x

Self Organizing Maps

R Programming, FOSSEE Team

# Table of Contents

* Chapter 1: Introduction to SOM
* Chapter 2: Algorithm
* Chapter 3: Guided Tutorial in R – 3.1: Initialization – 3.2: Sampling – 3.3: Competition – 3.4: Cooperation – 3.5: Adaptation
* Chapter 4: Implementation – 4.1: Data Generation – 4.2: Initialization – 4.3: Best Matching Unit – 4.4:Training the SOM
* Chapter 5: Optimization in R
* Chapter 6: References

## Chapter 1: Introduction to SOM

Self organizing maps are a class of artificial neural networks based on competitive learning that helps to organize and understand high dimensional data by reducing the number of dimensions from a high dimensional space to a 2 D map. With SOM, clustering is performed by having several units compete for the current object. Once the data have been entered into the system, the newtwork of artificial neurons is trained by providing information about inputs. The weight vector of the unit is closest to the current object becomes the winning or active unit. During the training stage, the values for the input variables are gradually adjusted in an attempt to preserve neighborhood relationships that exist within the input data set. As it gets closer to the input object, the weights of the winning unit are adjusted as well as its neighbors.(Uoolc, n.d.)

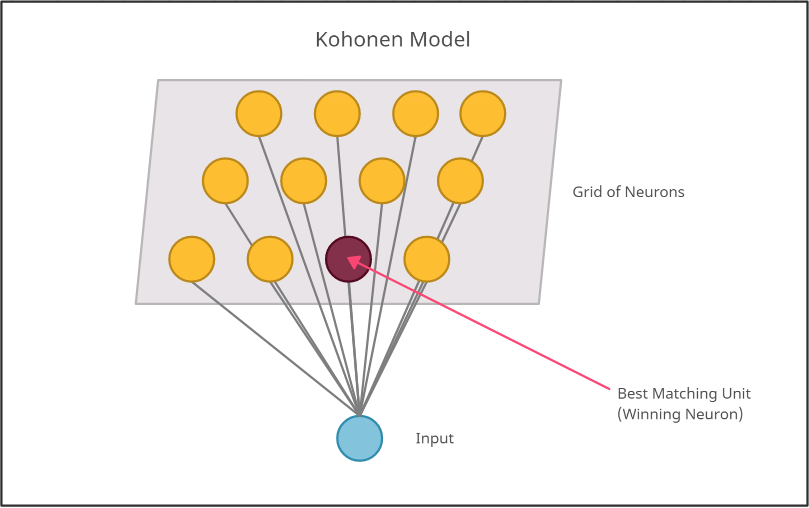


Figure 1: Kohonen model

#### Competitive Learning

The model utilizes using unsupervised learning to map the input through competitive learning in which the output neurons compete amongst themselves to be activated, with the result that only one is activated at any one time. Getting the Best Matching Unit is done by running through all wright vectors and calculating the distance from each weight to the sample vector. The weight with the shortest distance is the winner. There are numerous ways to determine the distance, however, the most commonly used method is the Euclidean Distance and/or Consine Distance.Due to the negative feedback connections between the neurons, the neurons are forced to organise themselves which gave rise to the name Self Organizing Map (SOM).

