Module 1

Introduction to Self Organizing Maps

Table of Contents

Chapter 1: Introduction to SOM

Self organizing maps are a type of artificial neural network based on competitive learning that work 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 network 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.)

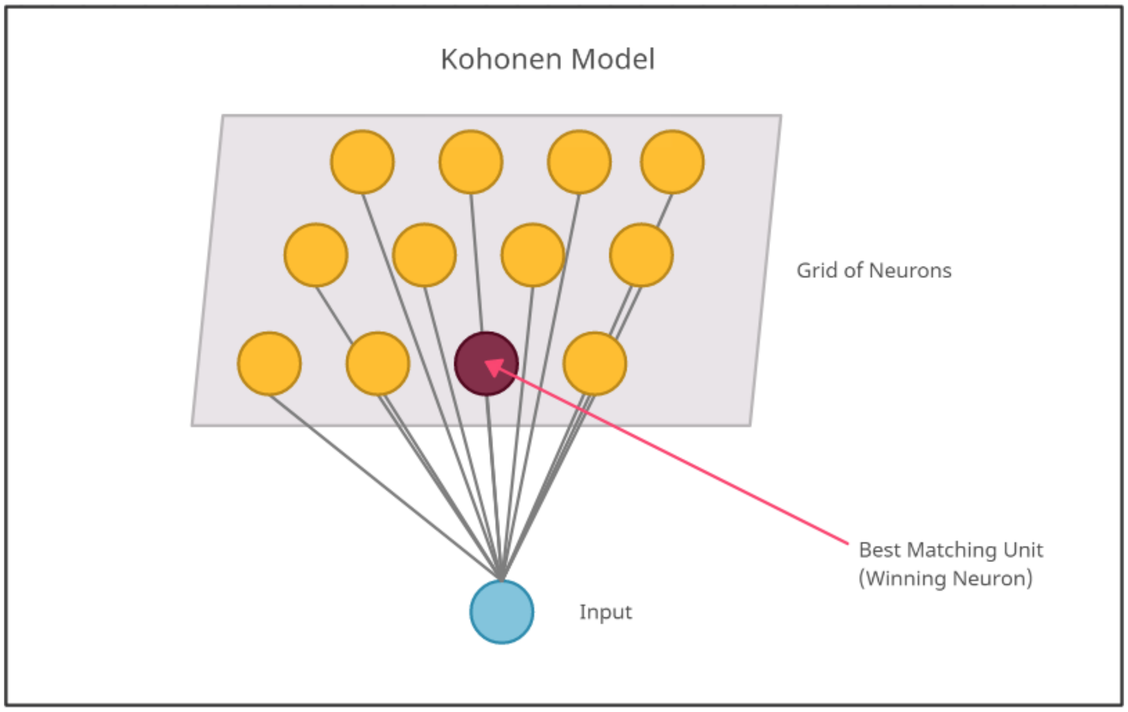


Figure 1: Kohonen model

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 Cosine 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).(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.)

