CS 229 - Machine Learning
Xiangliang Zhang
Computer Science (CS)/Statistics (STAT) Program
Computer, Electrical and Mathematical Sciences & Engineering (CEMSE) Division
King Abdullah University of Science and Technology (KAUST)

HOMEWORK VIII

Henrique Aparecido Laureano

Spring Semester 2018

Contents

Data : All questions will use the data in: `clu_data.txt`.

```
# <r code> ============================================================================ #
path <- "~/Dropbox/KAUST/machine_learning/hw8/"                          # files path
                                                  # df: dataframe. reading the dataset
df <- read.table(paste0(path, "clu_data.txt"))
x <- df$V1 ; y <- df$V2                                  # creating data vectors x and y
par(mar = c(4, 4, 0, 0) + .1) ; plot(x, y)  # graph. definitions and data plotting
# </r code> ============================================================================ #
```

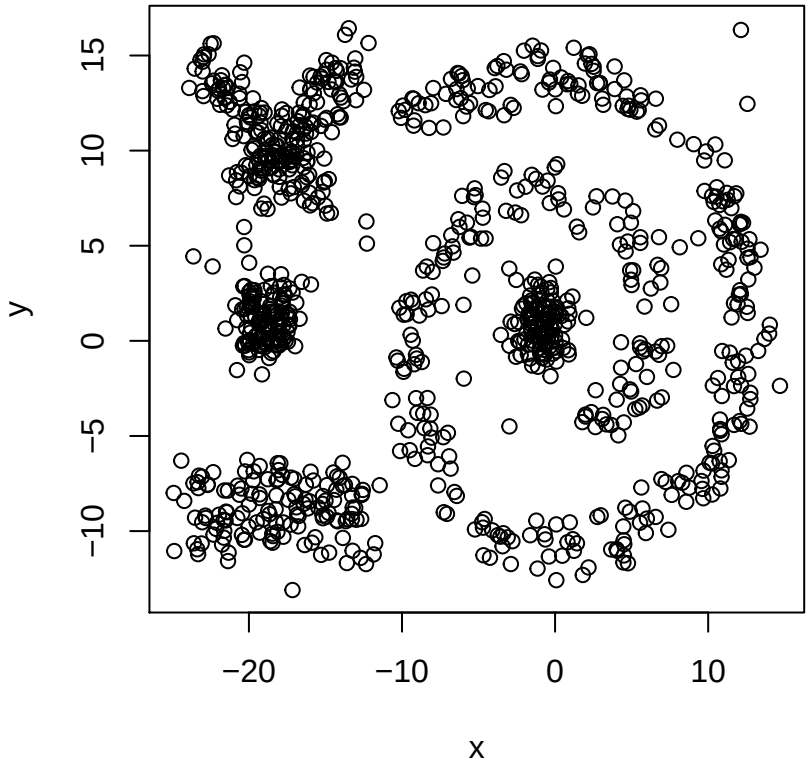

Figure 1: `clu_data` scatter plot.

[20 points] Question 1: Programming k-means

Write your own code of k-*means* algorithm.

Here I'm computing the distance by

$$\sqrt{(x_i - C_k^x)^2 + (y_i - C_k^y)^2},$$

where C_k represents the centroid of the cluster k and i represents the data points.

The sum of squared error (SSE) is also computed, it is given by the following expression

$$\text{SSE} = \sum_{k=1}^{K} \sum_{x_i \in \text{Cluster}_k} (x_i - C_k)^2,$$

where x_i represents the data points, Cluster_k represents the cluster k, and C_k the respective cluster centroid.

```r
# <r code> ============================================================================ #
kmeans2.0 <- function(x, y, nclus, random.seed = 1) {
  set.seed(random.seed)                 # the seed make all the values reproducible
                                                    # defining random centroids
  cen.x = runif(n = nclus, min = min(x), max = max(x))
  cen.y = runif(n = nclus, min = min(y), max = max(y))
                                                                  # clusters centroids
  clus = data.frame(cluster = 1:nclus, cen.x = cen.x, cen.y = cen.y)
  df = data.frame(xs = x, ys = y, cluster = NA)  # data & cluster assignment in df
                  # vector to keep the sum of squared error (sse) for each cluster
  sse.clus = numeric(length(nclus))
  iter = 1                                                        # iterations counter
  sse = numeric(iter)                     # vector to keep the sum of squared error
  done = FALSE                                                    # stopping criterion
  while(done == FALSE) {                    # while loop (where the beauty happens)
                           # computing the centroid distances for each data point
    for(i in 1:length(x)) { #    and assignment to the cluster of minimum distance
      distance = sqrt( (x[i] - clus$cen.x)**2 + (y[i] - clus$cen.y)**2 )
      df$cluster[i] = which.min(distance)
    }
    cen.xold = clus$cen.x ; cen.yold = clus$cen.y
    for(i in 1:nclus) {                  # updating centroids and computing the sse
      xs.clus = subset(df$xs, df$cluster == i)
      ys.clus = subset(df$ys, df$cluster == i)
      clus[i, 2] = mean(xs.clus)                                          # cen.x
      clus[i, 3] = mean(ys.clus)                                          # cen.y
      sse.clus[i] = sum( (xs.clus - clus[i, 2])**2 + (ys.clus - clus[i, 3])**2 )
    }
    sse[iter] = sum(sse.clus)                 # computing the sse of the iteration
                    # checking stopping criterion:
                    # stop the loop *if* there is no change in cluster assignment
    if(identical(cen.xold, clus$cen.x) & identical(cen.yold, clus$cen.y)) {
      done = TRUE
    } else iter = iter + 1                  # else, increase the iteration counter
  }
  return(list(data = df, sse = sse))                # returning data frame and sse
}
# </r code> ============================================================================ #
```

1)

