# Fundamentos de la Ciencia de Datos Práctica 4

Fernández Díaz, Daniel
Cano Díaz, Francisco
Fernández Hernández, Alberto

19 de noviembre del 2019

# Índice

**1. Apartado 1** **3**
1.1. Algoritmo K-means en R . . . 3
1.1.1. Obtención de los datos . . . 3
1.1.2. Ejecución del algoritmo . . . 3
1.1.3. Resultados . . . 4

**2. Apartado 2** **5**
2.1. Programación de nuestro propio **k-means** . . . 5
2.1.1. El código . . . 5
2.1.2. Ejemplo de ejecución en 2D . . . 9
2.1.3. Ejemplo de ejecucion en 3D . . . 10
2.1.4. Visualización de los resultados . . . 11
2.2. Análisis del número de *clusters* óptimos. Comparación K-Means y *Hierarchical Clustering*. Algoritmo DBSCAN . . . 17
2.2.1. K-Means . . . 17
2.2.2. Hierarchical Clustering . . . 28
2.2.3. DBSCAN . . . 34

# 1. Apartado 1

En este apartado se resolverá el problema visto en teoría realizando un análisis de clasificación no supervisada con R tal y como vimos en clase. Por último, se realizará el proceso paso a paso de forma similar a la vista en teoría para comprobar que ambos resultados coinciden.

## 1.1. Algoritmo K-means en R

### 1.1.1. Obtención de los datos

En primer lugar, leeremos los datos de los puntos y los centroides iniciales del fichero ***apartado1.txt*** utilizando la función **muestra.leer** desarrollada para la práctica 3, la cual permite leer varias tablas o data frames de un mismo archivo.

```
> muestra <- muestra.leer("apartado1.txt")
> m_clasif <- muestra[[1]]
> c_clasif <- muestra[[2]]
```

De esta forma cargamos nuestra matriz de clasificaciones como un data frame llamado **m_clasif**:

```
  d1 d2
1  4  4
2  3  5
3  1  2
4  5  5
5  0  1
6  2  2
7  4  5
8  2  1
```

Y otro dataframe llamado **c_clasif** con las coordenadas iniciales de nuestros centroides, que en este caso son dos:

```
  d1 d2
1  0  1
2  2  2
```

### 1.1.2. Ejecución del algoritmo

Con esto, ya podemos ejecutar la función ***kmeans*** donde además de los dos data frames ya mencionados, le pasamos como tercer argumento un 4, que indica el número de iteraciones máximas que permitimos al algoritmo. Este parámetro es importante ya que el algoritmo puede no converger. Por último mencionar que típicamente se le pasan a esta función dos matrices, una con los puntos y otra con los centroides; pero funciona igual con data frames (los convierte internamente) por lo que no es necesario que nossotros hagamos esa conversión previamente.

```
> (clasif_ns <- kmeans(m_clasif,c_clasif,4))

K-means clustering with 2 clusters of sizes 4, 4

Cluster means:
    d1   d2
1 1.25 1.50
2 4.00 4.75

Clustering vector:
```

```
1 2 3 4 5 6 7 8
2 2 1 2 1 1 2 1

Within cluster sum of squares by cluster:
[1] 3.75 2.75
 (between_SS / total_SS =  84.8 %)

Available components:

[1] "cluster"      "centers"      "totss"        "withinss"     "tot.withinss"
[6] "betweenss"    "size"         "iter"         "ifault"
```

### 1.1.3. Resultados

Como resultado, obtenemos una estructura de datos con toda la información calculada por el algoritmo: un data frame con las coordenadas finales de los centroides, un vector de asociaciones, etc.
En ese vector de asociaciones o *clustering vector* figura la clase a la que pertenece cada uno de los puntos; y es precisamente este vector el que utilizaremos en un último paso para dividir los datos iniciales y extraer información del análisis. Para ello añadimos dicho vector como columna a la estructura de datos que contenía los puntos:

```
> cluster <- clasif_ns$cluster
> (m_clasif <- cbind(cluster,m_clasif))

  cluster d1 d2
1       2  4  4
2       2  3  5
3       1  1  2
4       2  5  5
5       1  0  1
6       1  2  2
7       2  4  5
8       1  2  1
```

Y dividimos dicha estructura en dos, cada una con los puntos de cada cluster:

```
> m_c1 <- subset(m_clasif,m_clasif[,1]==1)
> (m_c1 <- m_c1[,-1])

  d1 d2
3  1  2
5  0  1
6  2  2
8  2  1

> m_c2 <- subset(m_clasif,m_clasif[,1]==2)
> (m_c2 <- m_c2[,-1])

  d1 d2
1  4  4
2  3  5
4  5  5
7  4  5
```