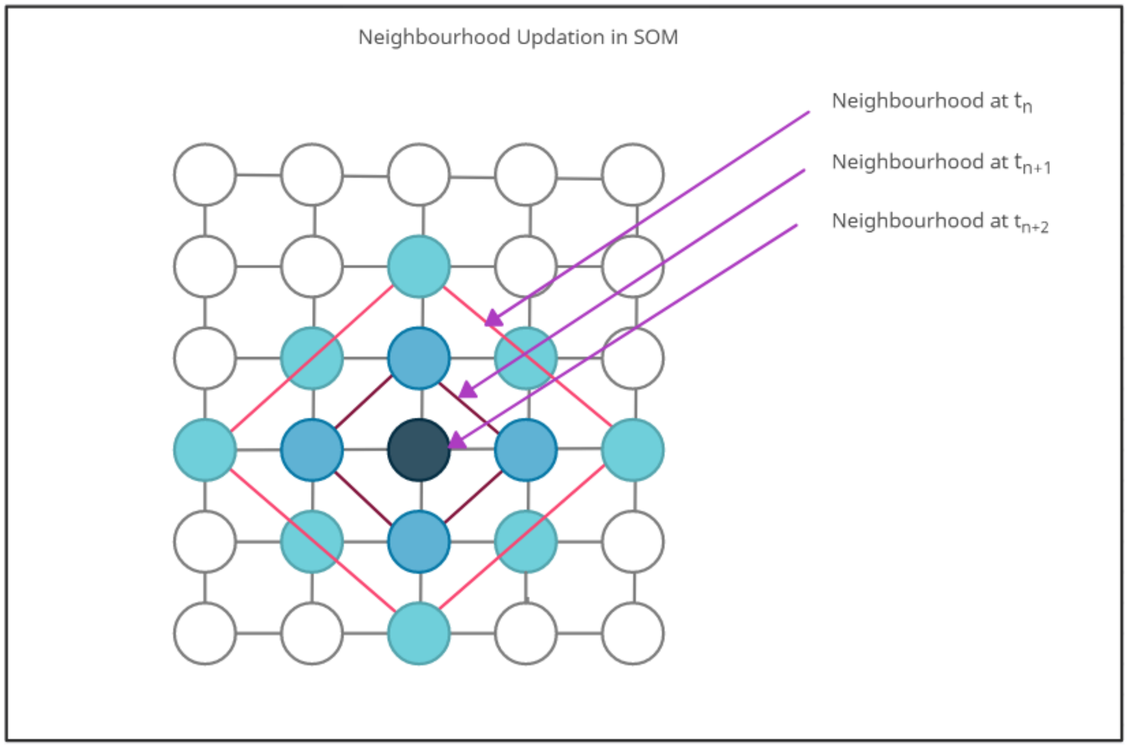

Neighbourhood Updation in SOM
Neighbourhood at t_n
Neighbourhood at t_{n+1}
Neighbourhood at t_{n+2}

