3 by 3 lattice SOM code:

```
import pandas as pd
from PIL import Image
from numpy import asarray
import numpy as np
import matplotlib.pyplot as plt
#!pip install pyexcel


import math
import pyexcel as pe
import pyexcel.ext.xls
import sklearn


#your_matrix = pe.get_array('RBF_Data.xlsx')









df = pd.read_excel('RBF_Data.xlsx')
print(df)


#Convert the column one also known as X to an array
X_array = df[["Unnamed: 0"]].to_numpy()
print("##########")
print('\n')
print("print out X")
print(X_array)





#convert column two also known as Y to an array
Y_array = df[["Unnamed: 1"]].to_numpy()
print("##########")
print('\n')
```

```
print("print out Y")
print(Y_array)




#convert column three also known as Label to an array
L_array = df[["Label"]].to_numpy()
print("##########")
print('\n')
print("print out Label")
print(L_array)
print("########")




##### print out the group to figure out the best K to use
# empty list, will hold color value
# corresponding to x
col =[]

for i in range(0, len(L_array)):
    if L_array[i] == 1:
        col.append('blue')
    else:
        col.append('magenta')

for i in range(len(L_array)):

    # plotting the corresponding x with y
    # and respective color
    plt.scatter(X_array[i], Y_array[i], c = col[i], s = 10,
                linewidth = 0)


print("print out plot. The K values will be chosen base on the cluster of the two different groups")
print("Group one cluster center/K is (1,7) and group negative one is (1,2)")
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Blue for group 1\nand Magenta for group -1')
plt.legend()
```

```
plt.show()

#count the winners of each node at the last epoch
Total_wins = [0 for i in range(9)]


#get the error at the last node of each epoch
Error_Matrix = [0 for i in range(9)]


#initialize the X and Y values for the results
plot_X = [0 for i in range(9)]
plot_Y = [0 for i in range(9)]

#initialize the weights
w = [[1, 4],
     [1.1, 4.1],
     [.94, 4],
     [1.5, 4],
     [1.3, 2.8],
     [1, 3],
     [1.2, 3.9],
     [1.4, 4],
     [1.1, 3.1]]

count = 0
while count < 9:
    #print(count)
    plot_X[count] = w[count][0]
    plot_Y[count] = w[count][1]
    count += 1


#initialize neighbor
bor = 0

#initialize learning rate
lr = 0.03

#initialize sigma
sigma = 0.1

oldsigma = 0
```

```
#keep track of the weights moving
moving = 0


#plot the graph to find the best weights

plt.title('Weights and Sample points')
plt.xlabel('X_axis')
plt.ylabel('Y_axis')
plt.scatter(X_array, Y_array, label = 'Sample points')
plt.scatter(plot_X, plot_Y, label = 'Weights')
plt.legend()
plt.show()





#initialize value for minimum distance and it's respective node
mindistance = 0
winning_node = 0



oldw = [[0 for i in range(2)] for j in range(9)]

# initialize distance
distance = [0 for i in range(9)]

# initialize lattice structure
D = [[0, 0],
    [1, 0],
    [2, 0],
    [0, 1],
    [1, 1],
    [2, 1],
    [0, 2],
    [1, 2],
    [2, 2]]

#initialize the placeholders use to tranverse through the dataset
X = 0
Y = 0
```

```
X_axis = [0 for i in range(75)]
count = 0
while count < 75:
    X_axis[count] = count + 1
    count += 1

Decay_lr = [0 for i in range(75)]
Decay_sigma = [0 for i in range(75)]




epoch = 0

while epoch < 75:

    X_array, Y_array = sklearn.utils.shuffle(X_array, Y_array)

