a. **IMPLEMENTATION**

Below is the code and first results for Kohonen SOM for RBF Dataset:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
import random, time, math
import pandas as pd
```

Necessary **libraries** are imported:

a. Numpy for mathematical calculations.
b. Matplotlib to plot necessary graphs.
c. Random, time and math for random arrays/numbers, time and some important mathematical notations respectively.
d. Pandas to read .xlsx file.

```
data = pd.read_excel('RBF_Data.xlsx')
data= data.drop(['Label'], axis=1)

max_number = data.max().max()

norm_input = data.divide(max_number)
norm_input_np=norm_input.to_numpy()
```

```
norm_input_np
```

```
array([[ 0.03670951,  0.53424789],
       [ 0.05852265,  0.49011634],
       [ 0.12113092,  0.48989856],
       ...,
       [-0.02194689,  0.19315027],
       [ 0.09141455,  0.09261915],
       [-0.12196734,  0.47573364]])
```

Data is first read, the label is removed as SOM does not require any labels, normalized between zero and one, and converted to numpy afterwards for further processing.

Resultant data is shown as output to visualize.

```
width = 20
height = 20
inputs_len = 2
```

```
lr = 0.9
lr_decay=0.1

batch_size=32
#gaussian parameters

neighbor_dist = None
nb_decay=1
```

The size of the feature map is then defined as by width times height, it can be changed and hardcoded. On the other hand input_len represents the number of inputs we have i.e number of columns in our datasets for input; for this dataset it is 2, therefore it is assigned.

Learning rate and lr_decay can be initialized on will. Here it is initialized to decay from higher learning factor and decrease gradually – exponentially as we will see it soon.

Batch size is introduced to input data, during training, in batches to fasten the process. The higher the batch size the more time the training is likely to consume but it will adhere better results.

neighbor_dist defines **the size of gaussian bell**, the percentage of winning trophy the neighborhood cluster will share. Here, nb_decay on the other hand contains a factor by which we would like to shrink our circle to be less generous, gradually.

A function is defined to be called for **training**. Training is nothing here but iterations to update weights with respect to Euclidean distance. We will in further code witness the maths behind the process of distance calculations, weight updates and neighborhood processes.

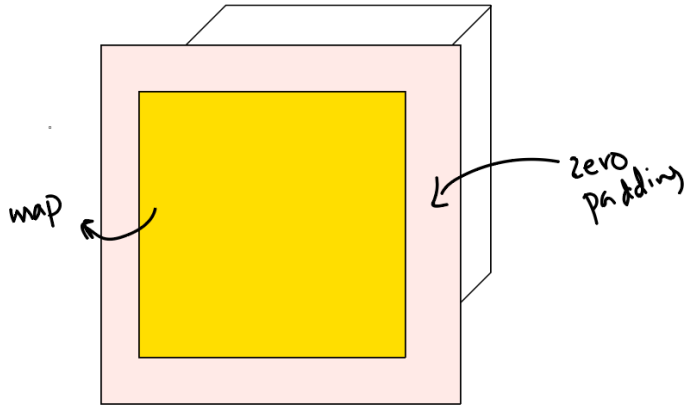
weights_final is most crucial here in code to be followed. It is first assigned with random floats between 0 and 1, with dimension of height times width times input length. This array will be further updated to get our final weight matrix.

neighbor_dist variable can be hardcoded to be some value, which is by default is None, but we do need this to be something to witness a gradual growing or decreasing phenomenon in our results. For that it be assigned a very small value:

$$neighborhood\ distance = \frac{\min\left(height, width\right)}{1.3}$$

Which is then converted to integer to make it callable for further calculations.

Since, to apply gaussian process onto the vector to share the profit, the gaussian vector will serve as a filter which is applied to an array – just like CNNs. Therefore, it is an intelligent choice **to pad the vector with zeros** to avoid errors while the process. After padding in two dimensions of feature map it is then saved back to our weights_final variable.



From training function, finally the training is started where first the **gaussian filter** is prepared. It is going to be a matrix which will be rubbed – as it is visualized like that- over the weight matrix keeping the mean -the max value- on mean.

Formula used to calculated the probabilities – which are used as factors here which are to be multiplied with the numbers- is as follows:

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

neighbor_dist decides the size of the vector, while sigma is calculated using that size. Here, it should be noted that sigma and neighbor_dist are directly proportional therefore, different values for neighbor_dist will yield different result. neighbor_dist can be hard assigned.

In the above code block it is worth noting that a new nb_weight – which defines a dummy neighbor weight filter- is initialized first:

And then gaussian prob is calculated using the formula:

Respective factors are saved in nb_weight array with the location/distance of every factor with the mean in change_x and change_y. these change variable defines the change in distance from mean.

After which necessary operations are done to make them callable to be processed along with normalizing and filtering the neighbor weight matrix/filter.

After creating a filter for gaussian process, the real training is initialized. The loop with run batch_count times, the variable was discussed in previous lines.

The most important line here is the **best_matches** array, which is going to save best match node indesxes.

After getting a batch of input matrix for every iteration, distance between a sample and the specific cluster for that iteration is calculated. Here for every iteration: node_vect contains the cluster weights and dist defines **the Euclidean distance** between both matrices. The best matches are then appended along with their positions in ‘x’ and ‘y’ variable in the array to be used in further code.

After initializing gaussian filter **and finding best matching clusters and their positions**, it is time to **update the winning cluster with their neighbors**. old_coeffs contains the cluster for given row, which is then update with a learning factor 'new_lr'. update_vect contains the updated values with the formula:

$$w_k = w_k + \eta(t) \cdot h_{ik}(t) \cdot (x^{(n} - w_k)$$

weights are then updated with the new addition, and hence after certain iterations over rows final weights are yielded.

The **average distance** is then printed which represents the overall performance – since there is nothing like error in this approach, all there is, is distance. Learning factor is continuously update with the formula:

$$\eta(t) = \eta_0 \exp\left(-\frac{t}{T_1}\right)$$

And is saved in new_lr. weights_final now contains unpadded weights which are to be observed further.

Training function finally returns the trained Boolean and weights yielded.

OUTPUT:

The training function can be used as follows:

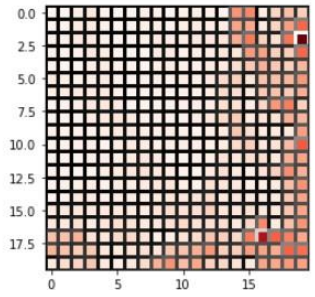
```
trained, weights_final= training(norm_input_np,n_iter=30)
```