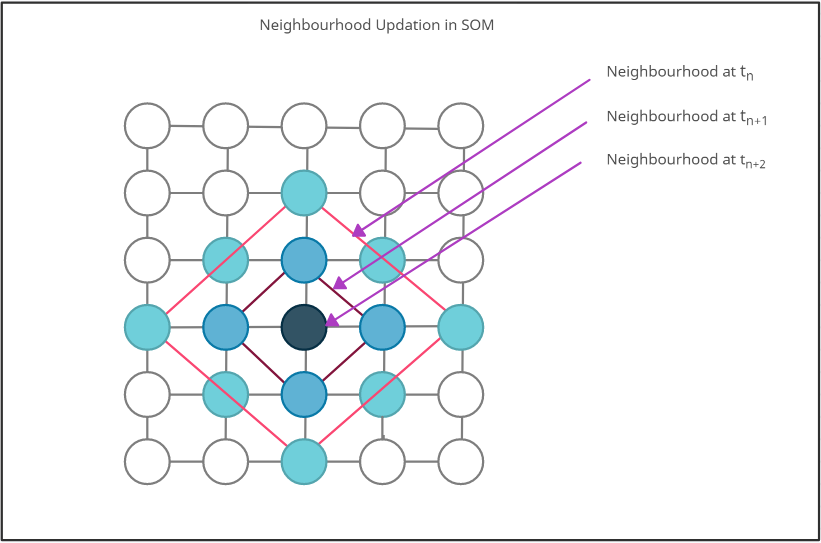


Figure 2: Updating neighbourhood after finding BMU

## Chapter 2: Algorithm

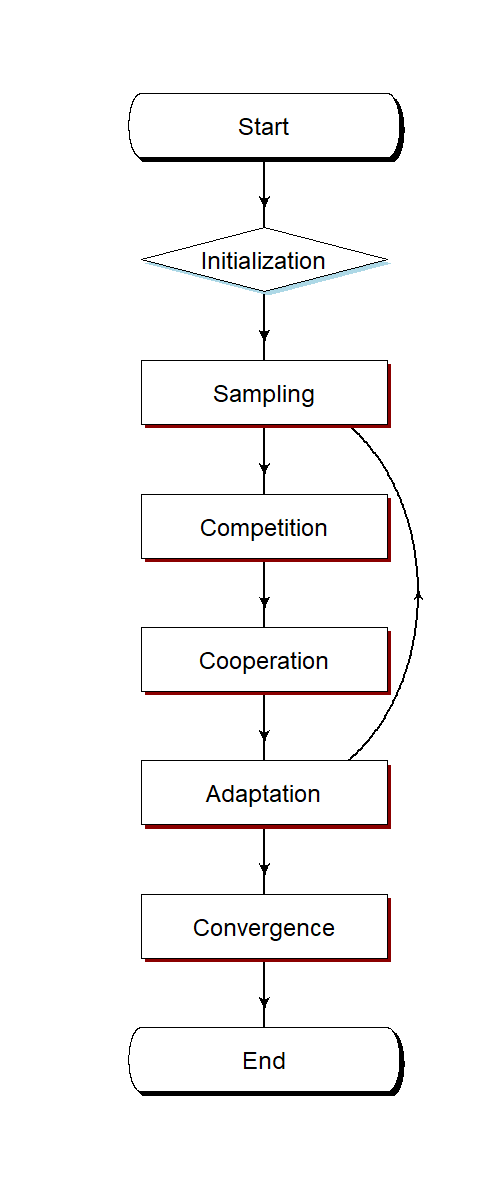


Figure 3: Flowchart

### Steps:

* Initialization - Create a grid of neurons and randomly initialize weights.
* Sampling- Select a random row (vector) from input data.
* Competition- Neurons fight to become the Best Matching Unit which is determined using the discriminant function.
* Cooperation- The winning neuron determines the spatial location of a topological neighbourhood of excited neurons which will cooperate.
* Adaptation- Weights are adjusted with respect to winning neuron, such that a similar input pattern is enhanced.
* We will go back to step 2 and keep repeating the process till the map stops changing or convergence is achieved.

## Chapter 3: Guided Tutorial in R

### 3.1: Initialization

Create a grid of neurons and randomly initialize weights. The neurons are represented by weight vectors of same dimensions as input. The random numbers are generated using the rnorm function which generates random numbers in the range -1 to 1.

Code snippet:

#Let's create a matrix of 10 rows and 5 columns  
t <- matrix(data = rnorm(50), nrow = (10), ncol = 5)  
t

## [,1] [,2] [,3] [,4] [,5]  
## [1,] 1.298608149 0.008340018 -0.11324826 0.1314727 0.03359912  
## [2,] -0.233055390 -0.459301531 -1.23499112 -0.5006449 -1.90712203  
## [3,] 0.524394301 -2.556522343 0.17094332 0.4939170 -0.53521982  
## [4,] -0.009370764 0.289105874 -1.93357036 1.9368278 -1.38490916  
## [5,] -0.808336203 0.157121818 1.59441802 -0.6130572 -0.54694723  
## [6,] -0.214896813 -1.001455589 0.02309109 -0.3933025 -0.44082015  
## [7,] -0.588716948 -0.555342952 -1.77669124 0.2436262 0.13113801  
## [8,] 1.009496254 -1.027393409 0.76837602 -0.1932311 1.03117496  
## [9,] -0.344363415 -0.225153918 1.23500887 -0.7171822 0.56583749  
## [10,] -0.316185561 1.684208639 0.16778926 -1.1407915 0.43536539