    #begins index  ########
    index = 0

    while index < len(X_array):

        X = X_array[index]
        Y = Y_array[index]

        count = 0
        while count < 9:

            distance[count] = math.sqrt ( ((X - w[count][0])**2) + ((Y - w[count][1])**2) )

            count += 1

        oldw = w

        mindist = min(distance)

        # get the node with the lowest distance
        N = np.argmin(distance)

        # count the wins at the last epoch
        if epoch == 74:
            Total_wins[N] += 1
```

```
    j = 0
    while j < 9:


        # get the neighborhood function
        bor = math.sqrt ( ((D[N][0] - D[j][0])**2) + ((D[N][1] - D[j][1])**2) )

        # updating the X side of the weight
        w[j][0] = oldw[j][0] + lr * math.exp( (-(bor)**2) / ( 2*((sigma)**2) ) )  *  (X - oldw[j][0])

        # updating the Y side of the weight
        w[j][1] = oldw[j][1] + lr * math.exp( (-(bor)**2) / ( 2*((sigma)**2) ) )  *  (Y - oldw[j][1])

        if epoch == 74:
            Error_Matrix[j] = ((abs(w[j][0] - oldw[j][0]) / w[j][0] )*100) + ((abs(w[j][1] - oldw[j][1]) /
w[j][1])*100)

        j += 1

    #decaying the learning rate
    lr = lr * math.exp(-(epoch/1000000))



    #decaying sigma
    oldsigma = sigma

    sigma = sigma * math.exp(-(epoch/1000000))

    index += 1

    # Plot the results of the graph at each number of epoch so we can see how the data is
moving
    if moving == 1:
        count = 0
        while count < 9:
            #print(count)
            plot_X[count] = w[count][0]
            plot_Y[count] = w[count][1]
            count += 1

        #plot the results of the graph
        print(epoch)
        plt.title('Weights and Sample points')
```

```
            plt.xlabel('X_axis')
            plt.ylabel('Y_axis')
            plt.scatter(X_array, Y_array, label = 'Sample points')
            plt.scatter(plot_X, plot_Y, label = 'Weights')
            plt.legend()
            plt.show()



            moving = 0

        Decay_lr[epoch] = lr
        Decay_sigma[epoch] = sigma

    moving += 1
    epoch += 1


plt.title('learning rate decay')
plt.xlabel('epoch')
plt.ylabel('decay')
plt.scatter(X_axis, Decay_lr)
plt.show()

plt.title('learning sigma decay')
plt.xlabel('epoch')
plt.ylabel('decay')
plt.scatter(X_axis, Decay_sigma)
plt.show()



#lattice structure generated to calculate the U_Matrix

Z = [[[w[0][0], w[0][1]],  [w[1][0], w[1][1]], [w[2][0], w[2][1]]],

     [[w[3][0], w[3][1]],  [w[4][0], w[4][1]], [w[5][0], w[5][1]]],

     [[w[6][0], w[6][1]],  [w[7][0], w[7][1]], [w[8][0], w[8][1]]]]

# store the middle weight values

Nearest_Next =  [0 for i in range(4)]
```

```
Next = [0 for i in range(6)]

count = 0
while count < 9:
    #print(count)
    plot_X[count] = w[count][0]
    plot_Y[count] = w[count][1]
    count += 1



# find the middle value of the Euclidean weights
count3 = 0
switch = 0

count1 = 0
count2 = 0
count = 0
while count < 3:

    if count > 0:
        switch = 1



    count1 = 0
    while count1 < 3:

        if switch == 1 and count1 < 2:

            x_value = Z[count][count1][0] - Z[count-1][count1+1][0]
            y_value = Z[count][count1][1] - Z[count-1][count1+1][1]

            sum_1 = math.sqrt( ((x_value)**2) + ((y_value)**2) )


            x_value2 = Z[count][count1+1][0] - Z[count-1][count][0]