Figure 2: Updating neighbourhood after finding BMU

Chapter 2: Algorithm

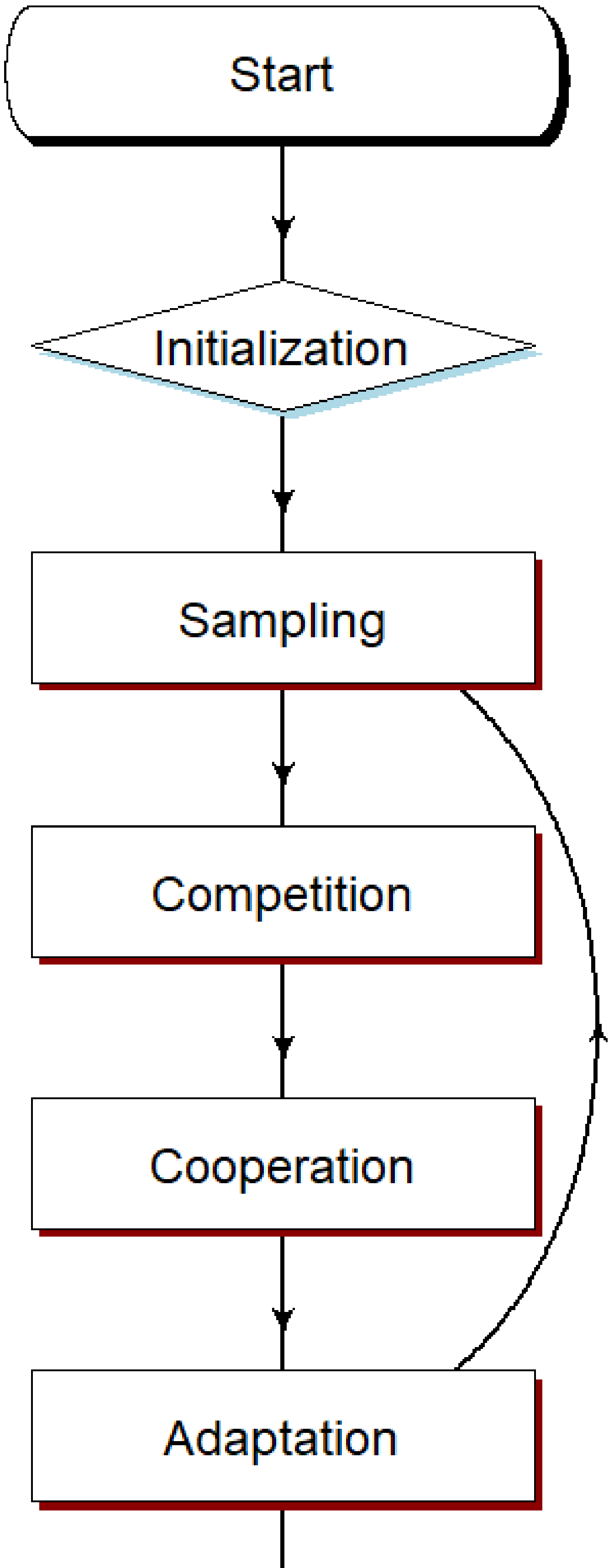

Start
Initialization
Sampling
Competition
Cooperation
Adaptation

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,] -0.12230677 -1.5922549 -0.95932543  1.8690040  0.39298454
##  [2,] -0.04083404  1.2130363 -0.95681839 -0.8318644 -0.88234092
##  [3,] -0.60736008 -2.1627850 -0.05226844  0.9583807  1.47477935
##  [4,] -1.31057389 -0.7345847  1.40792374  1.3257926  0.04208048
##  [5,] -0.42732910 -0.9057296  0.15250118 -1.0834193  2.03589522
##  [6,] -1.22880021  0.7948513 -0.50912704  0.1603020 -1.98191581
##  [7,]  0.35595697 -0.6377389 -0.40946713 -0.1856921  0.05731405
##  [8,]  0.27430192  0.1955348  0.50759722 -0.5837387 -1.94096162
##  [9,] -1.46454268 -0.8324308  0.64892888  0.2075777  2.30932579
## [10,]  0.24564618  0.4572430  0.25412534  0.9840664  0.59356109
```

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] 2.309326
```

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 x and y are the two points in n dimensional space and x_i and y_i 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.