Try different settings of parameter k. For *each value of k*, compute the *SSE (sum of squared error)* of clustering result. Plot the *SSE curve* w.r.t. the various k values.

```
# <r code> ============================================================================ #
par(mar = c(4, 4, 3, 1) + .1, mfrow = c(3, 3))            # graphical definitions
for (i in 4:10)                  # trying different k's and plotting the SSE curves
  plot(kmeans2.0(x, y, nclus = i)$sse, type = "b"
       , xlab = "Iterations", ylab = "SSE", main = paste(i, "clusters"))
# </r code> ============================================================================ #
```

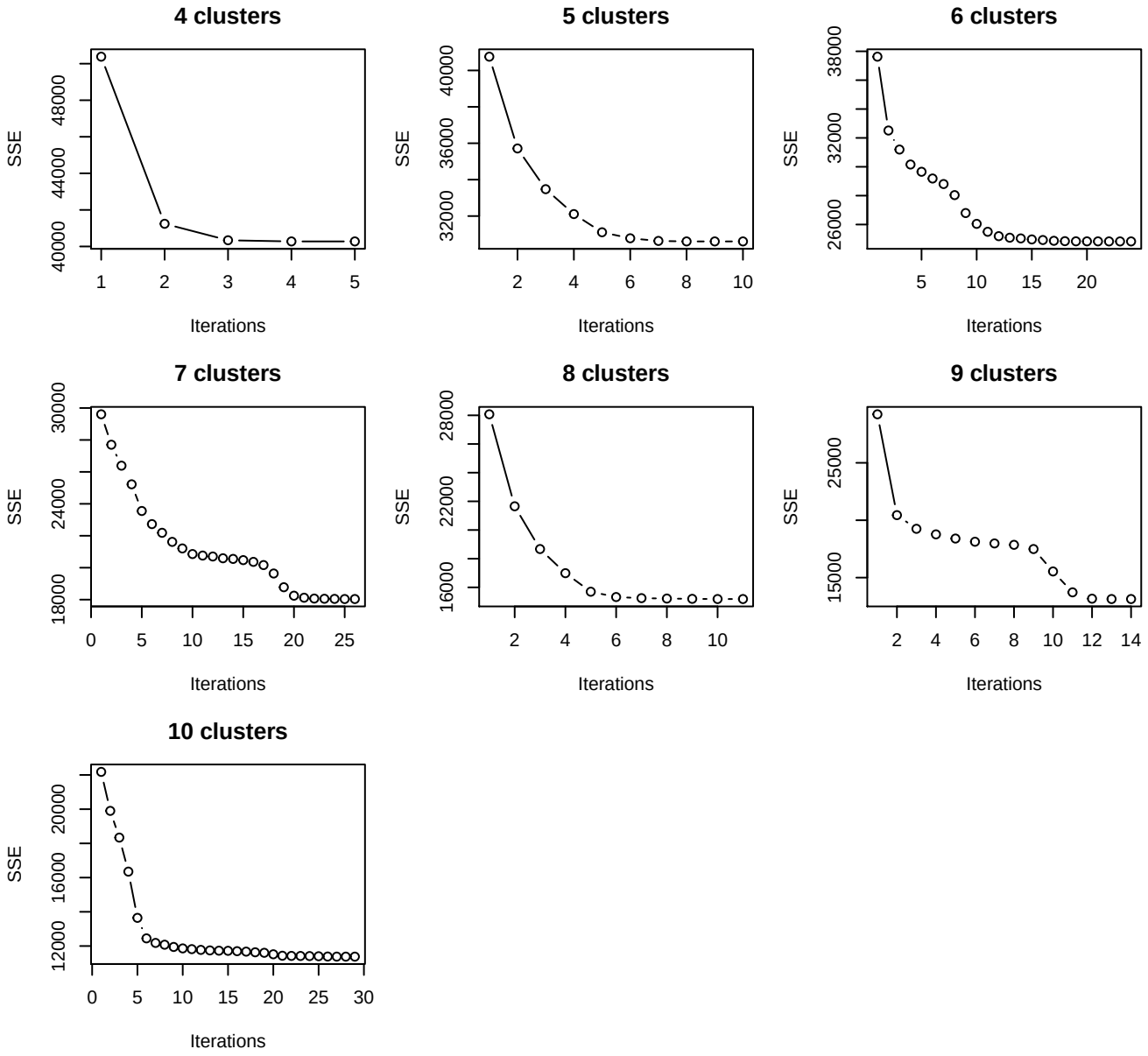

Figure 2: Sum of squared error (SSE) curve w.r.t. different k's (number of different clusters).

□

2)

Does the SSE curve suggest the best clustering results?

No. As k increases the SSE decrease. Therefore, choosing the smallest SSE doesn't mean that it correspond, exactly, to the best clustering results. The algorithm starts from random centroids, changing the initial centroids the results can change.

□

3)

Plot the best clustering result you think (using different colors to show the different clusters), and answer:

```
# <r code> ============================================================================ #
par(mar = c(2, 2, 3, 1) + .1, mfrow = c(1, 3))              # graphical definitions

for (i in c(4, 8, 10))                                  # plotting some clustering results
  plot(x, y, col = kmeans2.0(x, y, nclus = i)$data$cluster, xlab = NA, ylab = NA
       , main = paste(i, "clusters"))
# </r code> ============================================================================ #
```