### 3.2: Sampling

Select a random row (vector) from input data. The sampling is done using the sample() function in R which retrieves an input row.

i <- sample(t, 1, replace = F)  
i

## [1] 1.594418

### 3.3: Competition

Neurons fight to become the Best Matching Unit which is determined using the discriminant function.Here our discriminant function is Euclidean distance given by the formula:

where and are the two points in n dimensional space and and are the vectors representing their positions between which the Euclidean distance is to be calculated.

Euclidean distance formula

#Lets make a function to calculate euclidean distance.  
euclidean\_distance <- function(x, y) {  
 ret <- sum((x - y)^2)  
 return(ret)  
}  
  
#Let's run this on a sample input  
  
euclidean\_distance(2,4)

## [1] 4

The Best Matching Unit is the neuron which is closest to the input vector. The discriminant function is used to calculate this distance between all the neurons’ weights and the input vector.

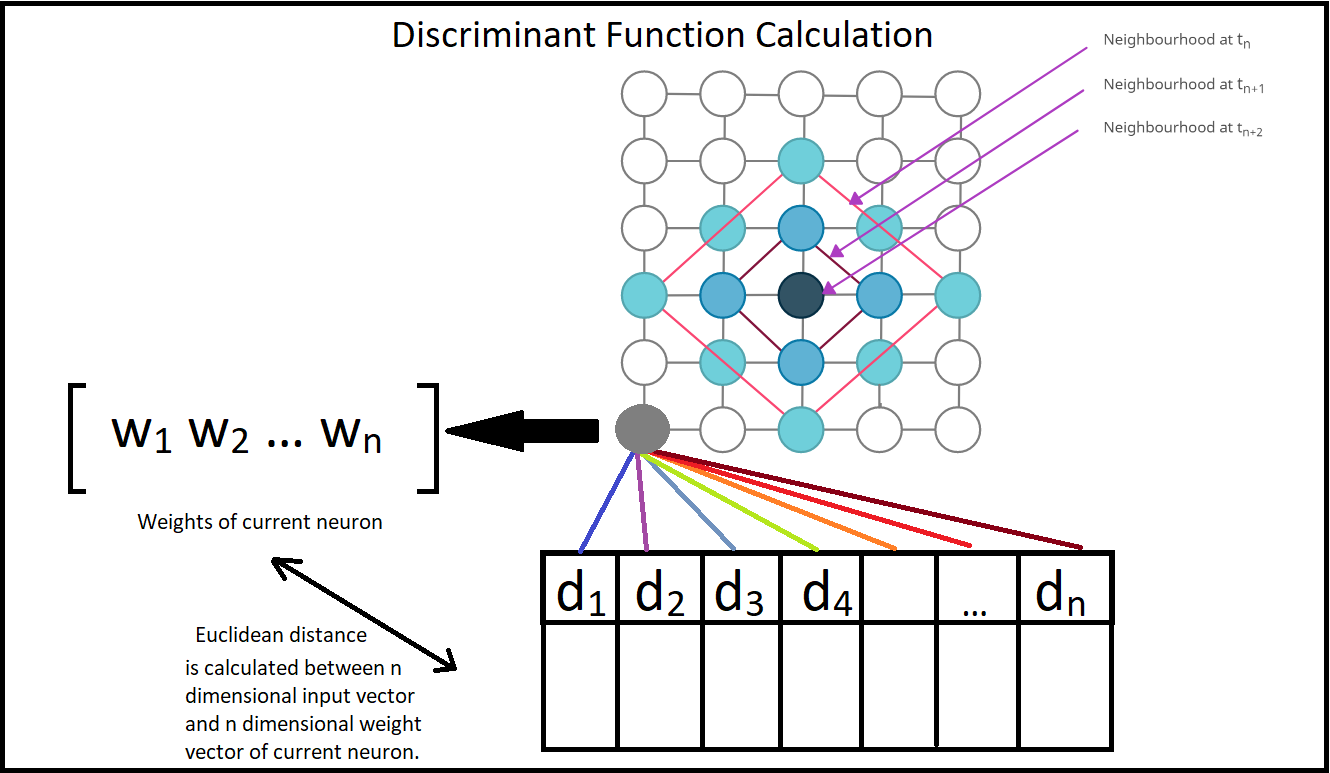


Figure 4: Discriminant function calculation

# x is a single input row of data and input\_grid is the grid  
BMU <- function(x, input\_grid) {   
 distance <- 0  
 min\_distance <- 10000000 # Setting high min dist value  
 min\_ind <- -1 # Setting random min\_ind value  
 for (e in 1:nrow(input\_grid)) # Iterating through grid  
 {  
 distance <- euclidean\_distance(x, input\_grid[e, ]) # euclidean\_distance distance  
 if (distance < min\_distance) {  
 min\_distance <- distance # Updating min distance for winning unit  
 min\_ind <- e # Updating winning neuron  
 }  
 }  
 return(min\_ind-1) #returns index of BMU  
}

### 3.4: Cooperation

The winning neuron determines the spatial location of a topological neighbourhood of excited neurons which will cooperate.

where is the lateral distance between neurons in the grid and is the radius of the neighbourhood over which influence is to be calculated.

Neighbourhood influence calculation formula

# Defining a function to calculate the neighbourhood influence using the radius of neighbourhood and lateral distance.  
influence\_calculation <- function(distance, radius) {  
 ret <- exp(-(distance^2) / (2 \* (radius^2)))  