# 2. Apartado 2

## 2.1. Programación de nuestro propio k-means

### 2.1.1. El código

En este apartado presentaremos el algoritmo **k-means** que hemos programado siguiendo los pasos vistos en la teoría. El código es el siguiente:

```
> our.kmeans <- function(muestra,centroides,aleatorio=TRUE){
+   if(aleatorio) centroides = elegir.centroides(muestra,centroides)
+
+   continuar = TRUE
+   asignacion.old = matrix(0,nrow=nrow(centroides),ncol=nrow(muestra))
+
+   while(continuar){
+     distancias = crear.matriz.distancias(muestra,centroides)
+     asignacion = crear.matriz.asignacion(distancias)
+     centroides = crear.centroides(muestra,distancias,asignacion)
+
+     if(comprobar.matriz.asignacion(asignacion,asignacion.old)) continuar = FALSE
+     else asignacion.old = asignacion
+   }
+
+   vector.clusters = crear.vector.clusters(asignacion)
+   list(m.distancias=distancias,m.asignacion=asignacion,centroides=centroides,v.clusteriz=vector.clusters)
+ }
> ## Distancia euclidea ##
> # Calcula la distancia euclidea entre dos puntos.
> distancia.euclidea <- function(p1,p2){
+   sum = 0
+   for(i in 1:length(p1)){
+           sum = sum + ((p1[1,i] - p2[1,i])^2)
+   }
+   sqrt(sum)
+ }
> ## Seleccion de centroides de forma aleatoria ##
> # A partir de la muestra utilizada para el K-MEANS obtiene
> # los centroides de forma aleatoria a partir de los puntos
> # que conforman la muestra.
> elegir.centroides <- function(muestra,num.centroides){
+   if(num.centroides > nrow(muestra)){
+           num.centroides <- nrow(muestra)
+   } else if(num.centroides < 1){
+           num.centroides <- 1
+   }
+
+   centroides = matrix(nrow=0,ncol=ncol(muestra))
+
+   # Obtenemos cada uno de los centroides. Estos centroides seran,
+   # de forma aleatoria, algunos o todos los puntos de la muestra.
+   for(i in 1:num.centroides){
+     centroide = floor(runif(1,min=1,max=nrow(muestra)))
+     centroides = rbind(centroides,muestra[centroide,])
+     muestra = muestra[-centroide,]
+   }
+   colnames(centroides) <- c(paste0("d",1:ncol(centroides)))
+   rownames(centroides) <- c(paste0("C",1:nrow(centroides)))
+   as.data.frame(centroides)
+ }
> ## Creacion de la matriz de distancias ##
> # Obtenemos en forma de matriz la distancia de cada uno de
> # los puntos de la muestra a cada uno de los centroides definidos.
```

```
> crear.matriz.distancias <- function(muestra,centroides){
+   vd = c()
+   md = matrix(nrow=0,ncol=nrow(muestra))
+   for(i in 1:nrow(centroides)){
+     # Obtenemos las coordenadas del centroide correspondiente.
+     c = centroides[i,]
+     for(j in 1:nrow(muestra)){
+       # Obtenemos las coordenadas del punto correspondiente.
+       m = muestra[j,]
+       # Calculamos la distancia entre ambos y la añadimos al vector.
+       vd = c(vd,distancia.euclidea(m,c))
+     }
+     md = rbind(md,vd)
+     vd = c()
+   }
+   colnames(md) = c(paste0("P",1:nrow(muestra)))
+   rownames(md) = c(paste0("C",1:nrow(centroides)))
+   as.data.frame(md)
+ }
> ## Creacion de la matriz de asignacion ##
> # Obtenemos en forma de matriz la pertenencia de cada punto
> # de la muestra a cada cluster de tal forma que se indicara
> # con un 1 aquellos puntos que pertenezcan y un 0 a aquellos
> # que no.
> crear.matriz.asignacion <- function(distancias){
+   #Creamos la matriz de asignacion con valores a 0
+   ma = matrix(0,nrow=nrow(distancias),ncol=ncol(distancias))
+   rownames(ma) = c(paste0("C",1:nrow(distancias)))
+   colnames(ma) = c(paste0("P",1:ncol(distancias)))
+
+   for(i in 1:ncol(distancias)){
+     # Obtiene el indice del menor valor del punto correspondiente.
+     # Este punto sera asignado al cluster mas cercano.
+     indice.menor = which.min(distancias[,i])
+
+     #Por tanto el valor de ese puto para ese cluster sera 1: pertenece
+     ma[indice.menor,i] = 1
+   }
+
+   # Una vez tenemos el vector con las asignaciones lo convertimos a dataframe.
+   as.data.frame(ma)
+ }
> ## Creacion de los nuevos centroides ##
> # Obtenemos los nuevos centroides a partir de los nuevos
> # puntos obtenidos en la matriz de asignacion. Para ello
> # calculamos el promedio de cada coordenada obteniendo asi
> # las coordenadas de cada uno de los centroides.
> crear.centroides <- function(muestra,distancias,asignacion){
+   puntos = matrix(nrow=0,ncol=ncol(muestra))
+   centroides = matrix(nrow=0,ncol=ncol(muestra))
+
+   for(i in 1:nrow(asignacion)){
+     for(j in 1:ncol(asignacion)){
+       # Para los puntos que pertenecen a cada cluster,
+       # obtenemos la coordenada x e y de todos ellos.
+       if(asignacion[i,j] == 1){
+         puntos = rbind(puntos,muestra[j,])
+       }
+     }
+     # Obtenemos el nuevo centroide mediante la media de las
+     # coordenadas.
+     centroides = rbind(centroides,calcular.centroide(puntos))
```

```
+     puntos = matrix(nrow=0,ncol=ncol(muestra))
+   }
+
+   # Una vez tenemos el vector con los centroides lo convertimos a dataframe.
+   rownames(centroides) = c(paste0("C",1:nrow(centroides)))
+   colnames(centroides) = c(paste0("d",1:ncol(centroides)))
+   as.data.frame(centroides)
+ }
> ## Calculo de las coordenadas del centroide ##
> # Se calculan las coordenadas del centroide a partir de la
> # media de los puntos de su cluster
> calcular.centroide <- function(puntos){
+   centroide = c()
+   sum = 0
+
+   # Calculo de la media de cada coordenada o dimension de los puntos
+   for(i in 1:ncol(puntos)){
+     for(j in 1:nrow(puntos)){
+       sum = sum + puntos[j,i]
+     }
+     centroide = c(centroide,sum/nrow(puntos))
+     sum = 0
+   }
+   centroide
+ }
> ## Condicion de parada ##
> # Indica al algoritmo si debe parar atendiendo a la comparacion de
> # la matriz de asignacion anterior y la nueva. Si no cambia
> # (iguales=TRUE) significa que los centroides no se han movido en
> # esta iteracion y podemos parar
> comprobar.matriz.asignacion <- function(asignacion.antigua, asignacion.nueva){
+   iguales = asignacion.antigua == asignacion.nueva
+   iguales = all(iguales)
+ }
> ## Creacion del vector de clusterizacion ##
> crear.vector.clusters <- function(asignacion){
+   mc = matrix(nrow=1,ncol=ncol(asignacion))
+   rownames(mc) = "C"
+   colnames(mc) = c(paste0("P",1:ncol(asignacion)))
+
+   for(i in 1:ncol(asignacion)){
+     for(j in 1:nrow(asignacion)){
+       if(asignacion[j,i]) mc[1,i] = j
+     }
+   }
+
+   as.data.frame(mc)
+ }
> ## Grafica de los resultados (2D)##
> # Muestra los puntos sobre el plano X-Y, asi como los centroides
> # y los clusters obtenidos
> plot2D.our.kmeans <- function(muestra,kmeans){
+   # Nos aseguramos de que el imput sea 2D
+   if(ncol(muestra)!=2) stop("Dimensiones erroneas. Asegurate de que las estructuras son de 2D")
+
+   # Estructura aux con vector de clusterizacion, muestra y matriz de distancias
+   aux = cbind(C=t(kmeans$v.clusteriz),muestra=muestra,dist=t(kmeans$m.distancias))
+
+   # Calculo del rango del plot
+   rango.x = range(aux$muestra.d1); rango.y = range(aux$muestra.d2)
+   centroide.min = centroide.minimo(kmeans$centroides,rango.x,rango.y) # Etiqueta del centroide min
+   centroide.max = centroide.maximo(kmeans$centroides,rango.x,rango.y) # Etiqueta del centroide max
```

```
+   clusters = crear.clusters(aux,nrow(kmeans$centroides)) # Datos clusterizados
+
+   rango.x = c(min(kmeans$centroides[centroide.min,][1]-max(clusters[[centroide.min]][paste0("dist.",centroide.min)]),rango.x[1],
+     max(kmeans$centroides[centroide.max,][1]+max(clusters[[centroide.max]][paste0("dist.",centroide.max)]),rango.x[2]))
+   rango.y = c(min(kmeans$centroides[centroide.min,][2]-max(clusters[[centroide.min]][paste0("dist.",centroide.min)]),rango.y[1],
+     max(kmeans$centroides[centroide.max,][2]+max(clusters[[centroide.max]][paste0("dist.",centroide.max)]),rango.y[2]))
+
+   # Plot centroides
+   plot(kmeans$centroides,pch=22,col='blue',xlim=rango.x,ylim=rango.y)
+   etiquetas = c("Centroides"); puntos = c(22); linea = c(0); color = c(4)
+
+   for(i in 1:length(clusters)){
+     # Plot puntos
+     points(clusters[[paste0("C",i)]]$muestra.d1,clusters[[paste0("C",i)]]$muestra.d2,pch=19,col=i)
+     etiquetas = c(etiquetas,paste0("Cluster ",i)); puntos = c(puntos,19); linea = c(linea,0); color = c(color,i)
+
+     # Plot contorno actual del cluster
+     symbols(kmeans$centroides[paste0("C",i),'d1'],kmeans$centroides[paste0("C",i),'d2'],
+       circle=0.1+max(clusters[[paste0("C",i)]][paste0("dist.C",i)]),add=TRUE,inches=FALSE)
+     etiquetas = c(etiquetas,paste0("Frontera ",i)); puntos = c(puntos,46); linea = c(linea,1); color = c(color,1)
+   }
+
+   # Titulo
+   title(sprintf("Clusterizacion K-means con %i clusters",length(clusters)))
+   # Leyenda
+   legend(rango.x[1], rango.y[2], legend=etiquetas, lty=linea, pch = puntos, col = color, cex=0.8)
+ }
> ## Calcular el centroide minimo ##
> # Calcula cual es el centroide con la distancia mas cercana al limite inferior del rango del plot
> centroide.minimo <- function(centroides,rango.x,rango.y){
+   distancias = c()
+   for(i in 1:nrow(centroides)){
+     distancias = c(distancias,distancia.euclidea(centroides[i,],matrix(c(rango.x[1],rango.y[1]),nrow=1)))
+   }
+   rownames(centroides[(order(distancias)[1]),])
+ }
> ## Calcular el centroide maximo ##
> # Calcula cual es el centroide con la distancia mas cercana al limite superior del rango del plot
> centroide.maximo <- function(centroides,rango.x,rango.y){
+   distancias = c()
+   for(i in 1:nrow(centroides)){
+     distancias = c(distancias,distancia.euclidea(centroides[i,],matrix(c(rango.x[2],rango.y[2]),nrow=1)))
+   }
+   rownames(centroides[(order(distancias)[1]),])
+ }
> ## Crear clusters ##
> # Dado un conjunto de datos con un vector de clusterizacion en una de las
> # columnas, agrupa los datos en grupos en funcion de a que cluster pertenece
> # cada punto y devulve una lista de dichos puntos
> crear.clusters <- function(datos,num.clusters){
+   grupos = list()
+   for(i in 1:num.clusters){
+     grupos[[paste0("C",i)]] = subset(datos,datos$C==i)
+   }
+   grupos
+ }
```

### 2.1.2. Ejemplo de ejecución en 2D

La función principal de nuestro algoritmo es ***our.kmeans(muestra,centroides,aleatorio)***, donde:

- **muestra** son los puntos a clusterizar
- **centroides** puede ser:
  - un dataframe con las coordenadas de los centroides que queremos utilizar en esa clusterizacion. Para ello el parametro aleatorio debe estar a FALSE, indicando al programa que no se desean coordenadas aleatorias para los centroides iniciales.
  - un entero, el numero de clusters en el que se desean dividir los datos. Para ello el parametro aleatorio debe estar a TRUE, indicando al programa que se desean coordenadas aleatorias para los centroides iniciales.
- **aleaorio** es una variable logica con valor TRUE por defecto. Indica al programa si se desean o no coordenadas aleatorias para los centroides iniciales.