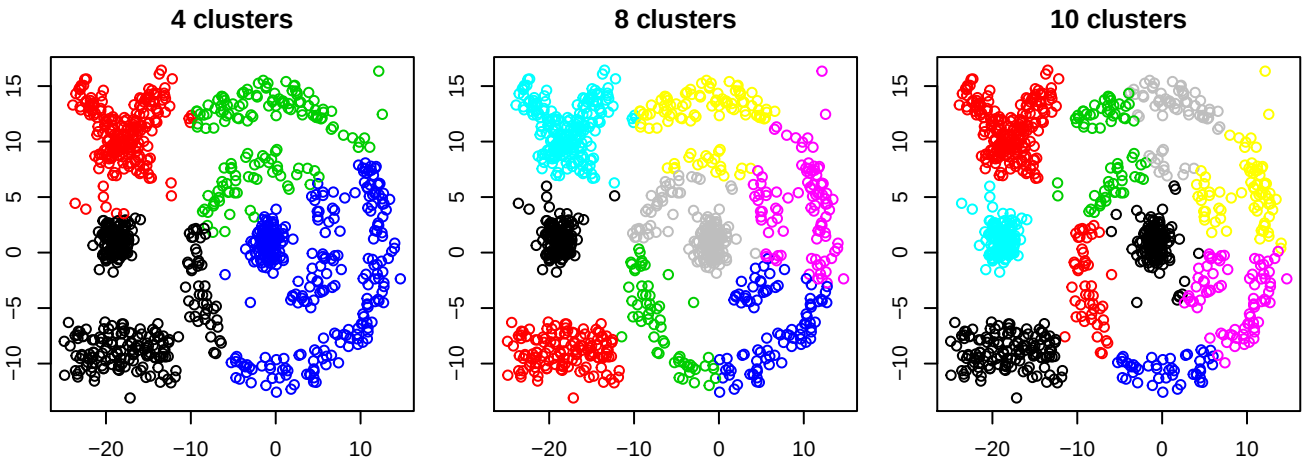

Figure 3: Clustering results for some number of clusters (in the graphic corresponding to 10 clusters the color black appears two times, but representing different clusters).

a)

How does k-means (with your setting of k) perform on clustering the data?

Looking to the data scatter plot (Figure 1) we could expect 4 clusters, 3 for the groups of points in the left and one big one for the rest of the points. However we don't get this result. The algorithm only create the 3 clusters for the left groups when we allow 6 clusters or more for the data.

With 8 and 10 clusters (Figure 3) the divisions are good, but in none of the cases the algorithm captures perfectly the non-convex shape of the data, instead, it breaks the data in small groups. This small, and closer, clusters can be putted together in the end and make a unique cluster.

□

b)

And why?

The k-means algorithm ins't suitable to discover clusters with different sizes, different density, and non-convex shapes. The data studied here present all this characteristics.

□

[40 points] Question 2: Programming hierarchical clustering algorithm

---

Write your own code of *agglomerative hierarchical clustering*.

The main `R` function responsable for hierarchical clustering is the `hclust`. To be able to build the dendrograms without some crazy coding I put a class `hclust` in my main function `cluster`, in this way with a `plot` I can generate the dendrograms. For this I also compute some things take the plot function need for the `hclust` class.

```
# <r code> ============================================================================ #
distance <- function(x) {                  # building the euclidian distance matrix
  x = as.matrix(x)
  u = apply(x*x, 1, sum) %*% matrix(1.0, 1, nrow(x))
  sqrt( abs( u + t(u) - 2 * x %*% t(x) ) )
}
iorder <- function(m) {                    # ordering to avoid crossing connections
  N = nrow(m) + 1
  iorder = rep(0, N)
  iorder[1] = m[N - 1, 1]
  iorder[2] = m[N - 1, 2]
  loc = 2
  for (i in seq(N - 2, 1)) {
```

```
    for (j in seq(1, loc)) {
      if (iorder[j] == i) {
        iorder[j] = m[i, 1]
        if (j == loc) {
          loc = loc + 1
          iorder[loc] = m[i, 2]
        } else {
          loc = loc + 1
          for (k in seq(loc, j + 2)) iorder[k] = iorder[k - 1]
          iorder[j + 1] = m[i, 2]
        }
      }
    }
  }
  - iorder
}                                                                    # main function
cluster <- function(d, method = c("single", "complete", "average")) {
  if (!is.matrix(d)) d = as.matrix(d)
  method_fn = switch(match.arg(method)              # picking a clustering function
                     , single = min, complete = max, average = mean)
  N = nrow(d)
  diag(d) = Inf
  n = -(1:N)                                                # tracks group membership
  m = matrix(0, nrow = N - 1, ncol = 2)                        # hclust merge output
  h = rep(0, N - 1)                                           # hclust height output
  for (j in seq(1, N - 1)) {
    h[j] = min(d)             # finding smallest distance and corresponding indices
    i = which(d - h[j] == 0, arr.ind = TRUE)
    i = i[1, , drop = FALSE]
    p = n[i]
    p = p[order(p)]                                      # ordering each m[j, ] pair
    m[j, ] = p
                                  # agglomerating the pair and all previous groups
                                  #     they belong to into the current j-th group
    grp = c( i, which(n %in% n[i[1, n[i] > 0]]) )
    n[grp] = j
    r = apply(d[i, ], 2, method_fn)
                                # moving on to the next minimum distance, excluding
                                #      current one by modifying the distance matrix
    d[min(i), ] = d[ , min(i)] = r
    d[min(i), min(i)] = Inf
    d[max(i), ] = d[ , max(i)] = Inf
  }
                             # returning something similar to the output from hclust
  structure(list(merge = m, height = h, order = iorder(m)), class = "hclust")
}
# </r code> ============================================================== #
```

1)

Use *single linkage* (min, shortest distance). Plot the best clustering result you think. How do you think the performance of single linkage?

```
# <r code> ============================================================================ #
par(mar = c(0, 4, 0, 0) + .1)                                    # graphical definition
                                                                 # computing and plotting
plot(cluster(distance(df), method = "single"), main = NULL, labels = FALSE)
# </r code> ============================================================================ #
```

