['b'+str(L)],
    caches.append(cache)

    return AL,caches


def compute_cost(AL,Y):
    m = Y.shape[1]
    cost = -(1/m)*np.sum(Y*np.log(AL)+(1-Y)*np.log(1-AL))
    cost = np.squeeze(cost)
    return cost


def linear_backward(dZ, cache):
    A_prev, W, b = cache
    m = A_prev.shape[1]
    dW = (1/m)*np.dot(dZ,A_prev.T)
    db = (1/m)*np.sum(dZ,axis=1,keepdims=True)
    dA_prev = np.dot(W.T,dZ)

    return dA_prev, dW, db


def linear_activation_backward(dA, cache, activation):
    linear_cache, activation_cache = cache

    if activation == "relu":
        dZ = relu_backward(dA,activation_cache)
        dA_prev, dW, db = linear_backward(dZ,linear_cache)

    elif activation == "sigmoid":
        dZ = sigmoid_backward(dA,activation_cache)
        dA_prev, dW, db = linear_backward(dZ,linear_cache)

    return dA_prev, dW, db


def L_model_backward(AL, Y, caches):
    grads = {}
    L = len(caches)
    m = AL.shape[1]
    Y = Y.reshape(AL.shape)
```

```
    dAL = -(np.divide(Y,AL)-np.divide(1-Y,1-AL))

    current_cache = caches[L-1]
    grads["dA" + str(L-1)], grads["dW" + str(L)], grads["db" + str(L)] = linear_activation_

    for l in reversed(range(L-1)):
        current_cache = caches[l]
        dA_prev_temp, dW_temp, db_temp = linear_activation_backward(grads["dA"+str(l+1)],c
        grads["dA" + str(l)] = dA_prev_temp
        grads["dW" + str(l + 1)] = dW_temp
        grads["db" + str(l + 1)] = db_temp

    return grads


def update_parameters(parameters, grads, learning_rate):

    L = len(parameters)//2
    for l in range(L):
        parameters["W" + str(l+1)] = parameters["W" + str(l+1)]-learning_rate*grads["dW"+s
        parameters["b" + str(l+1)] = parameters["b" + str(l+1)]-learning_rate*grads["db"+s

    return parameters


def L_layer_model(X, Y, layers_dims, learning_rate, num_iterations, print_cost=False):

    costs = []

    parameters = initialize_parameters_deep(layers_dims)

    for i in range(0, num_iterations):

        AL, caches = L_model_forward(X,parameters)
        cost = compute_cost(AL,Y)
        grads = L_model_backward(AL,Y,caches)
        parameters = update_parameters(parameters,grads,learning_rate)
        costs.append(cost)

    plt.plot(np.squeeze(costs),'g')
    plt.ylabel('cost')
    plt.xlabel('Iterations')
    plt.title("Learning rate =" + str(learning_rate))
    plt.show()

    return parameters,costs


X=data.iloc[:,:-1].values
Y=data.iloc[:,-1].values
xmin=np.min(X,axis=0)
xmax=np.max(X,axis=0)
X=(X-xmin)/(xmax-xmin)#performing normalization on input features
m=X.shape[0]
pp=np.ones([m,1])
```

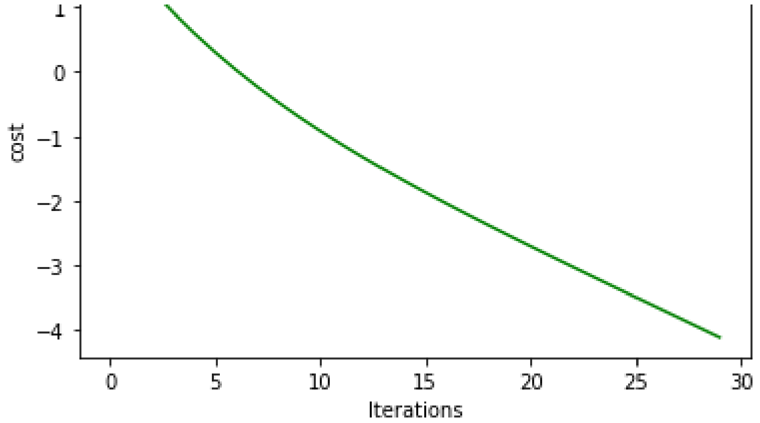
```
pp=np.ones([m,1])
X=np.append(pp,X,axis=1)
train_x,test_x,train_y,test_y=train_test_split(X,Y,test_size=0.2,train_size=0.8,shuffle=Tr
train_x,valid_x,train_y,valid_y=train_test_split(train_x,train_y,test_size=0.125,train_siz
train_x=train_x.T
train_y=train_y.T
train_y=np.reshape(train_y,newshape=(1,train_y.shape[0]))
layers_dims = [8,4,1]
parameters,costs = L_layer_model(train_x, train_y, layers_dims, 0.09, 30, print_cost = Tru
```



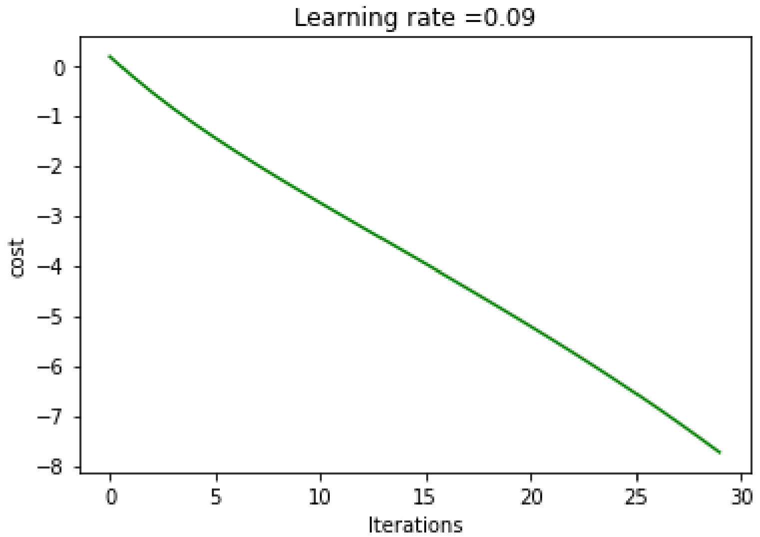

```
def accuracy(parameters,test_x,test_acc):
    p,l=L_model_forward(test_x.T,parameters)
    p=(p>0.5).astype(int)
    a=test_acc-p
    a=np.sum((a!=0).astype(int))
    return (p.shape[1]-a)/p.shape[1]
```

```
kf = KFold(n_splits=5)
X=data.iloc[:,:-1].values
X=(X-np.mean(X,axis=0))/(np.std(X,axis=0))
m=X.shape[0]
pp=np.ones([m,1])
X=np.append(pp,X,axis=1)

Y=data.iloc[:,-1].values
Y=np.reshape(Y,newshape=(-1,1))
layers_dims = [8,4,1]

kf.get_n_splits(X)
fold=0
acc=0
overall=0
for train_index, test_index in kf.split(X):
    fold+=1
    X_train=X[train_index]
    Y_train=Y[train_index]
    X_train=X_train.T
    Y_train=Y_train.T
```

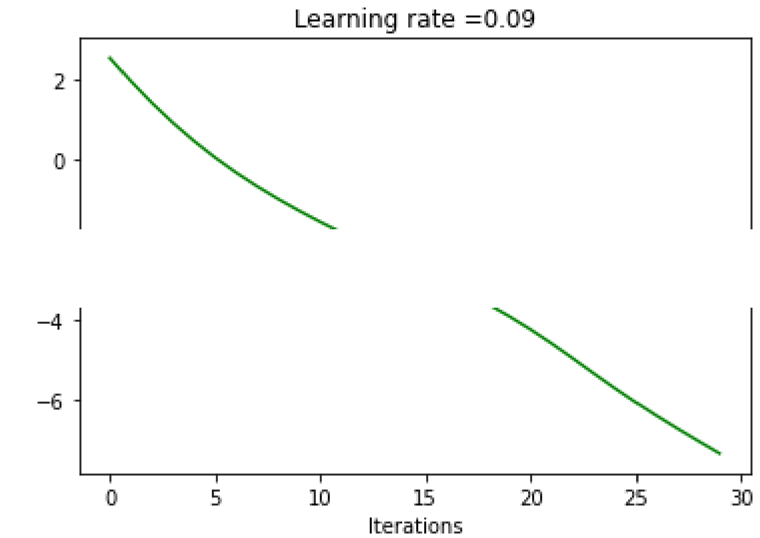
```
        Y_train=np.reshape(Y_train,newshape=(1,Y_train.shape[1]))
        X_test=X[test_index]
        Y_test=Y[test_index]
        parameters,costs = L_layer_model(X_train, Y_train, layers_dims, 0.09, 30, print_cost =
        acc=accuracy(parameters,X_test,Y_test)
        overall+=acc
        print("Fold: {0}, Accuracy: {1}%".format(fold,round(acc*100,2)))
    print("overall accuracy is for is:",round(overall/5*100,2))
```



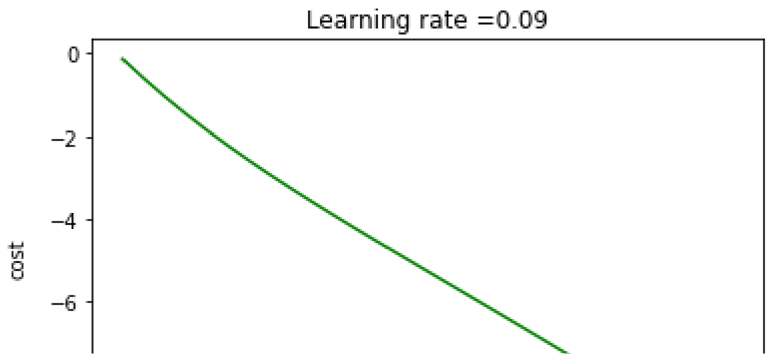
Fold: 1, Accuracy: -2700.0%



Fold: 2, Accuracy: -2800.0%



Fold: 3, Accuracy: -2700.0%



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

```
    Mounted at /content/drive
```

```
import pandas as pd
import numpy as np
import scipy.io as sio
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
```

```
data=pd.read_excel('/content/drive/MyDrive/NNFL data/Data_A2/data5.xlsx')
```

```
data.dropna(inplace=True)
```

```
data[data.columns[-1]].unique()
```

```
    array([1., 2., 3.])
```

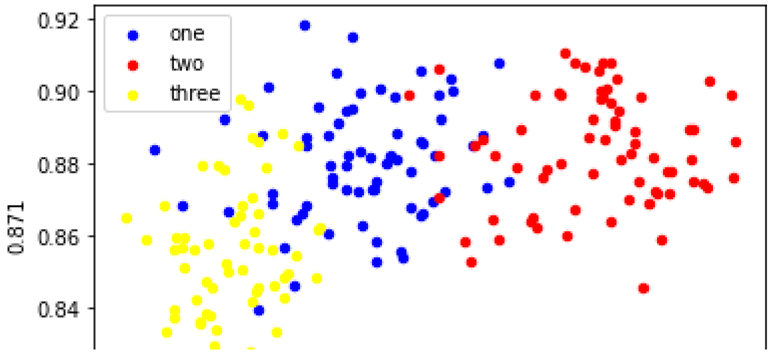
```
data=data.sample(frac=1)
data.head()
```

	15.260	14.840	0.871	5.763	3.312	2.221	5.220	1.000
131	15.38	14.90	0.8706	5.884	3.268	4.462	5.795	2.0
147	12.70	13.71	0.8491	5.386	2.911	3.260	5.316	3.0
44	13.80	14.04	0.8794	5.376	3.155	1.560	4.961	1.0
148	10.79	12.93	0.8107	5.317	2.648	5.462	5.194	3.0
26	12.74	13.67	0.8564	5.395	2.956	2.504	4.869	1.0