Si ejecutamos la función con los datos del apartado 1 (muestra y centroides) obtenemos unos resultados muy similares al algoritmo de el paquete *stats* de CRAN:

```
> m_clasif <- muestra[[1]]
> c_clasif <- muestra[[2]]
> (km <- our.kmeans(m_clasif,c_clasif,F))

$m.distancias
         P1       P2       P3       P4       P5        P6       P7        P8
C1 3.716517 3.913119 0.559017 5.129571 1.346291 0.9013878 4.451123 0.9013878
C2 0.750000 1.030776 4.069705 1.030776 5.482928 3.4003676 0.250000 4.2500000

$m.asignacion
   P1 P2 P3 P4 P5 P6 P7 P8
C1  0  0  1  0  1  1  0  1
C2  1  1  0  1  0  0  1  0

$centroides
     d1   d2
C1 1.25 1.50
C2 4.00 4.75

$v.clusteriz
  P1 P2 P3 P4 P5 P6 P7 P8
C  2  2  1  2  1  1  2  1
```

### 2.1.3. Ejemplo de ejecucion en 3D

Este algoritmo funciona también con puntos de más de dos dimensiones. Veamos un pequeño ejemplo: cojamos los puntos de la muestra original que acabamos de utilizar y añadamosle una tercera dimensión pero manteniendo los puntos en un mismo plano, de forma que no perdamos la agrupación original de estos. El algoritmo **K-means** solo funciona bien con puntos claramente agrupados.

```
> (m3 <- cbind(muestra[[1]],d3=(1:8)/4))

  d1 d2   d3
1  4  4 0.25
2  3  5 0.50
3  1  2 0.75
4  5  5 1.00
5  0  1 1.25
6  2  2 1.50
7  4  5 1.75
8  2  1 2.00

> our.kmeans(m3,2)

$m.distancias
          P1       P2        P3       P4       P5        P6        P7       P8
C1 3.8830561 4.009754 0.8385255 5.143260 1.352082 0.9100137 4.4668921 1.096871
C2 0.9762812 1.096871 4.0716244 1.038328 5.495737 3.4573292 0.9100137 4.396376

$m.asignacion
   P1 P2 P3 P4 P5 P6 P7 P8
C1  0  0  1  0  1  1  0  1
C2  1  1  0  1  0  0  1  0

$centroides
     d1   d2    d3
C1 1.25 1.50 1.375
C2 4.00 4.75 0.875

$v.clusteriz
  P1 P2 P3 P4 P5 P6 P7 P8
C  2  2  1  2  1  1  2  1
```

Como podemos ver, al estar todos en un mismo plano los clusters solucion se mantienen inalterados aunque con una dimensión más.

### 2.1.4. Visualización de los resultados

Además, hemos creado también una función ***plot2D.our.kmeans(muestra,kmeans)*** con los siguientes parámetros:

- **muestra** es la muestra inicial utilizada en el algoritmo **k-means**
- **kmeans** es la estructura (lista) devuelta por dicho algoritmo

Esta función solo sirve para puntos en **dos dimensiones**. Dados dichos puntos y la información de la clusterizacion obtenida de la estructura kmeans, realiza un plot de dichos puntos agrupados en sus respectivos clusters (dibujados con diferente color para cada cluster); así como los centroides de dichos clusters y una especie de frontera imaginaria del cluster que contiene los puntos que lo conforman hasta el momento. Este círculo no tiene valor analítico pero se ha considerado incluirlo en estas gráficas de ejemplo para hacer la clusterización más visual. En un estudio serio, estos círculos se quitarían.
Veamos ahora algunos ejemplos con los puntos originales en 2D y con una división en 2, 3, 4, 5 y 6 clusters:

```
> plot2D.our.kmeans(m_clasif,our.kmeans(m_clasif,2))
```

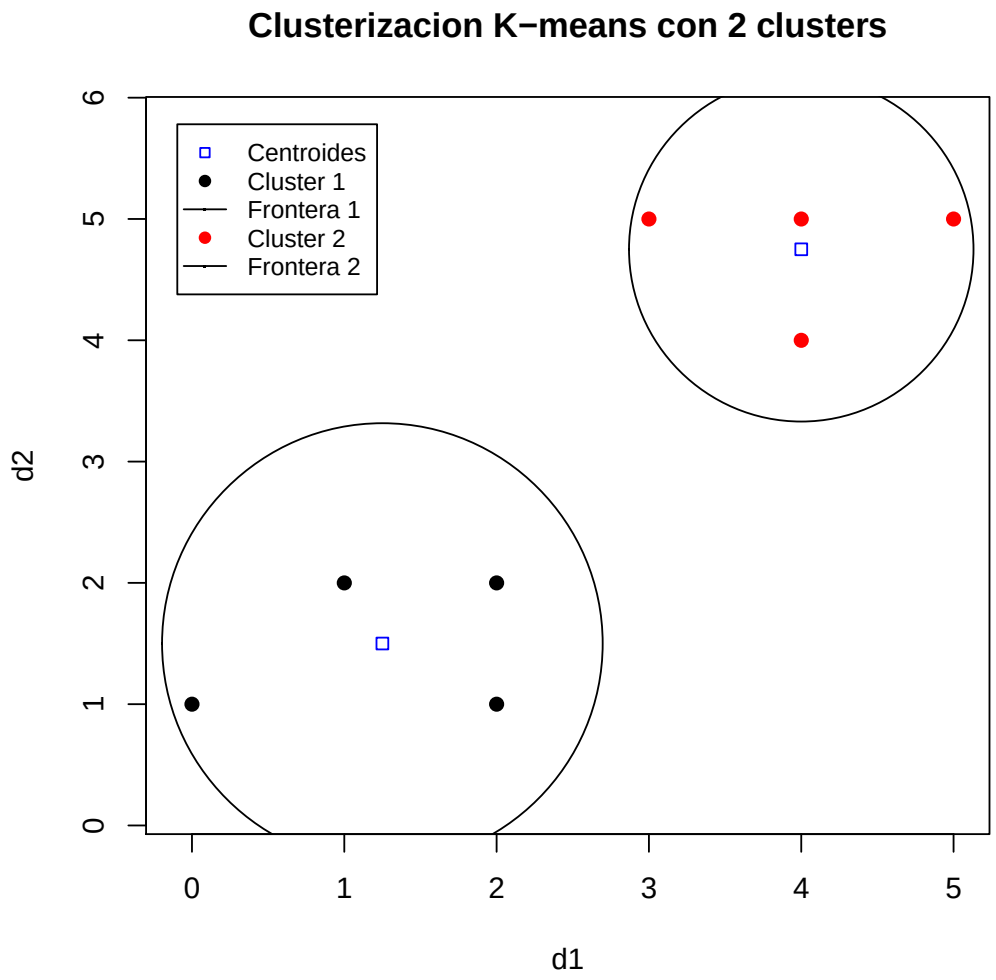



Figura 1: División en 2 clusters

```
> plot2D.our.kmeans(m_clasif,our.kmeans(m_clasif,3))
```