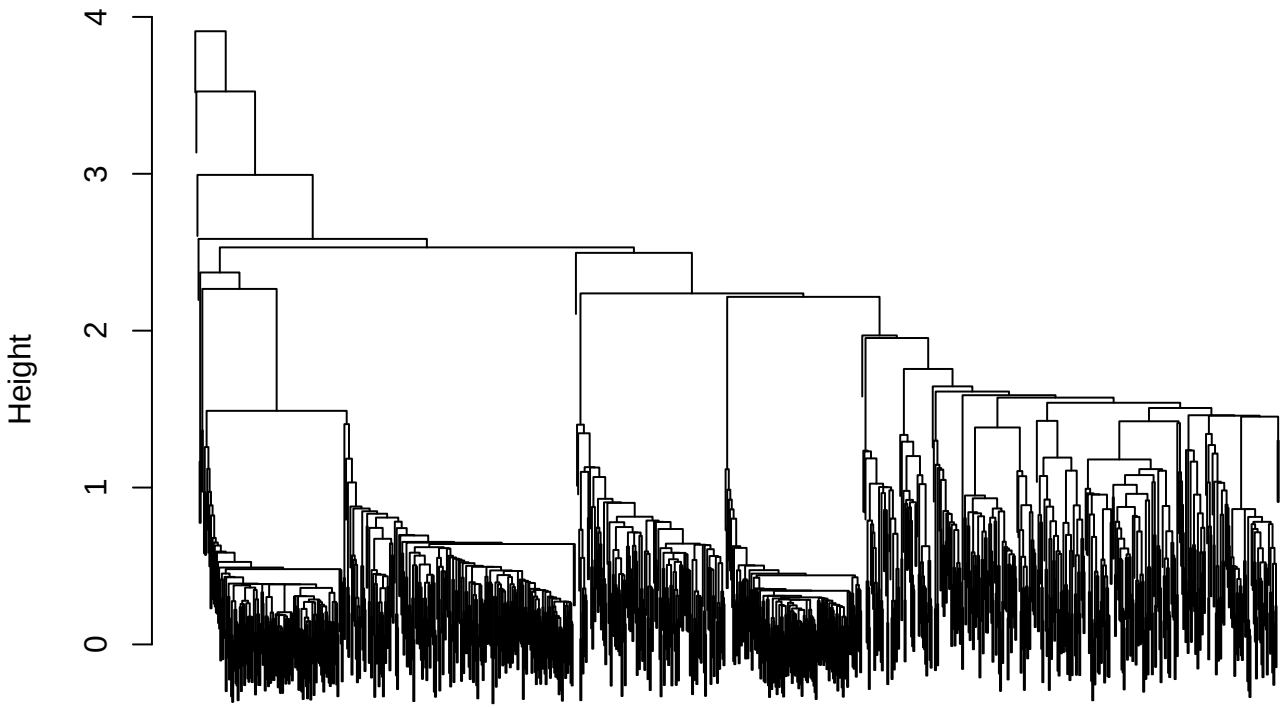

Figure 4: Cluster dendrogram using *single linkage*.

By the dendrogram we can see five clusters well-defined.

□

2)

Use *complete linkage* (max, furthest distance). Plot the best clustering result you think. How do you think the performance of complete linkage?

```r
# <r code> ============================================================================ #
par(mar = c(0, 4, 0, 0) + .1)                                          # graphical definition
                                                                       # computing and plotting
plot(cluster(distance(df), method = "complete"), main = NULL, labels = FALSE)
# </r code> ============================================================================ #
```

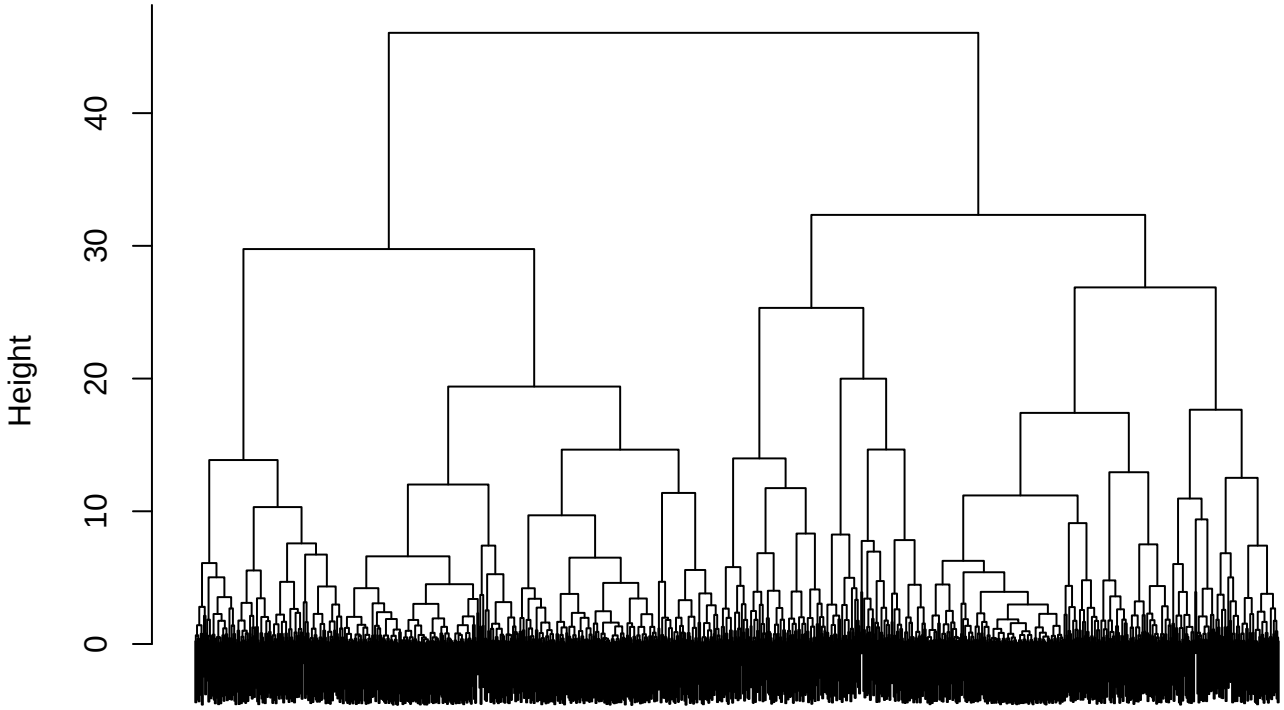

Figure 5: Cluster dendrogram using *complete linkage*.

Comparing with the *simple linkage* we see a very different behaviour. By the dendrogram we can see four clusters (with the *simple linkage* we see five). Here, if you want to be more specific each chunk can be broken in more clusters, here this divisions/cutting points are more clear.