```
class_one = data[data.iloc[:,-1]==1][0:210]
class_two = data[data.iloc[:,-1]==2][0:210]
class_three = data[data.iloc[:,-1]==3][0:210]
```

```
axes = class_one.plot(kind='scatter', x=1, y=2, color='blue', label='one')
class_two.plot(kind='scatter', x=1, y=2, color='red', label='two', ax=axes)
class_three.plot(kind='scatter', x=1, y=2, color='yellow', label='three', ax=axes)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fdb44b82310>
```




```
def clustering(k):
    kmeans=KMeans(n_clusters=k).fit(train_x)
    return kmeans


def kernel(x,mu,sigma,kernel_funct):
    beta=1/(2*sigma*sigma)
    if kernel_funct == "gaussian":
        return np.exp(-beta*(np.linalg.norm(x-mu))**2)
    elif kernel_funct == "multi-quadric":
        return ((np.linalg.norm(x-mu))**2+sigma**2)**0.5
    elif kernel_funct == "linear":
        return np.linalg.norm(x-mu)


def compute_sigma(x,labels,mu):
    c=mu.shape[0]
    sigma=np.zeros(c)
    for i in range(c):
        x_temp=x[labels==i]
        k=0
        for j in range(x_temp.shape[0]):
            k+=np.linalg.norm(x_temp[j]-mu[i])
        sigma[i]=k/x_temp.shape[0]
    return sigma


def compute_H(X,mu,sigma,kernel_funct):
    c=mu.shape[0]
    H=np.zeros((X.shape[0],c))
    for i in range(H.shape[0]):
        for j in range(H.shape[1]):
            H[i][j]=kernel(X[i],mu[j],sigma[j],kernel_funct)
    return H


X=data.iloc[:,:-1].values
Y=data.iloc[:,-1].values
xmin=np.min(X,axis=0)
xmax=np.max(X,axis=0)
X=(X-xmin)/(xmax-xmin)#performing normalization on input features
# m=X.shape[0]
# pp=np.ones([m,1])
# X=np.append(pp,X,axis=1)
train_x,test_x,train_y,test_y=train_test_split(X,Y,test_size=0.2,train_size=0.8,shuffle=Tr
```

```
train_x,valid_x,train_y,valid_y=train_test_split(train_x,train_y,test_size=0.125,train_siz
train_y=np.reshape(train_y,newshape=(train_y.shape[0],1))
valid_y=np.reshape(valid_y,newshape=(valid_y.shape[0],1))
test_y=np.reshape(test_y,newshape=(test_y.shape[0],1))
```

```
def compute(kernel_funct,xtrain,ytrain,xtest,ytest):
    kmeans=clustering(15)
    mu=kmeans.cluster_centers_
    sigma=compute_sigma(xtrain,kmeans.labels_,mu)
    H=compute_H(xtrain,mu,sigma,kernel_funct)
    W=np.dot(np.linalg.pinv(H),ytrain)
    H=compute_H(xtest,mu,sigma,kernel_funct)
    pred=np.dot(H,W)
    p=(pred>0.5).astype(int)
    a=(p!=ytest).astype(int)

    return (ytest.shape[0]-np.sum(a))/ytest.shape[0]


muquad_acc = compute("multi-quadric",train_x,train_y,test_x,test_y)
lin_acc = compute("linear",train_x,train_y,test_x,test_y)
gaussian_acc=compute("gaussian",train_x,train_y,test_x,test_y)


print("Accuracy for Multi quadric kernel is: {0}%".format(round(muquad_acc*100,2)))
print("Accuracy for Linear kernel is: {0}%".format(round(lin_acc*100,2)))
print("Accuracy for Gaussian kernel is: {0}%".format(round(gaussian_acc*100,2)))
```

```
Accuracy for Multi quadric kernel is: 40.48%
Accuracy for Linear kernel is: 40.48%
Accuracy for Gaussian kernel is: 38.1%
```

```
kf = KFold(n_splits=5)
X=data.iloc[:,:-1].values
X=(X-np.mean(X,axis=0))/(np.std(X,axis=0))
Y=data.iloc[:,-1].values
Y=np.reshape(Y,newshape=(-1,1))
kf.get_n_splits(X)
fold = 0
accuracy = 0
overall_mq = 0
overall_lin = 0
overall_gauss=0
for train_index, test_index in kf.split(X):
    fold+=1
    X_train=X[train_index]
    Y_train=Y[train_index]
    X_test=X[test_index]
    Y_test=Y[test_index]
    mquad_acc_val = compute("multi-quadric",train_x,train_y,test_x,test_y)
    lin_acc_val = compute("linear",train_x,train_y,test_x,test_y)
    gaussian_acc_val=compute("gaussian",train_x,train_y,test_x,test_y)
    overall_mq+=mquad_acc_val
```

```
        overall_lin+=lin_acc_val
        overall_gauss+=gaussian_acc_val
        print("Fold: {0}, Accuracy for multi quadratic kernel: {1}%".format(fold,round(mquad_a
        print("Fold: {0}, Accuracy for linear kernel: {1}%".format(fold,round(lin_acc_val*100,
        print("Fold: {0}, Accuracy for gaussian kernel: {1}%".format(fold,round(gaussian_acc_v
    print("overall accuracy is for Multiquad is: {0}%".format(round(overall_mq/5*100,2)))
    print("overall accuracy is for linear  is: {0}%".format(round(overall_lin/5*100,2)))
    print("overall accuracy is for Gaussian  is: {0}%".format(round(overall_gauss/5*100,2)))
```

```
Fold: 1, Accuracy for multi quadratic kernel: 40.48%
Fold: 1, Accuracy for linear kernel: 40.48%
Fold: 1, Accuracy for gaussian kernel: 38.1%
Fold: 2, Accuracy for multi quadratic kernel: 40.48%
Fold: 2, Accuracy for linear kernel: 40.48%
Fold: 2, Accuracy for gaussian kernel: 35.71%
Fold: 3, Accuracy for multi quadratic kernel: 40.48%
Fold: 3, Accuracy for linear kernel: 40.48%
Fold: 3, Accuracy for gaussian kernel: 35.71%
Fold: 4, Accuracy for multi quadratic kernel: 40.48%
Fold: 4, Accuracy for linear kernel: 40.48%
Fold: 4, Accuracy for gaussian kernel: 38.1%
Fold: 5, Accuracy for multi quadratic kernel: 40.48%
Fold: 5, Accuracy for linear kernel: 40.48%
Fold: 5, Accuracy for gaussian kernel: 33.33%
overall accuracy is for Multiquad is: 40.48%
overall accuracy is for linear  is: 40.48%
overall accuracy is for Gaussian  is: 36.19%
```

✓ 0s completed at 8:09 PM

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

```
Mounted at /content/drive
```

+ Code + Text

```
# Changing directory to the directory containing dataset
%cd drive/MyDrive/NNFL data/Data_A2/
```