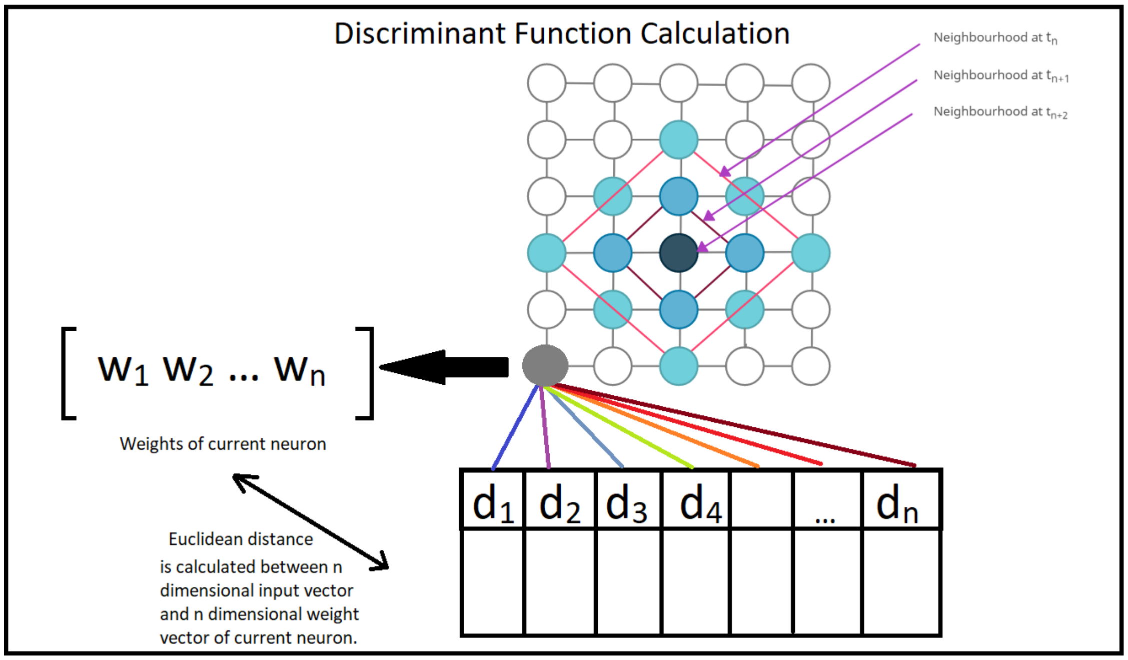

Discriminant Function Calculation
Neighbourhood at t_n
Neighbourhood at t_{n+1}
Neighbourhood at t_{n+2}
$[w_1\ w_2 \ldots w_n]$
Weights of current neuron
Euclidean distance is calculated between n dimensional input vector and n dimensional weight vector of current neuron.
d_1 d_2 d_3 d_4 … d_n

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

Neighbourhood influence calculation formula

```r
# 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 *radius* is the initial radius of neighbourhood, *current_iteration* is the iteration of data sampling that we are currently on and *time_constant* is the time constant which is incremented at each iteration, when the SOM gets updated.

Radius decay formula

<>

```r
#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 *learning_rate* is the old learning rate to be updated, *current_iteration* is the iteration of data sampling that we are currently on and *n_iteration* is the total number of iterations the SOM is trained over.

Learning rate decay formula

```r
#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.

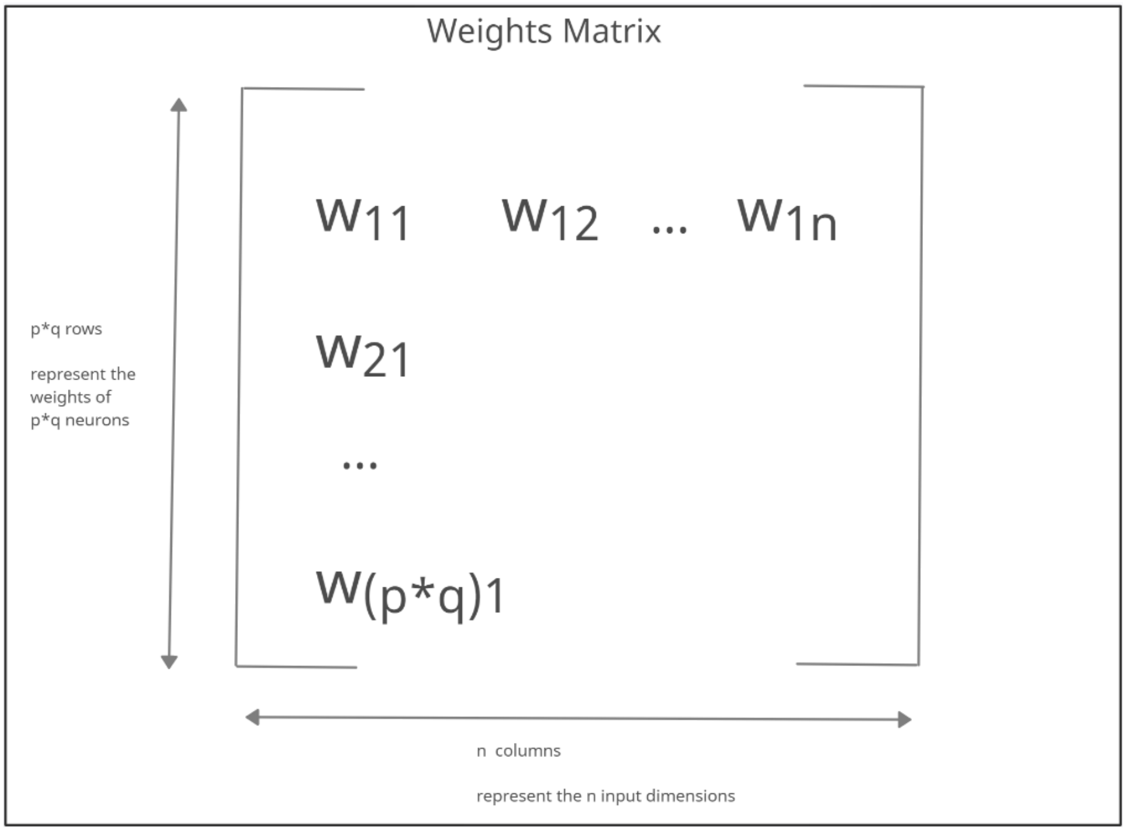

Weights Matrix
W11
W12
...
W1n
W21
...
W(p*q)1
p*q rows
represent the weights of p*q neurons
n columns
represent the n input dimensions

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(9,3)
grid

##               [,1]         [,2]       [,3]
##  [1,]  1.487757090 -1.201023506  0.5194904
##  [2,] -0.001891901  1.052458495 -0.7462955
##  [3,]  1.381020790 -1.305063566  0.7264546
##  [4,] -0.380213631 -0.692607634  0.7136567
##  [5,]  0.184136230  0.602648854 -0.6500629
##  [6,] -0.246895883 -0.197753074  1.4986962
##  [7,] -1.215560910 -1.185874517 -1.4358281
##  [8,]  1.561405098 -2.005512989 -2.1613182
##  [9,]  0.427310197  0.007509885  0.3952199
```