 return(ret)  
}  
  
# Calculating sample neighbourhood for lateral distance 2 and radius 4.  
influence\_calculation(2,4)

## [1] 0.8824969

### 3.5: Adaptation

Weights are adjusted with respect to winning neuron, such that a similar input pattern is enhanced.

where is the initial radius of neighbourhood, is the iteration of data sampling that we are currently on and is the time constant which is incremented at each iteration, when the SOM gets updated.

Radius decay formula

<>

#Function for the decaying radius for a given iteration current\_iteration  
decay\_radius\_function <- function(radius, current\_iteration, time\_constant) {  
 ret <- radius \* exp(-current\_iteration / time\_constant)  
 return(ret)  
}  
  
# Calculate radius of neighborhood at a iteration 4, with radius 3 and at the 4th iteration  
decay\_radius\_function(3,4,4)

## [1] 1.103638

where is the old learning rate to be updated, is the iteration of data sampling that we are currently on and is the total number of iterations the SOM is trained over.

Learning rate decay formula

#Function for the decaying learning rate  
decay\_learning\_rate <- function(learning\_rate, current\_iteration, n\_iteration) {  
 ret <- learning\_rate \* exp(-current\_iteration / n\_iteration)  
 return(ret)  
}  
  
#Calculating the learning rate of model at the 3rd iteration out of a total of 100 iterations and initial learning rate of 0.1.  
decay\_learning\_rate(0.1,3,100)

## [1] 0.09704455

## Chapter 4: Implementation

### 4.1: Data Generation

For this tutorial, we will demonstrate the working of SOM on a given dataset of 3 dimensions. We will load this dataset from the working directory. We will also import the necessary libraries.

Code Snippet

set.seed(222)  
library(dplyr)  
  
# 1) Reading the data and scaling it  
data <- read.csv("binary.csv", header = T)  
X <- scale(data[, -1])  
data <- X

### 4.2: Initialization

The SOM is in its essence a grid of neurons, each neuron containing a weight vector and a position i,j in the grid. We begin by assigning random values for the initial weight vectors w. The dimensions of the weight vector are equal to the number of input dimensions.