```
/content/drive/MyDrive/NNFL data/Data_A2
```

```
%ls -l
```

```
total 87
-rw------- 1 root root   259 Apr 29 07:23 class_label.mat
-rw------- 1 root root 70617 Apr 22 07:54 data55.xlsx
-rw------- 1 root root 17039 Apr 29 07:25 data5.xlsx
```

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import os
from pprint import pprint
```

```
import warnings
warnings.filterwarnings('ignore')
```

```
def sigmoidFuntion(Z):
    return 1/(1+np.exp(-Z)),Z

def relu(Z):
    A = np.maximum(0,Z)
    return (Z,A)

def reluBack(dA, cache):
    Z = cache

    dZ = np.array(dA, copy=True)
    dZ[Z <= 0] = 0

    return dZ

def sigmoidBack(dA, cache):
    Z = cache

    s = 1/(1+np.exp(-Z))
    dZ = dA * s * (1-s)

    return dZ

def initializeParams(layerDimensions, para, stack):
```

```
    parameters = {}
    L = len(layerDimensions)
    for l in range(1, L-1):
        if(stack==False):
            parameters['W' + str(l)] = np.random.randn(layerDimensions[l], layerDimensions
        else:
            parameters['W' + str(l)] = para[l-1]['W1']
        parameters['b' + str(l)] = np.zeros((layerDimensions[l], 1))
    parameters['W' + str(L-1)] = np.random.randn(layerDimensions[L-1], layerDimensions[L-2
    parameters['b' + str(L-1)] = np.zeros((layerDimensions[L-1],1))
    return parameters

def linearForward(A, W, bias):
    Z = np.dot(W, A) + bias
    cache = (A, W, bias)
    return Z, cache

def linearActivationForward(A_prev, W, bias, activation):

    if(activation == "sigmoid"):
        Z, linear_cache = linearForward(A_prev, W, bias)
        A, activation_cache = sigmoidFuntion(Z)

    elif(activation == "relu"):
        Z, linear_cache = linearForward(A_prev, W, bias)
        A, activation_cache = relu(Z)

    cache = (linear_cache, activation_cache)

    return A, cache

def forwardModelL(X, params):
    caches = []
    A = X
    L = (len(params) //2)
    for l in range(1, L):
        A_prev = A
        A, cache = linearActivationForward(A, params['W'+str(l)], params['b'+str(l)], "sig
        caches.append(cache)

    AL, cache = linearActivationForward(A, params['W'+str(L)], params['b'+str(L)], "sigmoi
    caches.append(cache)

    return AL,caches

def costComputation(AL, Y):
    m = Y.shape[1]
    cost = -(1/m)*np.sum(Y*np.log(AL)+(1-Y)*np.log(1-AL))
    cost = np.squeeze(cost)
    return cost

def costComputationAutoencoder(AL, Y, parameters):
    m = Y.shape[1]
    cost = (1/(2*m))*np.sum(np.linalg.norm(AL-Y))
    L = (len(parameters) //2)
```

```
        return cost

    def linearBack(dZ, cache):
        A_prev, W, _ = cache
        _, m = A_prev.shape

        dW = (1/m)*np.dot(dZ, A_prev.T)
        db = (1/m)*np.sum(dZ, axis=1, keepdims=True)
        dAP = np.dot(W.T, dZ)

        return dAP, dW, db

    def linearActivationBack(dA, cache, activation):
        linearCache, activationCache = cache

        if (activation == "relu"):
            dZ = reluBack(dA, activationCache)
            dAP, dW, db = linearBack(dZ, linearCache)

        elif (activation == "sigmoid"):
            dZ = sigmoidBack(dA, activationCache)
            dAP, dW, db = linearBack(dZ, linearCache)

        return dAP, dW, db

    def modelLBack(AL, Y, caches):
        gradients = {}
        L = len(caches)
        m = AL.shape[1]
        Y = Y.reshape(AL.shape)

        dAL = -(np.divide(Y,AL)-np.divide(1-Y,1-AL))

        current_cache = caches[L-1]
        gradients["dA" + str(L-1)], gradients["dW" + str(L)], gradients["db" + str(L)] = linea

        for l in reversed(range(L-1)):

            current_cache = caches[l]
            dA_prev_temp, dW_temp, db_temp = linearActivationBack(gradients["dA"+str(l+1)],cur
            gradients["dA" + str(l)] = dA_prev_temp
            gradients["dW" + str(l + 1)] = dW_temp
            gradients["db" + str(l + 1)] = db_temp

        return gradients

    def updateParams(parameters, gradients, learning_rate):

        L = (len(parameters) //2)

        for l in range(L):
            parameters["W" + str(l+1)] = parameters["W" + str(l+1)]-learning_rate*grads["dW"+s
            parameters["b" + str(l+1)] = parameters["b" + str(l+1)]-learning_rate*grads["db"+s
```

```
    return parameters

def layerLModel(X, Y, layers_dims, num_iterations,stack, learning_rate = 0.1):

    costs = []

    parameters = initializeParams(layers_dims,para,stack)

    for i in range(0, num_iterations):

        AL, caches = forwardModelL(X,parameters)

        if(stack==True):
            cost = costComputation(AL, Y)
        else:
            cost = costComputationAutoencoder(AL, Y, parameters)

        grads = modelLBack(AL,Y,caches)

        parameters = updateParams(parameters,grads,learning_rate)

        if(i%100==0):
            print ("Cost after iteration %i: %f" %(i, cost))
        costs.append(cost)
    print(cost)

    return parameters, costs

dataset = pd.read_excel('/content/drive/MyDrive/NNFL data/Data_A2/data55.xlsx')

row, col = dataset.shape
feats = col - 1

# normalization
dataset.loc[:, dataset.columns != feats] = (dataset.loc[:, dataset.columns != feats]-datas

# spliting dataset into train test and val
training_data, validation_data, testing_data = np.split(dataset.sample(frac=1),[int(0.7*le

training_data = np.array(training_data)
validation_data = np.array(validation_data)
testing_data = np.array(testing_data)
training_data_X = training_data[:, :feats]
training_data_y = training_data[:, feats]
validation_data_X = validation_data[:, :feats]
validation_data_y = validation_data[:, feats]
testing_data_X = testing_data[:, :feats]
testing_data_y = testing_data[:, feats]

train_row, train_col = training_data_X.shape


para = []
lr= 0.005
```

```
layerDemensions = [72,64,72]
params, costs = layerLModel(training_data_X, training_data_y, layerDemensions, num_iterati
paramsAE1 = params

W1=paramsAE1['W1']
b1=paramsAE1['b1']
X_new,_=linearActivationForward(X.T,W1,b1,"sigmoid")
X_new.shape


layerDemensions = [64,16,64]
params, costs = layerLModel(training_data_X, training_data_y, layerDemensions,num_iteratio
paramsAE2 = params

W1 = paramsAE2['W1']
b1 = paramsAE2['b1']
X_new, _ =  linearActivationForward(X.T, W1, b1, "sigmoid")


layerDemensions = [16,4,16]
params, costs = layerLModel(training_data_X, training_data_y, layerDemensions, learning_ra
paramsAE3= params


para=[paramsCopy, paramsAE2, paramsAE3]

layerDemensions = [72,64,16,4,1]
params, costs = layerLModel(training_data_X, training_data_y, layerDemensions,num_iteratio

p, l = forwardModelL(training_data_X, params)
p = (p>0.5).astype(int)
a = training_data_y-p
a = np.sum((a!=0).astype(int))
accuracy = (p.shape[1]-a)/p.shape[1]
print('Testing Accuracy: ')
metrics(p, data_testing_y)
print()
print('Validation Accuracy: ')
metrics(p, data_validation_y)
```

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

```
Mounted at /content/drive
```

```
# Changing directory to the directory containing dataset
%cd drive/MyDrive/NNFL data/Data_A2/
```

```
/content/drive/MyDrive/NNFL data/Data_A2
```

```
%ls -l
```

```
total 87
-rw------- 1 root root   259 Apr 29 07:23 class_label.mat
-rw------- 1 root root 70617 Apr 22 07:54 data55.xlsx
-rw------- 1 root root 17039 Apr 29 07:25 data5.xlsx
```

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import os
from pprint import pprint
```

```
import warnings
warnings.filterwarnings('ignore')
```

```
from sklearn.preprocessing import OneHotEncoder
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold

df = pd.read_excel('data5.xlsx', header = None)
df = df.sample(frac=1).reset_index(drop=True)
def sigmoidFunction(Z):
    return (1/(1+np.exp(-Z)), Z)

para=[]
def act1(x, a, b, act):
    if (act == "gaussian"):
        return np.exp(-b*np.linalg.norm(x-a))
    elif (act == "tanh"):
        num = 1-np.exp(-(np.dot(x.T,a)+b))
        den = 1+np.exp(-(np.dot(x.T,a)+b))
        return (num/den)

def init(hiddenLayer, dimensions):
    a = []
    b = []
    for i in range(hiddenLayer):
        a.append(np.random.rand(dimensions,1))
        b.append(np.random.rand(1))
    return (a,b)
```

```
def one_hot(y):
    onehotencoder = OneHotEncoder()
    y = onehotencoder.fit_transform(y).toarray()
    return y

def compute(hiddenLayer, train_x, test_x, train_y, test_y, act):

    Y_enc = one_hot(train_y)
    H = np.zeros((train_x.shape[0],hiddenLayer))
    for i in range(H.shape[1]):
        for j in range(H.shape[0]):
            H[j][i]=act1(train_x[j],a[i],b[i],act)
    W = np.dot(np.linalg.pinv(H),Y_enc)

    H = np.zeros((test_x.shape[0],hiddenLayer))
    for i in range(H.shape[1]):
        for j in range(H.shape[0]):
            H[j][i] = act1(test_x[j],a[i],b[i],act)

    p = np.dot(H,W)
    p = np.argmax(p,axis=1)
    p = np.reshape(p,newshape=(p.shape[0],1))
    accuracy = test_y-p
    accuracy = np.sum((accuracy!=0).astype(int))
    return (p.shape[0]-accuracy)/p.shape[0]

l = 256


kf = KFold(n_splits = 5)
X = df.iloc[:, 0:7].values
X = (X - np.mean(X, axis=0))/(np.std(X, axis=0))
Y = df.iloc[:,7].values
Y = np.reshape(Y, newshape=(-1,1))
a, b = init(l, X.shape[1])
kf.get_n_splits(X)
fold = 0
acctemp = 0
overall = 0
for train_index, test_index in kf.split(X):
    fold+=1
    training_data_X = X[train_index]
    training_data_Y = Y[train_index]
    testing_data_X = X[test_index]
    testing_data_Y = Y[test_index]
    acctemp = compute(l, training_data_X, testing_data_X, training_data_Y, testing_data_Y,
    overall+=acctemp
    print("Fold: ", fold, "Accuracy: ", acctemp)
print("Overall Accuracy (tanh) : ", overall/5, '\n')

kf = KFold(n_splits=5)
X=df.iloc[:,0:7].values
X=(X-np.mean(X,axis=0))/(np.std(X,axis=0))
Y=df.iloc[:,7].values
```

```
Y=np.reshape(Y,newshape=(-1,1))
a,b=init(l,X.shape[1])
kf.get_n_splits(X)
fold=0
accuracy=0
overall=0
for train_index, test_index in kf.split(X):
    fold+=1
    training_data_X = X[train_index]
    training_data_Y = Y[train_index]
    testing_data_X = X[test_index]
    testing_data_Y = Y[test_index]
    accuracy = compute(l,training_data_X,testing_data_X,training_data_Y,testing_data_Y,"ga
    overall+= accuracy
    print("Fold: ",fold," Accuracy: ", accuracy)
print("Overall Accuracy (Gaussian) : " , overall/5)
```

```
Fold:  1 Accuracy:  0.023809523809523808
Fold:  2 Accuracy:  0.023809523809523808
Fold:  3 Accuracy:  0.023809523809523808
Fold:  4 Accuracy:  0.0
Fold:  5 Accuracy:  0.047619047619047616
Overall Accuracy (tanh) :  0.023809523809523808

Fold:  1  Accuracy:  0.11904761904761904
Fold:  2  Accuracy:  0.07142857142857142
Fold:  3  Accuracy:  0.14285714285714285
Fold:  4  Accuracy:  0.09523809523809523
Fold:  5  Accuracy:  0.047619047619047616
Overall Accuracy (Gaussian) :  0.09523809523809523
```

✓ 8s completed at 8:44 PM

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

```
Mounted at /content/drive
```

```
# Changing directory to the directory containing dataset
%cd drive/MyDrive/NNFL data/Data_A2/
```