### 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.

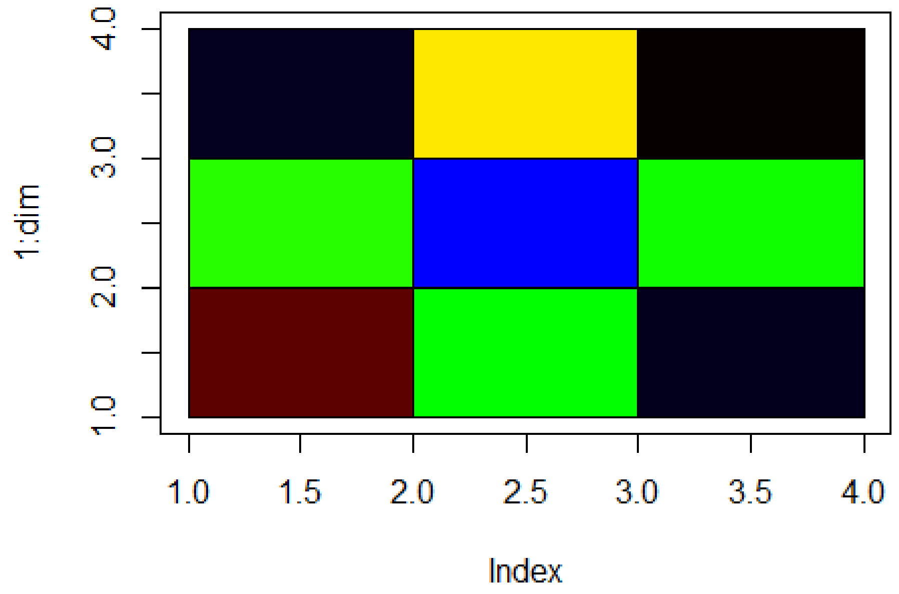
The SOM follows the algorithm mentioned above to fit the training data till the map stops changing or in other words till the model converges.