```
Average distance = 0.26581
Average distance = 0.27047
Average distance = 0.26714
Average distance = 0.25623
Average distance = 0.25617
Average distance = 0.25208
Average distance = 0.25292
Average distance = 0.24486
Average distance = 0.24095
Average distance = 0.23775
Average distance = 0.23089
Average distance = 0.23149
Average distance = 0.22510
Average distance = 0.22458
Average distance = 0.22496
Average distance = 0.23874
Average distance = 0.21797
Average distance = 0.22665
Average distance = 0.22883
Average distance = 0.22075
Average distance = 0.22157
Average distance = 0.22843
Average distance = 0.23219
Average distance = 0.23370
Average distance = 0.23416
Average distance = 0.22733
Average distance = 0.22982
Average distance = 0.22818
Average distance = 0.23085
Average distance = 0.23620
Training done in 297.023313 seconds.
```

Here, norm_input_np contains normalized input matrix along with 30 iterations to be done. So far, with 20 by 20 feature map we achieved a significant amount of convergence which can be seen by the umatrix:

```
norm_umatrix,umatrix =umatrix(height=height, width=width,weights =weights_final, trained=trained)
plt.imshow(umatrix, cmap='Reds')
```

```
<matplotlib.image.AxesImage at 0x1c6b0b8d400>
```



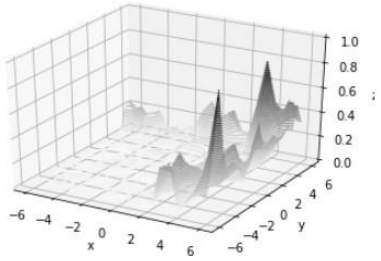
Here, dark pixels indicate more density, convergence and distance with neighbors. It is fascinating to know how with mere 30 iterations, the model managed to converge it self to some conclusion. Here, it is worth noting that the lines/ border color of the pixels also indicated the distance between the pixels.

The convergence can be more visualized via 3d graph:

```
x = np.linspace(-6, 6, umatrix.shape[0])
y = np.linspace(-6, 6,umatrix.shape[1])

X, Y = np.meshgrid(x, y)
Z = norm_umatrix


fig = plt.figure()
ax = plt.axes(projection='3d')
ax.contour3D(X, Y, Z, 50, cmap='binary')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z');
```



Here, we can witness two peaks. Which indicates that two classes exist in the data. Graph can be further strengthened via more iterations.

UMatrix plotting:

Even though plotting a matrix of n x m dimensions isn't a big task. But here we are more interested for each block to represent the distance between the pixels. We should remember that each node of the weight matrix contains two numbers – which are two values for both inputs. There for distance of one node to another will represent the color of that node which is coded as follows:

A check for weights matrix in added in the first line to acknowledge the user to train the matrix first along with the trained variable. After which umatrix matrix is assigned with zeroes. This matrix is going to contain all the values of distance for each node. A simple algorithm is applied to calculate the top, bottom, left and right distance of every node and then assign it the average distance. With this, a matrix of distance values is achieved and plotted.

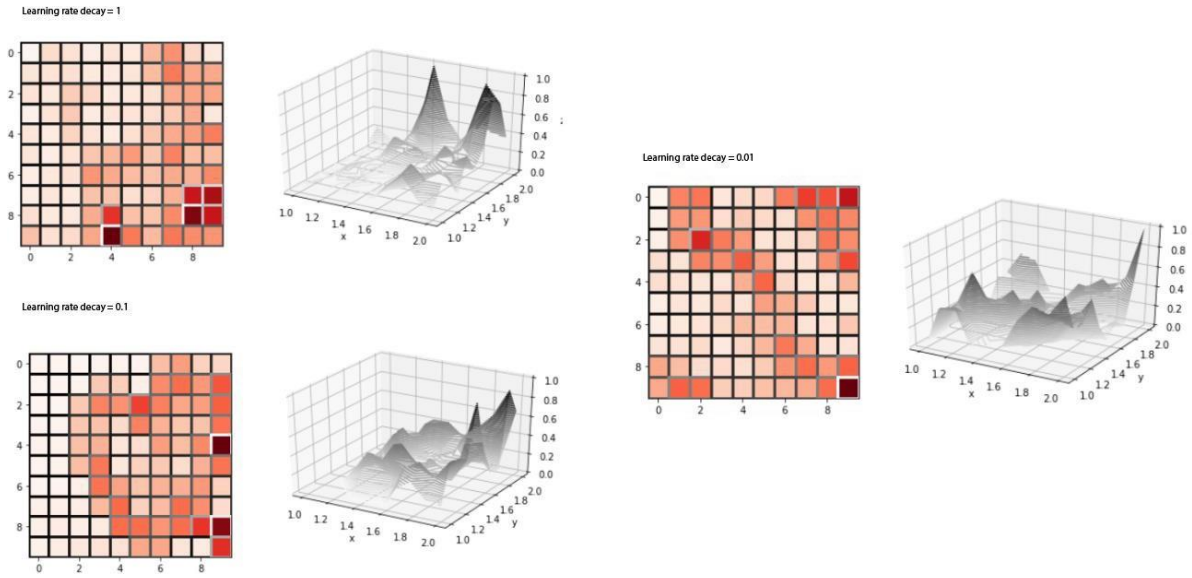
The above coded is to normalize the umatrix and for the fence representation – where the color of lines also represent the distance and makes it easier for the nodes/clusters to pop out.

b. **EFFECTS OF DIFFERENT PARAMETER CHANGES**

1. Changing learning rate:

Keeping number of epochs to 30, and map size to 10 by 10, we can easily observe the conversions.

Results:



Learning rate is decayed through iterations exponentially by the formula:

$$\eta(t) = \eta_0 \exp\left(-\frac{t}{T_1}\right)$$

We can see the learning rate by plotting for it:

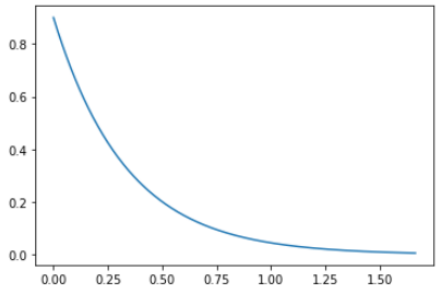
```
x=list()
y=list()
old_lr = 0.9
lr_decay=0.01