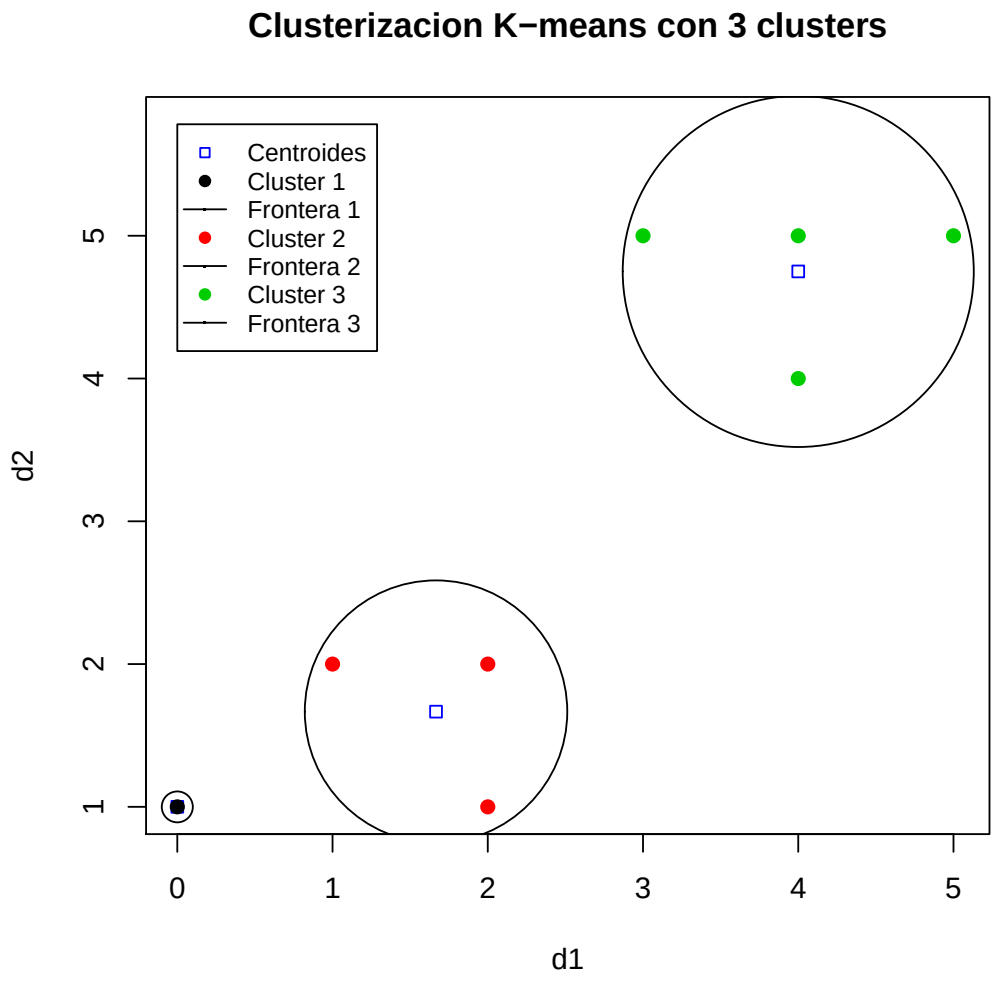



Figura 2: División en 3 clusters

```
> plot2D.our.kmeans(m_clasif,our.kmeans(m_clasif,4))
```

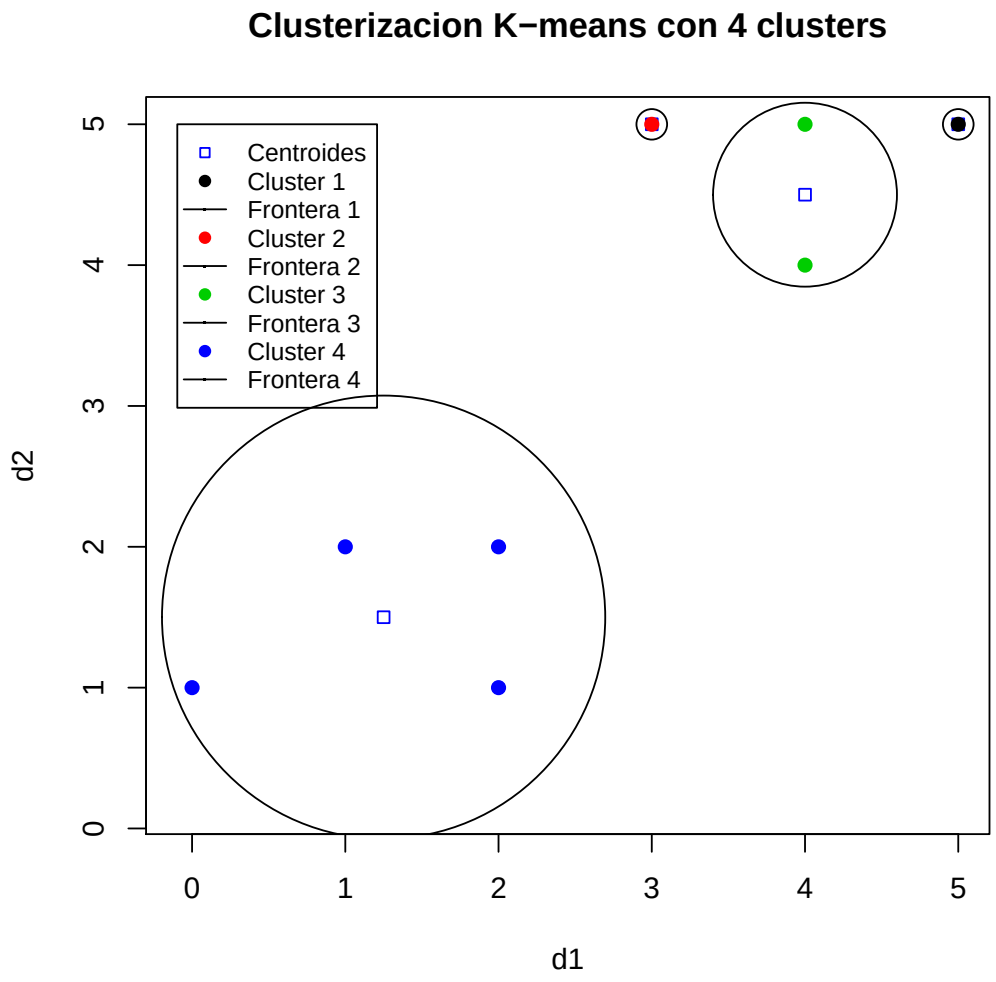



Figura 3: División en 4 clusters

```
> plot2D.our.kmeans(m_clasif,our.kmeans(m_clasif,5))
```