```
/content/drive/MyDrive/NNFL data/Data_A2
```

```
%ls -l
```

```
total 87
-rw------- 1 root root   259 Apr 29 07:23 class_label.mat
-rw------- 1 root root 70617 Apr 22 07:54 data55.xlsx
-rw------- 1 root root 17039 Apr 29 07:25 data5.xlsx
```

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import os
from pprint import pprint
```

```
import warnings
warnings.filterwarnings('ignore')
```

```
def metrics(Y_true, Y_pred):
    FP=0 # For counting the False Positives
    FN=0 # For counting the False Negatives
    TN=0 # For counting the True Negatives
    TP=0 # For counting the True Positives

    for i in range(len(Y_true)):
      if Y_true[i]==1:
        if Y_pred[i]==1:
          TP+=1
        else:
          FN+=1
      else:
        if Y_pred[i]==0:
          TN+=1
        else:
          FP+=1

    print('{}'.format('-'*75))

    sens= TP/(TP+FN)
    spes = TN/(TN+FP)

    print("Sensitivity : ", sens)
```

```
    print("Specificity : ", spes)
    print("Accuracy ((TN+TP)/(TN+TP+FN+FP)) : ", ((TP+TN)/(TN+FN+TP+FP)))

def sigmoidFunction(Z):
    return 1/(1+np.exp(-Z)),Z

def relu(Z):
    A = np.maximum(0,Z)
    return Z,A

def reluBackward(dA, cache):
    Z = cache

    dZ = np.array(dA, copy=True)
    dZ[Z <= 0] = 0

    return dZ

def sigmoidBackward(dA, cache):
    Z = cache

    s = 1/(1+np.exp(-Z))
    dZ = dA * s * (1-s)

    return dZ

def initializeNewLayer(layer_dims,para,stack):
    parameters = {}
    L = len(layer_dims)
    for l in range(1, L-1):
        if stack==False:
            parameters['W' + str(l)] = np.random.randn(layer_dims[l],layer_dims[l-1])
        else:
            parameters['W' + str(l)]=para[l-1]['W1']
        parameters['b' + str(l)] = np.zeros((layer_dims[l],1))
    parameters['W' + str(L-1)] = np.random.randn(layer_dims[L-1],layer_dims[L-2])
    parameters['b' + str(L-1)] = np.zeros((layer_dims[L-1],1))
    return parameters

def linearForward(A, W, b):
    Z = np.dot(W,A)+b
    cache = (A, W, b)
    return Z,cache

def linearActivationFunction(A_prev, W, b, activation):

    if(activation == "sigmoid"):
        Z, linear_cache = linearForward(A_prev,W,b)
        A, activation_cache = sigmoidFunction(Z)

    elif(activation == "relu"):
        Z, linear_cache = linearForward(A_prev,W,b)
        A, activation_cache = relu(Z)

    cache = (linear_cache, activation_cache)
```

```
        return A, cache

    def modelLForward(X, params):
        caches = []
        A = X
        L = (len(params) //2)
        for l in range(1, L):
            A_prev = A
            A, cache = linearActivationFunction(A, params['W'+str(l)], params['b'+str(l)],"sig
            caches.append(cache)

        AL, cache = linearActivationFunction(A, params['W'+str(L)], params['b'+str(L)],"sigmoi
        caches.append(cache)

        return AL,caches

    def costComputation(AL,Y):
        m = Y.shape[1]
        cost = -(1/m)*np.sum(Y*np.log(AL)+(1-Y)*np.log(1-AL))
        cost = np.squeeze(cost)
        return cost

    def costComputationAE(AL,Y,parameters):
        m=Y.shape[1]
        cost=(1/(2*m))*np.sum(np.linalg.norm(AL-Y))
        L = len(parameters) //2
        return cost

    def linearBack(dZ, cache):
        A_prev, W, b = cache
        m = A_prev.shape[1]

        dW = (1/m)*np.dot(dZ,A_prev.T)
        db = (1/m)*np.sum(dZ,axis=1,keepdims=True)
        dA_prev = np.dot(W.T,dZ)

        return dA_prev, dW, db

    def linearActivation(dA, cache, activation):
        linear_cache, activation_cache = cache

        if activation == "relu":
            dZ = reluBackward(dA,activation_cache)
            dA_prev, dW, db = linearBack(dZ,linear_cache)

        elif activation == "sigmoid":
            dZ = sigmoidBackward(dA,activation_cache)
            dA_prev, dW, db = linearBack(dZ,linear_cache)

        return dA_prev, dW, db

    def backModelL(AL, Y, caches):
        grads = {}
        L = len(caches)
```

```
    m = AL.shape[1]
    Y = Y.reshape(AL.shape)

    dAL = -(np.divide(Y,AL)-np.divide(1-Y,1-AL))

    current_cache = caches[L-1]
    grads["dA" + str(L-1)], grads["dW" + str(L)], grads["db" + str(L)] = linearActivation(

    for l in reversed(range(L-1)):

        current_cache = caches[l]
        dA_prev_temp, dW_temp, db_temp = linearActivation(grads["dA"+str(l+1)],current_cac
        grads["dA" + str(l)] = dA_prev_temp
        grads["dW" + str(l + 1)] = dW_temp
        grads["db" + str(l + 1)] = db_temp

    return grads

def updateParams(params, grads, learning_rate):

    L = len(params) //2

    for l in range(L):
        params["W" + str(l+1)] = params["W" + str(l+1)]-learning_rate*grads["dW"+str(l+1)]
        params["b" + str(l+1)] = params["b" + str(l+1)]-learning_rate*grads["db"+str(l+1)]

    return params

def layerLModel(X, Y, layers_dims, num_iterations,stack, learning_rate = 0.025):

    costs = []

    parameters = initializeNewLayer(layers_dims,para,stack)

    for i in range(0, num_iterations):

        AL, caches = modelLForward(X,parameters)

        if stack==True:
            cost = costComputation(AL,Y)
        else:
            cost = costComputationAE(AL,Y,parameters)

        grads = backModelL(AL,Y,caches)

        parameters = updateParams(parameters,grads,learning_rate)

    return (parameters, costs)

para = []
def ACT(x, a, b, act):
    if(act == "gaussian"):
        return np.exp(-b*np.linalg.norm(x-a))
    elif(act == "tanh"):
        num = 1 - np.exp(-(np.dot(x.T, a) + b))
```

```
        den = 1 + np.exp(-(np.dot(x.T, a) + b))
        return (num/den)

def init(l_hidden, dimensions):
    a = []
    b = []
    for i in range(l_hidden):
        a.append(np.random.rand(dimensions,1))
        b.append(np.random.rand(1))
    return (a,b)

def oneHotEnc(y):
    from sklearn.preprocessing import OneHotEncoder
    onehotencoder = OneHotEncoder()
    y = onehotencoder.fit_transform(y).toarray()
    return y

def compute(l_hidden,training_data_X,testing_data_X,training_data_y,testing_data_y,act):

    Y_enc = oneHotEnc(training_data_y)
    H = np.zeros((training_data_X.shape[0],l_hidden))
    for i in range(H.shape[1]):
        for j in range(H.shape[0]):
            H[j][i]=ACT(train_x[j],a[i],b[i],act)
    W=np.dot(np.linalg.pinv(H),Y_enc)

    H=np.zeros((testing_data_X.shape[0],l_hidden))
    for i in range(H.shape[1]):
        for j in range(H.shape[0]):
            H[j][i]=ACT(testing_data_X[j],a[i],b[i],act)

    p = np.dot(H,W)
    p = np.argmax(p,axis=1)
    p = np.reshape(p,newshape=(p.shape[0],1))
    metrics(p, testing_data_y)


dataset = pd.read_excel('/content/drive/MyDrive/NNFL data/Data_A2/data55.xlsx')

row, col = dataset.shape
feats = col - 1

# normalization
dataset.loc[:, dataset.columns != feats] = (dataset.loc[:, dataset.columns != feats]-datas

# spliting dataset into train test and val
training_data, validation_data, testing_data = np.split(dataset.sample(frac=1),[int(0.7*le

training_data = np.array(training_data)
validation_data = np.array(validation_data)
testing_data = np.array(testing_data)
training_data_X = training_data[:, :feats]
training_data_y = training_data[:, feats]
validation_data_X = validation_data[:, :feats]
validation_data_y = validation_data[:, feats]
```

```
testing_data_X = testing_data[:, :feats]
testing_data_y = testing_data[:, feats]

train_row, train_col = training_data_X.shape

layers_dims = [72,32,72]
parameters, costs = layerLModel(training_data_X, training_data_y, layers_dims, num_iterati
paramsAE1 = parameters

W1 = paramsAE1['W1']
b1 = paramsAE1['b1']
X_new, _ = linearActivationFunction(X.T,W1,b1,"sigmoid")

layers_dims = [32,16,32]
parameters,costs = layerLModel(training_data_X, training_data_y, layers_dims,num_iteration
paramsAE2 = parameters

W1 = paramsAE1['W1']
W2 = paramsAE2['W1']
b1 = paramsAE1['b1']
b2 = paramsAE2['b1']

x, _ = sigmoidFunction(np.dot(W1,X.T) + b1)
x, _ = sigmoidFunction(np.dot(W2,x) + b2)

a, b = init(256, X.shape[1])
print("Tanh Accuracy: ")
compute(256, training_data_X, testing_data_X, training_data_y, testing_data_y, "tanh")
print()
a, b = init(256,X.shape[1])
print("Gaussian Accuracy: ")
compute(256,training_data_X,testing_data_X,training_data_y,testing_data_y,"gaussian")
```

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

    Mounted at /content/drive
```

```
import numpy as np

import matplotlib.pyplot as plt
import pandas as pd
from sklearn.preprocessing import normalize
from sklearn.datasets import make_blobs, make_circles, make_moons
from sklearn.preprocessing import StandardScaler
```

```
data=pd.read_excel('/content/drive/MyDrive/NNFL data/Data_A2/data5.xlsx')
```

```
data[data.columns[-1]].unique()

    array([1., 2., 3.])
```

```
data=data.sample(frac=1)
data.head()
```

	15.260	14.840	0.871	5.763	3.312	2.221	5.220	1.000
60	11.23	12.63	0.8840	4.902	2.879	2.269	4.703	1.0
20	14.11	14.26	0.8722	5.520	3.168	2.688	5.219	1.0
135	17.36	15.76	0.8785	6.145	3.574	3.526	5.971	2.0
46	14.99	14.56	0.8883	5.570	3.377	2.958	5.175	1.0
12	13.78	14.06	0.8759	5.479	3.156	3.136	4.872	1.0

```
data.to_numpy()

    array([[11.23  , 12.63  ,  0.884 , ...,  2.269 ,  4.703 ,  1.    ],
           [14.11  , 14.26  ,  0.8722, ...,  2.688 ,  5.219 ,  1.    ],
           [17.36  , 15.76  ,  0.8785, ...,  3.526 ,  5.971 ,  2.    ],
           ...,
           [18.96  , 16.2   ,  0.9077, ...,  4.334 ,  5.75  ,  2.    ],
           [12.11  , 13.27  ,  0.8639, ...,  4.132 ,  5.012 ,  3.    ],
           [14.38  , 14.21  ,  0.8951, ...,  2.462 ,  4.956 ,  1.    ]])
```

```
class SMOModel:


    def __init__(self, X, y, C, kernel, alphas, b, errors):
        self.X = X               # training data vector
        self.y = y               # class label vector
        self.C = C               # regularization parameter
```

```
        self.kernel = kernel     # kernel function
        self.alphas = alphas     # lagrange multiplier vector
        self.b = b               # scalar bias term
        self.errors = errors     # error cache
        self._obj = []           # record of objective function value
        self.m = len(self.X)     # store size of training set


def gaussian_rbf_kernel(x, y, sigma=1):


    if np.ndim(x) == 1 and np.ndim(y) == 1:
         result = np.exp(- (np.linalg.norm(x - y, 7)) ** 2 / (2 * sigma ** 2))
    elif (np.ndim(x) > 1 and np.ndim(y) == 1) or (np.ndim(x) == 1 and np.ndim(y) > 1):
         result = np.exp(- (np.linalg.norm(x - y, 7, axis=1) ** 2) / (2 * sigma ** 2))
    elif np.ndim(x) > 1 and np.ndim(y) > 1:
         result = np.exp(- (np.linalg.norm(x[:, np.newaxis] - y[np.newaxis, :], 7, axis=2)
    #result=np.exp(-np.linalg.norm(x-y)**2 / (2 * (sigma ** 2)))
    return result

def polynomial_kernel(X, Y):
    k = (1 + np.matmul(X, Y.transpose()))**7
    return k
```

Testing kernels

```
x_len, y_len = 10, 5
```

```
polynomial_kernel(np.random.rand(x_len, 1), np.random.rand(y_len, 1)).shape == (x_len,y_le
```

```
True
```

```
gaussian_rbf_kernel(np.random.rand(x_len, 1), np.random.rand(y_len, 1)).shape == (10,5)
```

```
True
```

```
# Objective function to optimize

def objective_function(alphas, target, kernel, X_train):
    '''
    `alphas`: vector of Lagrange multipliers
    `target`: vector of class labels (-1 or 1) for training data
    `kernel`: kernel function
    `X_train`: training data for model."""'''

    return np.sum(alphas) - 0.5 * np.sum((target[:, None] * target[None, :]) * kernel(X_tr


# Decision function
```

```
def decision_function(alphas, target, kernel, X_train, x_test, b):


    result = (alphas * target) @ kernel(X_train, x_test) - b
    return result



def plot_decision_boundary(model, ax, resolution=100, colors=('b', 'k', 'r'), levels=(-1,
        """Plotting the model's decision boundary on the input axes object.
        Range of the decision boundary grid is determined by the training data.
        Returns the decision boundary grid and axes object (`grid`, `ax`)."""