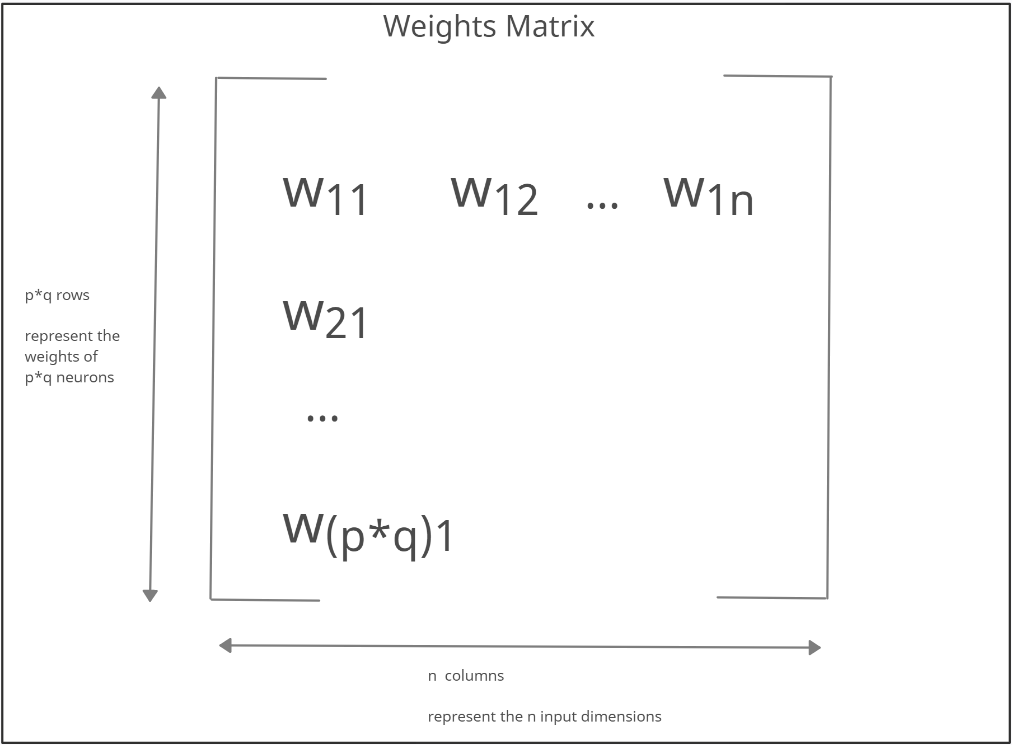


Figure 5: Weights matrix

Code Snippet

#Now lets initialize the weights of the neural network.  
#Creating a 4x4 neural network with 3 dimensions to match the input.  
  
# -----------------------------------------------------  
# This is Step 1 of the Algorithm: Initialization  
# -----------------------------------------------------  
  
create\_grid <- function(n,p) {  
 ret <- matrix(data = rnorm(n \* p), nrow = n, ncol = p)  
 return(ret)  
}  
grid <- create\_grid(16,3)  
grid

## [,1] [,2] [,3]  
## [1,] 1.487757090 -2.005512989 -0.2154811  
## [2,] -0.001891901 0.007509885 -0.1146497  
## [3,] 1.381020790 0.519490356 -0.2022654  
## [4,] -0.380213631 -0.746295471 0.4064927  
## [5,] 0.184136230 0.726454576 0.6567724  
## [6,] -0.246895883 0.713656667 0.1061908  
## [7,] -1.215560910 -0.650062920 -0.1843974  
## [8,] 1.561405098 1.498696215 0.9460342  
## [9,] 0.427310197 -1.435828082 0.2023869  
## [10,] -1.201023506 -2.161318185 0.4951010  
## [11,] 1.052458495 0.395219851 -0.5693554  
## [12,] -1.305063566 -0.394833956 1.1192945  
## [13,] -0.692607634 -0.309758382 2.2090779  
## [14,] 0.602648854 1.330826619 0.3171826  
## [15,] -0.197753074 -0.817428629 -0.9352971  
## [16,] -1.185874517 0.675893491 0.8136619

### 4.3: Best Matching Unit

The SOM works using competitive learning which selects a best matching unit at each iteration using the discriminant function value closest to the randomly sampled input vector.