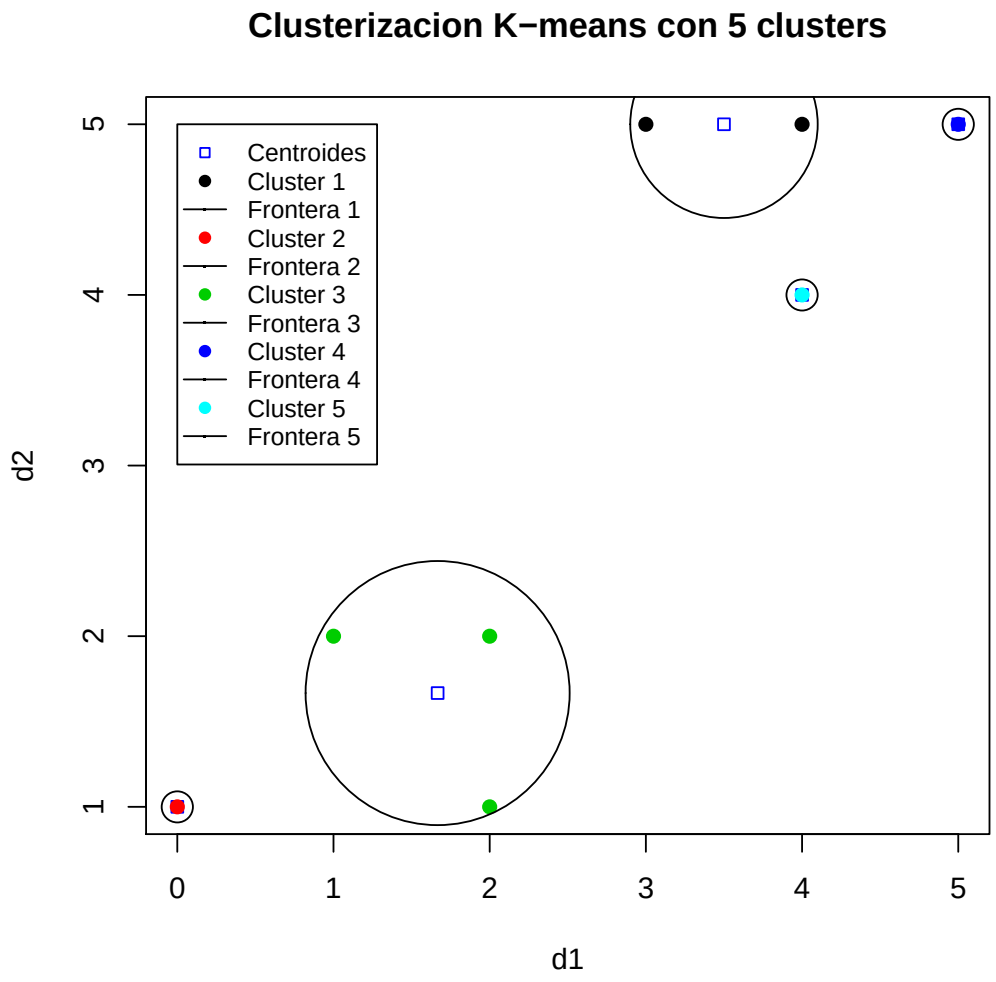



Figura 4: División en 5 clusters

```
> plot2D.our.kmeans(m_clasif,our.kmeans(m_clasif,6))
```

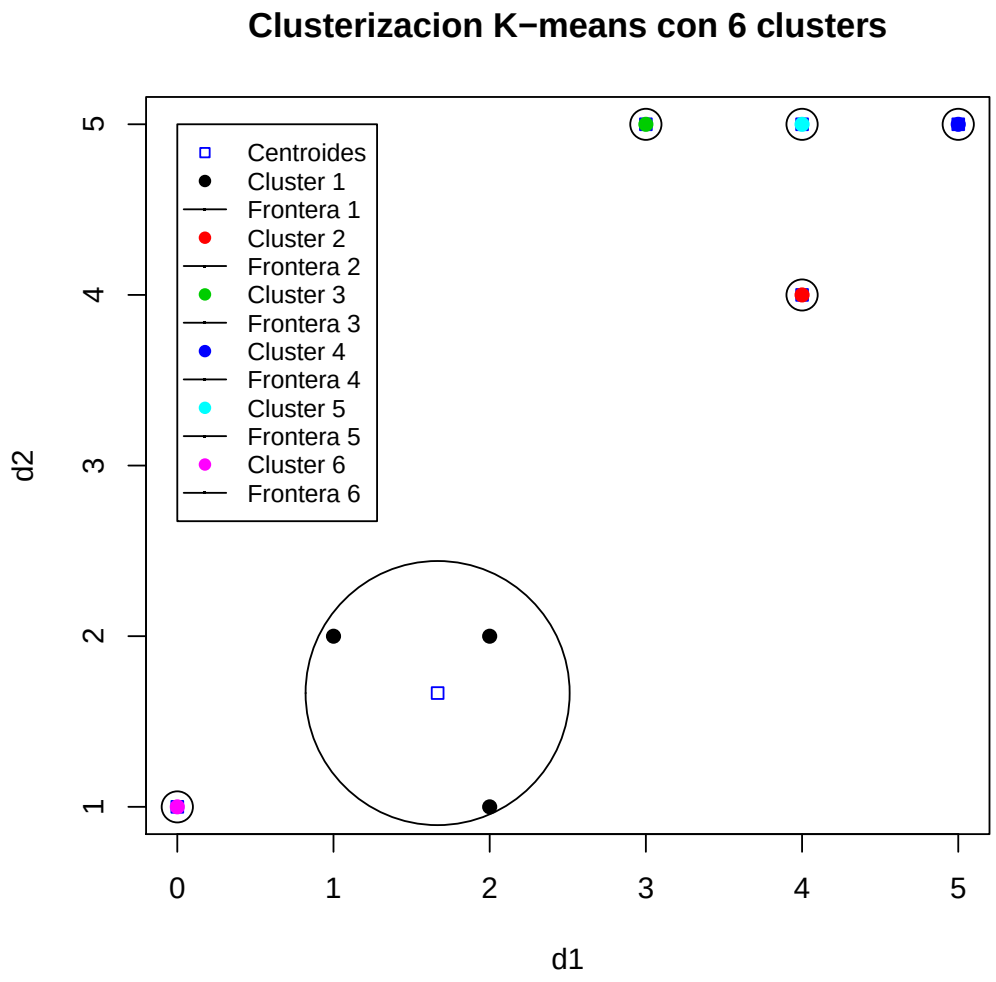



Figura 5: División en 6 clusters

## 2.2. Análisis del número de *clusters* óptimos. Comparación K-Means y *Hierarchical Clustering*. Algoritmo DBSCAN

### 2.2.1. K-Means

Vamos a realizar una modificación del ejercicio de clase, en el **visualizaremos puntos de acceso *Wi-Fi***, agrupando los diferentes puntos mediante *K-Means*. Para ello utilizaremos un *dataset* geográfico con los puntos de acceso *Wi-Fi* de la ciudad de Nueva York. [1] Para visualizar los datos, utilizaremos la librería *dplyr*, el cual nos permite manipular un *dataframe* de forma similar a una consulta `SQL`.

```
> # Incluimos la libreria dplyr
> if(!require(dplyr)){
+   install.packages("dplyr")
+   require(dplyr)
+ }
```

```
--- Please select a CRAN mirror for use in this session ---

  There is a binary version available but the source version is later:
      binary source needs_compilation
rlang  0.4.1  0.4.2              TRUE

  Binaries will be installed
package 'rlang' successfully unpacked and MD5 sums checked
package 'dplyr' successfully unpacked and MD5 sums checked

The downloaded binary packages are in
        C:\Users\solol\AppData\Local\Temp\RtmpCgG4GG\downloaded_packages
```

```
> # Preparamos el dataframe
> newyork <- read.csv("NYC_Free_Public_WiFi_03292017.csv")
> # Similar a la consulta SQL:
> # SELECT * FROM newyork LIMIT 5
> newyork %>% head(5)
```

```
  BORO                                  the_geom OBJECTID TYPE             PROVIDER         NAME              LOCATION
1   BK POINT (-73.87053740957452 40.68406083967918)    10321 Free LinkNYC - Citybridge bk-05-145941     3386 FULTON STREET 40.68
2   BK POINT (-73.86897452703059 40.68462509021575)    10322 Free LinkNYC - Citybridge bk-05-145940     3435 FULTON STREET 40.68
3   BK POINT (-73.86830878947508 40.68470155389536)    10323 Free LinkNYC - Citybridge bk-05-145939     3450 FULTON STREET 40.68
4   BK POINT (-73.86677732990765 40.68513094043811)    10324 Free LinkNYC - Citybridge bk-05-145938     3480 FULTON STREET 40.68
5   BK POINT (-73.89716745051707 40.67647466963193)    10325 Free LinkNYC - Citybridge bk-05-145932 62 PENNSYLVANIA AVENUE 40.67
               SSID    SOURCEID                    ACTIVATED BOROCODE BORONAME NTACODE                NTANAME COUNDIST POSTCOD
1 LinkNYC Free Wi-Fi LINK-021921 11/21/2017 12:00:00 AM +0000        3 Brooklyn    BK83 Cypress Hills-City Line       37    1120
2 LinkNYC Free Wi-Fi LINK-021922 11/21/2017 12:00:00 AM +0000        3 Brooklyn    BK83 Cypress Hills-City Line       37    1120
3 LinkNYC Free Wi-Fi LINK-021923 11/21/2017 12:00:00 AM +0000        3 Brooklyn    BK83 Cypress Hills-City Line       37    1120
4 LinkNYC Free Wi-Fi LINK-021925 12/20/2017 12:00:00 AM +0000        3 Brooklyn    BK83 Cypress Hills-City Line       37    1120
5 LinkNYC Free Wi-Fi LINK-021926 02/06/2018 12:00:00 AM +0000        3 Brooklyn    BK82           East New York       37    1120
```

El *dataframe* contiene, entre otros campos, los siguientes atributos:

1. *BORO*: barrio de la ciudad (ejemplo: BK = *Brooklyn*).
2. *the_geom*: coordenadas de latitud y longitud.
3. *OBJECTID*: id del punto de conexión.
4. *TYPE*: tipo de red (*free, limited-free...*)
5. *PROVIDER*: nombre del proveedor.
6. *NAME*: nombre de la red.
7. *LOCATION*: localización.