        # Generate coordinate grid of shape [resolution x resolution]

        xrange = np.linspace(model.X[:,0].min(), model.X[:,0].max(), resolution)
        yrange = np.linspace(model.X[:,1].min(), model.X[:,1].max(), resolution)
        grid = [[decision_function(model.alphas, model.y,
                                   model.kernel, model.X[:,0:2],
                                   np.array([xr, yr]), model.b) for xr in xrange] for yr i
        grid = np.array(grid).reshape(len(xrange), len(yrange))


        # make a scatter plot of training data
        ax.contour(xrange, yrange, grid, levels=levels, linewidths=(1, 1, 1),
                   linestyles=('--', '-', '--'), colors=colors)
        ax.scatter(model.X[:,0], model.X[:,1],
                   c=model.y, cmap=plt.cm.viridis, lw=0, alpha=0.25)

        # Plot support vectors (non-zero alphas)

        mask = np.round(model.alphas, decimals=2) != 0.0
        ax.scatter(model.X[mask,0], model.X[mask,1],
                   c=model.y[mask], cmap=plt.cm.viridis, lw=1, edgecolors='k')

        return grid, ax


def take_step(i1, i2, model,eps):

    # Skip if chosen alphas are the same
    if i1 == i2:
        return 0, model

    alph1 = model.alphas[i1]
    alph2 = model.alphas[i2]
    y1 = model.y[i1]
    y2 = model.y[i2]
    E1 = model.errors[i1]
    E2 = model.errors[i2]
    s = y1 * y2

    # Compute L & H, the bounds on new possible alpha values
    if (y1 != y2):
        L = max(0, alph2 - alph1)
        H = min(model.C, model.C + alph2 - alph1)
```

```
    elif (y1 == y2):
        L = max(0, alph1 + alph2 - model.C)
        H = min(model.C, alph1 + alph2)
    if (L == H):
        return 0, model

    # Compute kernel & 2nd derivative eta
    k11 = model.kernel(model.X[i1], model.X[i1])
    k12 = model.kernel(model.X[i1], model.X[i2])
    k22 = model.kernel(model.X[i2], model.X[i2])
    eta = 2 * k12 - k11 - k22

    # Compute new alpha 2 (a2) if eta is negative
    if (eta < 0):
        a2 = alph2 - y2 * (E1 - E2) / eta
        # Clip a2 based on bounds L & H
        if L < a2 < H:
            a2 = a2
        elif (a2 <= L):
            a2 = L
        elif (a2 >= H):
            a2 = H

    # If eta is non-negative, move new a2 to bound with greater objective function value
    else:
        alphas_adj = model.alphas.copy()
        alphas_adj[i2] = L
        # objective function output with a2 = L
        Lobj = objective_function(alphas_adj, model.y, model.kernel, model.X)
        alphas_adj[i2] = H
        # objective function output with a2 = H
        Hobj = objective_function(alphas_adj, model.y, model.kernel, model.X)
        if Lobj > (Hobj + eps):
            a2 = L
        elif Lobj < (Hobj - eps):
            a2 = H
        else:
            a2 = alph2

    # Push a2 to 0 or C if very close
    if a2 < 1e-8:
        a2 = 0.0
    elif a2 > (model.C - 1e-8):
        a2 = model.C

    # If examples can't be optimized within epsilon (eps), skip this pair
    if (np.abs(a2 - alph2) < eps * (a2 + alph2 + eps)):
        return 0, model

    # Calculate new alpha 1 (a1)
    a1 = alph1 + s * (alph2 - a2)

    # Update threshold b to reflect newly calculated alphas
    # Calculate both possible thresholds
    b1 = E1 + y1 * (a1 - alph1) * k11 + y2 * (a2 - alph2) * k12 + model.b
```

```
    b2 = E2 + y1 * (a1 - alph1) * k12 + y2 * (a2 - alph2) * k22 + model.b

    # Set new threshold based on if a1 or a2 is bound by L and/or H
    if 0 < a1 and a1 < model.C:
        b_new = b1
    elif 0 < a2 and a2 < model.C:
        b_new = b2
    # Average thresholds if both are bound
    else:
        b_new = (b1 + b2) * 0.5

    # Update model object with new alphas & threshold
    model.alphas[i1] = a1
    model.alphas[i2] = a2

    # Update error cache
    # Error cache for optimized alphas is set to 0 if they're unbound
    for index, alph in zip([i1, i2], [a1, a2]):
        if 0.0 < alph < model.C:
            model.errors[index] = 0.0

    # Set non-optimized errors
    non_opt = [n for n in range(model.m) if (n != i1 and n != i2)]
    model.errors[non_opt] = model.errors[non_opt] + \
                            y1*(a1 - alph1)*model.kernel(model.X[i1], model.X[non_opt]) +
                            y2*(a2 - alph2)*model.kernel(model.X[i2], model.X[non_opt]) +

    # Update model threshold
    model.b = b_new

    return 1, model


def examine_example(i2, model,tol,eps):

    y2 = model.y[i2]
    alph2 = model.alphas[i2]
    E2 = model.errors[i2]
    r2 = E2 * y2

    # Proceed if error is within specified tolerance (tol)
    if ((r2 < -tol and alph2 < model.C) or (r2 > tol and alph2 > 0)):

        if len(model.alphas[(model.alphas != 0) & (model.alphas != model.C)]) > 1:
            # Use 2nd choice heuristic is choose max difference in error
            if model.errors[i2] > 0:
                i1 = np.argmin(model.errors)
            elif model.errors[i2] <= 0:
                i1 = np.argmax(model.errors)
            step_result, model = take_step(i1, i2, model,eps)
            if step_result:
                return 1, model

        # Loop through non-zero and non-C alphas, starting at a random point
```

```
        for i1 in np.roll(np.where((model.alphas != 0) & (model.alphas != model.C))[0],
                          np.random.choice(np.arange(model.m))):
            step_result, model = take_step(i1, i2, model,eps)
            if step_result:
                return 1, model

        # loop through all alphas, starting at a random point
        for i1 in np.roll(np.arange(model.m), np.random.choice(np.arange(model.m))):
            step_result, model = take_step(i1, i2, model,eps)
            if step_result:
                return 1, model

    return 0, model

def train_mod(model,tol,eps):

    numChanged = 0
    examineAll = 1

    while(numChanged > 0) or (examineAll):
        numChanged = 0
        if examineAll:
            # loop over all training examples
            for i in range(model.alphas.shape[0]):
                examine_result, model = examine_example(i, model,tol,eps)
                numChanged += examine_result
                if examine_result:
                    obj_result = objective_function(model.alphas, model.y, model.kernel, m
                    model._obj.append(obj_result)
        else:
            # loop over examples where alphas are not already at their limits
            for i in np.where((model.alphas != 0) & (model.alphas != model.C))[0]:
                examine_result, model = examine_example(i, model, tol,eps)
                numChanged += examine_result
                if examine_result:
                    obj_result = objective_function(model.alphas, model.y, model.kernel, m
                    model._obj.append(obj_result)
        if examineAll == 1:
            examineAll = 0
        elif numChanged == 0:
            examineAll = 1

    return model


#normalizing data
data.iloc[:,0:7]=(data.iloc[:,0:7]-(data.iloc[:,0:7]).min())/((data.iloc[:,0:7]).max()-(da


data=data.sample(frac=1, random_state=50)#randomising data
train,validate,test= np.split(data,[int(.7*len(data)),int(.8*len(data))])#splitting data i



data.head()
```

	15.260	14.840	0.871	5.763	3.312	2.221	5.220	1.000
57	0.452314	0.487603	0.704174	0.429617	0.562366	0.160436	0.346135	1.0
150	0.134089	0.229339	0.152450	0.284910	0.104063	0.809645	0.369769	3.0
27	0.332389	0.365702	0.670599	0.361486	0.421240	0.258604	0.255539	1.0
16	0.481586	0.483471	0.886570	0.353604	0.630078	0.108427	0.259478	1.0
49	0.362606	0.411157	0.607985	0.386261	0.457591	0.417363	0.307730	1.0

```
data[data.columns[-1]].unique()
```

```
array([1., 3., 2.])
```

```
print(train.shape)
print(validate.shape)
print(test.shape)
#converting to array for slicing
train=np.array(train)
print(train)
validate=np.array(validate)
print(validate)
test=np.array(test)
print(test)
```

```
[7.40321058e-01 7.35537190e-01 9.03811252e-01 6.08671171e-01
 8.13257306e-01 2.88509797e-01 6.82422452e-01 2.00000000e+00]
[2.08687441e-01 2.19008264e-01 7.06896552e-01 1.46959459e-01
 3.53528154e-01 5.34124745e-01 1.94485475e-01 3.00000000e+00]
[8.07365439e-01 8.67768595e-01 5.81669691e-01 7.65765766e-01
 7.89023521e-01 7.69337789e-01 7.55292959e-01 2.00000000e+00]
[8.11142587e-01 8.71900826e-01 5.77132486e-01 8.27702703e-01
 7.49109052e-01 3.37008673e-01 8.41949778e-01 2.00000000e+00]
[4.90084986e-01 5.16528926e-01 7.64065336e-01 4.36373874e-01
 5.73057733e-01 6.27741877e-01 3.03791236e-01 1.00000000e+00]
[6.64778093e-01 7.37603306e-01 5.37205082e-01 7.27477477e-01
 6.63578047e-01 4.30495781e-01 7.58739537e-01 2.00000000e+00]
[7.89423985e-01 8.28512397e-01 6.78765880e-01 7.59572072e-01
 8.01853172e-01 3.38438934e-01 8.02067947e-01 2.00000000e+00]
[6.04343720e-02 8.47107438e-02 4.65517241e-01 1.06981982e-01
 1.36136850e-01 8.78817824e-01 2.15657312e-01 3.00000000e+00]
[7.79981114e-01 7.76859504e-01 8.84754991e-01 7.05518018e-01
 8.38203849e-01 2.70176442e-01 8.27671098e-01 2.00000000e+00]
[6.29839471e-01 6.85950413e-01 6.18874773e-01 6.07545045e-01
 6.87099073e-01 4.90696798e-01 6.26292467e-01 2.00000000e+00]
[5.09915014e-01 5.12396694e-01 8.92014519e-01 2.61261261e-01
 6.78545973e-01 3.34278173e-01 3.07730182e-01 2.00000000e+00]
[2.17186025e-01 2.80991736e-01 4.17422868e-01 3.35585586e-01
 2.82252316e-01 7.04715963e-01 3.92417528e-01 3.00000000e+00]
[5.19357885e-02 7.85123967e-02 4.32849365e-01 6.30630631e-02
 1.16892373e-01 7.31110793e-01 2.60955194e-01 3.00000000e+00]
[2.01133144e-01 2.39669421e-01 5.49001815e-01 1.84121622e-01
 2.98645759e-01 4.33876399e-01 1.94485475e-01 1.00000000e+00]
[2.78564684e-01 2.97520661e-01 7.16878403e-01 2.52815315e-01
 3.74910905e-01 2.36890351e-01 3.24470704e-01 1.00000000e+00]
[8.30028329e-01 8.90495868e-01 5.76225045e-01 7.90540541e-01
```