for c in range(500):


    new_lr = old_lr*math.e**(-c*lr_decay)
    #old_lr=new_lr
    x.append(c/300)
    y.append(new_lr)
```

```
plt.plot(x,y)
```

```
[<matplotlib.lines.Line2D at 0x1eb1578dcd0>]
```



Which is implemented by this line in the code:

```
new_lr = lr*math.e**(-step*lr_decay)
```

2. Changing Gaussian neighborhood size:

Decaying the neighborhood size in necessary during iteration to get a conical yet slanted curve. The sharp point on top indicates the winning cluster clearly. If neighborhood size is kept intact, we might get less clear and a blob like maxima over curve with no clear indication for winning cluster.

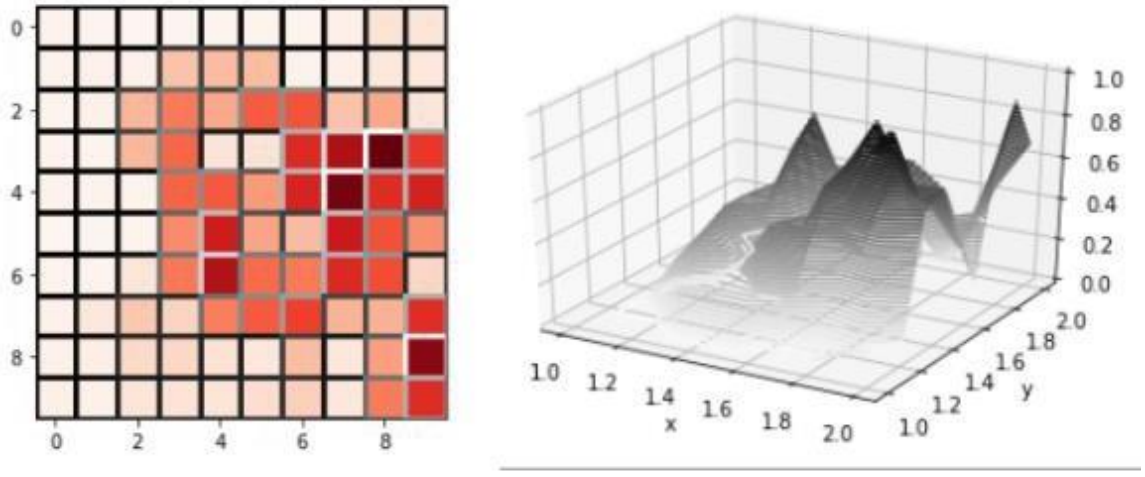
Decaying in implemented by this line in codding:

```
sigma = size/(7+iteration/nb_decay) #to decay gaussian size
```

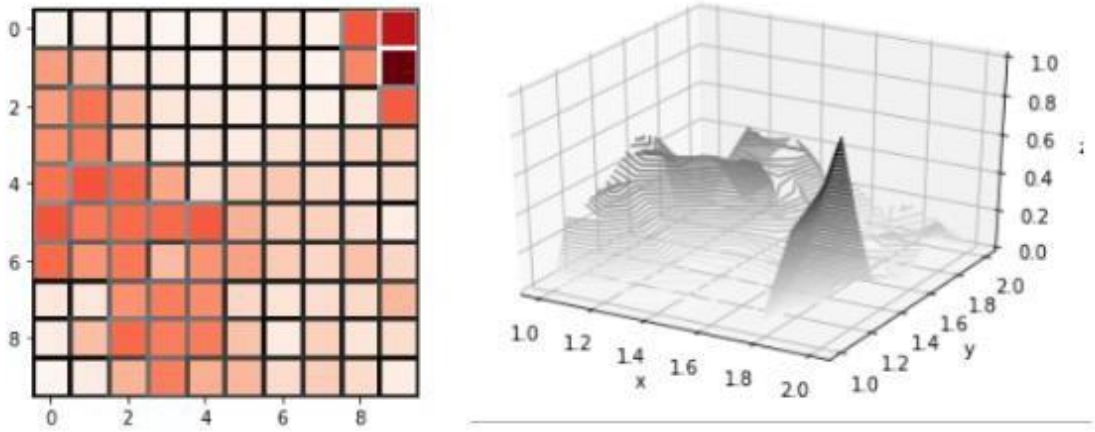
here, are is decayed with every iteration.

We can see the effect of changing decay factor/ neighborhood decay as:

neighborhood decay rate= 1

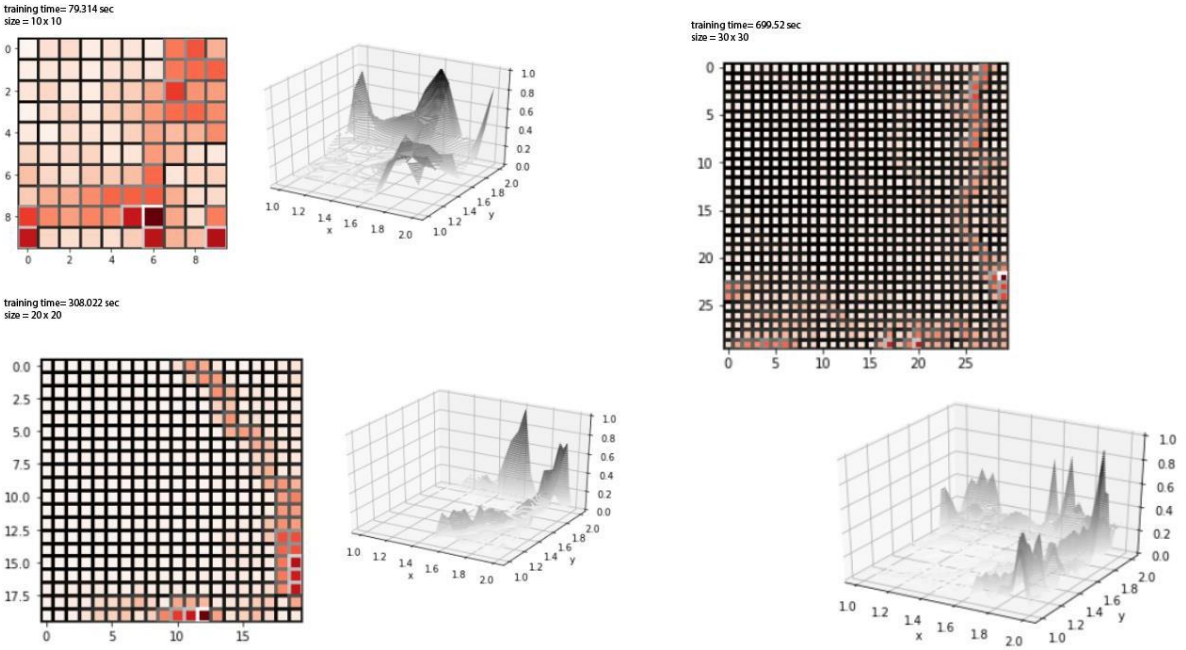


neighborhood decay rate = 0.1



3. Varying Size of SOM Lattice:

Keeping above initials, the feature map size will be varied from 10 by 10 to 30 by 30 to keep sizes relevant and witness the results. Here, time for training is relevant too due to which it will also be observed.



Above results are with 30 iterations, further clarity in graph can be achieved by training it for more period of time.

Findings:

1. Larger feature map yields more points for the graph, on the other hand takes significant amount of training time. such visible winning cluster should be achieved by training a lesser dimension feature map for more epochs.
2. More clusters means more nodes to claim similarity, while larger feature map provides more detail but it can be misleading towards the data sometimes.
3. We can see that 20 by 20 feature map produces somewhat better results indicating 2 classes, it is important to play with parameters to deduce the best parameters for the given data set.