□

3)

Use *average linkage* (average distance). Plot the best clustering result you think. How do you think the performance of average linkage?

```r
# <r code> ============================================================================ #
par(mar = c(0, 4, 0, 0) + .1)                                          # graphical definition
                                                                       # computing and plotting
plot(cluster(distance(df), method = "average"), main = NULL, labels = FALSE)
# </r code> ============================================================================ #
```

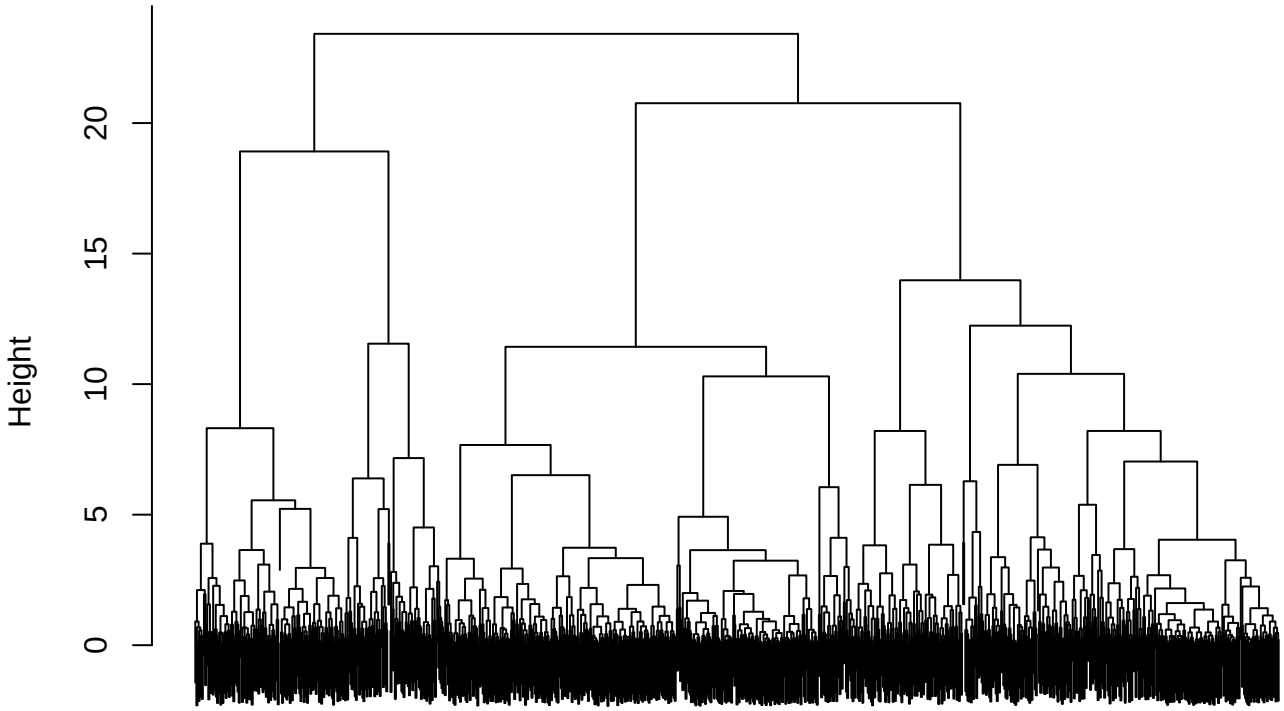

Figure 6: Cluster dendrogram using *average linkage*.

The behaviour here is more similar with the behaviour of the *complete linkage*. By the dendrogram we can see five clusters. If you want to be more specific this clusters can be broken in more clusters, the divisions are more evident with the last cluster (in the right).

□

[40 points] Question 3: Programming DbScan algorithm

Write your own code of *DbScan* algorithm. Try different settings of parameter *Minpts* and *Eps*. Plot the best clustering result you think (using different colors to show the different clusters), and answer:

```
# <r code> ============================================================================ #
dbscan2.0 <- function(data, eps, MinPts) {
  distcomb <- function(x, data) {
    data = t(data)
    temp = apply(x, 1, function(x) sqrt(colSums((data - x)**2)))
    return(t(temp))
  }
  data = as.matrix(data)
  n = nrow(data)
  classn = cv = integer(n)
```

```
  is.seed = logical(n)
  cn <- integer(1)
  for (i in 1:n) {
    unclass = (1:n)[cv < 1]
    if (cv[i] == 0) {
      reachables = unclass[
        as.vector(distcomb(data[i, , drop = FALSE],
                           data[unclass, , drop = FALSE])) <= eps]
      if (length(reachables) + classn[i] < MinPts) cv[i] = (-1)
      else {
        cn = cn + 1
        cv[i] = cn
        is.seed[i] = TRUE
        reachables = setdiff(reachables, i)
        unclass = setdiff(unclass, i)
        classn[reachables] = classn[reachables] + 1
        while (length(reachables)) {
          cv[reachables] = cn
          ap = reachables
          reachables = integer()
          for (i2 in seq(along = ap)) {
            j = ap[i2]
            jreachables = unclass[
              as.vector(distcomb(data[j, , drop = FALSE],
                                 data[unclass, , drop = FALSE])) <= eps]
            if (length(jreachables) + classn[j] >= MinPts) {
              is.seed[j] = TRUE
              cv[jreachables[cv[jreachables] < 0]] = cn
              reachables = union(reachables, jreachables[cv[jreachables] == 0])
            }
            classn[jreachables] = classn[jreachables] + 1
            unclass = setdiff(unclass, j)
          }
        }
      }
    }
    if (!length(unclass)) break
  }
  rm(classn)
  if (any(cv == (-1))) cv[cv == (-1)] <- 0
  out = list(cluster = cv, eps = eps, MinPts = MinPts)
  if (cn > 0) out$is.seed = is.seed
  return(out)
}
# </r code> ========================================================================== #
```

1)