```
 8.27512473e-01 3.78746311e-01 7.11964549e-01 2.00000000e+00]
[1.43531634e-01 2.19008264e-01 2.82214156e-01 1.46396396e-01
 2.86528867e-01 9.58145341e-02 0.00000000e+00 1.00000000e+00]
[3.56940510e-01 4.09090909e-01 5.85299456e-01 3.77252252e-01
 3.72772630e-01 9.08736299e-02 3.84539636e-01 1.00000000e+00]
[7.92256846e-01 8.78099174e-01 4.61887477e-01 9.29054054e-01
 7.41268710e-01 3.80436620e-01 9.74396849e-01 2.00000000e+00]
[6.39282342e-01 6.92148760e-01 6.38838475e-01 7.01576577e-01
 6.72843906e-01 3.58982694e-01 7.14918759e-01 2.00000000e+00]
[1.37865911e-01 2.06611570e-01 3.03992740e-01 2.07207207e-01
 1.54668567e-01 5.49077481e-01 2.59478090e-01 3.00000000e+00]
[1.88857413e-02 1.07438017e-01 2.35934664e-02 2.35360360e-01
 1.28296507e-02 6.10708760e-01 3.32348597e-01 3.00000000e+00]
[2.46458924e-01 2.58264463e-01 7.27767695e-01 1.89752252e-01
 4.29080542e-01 9.81666645e-01 2.64401773e-01 3.00000000e+00]
[7.88479698e-01 8.07851240e-01 7.81306715e-01 7.01013514e-01
 8.51746258e-01 2.78627989e-01 7.04086657e-01 2.00000000e+00]
[9.71671388e-01 9.58677686e-01 8.62068966e-01 8.73310811e-01
 9.99287242e-01 5.52718147e-01 8.87247661e-01 2.00000000e+00]
[5.66572238e-02 1.32231405e-01 1.56079855e-01 1.97635135e-01
 3.20741269e-02 6.56347112e-01 3.44657804e-01 3.00000000e+00]
[3.54107649e-01 4.04958678e-01 5.85299456e-01 4.11599099e-01
 3.99144690e-01 7.12400369e-02 3.10684392e-01 1.00000000e+00]
[6.57223796e-01 6.71487603e-01 8.25771325e-01 5.02252252e-01
 7.55523877e-01 5.98226475e-01 5.62284589e-01 2.00000000e+00]
[1.85080264e-01 2.39669421e-01 4.32849365e-01 2.44369369e-01
 2.40912331e-01 4.75093942e-01 3.23485968e-01 3.00000000e+00]]
```

Preparing data for 1v1 SVM classification

```
train1, train2, train3 = [], [], []


for row in train:
  if row[-1] == 1:
        train1.append(row)
        train2.append(row)

  elif row[-1] == 2:
        train1.append(row)
        train3.append(row)

  elif row[-1] == 3:
        train2.append(row)
        train3.append(row)
train1, train2, train3 = np.array(train1), np.array(train2), np.array(train3)

for row in train1:
    row[-1] = 1 if row[-1] == 1 else -1

for row in train2:
    row[-1] = 1 if row[-1] == 1 else -1

for row in train3:
    row[-1] = 1 if row[-1] == 2 else -1
```

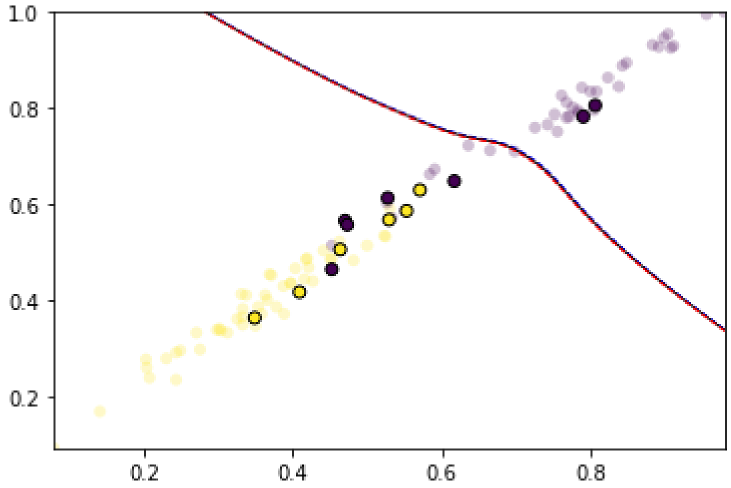
```
X_train1=train1[:,0:7]
y_train1=train1[:,7]
#y_train1=np.reshape(y_train1,newshape=(y_train1.shape[0],1))
print(X_train1.shape)
print(y_train1.shape)
X_train2=train2[:,0:7]
y_train2=train2[:,7]
print(X_train2.shape)
print(y_train2.shape)
X_train3=train3[:,0:7]
y_train3=train3[:,7]
print(X_train3.shape)
print(y_train3.shape)
```

```
(97, 7)
(97,)
(103, 7)
(103,)
(92, 7)
(92,)
```

```
def train_model(C,X_train,y_train,kernel_name):
  # Set model parameters and initial values

  m = len(X_train)
  initial_alphas = np.zeros(m)
  initial_b = 0.0
  # Set tolerances
  tol = 0.01 # error tolerance
  eps = 0.01 # alpha tolerance

  # Instantiate model
  if(kernel_name=="polynomial_kernel"):
    model = SMOModel(X_train, y_train, C, polynomial_kernel,
                     initial_alphas, initial_b, np.zeros(m))
  else:
    model = SMOModel(X_train, y_train, C, gaussian_rbf_kernel,
                     initial_alphas, initial_b, np.zeros(m))


  # Initialize error cache
  initial_error = decision_function(model.alphas, model.y, model.kernel,
                                    model.X, model.X, model.b) - model.y
  model.errors = initial_error
  np.random.seed(0)
  output = train_mod(model,tol,eps)
  print(output.alphas.sum())
  fig, ax = plt.subplots()
  grid, ax = plot_decision_boundary(output, ax)
```

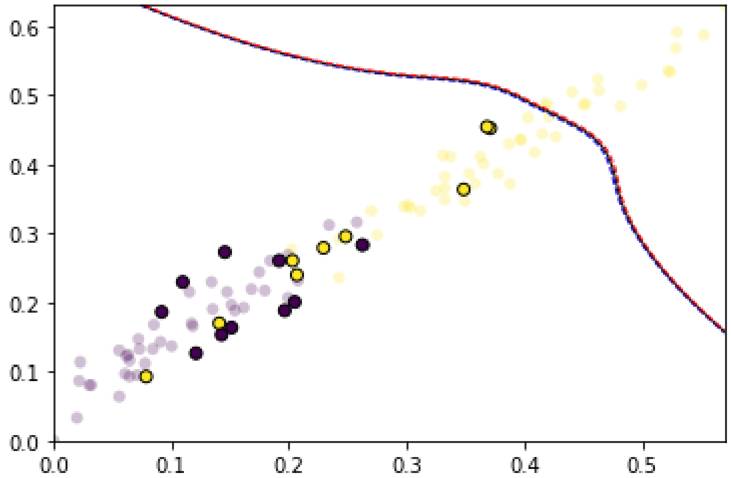
```
train_model(1000,X_train1,y_train1,polynomial_kernel)
```

```
11712.925926342334
```



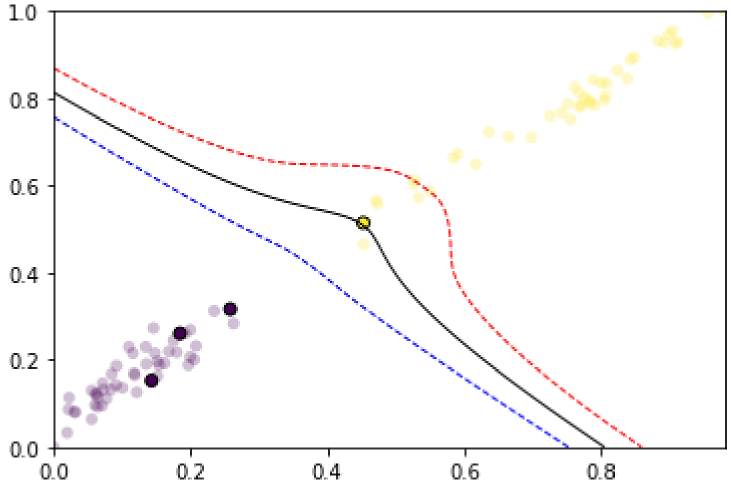
```
train_model(1000,X_train2,y_train2,polynomial_kernel)
```

```
17190.07646234499
```



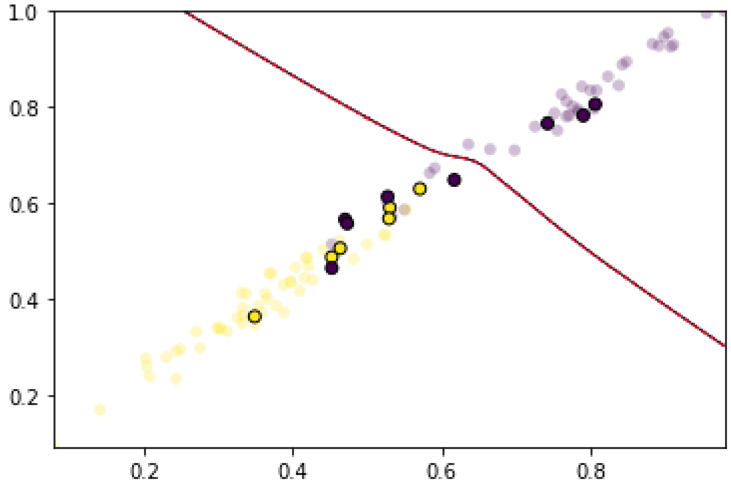
```
train_model(1000,X_train3,y_train3,polynomial_kernel)
```

```
85.5825378978646
```



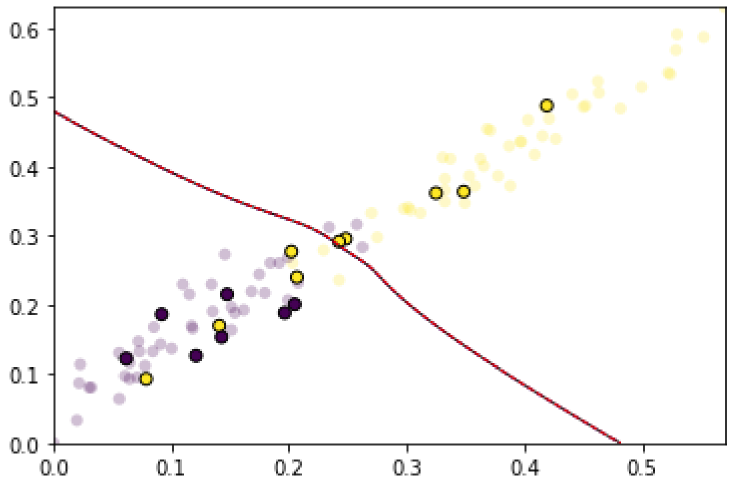
```
train_model(10000,X_train1,y_train1,gaussian_rbf_kernel)
```