Code Snippet

# -----------------------------------------------------  
# This is Step 3 of the Algorithm: Competition  
# -----------------------------------------------------  
  
# euclidean\_distance function  
euclidean\_distance <- function(x, y) {  
 ret <- sum((x - y)^2)  
 return(ret)  
}  
  
# Function to return winning neuron  
BMU <- function(x, input\_grid) {   
 distance <- 0  
 min\_distance <- 10000000 # Setting high min dist value  
 min\_ind <- -1 # Setting random min\_ind value  
 for (e in 1:nrow(input\_grid)) # Iterating through grid  
 {  
 distance <- euclidean\_distance(x, input\_grid[e, ]) # euclidean\_distance distance  
 if (distance < min\_distance) {  
 min\_distance <- distance # Updating min distance for winning unit  
 min\_ind <- e # Updating winning neuron  
 }  
 }  
 return(min\_ind-1) #returns index of BMU  
}

### 4.4: Training the SOM.

The SOM follows the algorithm mentioned above to fit the training data till the map stops changing or in other words till the model converges.

Code Snippet

# -----------------------------------------------------  
# This is Step 5 of the Algorithm: Adaptation  
# -----------------------------------------------------  
  
# Defining the updation function first.  
# 1) Decaying radius function  
decay\_radius\_function <- function(radius, current\_iteration, time\_constant) {  
 ret <- radius \* exp(-current\_iteration / time\_constant)  
 return(ret)  
  
}  
  
# -----------------------------------------------------  
# This is Step 4 of the Algorithm: Cooperation  
# -----------------------------------------------------  
  
  
# 2) Decaying learning rate  
decay\_learning\_rate <- function(learning\_rate, current\_iteration, n\_iteration) {  
 ret <- learning\_rate \* exp(-current\_iteration / n\_iteration)  
 return(ret)  
}  
  
# 3) A function to calculate influence over neighboring neurons  
influence\_calculation <- function(distance, radius) {  
 ret <- exp(-(distance^2) / (2 \* (radius^2)))  
 return(ret)  
}  
  
  
SOM <- function(x, input\_grid) {  
   
   
# Defining the training parameters.  
   
 n\_iteration <- 400 # Defining number of iterations  
 initial\_learning\_rate <- 0.05 # Defining initial learning rate  
 initial\_radius <- 3 # Defining initial radius  
 time\_constant <- n\_iteration / log(initial\_radius) # Initializing time constant  
 lateral\_distance\_points=expand.grid(1:sqrt(nrow(input\_grid)),1:sqrt(nrow(input\_grid)))#Initialising physical locations of neurons to figure out lateral distance.  
 rows=sqrt(nrow(input\_grid)) #The square grid is used here - so taking the number of rows as square root of number of entries in the grid.  
 n\_epochs=10 #Defining the number of epochs.  
 for(ne in 1:n\_epochs)  
 {  
 print(ne)  
 old\_grid=input\_grid  
 for (i in 1:n\_iteration) # Looping through for training  
 {  
 sample\_input\_row <- as.vector(unlist(x[sample(1:nrow(x), size = 1, replace = F), ])) # Selecting random input row from given data set  
 new\_radius <- decay\_radius\_function(initial\_radius, i, time\_constant) # Decaying radius  
 new\_learning\_rate <- max(decay\_learning\_rate(initial\_learning\_rate, i, n\_iteration), 0.01) # Decaying learning rate  
 index\_temp <- BMU(sample\_input\_row, input\_grid) # Finding best matching unit for given input row  
 index\_new=c((as.integer(index\_temp/rows))+1,(index\_temp%%rows)+1) #Converting a 1D co-ordinate to a 2D co-ordinate for finding lateral distance on the map.  
 lateral\_distance=sqrt(rowSums(sweep(lateral\_distance\_points,2,index\_new)^2)) #Finding Euclidean distance between the given best matching units and all units on the map.  
 rn=which(lateral\_distance<=new\_radius) #Finding neurons that are within the radius of the winning unit.  
 inf=influence\_calculation(lateral\_distance[rn],new\_radius) #Calculating the influence of the winning neuron on neighbours.  
 diff\_grid=(sweep(input\_grid[rn,],2,sample\_input\_row))\*-1 #A temporary matrix that stores the difference between the data point and the weights of the winning neuron & neighbours.  
 updated\_weights=new\_learning\_rate\*inf\*diff\_grid #The updating operation on the winning and neighbouring neurons.  
 input\_grid[rn,]=input\_grid[rn,]+updated\_weights #Now updating those grid entries that are either the winning neuron or its neighbours.  
 if(isTRUE(all.equal(old\_grid,input\_grid)))  
 {  
 print(i)  
 print("Converged")  
 }  
 }  
 }  
 return(input\_grid) #Returning the updated SOM weights.  
}  
start <- Sys.time()  
gridSOM=SOM(data,grid)