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(inpu
t_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
```

```r
    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,]  0.83783459  0.7460597 -0.120488929
##  [2,]  0.68457691  0.3751961 -0.062564377
##  [3,]  0.08935171 -0.1923388 -0.204433588
##  [4,]  0.63571536  0.3166299 -0.086911459
##  [5,]  0.30964489 -0.2942021  0.196228014
##  [6,] -0.36909612 -0.7618715  0.048797707
##  [7,]  0.09789500 -0.1890788 -0.202367096
##  [8,] -0.32060957 -0.7080236  0.003513688
##  [9,] -0.77499625 -0.9723701  0.021289937

time_taken <- end - start
print(time_taken)

## Time difference of 6.707737 secs

library(photobiology)
drawGrid<- function(weight,dimension){

  # Converting to a matrix
  weight<-as.matrix(weight, ncol = ncol(weight))

  norm.matrix<-NULL

  # Calculation of the norm
  for(i in 1:length(weight[,1])){
    a<-norm(weight[i,], type = "2")
    norm.matrix<-rbind(norm.matrix,a)
  }

  ## Mapping to range 400 to 700
  input_start<-min(norm.matrix)
  input_end<-max(norm.matrix)
  output_start<-400
  output_end<-700


  ## Calculating wavelength based on norm
  color<-NULL
  for(i in 1:length(norm.matrix)){
    input = norm.matrix[i]
    output = output_start + ((output_end - output_start) / (input_end -
input_start)) * (input - input_start)
    color<-rbind(color,output)
  }

  # Getting the colors (hex values) from the wavelength
  color.rgb<-w_length2rgb(color)


  # Plotting the grid
  dim<-max(dimension)+1
  plot(1:dim, type = "n")

  for (i in 1:dimension[1]) {
    for(j in 1:dimension[2]){
      #draw.circle(i*2,j*6, radius =.5, col = color.rgb[i*dimension[1]+j -
dimension[1]])
      rect(i,j,i+1,j+1, col = color.rgb[i*dimension[1]+j - dimension[1]])
    }
  }
}
gridSOM=matrix(unlist(gridSOM),ncol=3)
drawGrid(gridSOM,c(3,3))
```



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(inpu
t_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.
```

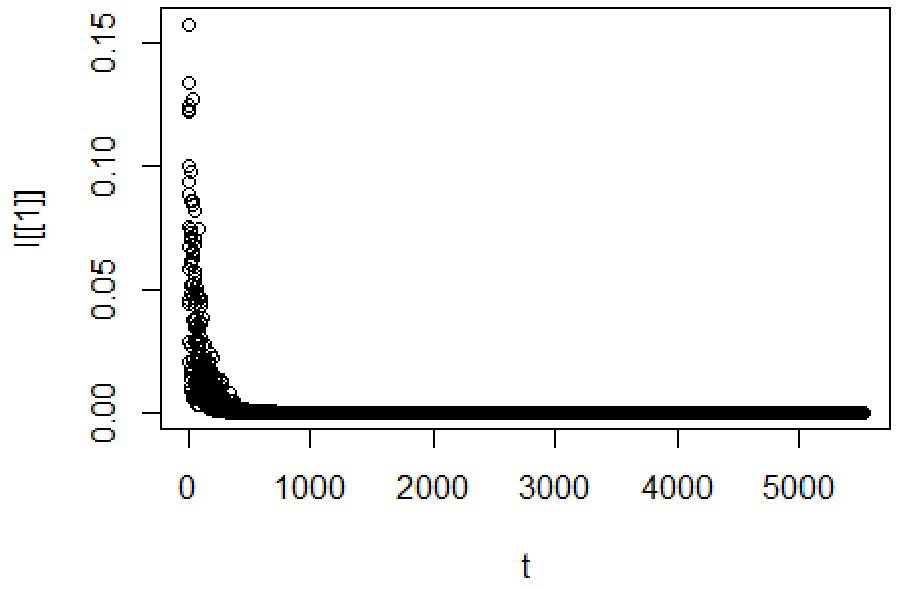
```
      rn=which(lateral_distance<=new_radius) #Finding neurons that are within
the radius of the winning unit.
      inf=influence_calculation(lateral_distance[rn],new_radius) #Calculating
the influence of the winning neuron on neighbours.
      diff_grid=(sweep(input_grid[rn,],2,sample_input_row))*-1 #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,]  0.93563378  0.5028079 -0.432887318
##  [2,]  0.61205256  0.4273599 -0.249594103
##  [3,] -0.09235935  0.1561476 -0.003106676
##  [4,]  0.59269828  0.4116258 -0.247425108
##  [5,]  0.07422356  0.1578972  0.142911783
##  [6,] -0.49782169 -0.3827718  0.367834940
##  [7,] -0.03019991  0.1661333 -0.082931683
##  [8,] -0.49455171 -0.3881704  0.363960276
##  [9,] -0.70956158 -0.7723888  0.488078157

time_taken <- end - start
print(time_taken)
```

```
## Time difference of 7.623296 secs
```