[1] `https://data.cityofnewyork.us/api/views/varh-9tsp/rows.csv?accessType=DOWNLOAD`

8. *LAT*: latitud.

9. *LON*: longitud.

Para aplicar el algoritmo *K-Means*, utilizaremos únicamente los atributos *LAT* y *LON*:

```
> newyork.df <- data.frame(newyork$LAT, newyork$LON)
```

Una vez tengamos estas columnas, vamos a determinar el número de *clusters* óptimos. Para ello vamos a trabajar con dos algoritmos básicos:

- *Elbow Method*
- *Average Silhouette Method*

**Elbow Method** : El objetivo de este algoritmo es **minimizar la suma de los cuadrados de las distancias de cada punto con respecto a su cluster correspondiente**. Supongamos que queremos comparar cuál es el número óptimo de **centroides** entre 1 y $k$:

1. *Por cada k iteración, ejecutamos el algoritmo kmeans con un número k de clusters*
2. *A continuación, calculamos la* ***suma de los cuadrados de las distancias de cada punto con su cluster respectivo***

Una vez realizada la iteración de 1 a $k$, elegimos el número de clusters cuya distancia sea mínima, es decir, el objetivo será **minimizar la suma de los cuadrados de las distancias**.
Con el dataset anterior, vamos a determinar el número óptimo de clusters, comprendido entre 1 y 20. Para ello creamos inicialmente una función para el cálculo de la suma del cuadrado de las distancias, al que llamaremos *wss*, del inglés *within-cluster sum of squares*; a continuación ejecutamos el algoritmo *kmeans* con un número $k$ de *clusters*, devolviendo la columna *tot.withinss*, la cual contiene la **suma total de los cuadrados de cada punto a su *cluster***:

```
> # Funcion para el calculo
> wss <- function(k){
+   # DataFrame = newyork.df
+   # nstart = numero de conjuntos que se crean inicialmente
+   # Al ser aleatorio este campo por defecto, vamos a establecer
+   # un valor determinado
+   kmeans(newyork.df, k, nstart = 10)$tot.withinss
+ }
```

A continuación, creamos un vector de 1 a $k$ clusters.

```
> k.values <- 1:20
```

Para ejecutar el algoritmo *kmeans* para cada valor de $k$ vamos a utilizar una función denominada *map*, la cual nos permite evaluar una misma función para diferentes parámetros, a través de un vector. Esta función se encuentra incluida en el paquete *purrr*, por lo que debemos instalarla y añadirla de forma previa:

```
> # Incluimos el paquete purrr
> if(!require(purrr)){
+   install.packages("purrr")
+   require(purrr)
+ }
```

```
> # Ejecutamos map, aplicando la funcion k.values
> # sobre un vector k de centroides. Como el resultado
> # sera de tipo double, existen variantes de la funcion
> # map como map_dbl, la cual devuelve la solucion en
> # formato double.
> wss.values <- map_dbl(k.values,wss)
```

Una vez ejecutado el algoritmo, mostramos los resultados por pantalla, mostrando **la suma total del cuadrado de las distancias en función del número de *clusters***:

```
> plot(k.values, wss.values, type = "b", pch = 19, frame = F,
+ xlab = "Numero k de clusters (1:20)",
+ ylab = "Suma del cuadrado de las distancias")
```

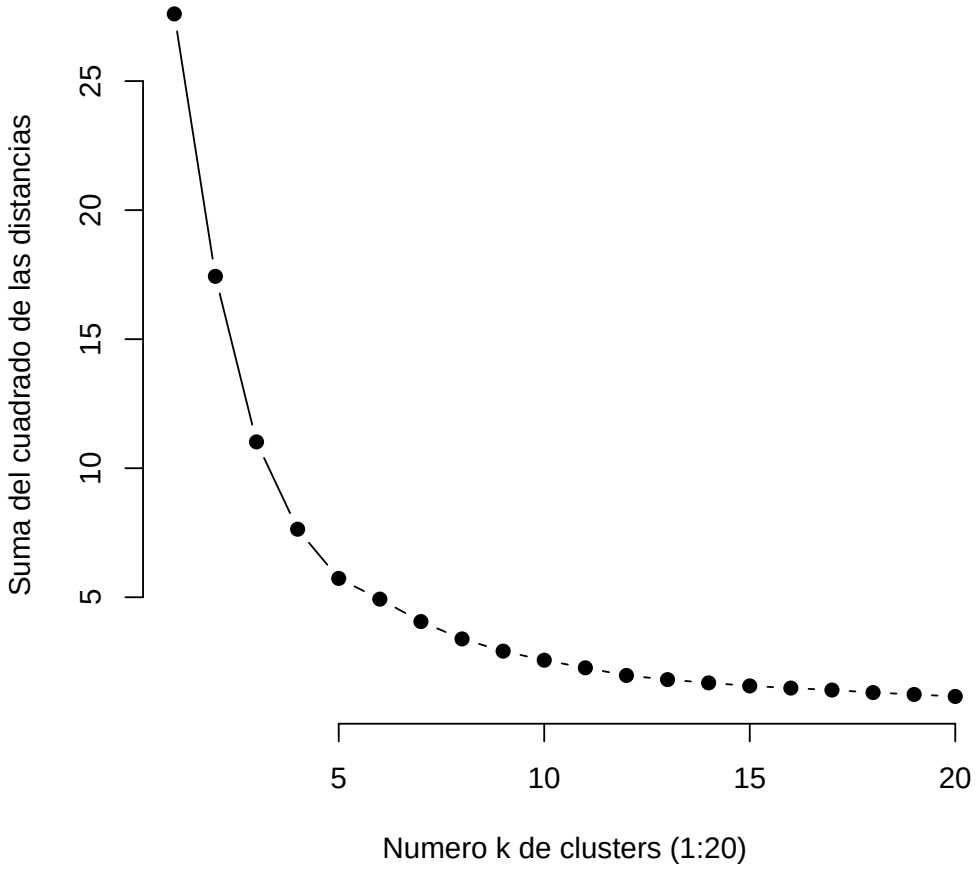



Figura 6: Elbow Method

Afortunadamente, el método *Elbow* ya está implementado en una función, denominada *fviz_nbclust*, disponible en el paquete *factoextra*:

```
> if(!require(factoextra)){
+   install.packages("factoextra")
+   library(factoextra)
+ }

> # Ejecutamos la funcion fviz_nbclust
> fviz_nbclust(newyork.df, kmeans, method = "wss", k.max = 20)
```