            y_value2 = Z[count][count1+1][1] - Z[count-1][count][1]

            sum_2 = math.sqrt( ((x_value2)**2) + ((y_value2)**2) )

            Nearest_Next[count2] = (sum_1 + sum_2) / 2
            count2 += 1
        count1 += 1
```

```
    count += 1



count2 = 0

count1 = 0
count = 0
while count < 3:

    count1 = 0
    while count1 < 3:
        if count1 < 2:

            x_value = Z[count][count1][0] - Z[count][count1+1][0]
            y_value = Z[count][count1][1] - Z[count][count1+1][1]

            sum_1 = math.sqrt( ((x_value)**2) + ((y_value)**2) )

            Next[count2] = sum_1

            count2 += 1
        count1 += 1

    count += 1

# initialize the U_Matrix
U_Ma = [0 for i in range(19)]

count1 = 0

count6 = 0
count5 = 0
count4 = 0
count3 = 0
count2 = 0
switch1 = 0
switch = 0
count = 0

while count < 5:

    if switch == 0:
```

```
        count1 = 0

        while count1 < 5:
            if switch1 == 0:
                #print("print count2")
                #print(count2)
                U_Ma[count2] = round(Total_wins[count3], 2)
                count2 += 1
                count3 += 1
                switch1 = 1
                if count1 == 4:
                    switch1 = 0

            else:

                U_Ma[count2] = round(Next[count4], 2)
                count4 += 1
                count2 += 1
                switch1 = 0

            count1 += 1

        switch = 1

    else:
        count5 = 0
        while count5 < 2:
            U_Ma[count2] = round(Nearest_Next[count6], 2)
            count2 += 1
            count6 += 1

            count5 += 1

        switch = 0

    count += 1



#print out the U_Matrix
switch = 0
a = 0
b = 0
c = 0
```

```
d = 0
e = 0
f = 0
h = 0
count = 0

print('\n')

while count < 5:
   if switch == 0:

      b  = a + 1
      c  = b + 1
      d  = c + 1
      e  = d + 1

      print('' + str(U_Ma[a]) +'         '''+ str(U_Ma[b])
      +'''     '+ str(U_Ma[c]) +'         '''+ str(U_Ma[d]) +'''       ' + str(U_Ma[e]) )
      switch = 1

   else:

      f = e + 1
      h = f + 1
      a = h + 1
      print('\n')
      print('          ''' + str(U_Ma[f]) +'''              '''+ str(U_Ma[h]) +'''            ')
      print('\n')
      switch = 0

   count += 1

print('\n')

print("print out the U_matrix")
print(U_Ma)

print("print out the nearest next in the U_matrix")
print(Nearest_Next)

print("print out the next in the U_matrix")
print(Next)

print("print the winning number for each node")
```

```
print(Total_wins)

print("print the error for each weight")
print(Error_Matrix)

print("print the new lattice structure")
print(Z)


count = 0
while count < 9:
    #print(count)
    plot_X[count] = w[count][0]
    plot_Y[count] = w[count][1]
    count += 1


#plot the results of the graph
plt.title('Weights and Sample points')
plt.xlabel('X_axis')
plt.ylabel('Y_axis')
plt.scatter(X_array, Y_array, label = 'Sample points')
plt.scatter(plot_X, plot_Y, label = 'Weights')
plt.legend()
plt.show()
```

decay the learn rate and Gaussian neighborhood

vary the size of the lattice

show a typical result in data space with the prototypes “connected” based on their relationship in lattice space

Implement the modified U-Matrix (fences in the SOM) and the Density representation (simultaneously).