How the clustering result is changing when you increase *Minpts*?

```
# <r code> ============================================================================ #
par(mar = c(2, 2, 3, 1) + .1, mfrow = c(3, 3))
for (i in 5:13) {
  clust = dbscan2.0(df, eps = 2, MinPts = i)$cluster
  plot(x, y, col = clust + 1, pch = ifelse(clust == 0, 4, ifelse(clust > 8, 2, 1))
       , xlab = NA, ylab = NA, main = paste("Minpts:", i))}
# </r code> ============================================================================ #
```

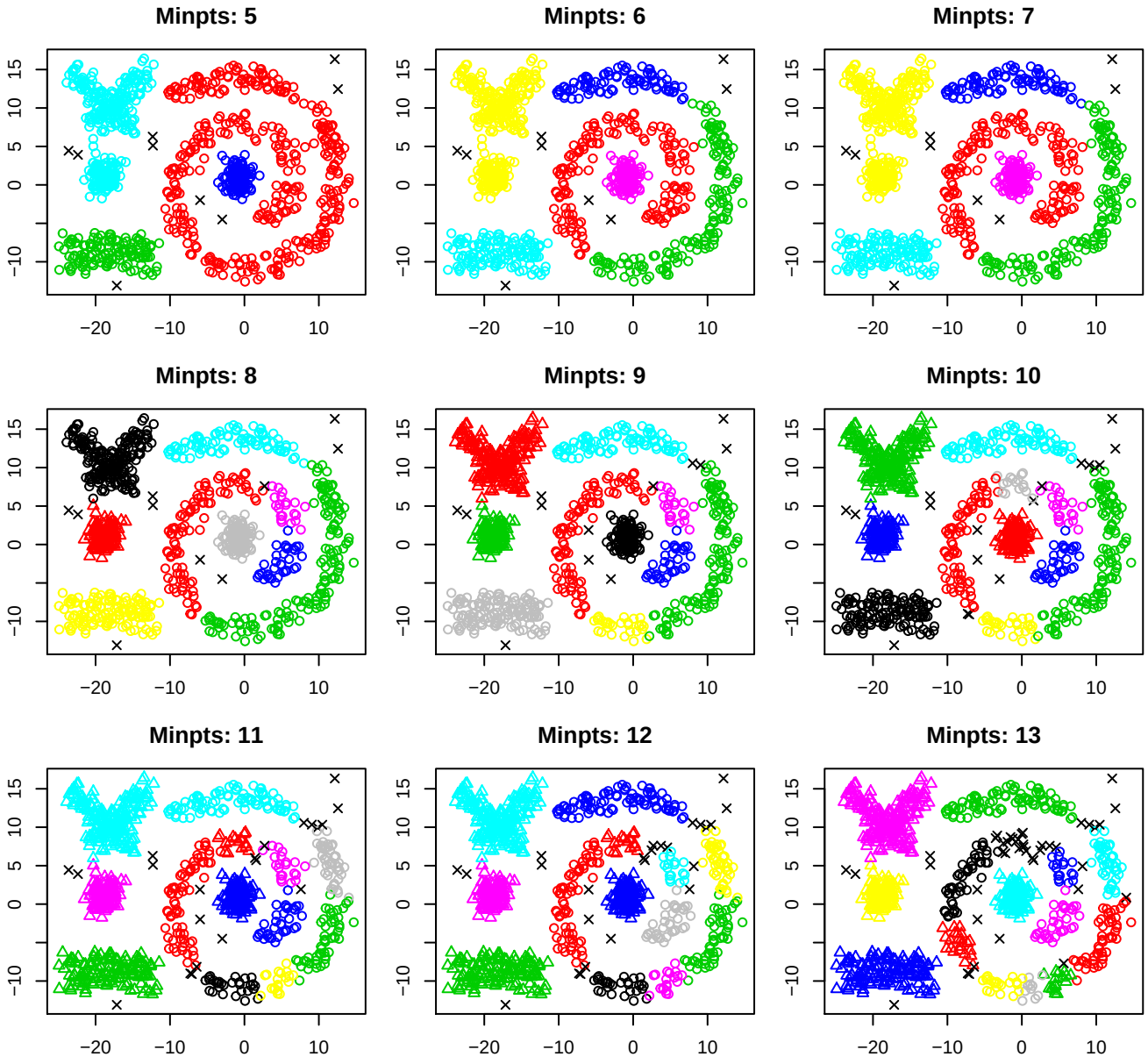

Figure 7: Clustering results for different *Minpts*. To differentiate some repetead colors we use different characters (circles and triangles). Outliers are represented by x's.

Here we used *Eps* equal 2.
With small values for *Minpts* we see that few clusters are identified. As *Minpts* increase the number of detected clusters also increase, reaching a point where the clusters begin to divide into several small clusters. The best clustering result in observed with *Minpts* equal 8. With less than this the three clusters of the left are not identified, and with more than 8 the clusters start to split in several small clusters.

□

2)

How the clustering result is changing when you increase *Eps*?

```
# <r code> ============================================================================ #
par(mar = c(2, 2, 3, 1) + .1, mfrow = c(2, 3))
for (i in c(.5, .75, 1, 1.25, 1.5, 1.75)) {
  clust = dbscan2.0(df, eps = i, MinPts = 8)$cluster
  plot(x, y, col = clust + 1, pch = ifelse(clust == 0, 4, ifelse(clust > 8, 2, 1))
       , xlab = NA, ylab = NA, main = paste("Eps:", i))}
# </r code> ============================================================================ #
```