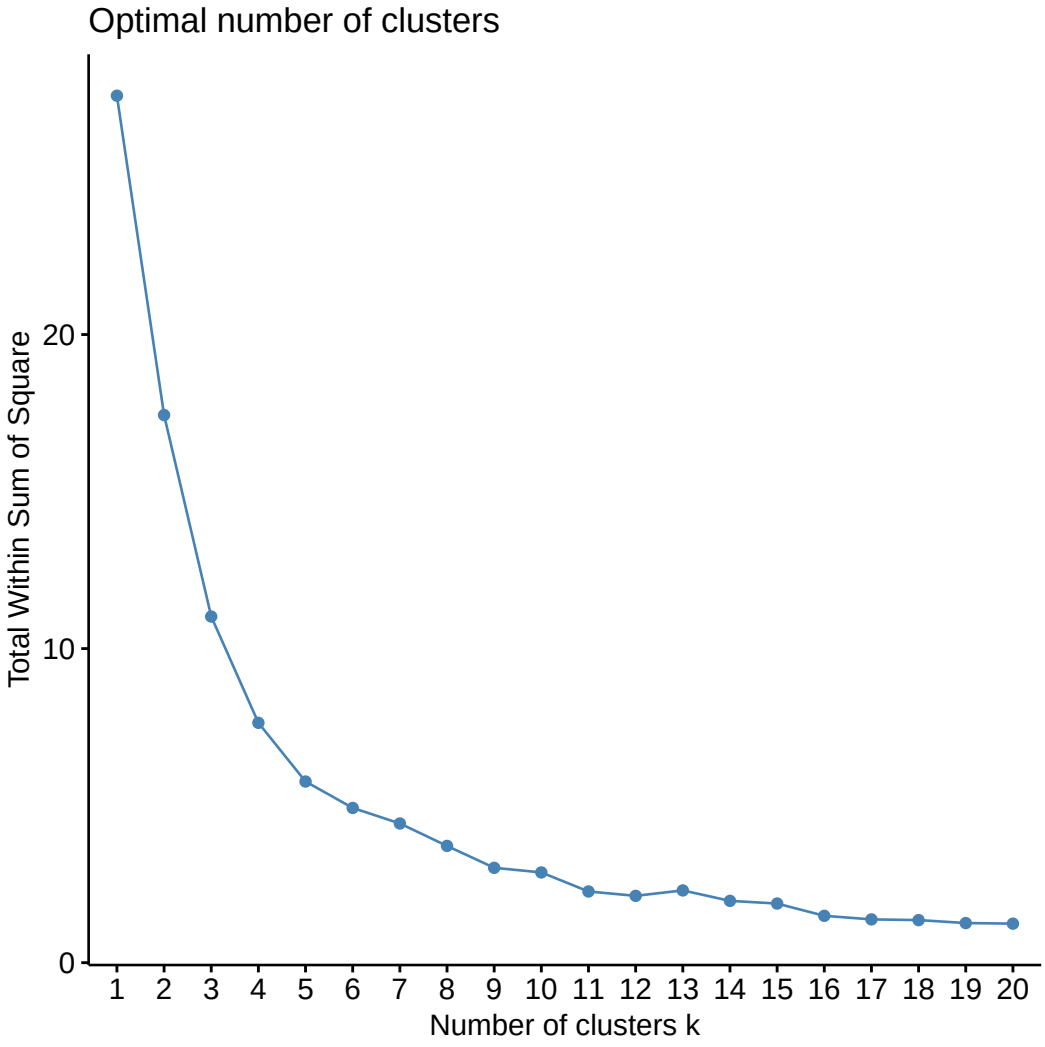



Figura 7: Elbow Method

Como podemos observar, **la suma del cuadrado de las distancias se reducen a partir de $k = 11$ centroides hasta llegar al mínimo con $k = 20$.**
Sin embargo, el hecho de escoger un mayor número de *clusters* no implica que la clasificación sea más óptima. Por ello, vamos a realizar un segundo análisis:

**Average Silhouette** : Este algoritmo se encarga de **medir el grado de calidad de un *cluster***, es decir si un objeto ha sido bien clasificado o no:
En primer lugar, **escogemos un punto *i* perteneciente al *cluster*** $C_i$

$$i \in C_i$$

calculamos la **distancia media entre el punto *i* elegido y el resto de puntos pertenecientes al mismo *cluster***:

$$a(i) = \frac{1}{C_i - 1} \sum_{i \neq j, j \in C_i} d(i,j)$$

Donde $C_i$ es el número de elementos contenidos en el *cluster*. En la ecuación anterior, restamos $C_i$ - *1* porque no tenemos en cuenta la distancia al elemento *i*, es decir, *d(i,i)*. Por tanto, *a(i)* permite medir el grado de pertenencia del elemento *i* al *cluster*.
A continuación, con ese mismo punto *i* calculamos **la distancia media del punto *i* a todos los puntos *j* perteneciente a otro *cluster*** $C_k$**, donde** $C_j \neq C_i$:

$$\frac{1}{C_k} \sum_{j \in C_k} d(i,j)$$

Como tenemos varios *clusters*, obtendremos **el mínimo de entre todos los *clusters***:

$$b(i) = \text{mín} \frac{1}{C_k} \sum_{j \in C_k} d(i,j)$$

Por tanto, el *cluster* con la menor media será el ***cluster* vecino**. Finalmente, para medir el grado de pertenencia calculamos:

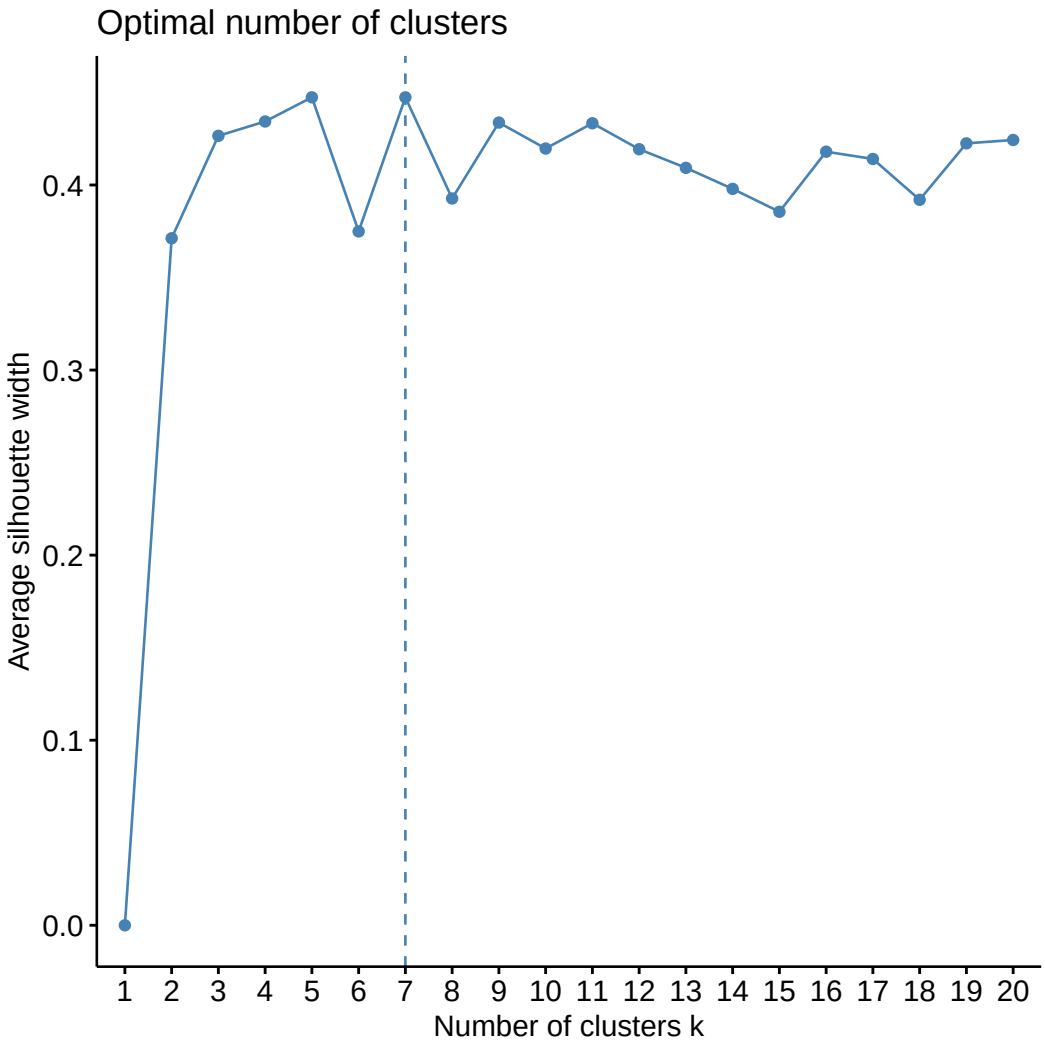
$$s(i) = \frac{b(i) - a(i)}{max\{a(i), b(i)\}}$$

Analicemos los posibles resultados:

- *s(i) = 0*: **significa que el dato se encuentra en la frontera entre dos *clusters***.
- *1 - a(i)/b(i)*: **significa que el dato se encuentra bien clasificado**, es decir, la media de las distancias del punto *i* a los elementos del *cluster* inicial (**a(i)**) es **menor** que la distancia media de *i* al siguiente cluster más cercano (**b(i)**), lo cual indica que *i* se ha clasificado correctamente.
- *1 - b(i)/a(i)*: **significa que el dato se encuentra mal clasificado**, es decir, la media de las distancias del punto *i* a los elementos del *cluster* vecino (**b(i)**) es **menor** que la distancia media de *i* al siguiente cluster original (**b(i)**), es decir, se encuentra más cerca del *cluster* vecino que del original.