5 by 5 lattice SOM code:

```
import pandas as pd
from PIL import Image
from numpy import asarray
import numpy as np
import matplotlib.pyplot as plt
#!pip install pyexcel


import math
import pyexcel as pe
import pyexcel.ext.xls



#your_matrix = pe.get_array('RBF_Data.xlsx')


df = pd.read_excel('RBF_Data.xlsx')
print(df)


#Convert the column one also known as X to an array
X_array = df[["Unnamed: 0"]].to_numpy()
print("##########")
print('\n')
print("print out X")
print(X_array)





#convert column two also known as Y to an array
Y_array = df[["Unnamed: 1"]].to_numpy()
print("##########")
print('\n')
print("print out Y")
print(Y_array)
```

```
#convert column three also known as Label to an array
L_array = df[["Label"]].to_numpy()
print("##########")
print('\n')
print("print out Label")
print(L_array)
print("########")





##### print out the group to figure out the best K to use
# empty list, will hold color value
# corresponding to x
col =[]

for i in range(0, len(L_array)):
    if L_array[i] == 1:
        col.append('blue')
    else:
        col.append('magenta')

for i in range(len(L_array)):

    # plotting the corresponding x with y
    # and respective color
    plt.scatter(X_array[i], Y_array[i], c = col[i], s = 10,
                linewidth = 0)


print("print out plot. The K values will be chosen base on the cluster of the two different groups")
print("Group one cluster center/K is (1,7) and group negative one is (1,2)")
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Blue for group 1\nand Magenta for group -1')
plt.legend()
plt.show()




#initialize the X and Y values for the results
```

```
plot_X = [0 for i in range(25)]
plot_Y = [0 for i in range(25)]

#count the winners of each node at the last epoch
Total_wins = [0 for i in range(61)]

Error_Matrix = [0 for i in range(25)]

#initialize the weights
w = [[1, 4],
     [1.1, 4.1],
     [.94, 4],
     [1.5, 4],
     [1.3, 2.8],
     [1, 3],
     [1.2, 3.9],
     [1.4, 4],
     [1.1, 3.1],
     [0.4, 4],
     [0.5, 3.7],
     [1.8, 4.1],
     [1, 4.2],
     [1, 5],
     [1.1, 4.8],
     [1.9, 4.6],
     [1.2, 4.2],
     [1.2, 3.8],
     [1, 5.1],
     [1.1, 2.3],
     [1.2, 2.3],
     [0.8, 2.4],
     [0.78, 2.8],
     [1.2, 3.69],
     [1,2]]



count = 0
while count < 25:
    #print(count)
    plot_X[count] = w[count][0]
    plot_Y[count] = w[count][1]
    count += 1
```

```
#initialize neighbor
bor = 0

#initialize learning rate
lr = 0.03

#initialize sigma
sigma = 0.1

oldsigma = 0

#keep track of the weights moving
moving = 0


#plot the graph to find the best weights


plt.title('Weights and Sample points')
plt.xlabel('X_axis')
plt.ylabel('Y_axis')
plt.scatter(X_array, Y_array, label = 'Sample points')
plt.scatter(plot_X, plot_Y, label = 'Weights')
plt.legend()
plt.show()



#initialize value for minimum distance and it's respective node
mindistance = 0
winning_node = 0



oldw = [[0 for i in range(2)] for j in range(25)]

# initialize distance
distance = [0 for i in range(25)]


# initialize lattice structure
D = [[0, 0],
    [1, 0],
```

```
    [2, 0],
    [3, 0],
    [4, 0],
    [0, 1],
    [1, 1],
    [2, 1],
    [3, 1],
    [4, 1],
    [0, 2],
    [1, 2],
    [2, 2],
    [3, 2],
    [4, 2],
    [0, 3],
    [1, 3],
    [2, 3],
    [3, 3],
    [4, 3],
    [0, 4],
    [1, 4],
    [2, 4],
    [3, 4],
    [4, 4]]

#initialize the placeholders use to tranverse through the dataset
X = 0
Y = 0

X_axis = [0 for i in range(280)]
count = 0
while count < 280:
   X_axis[count] = count + 1
   count += 1

Decay_lr = [0 for i in range(280)]
Decay_sigma = [0 for i in range(280)]


epoch = 0

while epoch < 280:

   X_array, Y_array = sklearn.utils.shuffle(X_array, Y_array)
```

```
#print("print the length of X_array")
#print(len(X_array))

#begins index  ########
index = 0

while index < len(L_array):

    X = X_array[index]
    Y = Y_array[index]

    count = 0
    while count < 25:

        distance[count] = math.sqrt ( ((X - w[count][0])**2) + ((Y - w[count][1])**2) )

        count += 1

    oldw = w

    mindist = min(distance)

    # get the node with the lowest distance
    N = np.argmin(distance)

    # count the wins at the last epoch
    if epoch == 279:
        Total_wins[N] += 1

    j = 0
    while j < 25:


        # get the neighborhood function
        bor = math.sqrt ( ((D[N][0] - D[j][0])**2) + ((D[N][1] - D[j][1])**2) )

        # updating the X side of the weight
        w[j][0] = oldw[j][0] + lr * math.exp( (-(bor)**2) / ( 2*((sigma)**2) ) )  *  (X - oldw[j][0])

        # updating the Y side of the weight
        w[j][1] = oldw[j][1] + lr * math.exp( (-(bor)**2) / ( 2*((sigma)**2) ) )  *  (Y - oldw[j][1])

        if epoch == 279:
```

```
            Error_Matrix[j] = ((abs(w[j][0] - oldw[j][0]) / w[j][0] )*100) + ((abs(w[j][1] - oldw[j][1]) /
w[j][1])*100)

          j += 1

        #decaying the learning rate
        lr = lr * math.exp(-(epoch/10000000))


        #decaying sigma
        oldsigma = sigma

        sigma = sigma * math.exp(-(epoch/10000000))

        index += 1


    if moving == 10:
          count = 0
          while count < 9:
            #print(count)
            plot_X[count] = w[count][0]
            plot_Y[count] = w[count][1]
            count += 1

          #plot the results of the graph
          print(epoch)
          plt.title('Weights and Sample points')
          plt.xlabel('X_axis')
          plt.ylabel('Y_axis')
          plt.scatter(X_array, Y_array, label = 'Sample points')
          plt.scatter(plot_X, plot_Y, label = 'Weights')
          plt.legend()
          plt.show()

          #print(epoch)
          moving = 0


    Decay_lr[epoch] = lr
    Decay_sigma[epoch] = sigma

    moving += 1
    epoch += 1
```

```
plt.title('learning rate decay')
plt.xlabel('epoch')
plt.ylabel('decay')
plt.scatter(X_axis, Decay_lr)
plt.show()

plt.title('learning sigma decay')
plt.xlabel('epoch')
plt.ylabel('decay')
plt.scatter(X_axis, Decay_sigma)
plt.show()


Z = [[[w[0][0],  w[0][1]],   [w[1][0],  w[1][1]],   [w[2][0], w[2][1]],    [w[3][0], w[3][1]],    [w[4][0],
w[4][1]]   ],

     [[w[5][0],  w[5][1]],   [w[6][0],  w[6][1]],   [w[7][0], w[7][1]],    [w[8][0], w[8][1]],    [w[9][0], w[9][1]]
],

     [[w[10][0], w[10][1]],  [w[11][0], w[11][1]],  [w[12][0], w[12][1]],  [w[13][0], w[13][1]],  [w[14][0],
w[14][1]] ],

     [[w[15][0], w[15][1]],  [w[16][0], w[16][1]],  [w[17][0], w[17][1]],  [w[18][0], w[18][1]],  [w[19][0],
w[19][1]] ],

     [[w[20][0], w[20][1]],  [w[21][0], w[21][1]],  [w[22][0], w[22][1]],  [w[23][0], w[23][1]],  [w[24][0],
w[24][1]] ]

    ]



# store the middle weight values

Nearest_Next =  [0 for i in range(16)]

Next = [0 for i in range(20)]

count = 0
while count < 25:
```

```
    #print(count)
    plot_X[count] = w[count][0]
    plot_Y[count] = w[count][1]
    count += 1



# find the middle value of the Euclidean weights
count3 = 0
switch = 0

count1 = 0
count2 = 0
count = 0
while count < 5:

    if count > 0:
        switch = 1



    count1 = 0
    while count1 < 5:

        if switch == 1 and count1 < 4:

            x_value = Z[count][count1][0] - Z[count-1][count1+1][0]
            y_value = Z[count][count1][1] - Z[count-1][count1+1][1]

            sum_1 = math.sqrt( ((x_value)**2) + ((y_value)**2) )