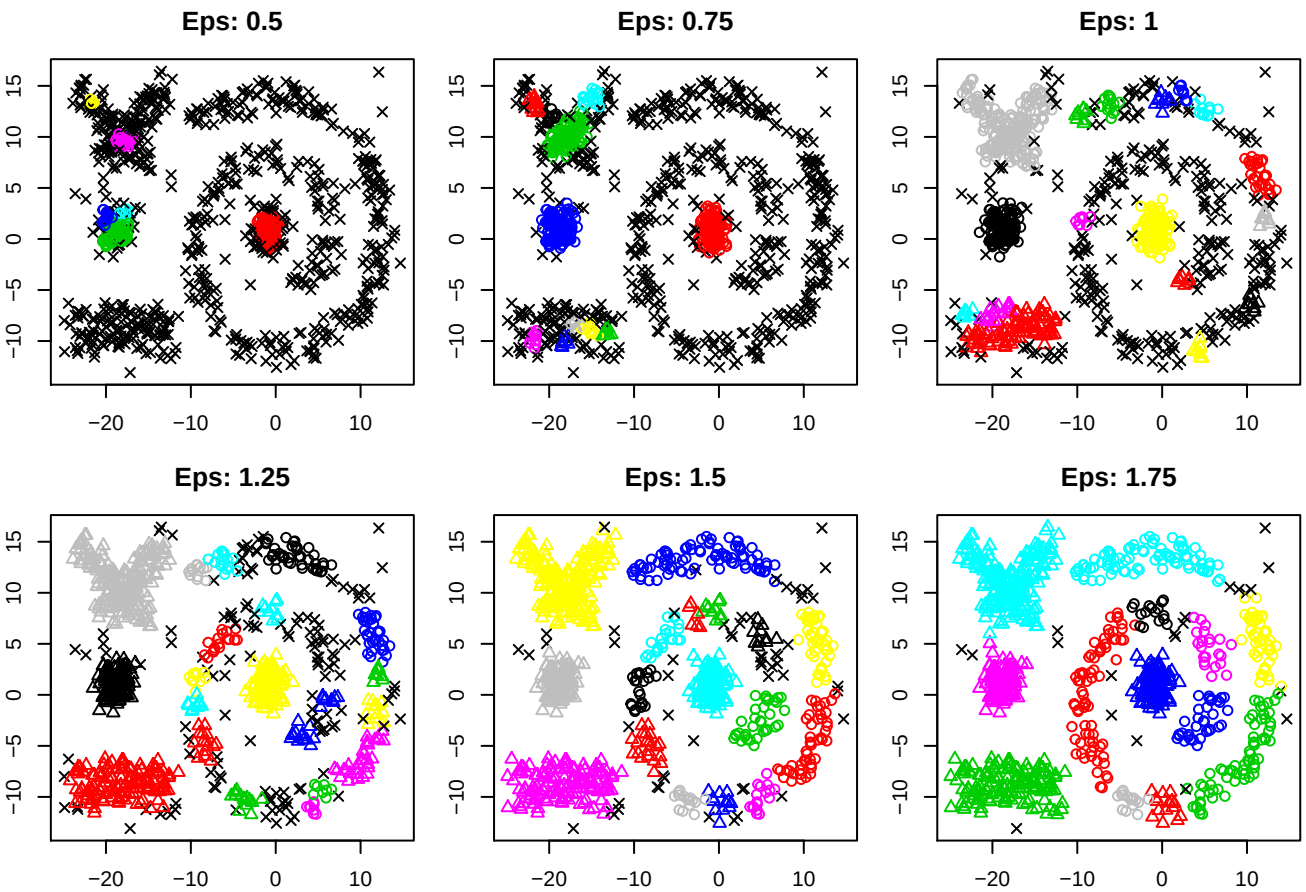

Figure 8: Clustering results for different *Eps*. To differentiate some repetead colors we use different characters (circles and triangles). Outliers are represented by x's.

```r
# <r code> ============================================================================ #
par(mar = c(2, 2, 3, 1) + .1, mfrow = c(3, 3))
for (i in c(1.8, 1.9, 2, 2.1, 2.15, 2.2, 2.5, 2.75, 3)) {
  clust = dbscan2.0(df, eps = i, MinPts = 8)$cluster
  plot(x, y, col = clust + 1, pch = ifelse(clust == 0, 4, ifelse(clust > 8, 2, 1))
       , xlab = NA, ylab = NA, main = paste("Eps:", i))}
# </r code> ============================================================================ #
```