Chapter 6: Running the model over multiple epochs

```
SOM <- function(x, input_grid) {
  breaker <- 0
  n_iteration <- nrow(x) # 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(inpu
t_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=40 #Defining the number of epochs.
  new_radius <- initial_radius
  l <- c()
  for(ne in 1:n_epochs)
  {
    extra <- ((ne-1)*400)
    for (i in 1:n_iteration) # Looping through for training
    {
      old_grid=input_grid
      curr_i <- extra + i
      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, curr_i,
time_constant) # Decaying radius
      new_learning_rate <- decay_learning_rate(initial_learning_rate,curr_i,
n_iteration) # 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(abs(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.
      if(length(rn)!=1) #Updating multiple rows if neighbourhood is large
      {
        #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.
      }
      else #Updating only winning neuron.
      {
        diff_row=(input_grid[rn,]-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_row #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.
      }
      l <- c(l,euclidean_distance(old_grid,input_grid))
      if(isTRUE(all.equal(old_grid,input_grid)))
      {
        print(curr_i)
        print("Converged")
        breaker <- 1
        break
      }
    }
    if(breaker ==1)
    {
      break
    }
  }
  return(list(input_grid,l)) #Returning the updated SOM weights.
}
y <- SOM(data,grid)

## [1] 5520
## [1] "Converged"

gridSOM <- y[1]
gridSOM

## [[1]]
##             [,1]       [,2]       [,3]
##  [1,]  1.00508462  1.0722503  0.1352241
##  [2,]  0.46464851  0.7124700 -0.6238244
##  [3,]  0.09451207 -0.3290118 -0.6349856
##  [4,]  0.46464852  0.7124700 -0.6238244
##  [5,] -0.20563534  0.3614995  0.0752349
##  [6,] -0.61770607 -0.6731773 -0.1049023
##  [7,]  0.09451208 -0.3290117 -0.6349856
##  [8,] -0.61770605 -0.6731773 -0.1049022
##  [9,] -0.77777217 -0.7624095  1.0044831
```

```
l <- y[2]

t=1:length(l[[1]])
plot(t,l[[1]])
```



The above plot shows the decay in learning rate in a visual format. The points on the graph represent the difference between the weights of the neural network in consecutive iterations of training. Thus the downwards curve represents a decay in rapidness with which the SOM updates its weights, or in visual terms adjusts itself to map the input space. As the training of the SOM progresses, the neighbourhood radii decrease and the map fixates on finer details, but has learnt a majority of the representation and does not move rapidly. Initially the map has a steep curve which indicates that it is initially learning rapidly.

Thus we have successfully implemented Self Organizing Maps in R and have visualized the training progress, results on Admissions data and also have optimized the slower R implementation.

Chapter 7: 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."