Module 2

Solved Example of Self Organizing Maps: Iris Dataset

In this module we will apply our knowledge from module 1 and implement SOM for dimensionality reduction for the Iris dataset.(“Iris Dataset,” n.d.)

Table of Contents

Part 1: Loading and installing necessary libraries.

```
library(dplyr)
library(photobiology)
library(ggplot2)
```

Part 2: Creation of functions.

```
#-------------------------------------------------------------------------
-------------------------------------------------------------------------
--------

# 1) A function to return the sum of squared distance between x and y.

euclidean_distance <- function(x, y) {
  ret <- sqrt(sum((x - y)^2))
  return(ret)
}

#-------------------------------------------------------------------------
-------------------------------------------------------------------------
--------

# 2) Function to create a SOM grid.
# n is the number of neurons.
# p is the number of columns in the original dataframe.
```

```r
# data table is used for faster computation.

create_grid <- function(x,y,p) {
  ret <- matrix(data = rnorm(x * y * p), nrow = x * y, ncol = p)
  return(ret)
}


#-------------------------------------------------------------------------
-------------------------------------------------------------------------
--------

# 3) Function to decay the radius exponentially over time.
# rds is the initial radius that is passed.
# cur_iter represents the current iteration.
# time_constant is the time constant that is calculated before the

decay_radius_function <- function(radius, current_iteration, time_constant) {
  ret <- radius * exp(-current_iteration / time_constant)
  return(ret)
}

#-------------------------------------------------------------------------
-------------------------------------------------------------------------
--------

# 4) Function to decay the learning rate.
# lr is the current learning rate.
# cur_iter is the current iteration
# n_iteration is the number of iterations.

decay_learning_rate <- function(learning_rate, current_iteration,
n_iteration) {
  ret <- learning_rate * exp(-current_iteration / n_iteration)
  return(ret)
}

#-------------------------------------------------------------------------
-------------------------------------------------------------------------
--------

# 5) A function to calculate influence over neighboring neurons
#dstnc is the lateral distance.
#rds is the current neighbourhood radius.

influence_calculation <- function(distance, radius) {
  ret <- exp(-(distance^2) / (2 * (radius^2)))
  return(ret)
}
```

```r
#-------------------------------------------------------------------------
-------------------------------------------------------------------------
--------

# 6) A function to return the winning neuron.
# x is a single 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
}

#-------------------------------------------------------------------------
-------------------------------------------------------------------------
--------

#7) Fastest BMU Implementation using vectorisation. You can opt in for this
function over the regular BMU function for faster execution.


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.
}


#-------------------------------------------------------------------------
-------------------------------------------------------------------------
--------

#8) A function to encapulate the entire creation, working and updating of SOM
over the training period.
#x is the input and input_grid is the SOM grid that will be updated
```

```
iteratively.

SOM <- function(x, input_grid) {
  breaker <- 0
  n_iteration <- nrow(x) # Defining number of iterations
  initial_learning_rate <- 0.5 # Defining initial learning rate
  initial_radius <- 15 # 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)))
      {
        breaker <- 1
        break
      }
    }
    if(breaker ==1)
    {
      break
    }
  }
  return(list(input_grid,l)) #Returning the updated SOM weights.
}
```

Part 3: Loading the data files for implementation.

```
# 1) Reading the data and scaling it.
data<-scale(iris[,-5])
X <- scale(data[, ])
data <- X

# 2) Setting the seed for consistent results.
set.seed(222)

# 3) Creating the grid of neurons.
grid <- create_grid(5,5,4)
```