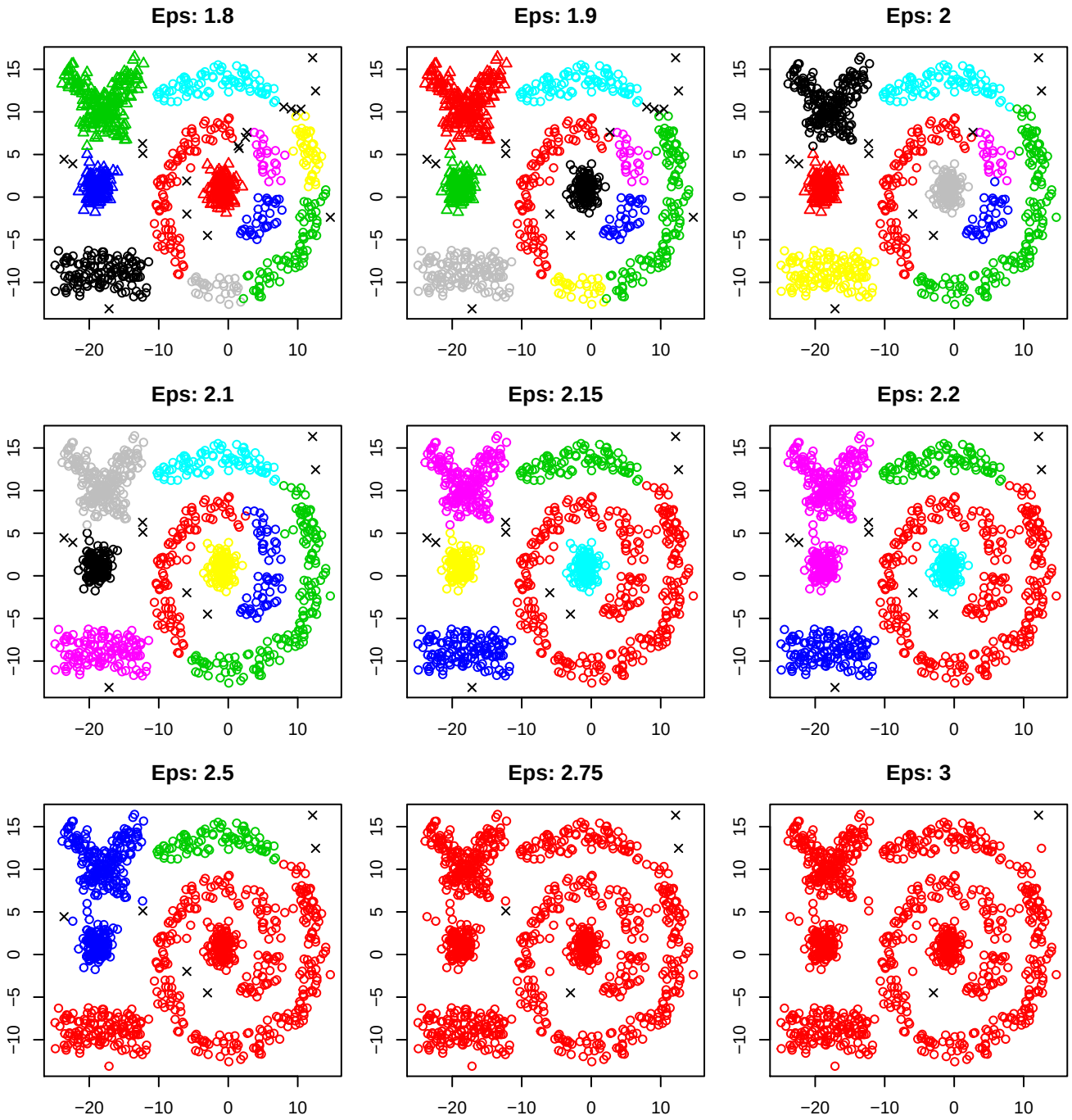

Figure 9: Clustering results for different *Eps*. To differentiate some repetead colors we use different characters (circles and triangles). Outliers are represented by x's.

Here we used *Minpts* equal 8.
With small *Eps*, 0.5 e.g., few points are detected and became clusters. As *Eps* increase reasonable clusters are formed. The best result is obtained with *Eps* equal 2.1. With 2.15, a very small difference, the difference in the clustering is huge. With bigger values we see that all the data became a cluster.
So, we see that with a small value the clusters aren't identified and with a big value all the data became a cluster. The point here is find the intermediate value that result in the good, reasonable, adequate clustering.

□

3)

How many *core points*, *border points* and *outliers* do you have in your best clustering result?

```
# <r code> ============================================================================ #
par(mar = c(2, 2, 3, 1) + .1)
clust = dbscan2.0(df, eps = 2.1, MinPts = 8)$cluster
plot(x, y, col = clust + 1, pch = ifelse(clust == 0, 4, 1), xlab = NA, ylab = NA
     , main = paste("Minpts: 8 and Eps: 2.1"))
legend(-3, -9, legend = 1:8, col = c(2:8, 1), pch = 1, bty = "n", ncol = 4)
# </r code> ============================================================================ #
```

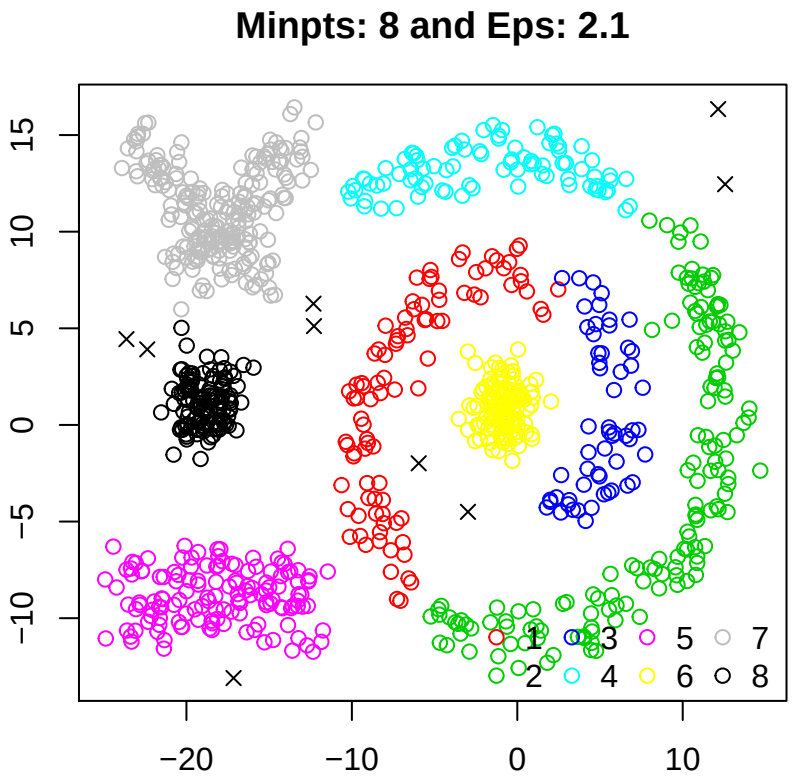

Figure 10: Best clustering result. Outliers are represented by x's.

```r
# <r code> ============================================================================ #
best.clus <- dbscan2.0(df, eps = 2.1, MinPts = 8)

cross.table <- function (x) {
  tab = table(c("seed", "border")[2 - x$is.seed], cluster = x$cluster)
  tab = rbind(tab, total = colSums(tab))
  tab = cbind(tab, total = rowSums(tab))
  print(tab)
}
cross.table(best.clus)
# </r code> ============================================================================ #
```

```
       0  1   2  3  4   5   6   7   8 total
border 9 10   8  5  1   5   1   3   1    43
seed   0 82 143 51 85 132 125 212 127   957
total  9 92 151 56 86 137 126 215 128  1000
```

In the R output above zero represent the outliers, so, we have 9 outliers and eight clusters, as we can see in Figure 10. In total we have 43 border points, and the cluster with more border points is the cluster one, with 10. In the R output we have the number of border points, the number of non-border points, and the total number of points for each cluster.

■