```
116705.82575851877
```



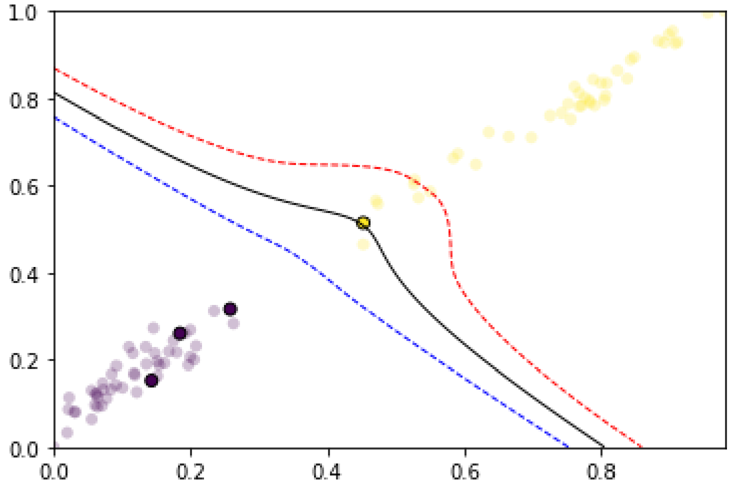

```
train_model(10000,X_train2,y_train2,gaussian_rbf_kernel)
```

```
140000.0
```




```
train_model(1000,X_train3,y_train3,gaussian_rbf_kernel)
```

```
85.5825378978646
```



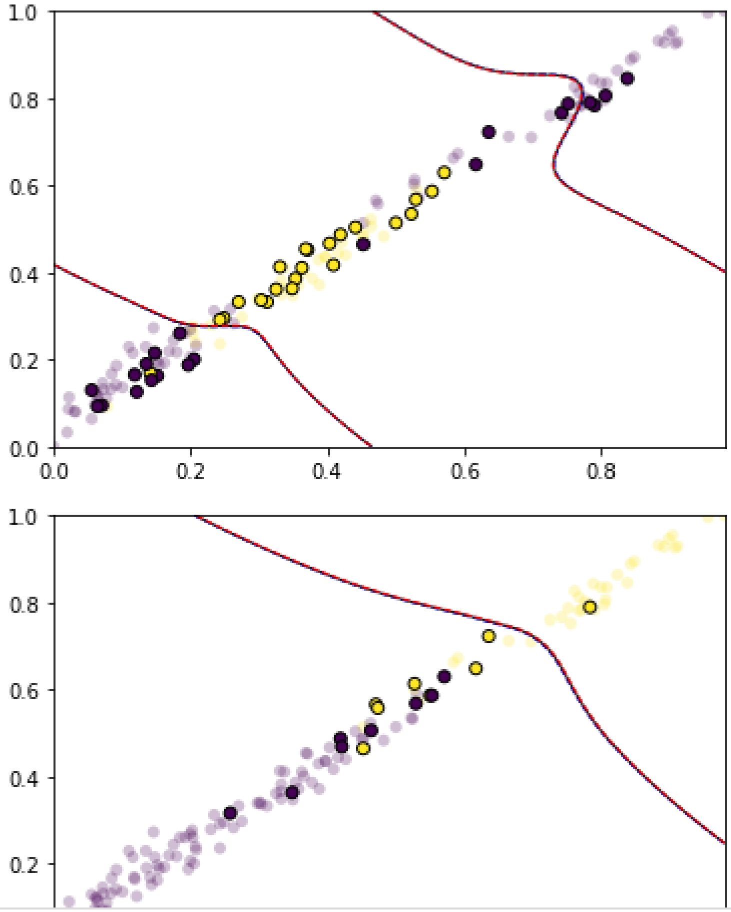
1 v ALL classification

```
for i in range(3):

    X = train.copy() #copy of the training samples
    for row in X:
        if row[-1] == i+1:
            row[-1] = 1
        else:
            row[-1] = -1

    X_train=X[:,0:7]
    y_train=X[:,7]
    train_model(1000,X_train,y_train,polynomial_kernel)
```

```
42000.0
14389.218038983792
16071.0638464454
```


```
for i in range(3):

    X = train.copy() #copy of the training samples
    for row in X:
        if row[-1] == i+1:
            row[-1] = 1
        else:
            row[-1] = -1

    X_train=X[:,0:7]
    y_train=X[:,7]
    train_model(1000,X_train,y_train,gaussian_rbf_kernel)
```

```
42000.0
14389.218038983792
16071.0638464454
```



✓ 27s completed at 8:15 PM

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

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.m
```

```
import tensorflow as tf
import matplotlib.pyplot as plt
import pandas as pd
import scipy.io
import numpy as np
import sklearn
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
```

```
input_data=scipy.io.loadmat('/content/drive/MyDrive/NNFL data/Data_A2/input.mat')
class_label_data=scipy.io.loadmat('/content/drive/MyDrive/NNFL data/Data_A2/class_label.ma
```

```
input_data=pd.DataFrame(input_data["x"])
#cols = input_data.select_dtypes(exclude=['float']).columns
#input_data[cols] = input_data[cols].apply(pd.to_numeric, downcast='float', errors='coerce
input_data=(np.asarray(input_data)).T
print(input_data.dtype)
class_label_data=np.asarray(class_label_data["y"])
```

```
object
```

```
data_input=[]
for i in range(len(input_data)):
    data_input.append(input_data[i][0])
data_input=np.asarray(data_input)
print(data_input)
#data_input=np.reshape(data_input, (data_input.shape[0], 1))
print(data_input.shape)
data_input=data_input.transpose(0,2,1)
for i in range(len(data_input)):
    data_input[i]=preprocessing.normalize(data_input[i])
```

```
[[[-0.02942547 -0.03004717 -0.03063448 ... -0.00171387 -0.00150202
   -0.00129178]
  [-0.03354165 -0.0335658  -0.03354647 ... -0.00397835 -0.00348932
   -0.00299711]
  [-0.0041002  -0.00350403 -0.00289871 ... -0.0023221  -0.00203756
   -0.00174853]
  ...
  [-0.01438948 -0.01302857 -0.01166126 ...  0.00079799  0.00065684
    0.0005333 ]
  [-0.02843199 -0.0272833  -0.02612498 ...  0.00120509  0.00103033
    0.0008662 ]
  [-0.00433259 -0.00382339 -0.00330092 ... -0.00397676 -0.00337576
   -0.00281225]]
```

```
[[-0.02942547 -0.03004717 -0.03063448 ... -0.00171387 -0.00150202
  -0.00129178]
 [-0.03354165 -0.0335658  -0.03354647 ... -0.00397835 -0.00348932
  -0.00299711]
 [-0.0041002  -0.00350403 -0.00289871 ... -0.0023221  -0.00203756
  -0.00174853]
 ...
 [-0.01438948 -0.01302857 -0.01166126 ...  0.00079799  0.00065684
   0.0005333 ]
 [-0.02843199 -0.0272833  -0.02612498 ...  0.00120509  0.00103033
   0.0008662 ]
 [-0.03536238 -0.03581917 -0.03625893 ... -0.00944339 -0.00816656
  -0.00691697]]

[[-0.02942547 -0.03004717 -0.03063448 ... -0.00171387 -0.00150202
  -0.00129178]
 [-0.03354165 -0.0335658  -0.03354647 ... -0.00397835 -0.00348932
  -0.00299711]
 [-0.0041002  -0.00350403 -0.00289871 ... -0.0023221  -0.00203756
  -0.00174853]
 ...
 [-0.01438948 -0.01302857 -0.01166126 ...  0.00079799  0.00065684
   0.0005333 ]
 [-0.02843199 -0.0272833  -0.02612498 ...  0.00120509  0.00103033
   0.0008662 ]
 [ 0.27783873  0.32234347  0.36762522 ... -0.00136926 -0.00118706
  -0.00100843]]

...

[[-0.01032181 -0.01064618 -0.01100247 ...  0.00702488  0.00598827
   0.00500332]
 [-0.01197618 -0.0093181  -0.00662519 ... -0.00213488 -0.00182275
  -0.00151613]
 [-0.00145384  0.00154284  0.0046053  ... -0.00917414 -0.00782367
  -0.00653038]
 ...
 [-0.00790134 -0.01163092 -0.01536316 ... -0.00465235 -0.00402374
  -0.00340576]
 [-0.04150407 -0.04452396 -0.04742369 ... -0.00698926 -0.00600713
  -0.00506403]
 [ 0.048734    0.00342764 -0.04154496 ...  0.00630108  0.00536445
   0.00447938]]
```

```
data_input=data_input.astype(np.float)
print(data_input)
print(data_input.dtype)
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:1: DeprecationWarnin
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdoc
  """Entry point for launching an IPython kernel.
[[[-0.40125279 -0.45738194 -0.05591131 ... -0.19621842 -0.38770543
   -0.05908019]
  [-0.41160141 -0.45980145 -0.04800002 ... -0.17847202 -0.37374054
   -0.0523747 ]
  [-0.42141005 -0.46146759 -0.03987491 ... -0.16041305 -0.35937711
   -0.0454077 ]
```

```
  ...
  [-0.19797961 -0.45956381 -0.26823974 ...  0.09218033  0.13920735
   -0.45938054]
  [-0.20131779 -0.46767903 -0.27309765 ...  0.08803666  0.13809689
   -0.45245752]
  [-0.20465272 -0.4748241  -0.27701496 ...  0.08449011  0.13722921
   -0.44553836]]

 [[-0.36193933 -0.41256913 -0.0504333  ... -0.17699357 -0.3497193
   -0.43496452]
  [-0.36992545 -0.41324509 -0.04313987 ... -0.16040116 -0.33589812
   -0.44098743]
  [-0.37741609 -0.41329174 -0.03571209 ... -0.1436664  -0.32185921
   -0.44670924]
  ...
  [-0.14073517 -0.3266841  -0.19068007 ...  0.06552702  0.0989565
   -0.77544818]
  [-0.14258944 -0.33124788 -0.19342971 ...  0.06235464  0.09781132
   -0.77526631]
  [-0.14462615 -0.33555373 -0.19576387 ...  0.05970837  0.0969786
   -0.77441837]]

 [[-0.10241317 -0.11673922 -0.01427044 ... -0.05008152 -0.09895543
    0.96699701]
  [-0.09091864 -0.10156554 -0.01060272 ... -0.03942268 -0.08255555
    0.97536734]
  [-0.08175097 -0.08952189 -0.00773549 ... -0.03111915 -0.06971696
    0.98104229]
  ...
  [-0.21943783 -0.50937412 -0.29731319 ...  0.10217139  0.15429548
   -0.17531535]
  [-0.22223755 -0.51627748 -0.30147635 ...  0.09718492  0.15244711
   -0.17563637]
  [-0.2250399  -0.52212532 -0.30461075 ...  0.09290688  0.15089977
   -0.175679  ]]

 ...

 [[-0.13052594 -0.15144652 -0.01838474 ... -0.09991761 -0.52484585
    0.61627302]
  [-0.17569061 -0.1537737   0.02546104 ... -0.1919415  -0.73476545
    0.05656528]
  [-0.15045611 -0.09059782  0.06297628 ... -0.21008747 -0.64850726
   -0.5681171 ]
  ...
  [ 0.36731449 -0.11162782 -0.47969449 ... -0.24326062 -0.3654523
    0.32946898]
```