## [1] 1  
## [1] 2  
## [1] 3  
## [1] 4  
## [1] 5  
## [1] 6  
## [1] 7  
## [1] 8  
## [1] 9  
## [1] 10

end <- Sys.time()  
gridSOM

## [,1] [,2] [,3]  
## [1,] 0.74352087 0.70412465 -0.88501576  
## [2,] 0.64275317 0.44668644 -0.74257737  
## [3,] 0.48175345 0.15306486 0.17853916  
## [4,] 0.32883768 0.21113676 0.75269190  
## [5,] 0.65806641 0.44590741 -0.77502593  
## [6,] 0.39796065 -0.05900396 -0.72718328  
## [7,] 0.15899425 -0.26817904 -0.22977991  
## [8,] -0.07254835 -0.28692345 0.44389629  
## [9,] 0.51655479 0.26968424 0.07805282  
## [10,] 0.12077430 -0.36825512 -0.28651102  
## [11,] -0.34586925 -0.82103510 -0.20255336  
## [12,] -0.80880456 -1.00700408 0.11426690  
## [13,] 0.31981819 0.18242487 0.73360545  
## [14,] -0.08786411 -0.29512278 0.44548830  
## [15,] -0.81518112 -1.05492560 0.12637825  
## [16,] -1.09844248 -1.19899201 0.16214440

time\_taken <- end - start  
print(time\_taken)

## Time difference of 3.703874 secs

## Chapter 5: Optimization in R

If you wish to explore further optimization of the SOM code, try running the below code cells and compare the running time of two approaches. The method of optimization here is vectorization.

BMU\_Vectorised <- function(x, input\_grid) {   
 dist\_mtrx=rowSums(sweep(input\_grid,2,x)^2) #Calculating the distance of this row from all the neurons using matrix operations.  
 min\_ind=which.min(dist\_mtrx) #Finding the location of the neuron with the minimum distance.  
 return (min\_ind-1) #Returning the zero-indexed value of the winning neuron.  
}

#Fastest BMU Implementation using vectorisation.  
#x is a single row of data and input\_grid is the grid  
  
  
SOM <- function(x, input\_grid) {  
   
   
# Defining the training parameters.  
   
 n\_iteration <- 400 # Defining number of iterations  
 initial\_learning\_rate <- 0.05 # Defining initial learning rate  
 initial\_radius <- 3 # Defining initial radius  
 time\_constant <- n\_iteration / log(initial\_radius) # Initializing time constant  
 lateral\_distance\_points=expand.grid(1:sqrt(nrow(input\_grid)),1:sqrt(nrow(input\_grid)))#Initialising physical locations of neurons to figure out lateral distance.  
 rows=sqrt(nrow(input\_grid)) #The square grid is used here - so taking the number of rows as square root of number of entries in the grid.  
 n\_epochs=10 #Defining the number of epochs.  
 for(ne in 1:n\_epochs)  
 {  
 print(ne)  
 old\_grid=input\_grid  
 for (i in 1:n\_iteration) # Looping through for training  
 {  
 sample\_input\_row <- as.vector(unlist(x[sample(1:nrow(x), size = 1, replace = F), ])) # Selecting random input row from given data set  
 new\_radius <- decay\_radius\_function(initial\_radius, i, time\_constant) # Decaying radius  
 new\_learning\_rate <- max(decay\_learning\_rate(initial\_learning\_rate, i, n\_iteration), 0.01) # Decaying learning rate  
 index\_temp <- BMU\_Vectorised(sample\_input\_row, input\_grid) # Finding best matching unit for given input row  
 index\_new=c((as.integer(index\_temp/rows))+1,(index\_temp%%rows)+1) #Converting a 1D co-ordinate to a 2D co-ordinate for finding lateral distance on the map.  
 lateral\_distance=sqrt(rowSums(sweep(lateral\_distance\_points,2,index\_new)^2)) #Finding Euclidean distance between the given best matching units and all units on the map.  
 rn=which(lateral\_distance<=new\_radius) #Finding neurons that are within the radius of the winning unit.  
 inf=influence\_calculation(lateral\_distance[rn],new\_radius) #Calculating the influence of the winning neuron on neighbours.  
 diff\_grid=(sweep(input\_grid[rn,],2,sample\_input\_row))\*-1 #A temporary matrix that stores the difference between the data point and the weights of the winning neuron & neighbours.  
 updated\_weights=new\_learning\_rate\*inf\*diff\_grid #The updating operation on the winning and neighbouring neurons.  
 input\_grid[rn,]=input\_grid[rn,]+updated\_weights #Now updating those grid entries that are either the winning neuron or its neighbours.  
 if(isTRUE(all.equal(old\_grid,input\_grid)))  
 {  
 print(i)  
 print("Converged")  
 }  
 }  
 }  
 return(input\_grid) #Returning the updated SOM weights.  
}  
start <- Sys.time()  
gridSOM=SOM(data,grid)