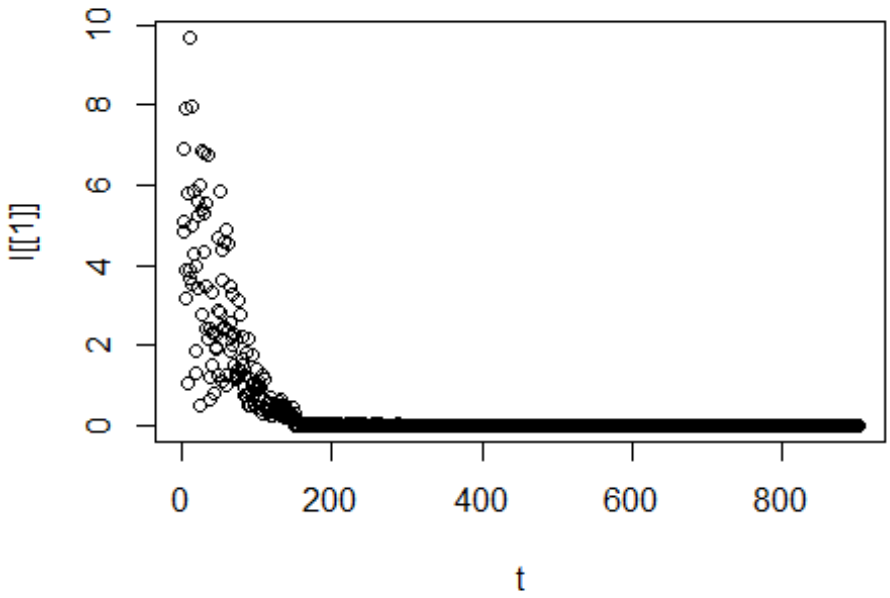
Part 4: Generating the Self Organizing Map.

```
# 1) Generating the SOM.
y <- SOM(data,grid)
```

```r
# 2) Transforming the SOM weights to usable form.
gridSOM <- y[1]
gridSOM
```

```
## [[1]]
##              [,1]        [,2]       [,3]        [,4]
##  [1,]  1.51979580  0.03237630  1.2644454  1.14230472
##  [2,]  1.24099155 -0.02410680  1.0516778  0.98407574
##  [3,]  0.73293736 -0.29868812  0.7240412  0.60851629
##  [4,]  0.15410923 -0.77490850  0.4535389  0.34719100
##  [5,]  0.01098373 -0.59608748  0.4244492  0.37848457
##  [6,]  1.20915449 -0.09180618  1.0353214  1.03089388
##  [7,]  1.02275478 -0.05746048  0.8055906  0.72338924
##  [8,]  0.63555127 -0.35351906  0.4973977  0.33937501
##  [9,]  0.22881853 -0.72438396  0.2992720  0.11486734
## [10,] -0.19835820 -0.62107698  0.1771841  0.09836969
## [11,]  0.68795681 -0.41771178  0.7460912  0.66425283
## [12,]  0.66586054 -0.34578508  0.5388258  0.37420532
## [13,]  0.26071271 -0.24531347  0.1953997  0.07510772
## [14,] -0.77764817 -0.06289039 -0.7263837 -0.73873748
## [15,] -0.62067355  0.19287913 -0.5233841 -0.56000999
## [16,]  0.20381249 -0.78914720  0.4569668  0.30244892
## [17,]  0.19980266 -0.75748213  0.3117550  0.10648900
## [18,] -0.71927940 -0.11619778 -0.7058482 -0.68951213
## [19,] -1.15931006  0.45744592 -1.2669385 -1.21821706
## [20,] -0.99092608  1.00557473 -1.2819808 -1.20593553
## [21,]  0.09036951 -0.67733186  0.4509914  0.32609800
## [22,] -0.19596576 -0.61017007  0.1541737  0.05987712
## [23,] -0.71114901  0.33176281 -0.7535654 -0.72404885
## [24,] -0.98814935  1.01000002 -1.2860587 -1.21374472
## [25,] -0.74348052  1.75151250 -1.2995359 -1.23497618
```

```r
l <- y[2]
t=1:length(l[[1]])
plot(t,l[[1]])
```



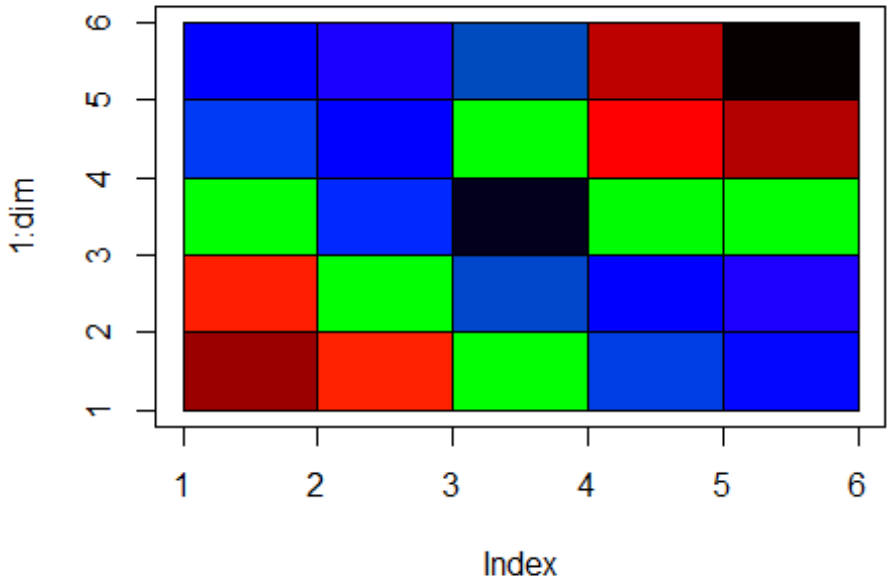

```
l=unlist(l)
l[which.min(unlist(l))]

## [1] 4.591932e-08
```

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.

Part 5: Visualizing the results.

```
# 1) Creating a function to visualize the SOM results by mapping the numbers
to colours.
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]])
    }
  }
}

# 2) Plotting  the grid of weights.
gridSOM=matrix(unlist(gridSOM),ncol=4)
drawGrid(gridSOM,c(5,5))
```



Using the grid we can see that there are predominantly 3 colours or clusters [green, blue and red] produced by the SOM, which correspond to the 3 original clusters Iris Setosa, Iris Versicolor and Iris Virginica of the Iris dataset.

Thus we have successfully implemented Self Organizing Maps in R for dimensionality reduction for the Iris dataset.

Part 6: References

"Iris Dataset." n.d. https://archive.ics.uci.edu/ml/datasets/iris.