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

```
X_train, X_test, y_train, y_test = train_test_split(data_input, data_class, test_size=0.2,
X_train, X_valid, y_train, y_valid = train_test_split(X_train, y_train, test_size=0.125, r
```

```
model=tf.keras.Sequential()
model.add(tf.keras.layers.Conv1D(input_shape=(800,12),filters=20,kernel_size=7,padding="sa
model.add(tf.keras.layers.Conv1D(filters=20,kernel_size=7,padding="same",activation="relu"
model.add(tf.keras.layers.MaxPool1D(pool_size=3,strides=3))
model.add(tf.keras.layers.Conv1D(filters=60,kernel_size=7,padding="same",activation="relu"
model.add(tf.keras.layers.Conv1D(filters=60,kernel_size=7,padding="same",activation="relu"
model.add(tf.keras.layers.MaxPool1D(pool_size=3,strides=3))
model.add(tf.keras.layers.Dropout(0.7))
model.add(tf.keras.layers.Conv1D(filters=120,kernel_size=7,padding="same",activation="relu
model.add(tf.keras.layers.Conv1D(filters=120,kernel_size=7,padding="same",activation="relu
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(units=2000,activation="relu"))
model.add(tf.keras.layers.Dense(units=700,activation="relu"))
model.add(tf.keras.layers.Dense(units=50,activation="relu"))
model.add(tf.keras.layers.Dense(units=7,activation="sigmoid"))
```

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv1d (Conv1D)             (None, 800, 20)           1700

 conv1d_1 (Conv1D)           (None, 800, 20)           2820

 max_pooling1d (MaxPooling1D  (None, 266, 20)          0
 )

 conv1d_2 (Conv1D)           (None, 266, 60)           8460

 conv1d_3 (Conv1D)           (None, 266, 60)           25260

 max_pooling1d_1 (MaxPooling  (None, 88, 60)           0
 1D)

 dropout (Dropout)           (None, 88, 60)            0

 conv1d_4 (Conv1D)           (None, 88, 120)           50520

 conv1d_5 (Conv1D)           (None, 88, 120)           100920

 flatten (Flatten)           (None, 10560)             0

 dense (Dense)               (None, 2000)              21122000

 dense_1 (Dense)             (None, 700)               1400700

 dense_2 (Dense)             (None, 50)                35050
```

```
 dense_3 (Dense)             (None, 7)                 357

=================================================================
Total params: 22,747,787
Trainable params: 22,747,787
Non-trainable params: 0
_________________________________________________________________
```

```
opt=tf.keras.optimizers.Adam(learning_rate=0.001)
model.compile(optimizer=opt,loss=tf.keras.losses.SparseCategoricalCrossentropy(),metrics=[
```

```
model.fit(X_train, y_train, epochs=2, batch_size=200, validation_data=(X_valid,y_valid))
```

```
Epoch 1/2
61/61 [==============================] - 112s 2s/step - loss: 0.0047 - accuracy: 0.9
Epoch 2/2
61/61 [==============================] - 109s 2s/step - loss: 0.0034 - accuracy: 0.9
<keras.callbacks.History at 0x7fd40488b650>
```

```
model.evaluate(X_test,y_test)
```

```
108/108 [==============================] - 10s 90ms/step - loss: 0.0016 - accuracy: 0
[0.0015695691108703613, 0.999417245388031]
```

```
y_pred=np.argmax(model.predict(X_test), axis=-1)
```

```
from sklearn.metrics import classification_report,confusion_matrix
```

```
confusion_m=confusion_matrix(y_test,y_pred)
for i in range(7):
    print("Accuracy for class {0}: {1}%".format(i+1,(confusion_m.diagonal()/confusion_m.su

print("\n")
print(confusion_m)
```

```
Accuracy for class 1: 100.0%
Accuracy for class 2: 99.49367088607595%
Accuracy for class 3: 97.12230215827337%
Accuracy for class 4: 98.48484848484848%
Accuracy for class 5: 87.44038155802862%
Accuracy for class 6: 85.34201954397395%
Accuracy for class 7: 100.0%


[[567   0   0   0   0   0   0]
 [  2 393   0   0   0   0   0]
 [ 16   0 540   0   0   0   0]
 [  5   0   0 325   0   0   0]
 [ 79   0   0   0 550   0   0]
```

```
 [ 26   0  19   0  45 524   0]
 [  0   0   0   0   0   0 341]]
```

```
print("Number of correctly predicted class labels are: {0}".format(np.sum(y_pred==y_test)))
print("Total number of class labels are: {0}".format(len(y_test)))
print("Overall Accuracy is: {0}%".format(np.sum(y_pred==y_test)/len(y_test)*100))
```

```
Number of correctly predicted class labels are: 3240
Total number of class labels are: 3432
Overall Accuracy is: 94.4055944055944%
```

0s completed at 9:04 PM

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

```
Mounted at /content/drive
```

```
import pandas as pd
import numpy as np
import scipy.io as sio
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
```

```
data=pd.read_excel('/content/drive/MyDrive/NNFL data/Data_A2/data55.xlsx')
```

```
data.dropna(inplace=True)
```

```
data[data.columns[-1]].unique()
```

```
array([0., 1.])
```

```
data=data.sample(frac=1)
data.head()
```

1 to

index	0.02	0.0371	0.0428	0.0207	0.0954	0.0986	0.1539	0.1601	0.3109	0.2111	0.1609	0.15
127	0.0374	0.0586	0.0628	0.0534	0.0255	0.1422	0.2072	0.2734	0.307	0.2597	0.3483	0.39
111	0.0454	0.0472	0.0697	0.1021	0.1397	0.1493	0.1487	0.0771	0.1171	0.1675	0.2799	0.33
124	0.0228	0.0853	0.1	0.0428	0.1117	0.1651	0.1597	0.2116	0.3295	0.3517	0.333	0.36
39	0.0068	0.0232	0.0513	0.0444	0.0249	0.0637	0.0422	0.113	0.1911	0.2475	0.1606	0.09
206	0.026	0.0363	0.0136	0.0272	0.0214	0.0338	0.0655	0.14	0.1843	0.2354	0.272	0.24

Show 25 per page
Like what you see? Visit the data table notebook to learn more about interactive tables.

```
data.hist
```

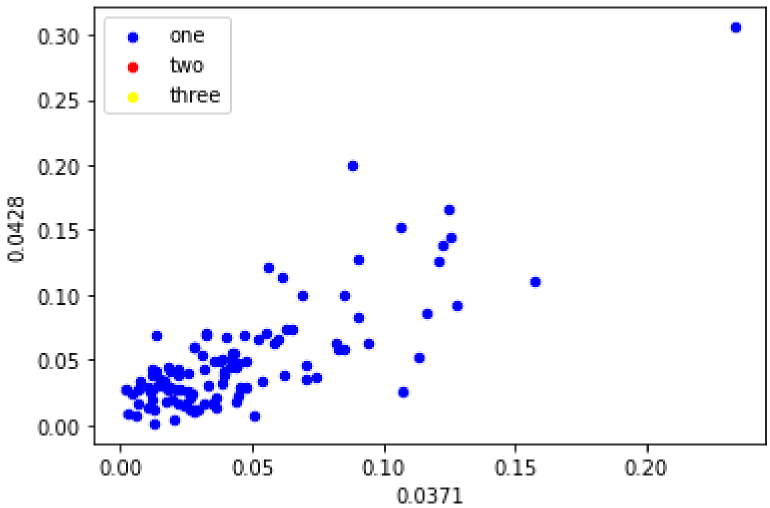
```
<bound method hist_frame of      0.0200  0.0371  0.0428  0.0207  0.0954  0.0986  0.1!
127  0.0374  0.0586  0.0628  0.0534  0.0255  0.1422  0.2072  0.2734  0.3070
111  0.0454  0.0472  0.0697  0.1021  0.1397  0.1493  0.1487  0.0771  0.1171
124  0.0228  0.0853  0.1000  0.0428  0.1117  0.1651  0.1597  0.2116  0.3295
39   0.0068  0.0232  0.0513  0.0444  0.0249  0.0637  0.0422  0.1130  0.1911
206  0.0260  0.0363  0.0136  0.0272  0.0214  0.0338  0.0655  0.1400  0.1843
..      ...     ...     ...     ...     ...     ...     ...     ...     ...
193  0.0392  0.0108  0.0267  0.0257  0.0410  0.0491  0.1053  0.1690  0.2105
171  0.0180  0.0444  0.0476  0.0698  0.1615  0.0887  0.0596  0.1071  0.3175
26   0.0177  0.0300  0.0288  0.0394  0.0630  0.0526  0.0688  0.0633  0.0624
99   0.0629  0.1065  0.1526  0.1229  0.1437  0.1190  0.0884  0.0907  0.2107
101  0.0587  0.1210  0.1268  0.1498  0.1436  0.0561  0.0832  0.0672  0.1372
```

```
     0.2111  ...  0.0027  0.0065  0.0159  0.0072  0.0167  0.0180  0.0084  \
127  0.2597  ...  0.0118  0.0063  0.0237  0.0032  0.0087  0.0124  0.0113
111  0.1675  ...  0.0120  0.0042  0.0238  0.0129  0.0084  0.0218  0.0321
124  0.3517  ...  0.0172  0.0191  0.0260  0.0140  0.0125  0.0116  0.0093
39   0.2475  ...  0.0173  0.0163  0.0055  0.0045  0.0068  0.0041  0.0052
206  0.2354  ...  0.0146  0.0129  0.0047  0.0039  0.0061  0.0040  0.0036
..      ...  ...     ...     ...     ...     ...     ...     ...     ...
193  0.2471  ...  0.0083  0.0080  0.0026  0.0079  0.0042  0.0071  0.0044
171  0.2918  ...  0.0122  0.0114  0.0098  0.0027  0.0025  0.0026  0.0050
26   0.0613  ...  0.0102  0.0122  0.0044  0.0075  0.0124  0.0099  0.0057
99   0.3597  ...  0.0089  0.0262  0.0108  0.0138  0.0187  0.0230  0.0057
101  0.2352  ...  0.0331  0.0111  0.0088  0.0158  0.0122  0.0038  0.0101

     0.0090  0.0032  0.0000
127  0.0098  0.0126     1.0
111  0.0154  0.0053     1.0
124  0.0012  0.0036     1.0
39   0.0194  0.0105     0.0
206  0.0061  0.0115     1.0
..      ...     ...     ...
193  0.0022  0.0014     1.0
171  0.0073  0.0022     1.0
26   0.0032  0.0019     0.0
99   0.0113  0.0131     1.0
101  0.0228  0.0124     1.0

[207 rows x 61 columns]>
```

```
class_one = data[data.iloc[:,-1]==1][0:210]
class_two = data[data.iloc[:,-1]==2][0:210]
class_three = data[data.iloc[:,-1]==3][0:210]
```

```
axes = class_one.plot(kind='scatter', x=1, y=2, color='blue', label='one')
class_two.plot(kind='scatter', x=1, y=2, color='red', label='two', ax=axes)
class_three.plot(kind='scatter', x=1, y=2, color='yellow', label='three', ax=axes)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fc11c4c2c90>
```



✓ 0s completed at 8:27 PM