## [1] 1  
## [1] 2  
## [1] 3  
## [1] 4  
## [1] 5  
## [1] 6  
## [1] 7  
## [1] 8  
## [1] 9  
## [1] 10

end <- Sys.time()  
gridSOM

## [,1] [,2] [,3]  
## [1,] 0.86971369 0.72137817 -1.05747447  
## [2,] 0.83370938 0.43827080 -0.72953666  
## [3,] 0.55844249 0.32569595 0.19847814  
## [4,] 0.29211398 0.40476938 0.67500121  
## [5,] 0.82555564 0.44605775 -0.78282413  
## [6,] 0.60455700 0.02287743 -0.65475780  
## [7,] 0.28287941 -0.13918350 -0.27057528  
## [8,] -0.09267776 0.03129728 0.27393822  
## [9,] 0.50070506 0.48507484 0.12533844  
## [10,] 0.25034941 -0.18206473 -0.36199324  
## [11,] -0.31555679 -0.57357808 -0.32284595  
## [12,] -0.70151862 -0.68319511 0.07412104  
## [13,] 0.27247810 0.42158405 0.67852646  
## [14,] -0.24898510 -0.03670322 0.29271284  
## [15,] -0.76004531 -0.74379983 0.08073197  
## [16,] -1.02314913 -0.86836238 0.30767985

time\_taken <- end - start  
print(time\_taken)

## Time difference of 5.779072 secs

nocite: | Kohonen (2012), “Guide to SOM” (n.d.), “SOM Lecture 1” (n.d.), “SOM Tutorial 1” (n.d.), “SOM Tutorial 2” (n.d.), “SOM Lecture 2” (n.d.), “SOM Tutorial 3” (n.d.), “SOM from Scratch” (n.d.)

## Chapter 6: References

“Guide to SOM.” n.d. https://www.superdatascience.com/blogs/the-ultimate-guide-to-self-organizing-maps-soms.

Kohonen, Teuvo. 2012. *Self-Organizing Maps*. Vol. 30. Springer Science & Business Media.

“SOM from Scratch.” n.d. https://inblog.in/Self-Organising-Maps-From-Scratch-7dSqjEKy9h.

“SOM Lecture 1.” n.d. https://www.cs.hmc.edu/ kpang/nn/som.html.

“SOM Lecture 2.” n.d. https://www.cs.bham.ac.uk/ jxb/NN/l16.pdf.

“SOM Tutorial 1.” n.d. https://users.ics.aalto.fi/jhollmen/dippa/node20.html.

“SOM Tutorial 2.” n.d.

“SOM Tutorial 3.” n.d. http://www.pitt.edu/ is2470pb/Spring05/FinalProjects/Group1a/tutorial/som.html.

Uoolc, A Bradford. n.d. “Self-Organizing Map Formation: Foundations of Neural Computation.”