            x_value2 = Z[count][count1+1][0] - Z[count-1][count][0]
            y_value2 = Z[count][count1+1][1] - Z[count-1][count][1]

            sum_2 = math.sqrt( ((x_value2)**2) + ((y_value2)**2) )

            Nearest_Next[count2] = (sum_1 + sum_2) / 2
            count2 += 1
        count1 += 1
    count += 1
```

```
count2 = 0

count1 = 0
count = 0
while count < 5:

    count1 = 0
    while count1 < 5:
        if count1 < 4:

            x_value = Z[count][count1][0] - Z[count][count1+1][0]
            y_value = Z[count][count1][1] - Z[count][count1+1][1]

            sum_1 = math.sqrt( ((x_value)**2) + ((y_value)**2) )

            Next[count2] = sum_1

            count2 += 1
        count1 += 1

    count += 1

# initialize the U_Matrix
U_Ma = [0 for i in range(61)]

count1 = 0

count6 = 0
count5 = 0
count4 = 0
count3 = 0
count2 = 0
switch1 = 0
switch = 0
count = 0

while count < 9:

    if switch == 0:

        count1 = 0

        while count1 < 9:
            if switch1 == 0:
```

```
            #print("print count2")
            #print(count2)
            U_Ma[count2] = round(Total_wins[count3], 2)
            count2 += 1
            count3 += 1
            switch1 = 1
            if count1 == 8:
                switch1 = 0

        else:

            U_Ma[count2] = round(Next[count4], 2)
            count4 += 1
            count2 += 1
            switch1 = 0

        count1 += 1

    switch = 1

else:
    count5 = 0
    while count5 < 4:
        U_Ma[count2] = round(Nearest_Next[count6], 2)
        count2 += 1
        count6 += 1

        count5 += 1

    switch = 0

count += 1




switch = 0

a = 0
b = 0
c = 0
d = 0
e = 0
f = 0
```

```
g = 0
h = 0
i = 0
k = 0
q = 0
r = 0
s = 0

count = 0

print('\n')

while count < 9:
    if switch == 0:

        b  = a + 1
        c  = b + 1
        d  = c + 1
        e  = d + 1
        f  = e + 1
        g  = f + 1
        h  = g + 1
        i  = h + 1


        print('' + str(U_Ma[a]) +'         '''+ str(U_Ma[b])
        +'''     '+ str(U_Ma[c]) +'         '''+ str(U_Ma[d]) +'''      ' + str(U_Ma[e])
        +'       '''+ str(U_Ma[f]) +'''     ' + str(U_Ma[g]) +  '    '''+ str(U_Ma[h]) +'''         ' + str(U_Ma[i]) )

        switch = 1

    else:

        k = i + 1
        q = k + 1
        r = q + 1
        s = r + 1
        a = s + 1
        print('\n')
        print('          ''' + str(U_Ma[k]) +'''                 '''+ str(U_Ma[q]) +'''    '+
        '          ''' + str(U_Ma[r]) +'''              '''+ str(U_Ma[s]) +'''             '     )
        print('\n')
        switch = 0
```

```
    count += 1

print('\n')

print("print out the U_matrix")
print(U_Ma)

print("print out the nearest next in the U_matrix")
print(Nearest_Next)

print("print out the next in the U_matrix")
print(Next)

print("print the winning number for each node")
print(Total_wins)

print("print the error for each weight")
print(Error_Matrix)

print("print the new lattice structure")
print(Z)




count = 0
while count < 25:
    #print(count)
    plot_X[count] = w[count][0]
    plot_Y[count] = w[count][1]
    count += 1


#plot the results of the graph
plt.title('Weights and Sample points')
plt.xlabel('X_axis')
plt.ylabel('Y_axis')
plt.scatter(X_array, Y_array, label = 'Sample points')
plt.scatter(plot_X, plot_Y, label = 'Weights')
plt.legend()
plt.show()
```

decay the learn rate and Gaussian neighborhood

vary the size of the lattice

show a typical result in data space with the prototypes “connected” based on their relationship in lattice space

Implement the modified U-Matrix (fences in the SOM) and the Density representation (simultaneously).