A continuación, vamos a analizar el grado de clasificación utilizando el algoritmo anterior, utilizando la función *fviz_nbclust* en la que cambiamos el método a *silhouette*:

```
> fviz_nbclust(newyork.df, kmeans, method = "silhouette", k.max = 20)
```




Analizando la gráfica anterior, observamos que $k = 7$ es el **número óptimo de *clusters***, ya que presenta el mayor *s(i)*.
Una vez determinado el número de *clusters*, comenzamos con el análisis de clasificación. En primer lugar, ejecutamos el algoritmo *kmeans*:

```
> # Numero de clusters = 7
> clasificacion.ns <- kmeans(newyork.df, 7)
```

A continuación, calculamos el valor medio de cada *cluster*. Para ello, agrupamos los datos en función del *cluster* al que pertenezca (para ello, la funcion *kmeans* dispone de una columna, denominada *cluster* que indica a qué conjunto pertenece cada dato):

```
> aggregate(newyork.df,by=list(clasificacion.ns$cluster),FUN=mean)

  Group.1 newyork.LAT newyork.LON
1       1    40.59225   -74.06737
2       2    40.72967   -73.85391
3       3    40.67242   -73.78497
4       4    40.73038   -73.92993
5       5    40.82619   -73.92717
6       6    40.67763   -73.97942
7       7    40.75507   -73.98198
```

Una vez calculada la media de cada conjunto, añadimos al *dataframe* original una nueva columna con el *cluster* al que pertenece cada dato:

```
> newyork.df <- data.frame(newyork.df, clasificacion.ns$cluster)
> # Mostrmoas las 10 primeras filas
> head(newyork.df,10)

   newyork.LAT newyork.LON clasificacion.ns.cluster
1     40.68406   -73.87054                       2
2     40.68463   -73.86897                       2
3     40.68470   -73.86831                       2
4     40.68513   -73.86678                       2
5     40.67647   -73.89717                       4
6     40.67722   -73.89746                       4
7     40.67699   -73.89836                       4
8     40.67704   -73.89921                       4
9     40.67754   -73.90195                       4
10    40.67789   -73.90299                       4

> sapply(newyork.df, class)

             newyork.LAT              newyork.LON clasificacion.ns.cluster
               "numeric"                "numeric"                "integer"
```

Sin embargo, el tipo de dato de la columna *cluster* es *integer*, por lo que debemos cambiarlo a *factor*:

```
> newyork.df$clasificacion.ns.cluster <- as.factor(newyork.df$clasificacion.ns.cluster)
```

Finalmente, vamos a representar gráficamente los *clusters*. Para ello, utilizaremos el paquete *ggplot2*. [2] *ggplot2* forma parte de un conjunto de subpaquetes del paquete *tidyverse* para análisis y manipulación de datos. En concreto, este paquete proporciona herramientas para una mejor visualización de la información, basándose en la **gramática de gráficos**, es decir, que cualquier gráfico pueda expresarse a partir de la combinación de:

- **Un conjunto de datos**
- **Un sistema de coordenadas**

[2] https://ggplot2.tidyverse.org

- **Un conjunto de herramientas para representar visualmente los datos, denominados *geoms* (puntos, líneas, líneas discontínuas etc.)**

Veamos un ejemplo con el *dataframe* anterior: en primer lugar instalamos y cargamos el paquete *ggplot2*:

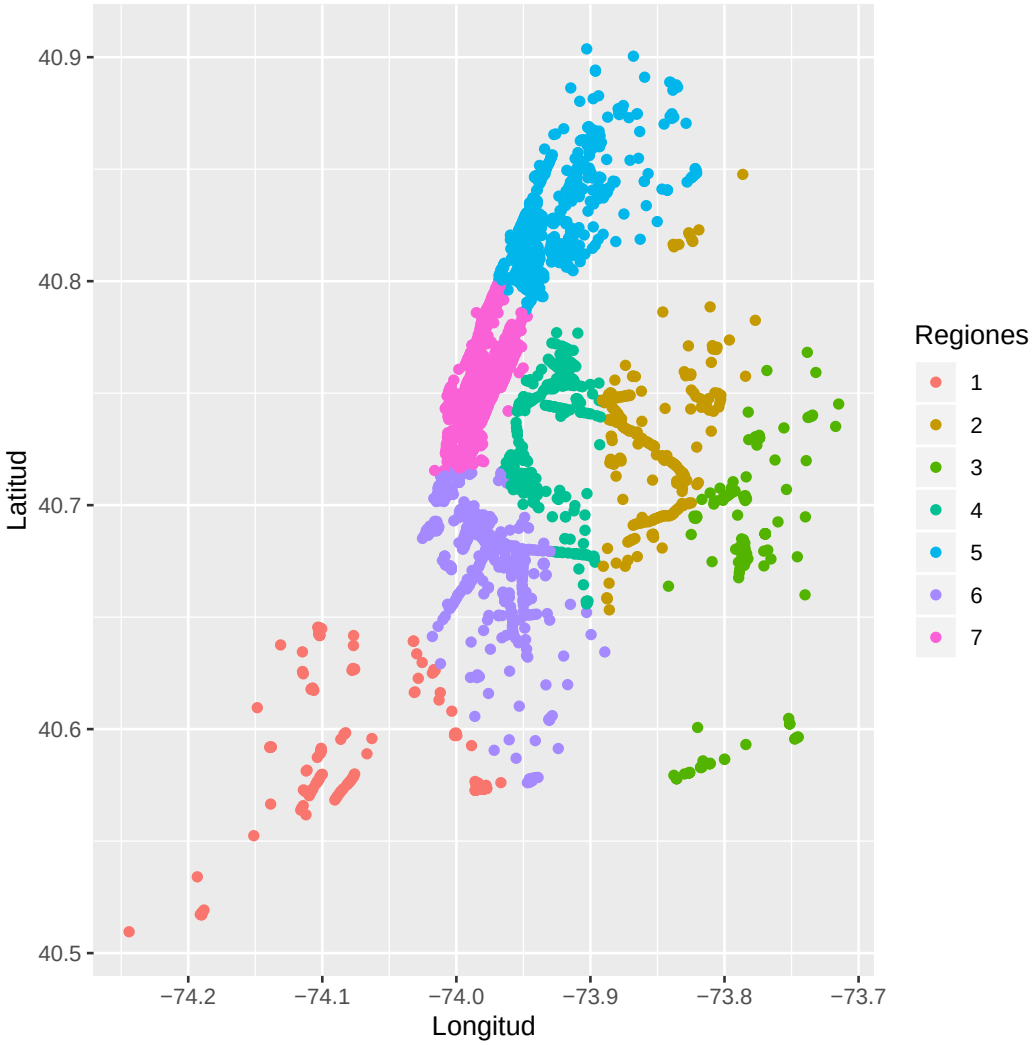
```
> if(!require(ggplot2)){
+   install.packages("ggplot2")
+   require(ggplot2)
+ }
```

Para este ejemplo, queremos **visualizar las coordenadas de latitud (LAT) y longitud (LON) del dataframe *newyork.df***:

- **Un conjunto de datos**: en nuestro caso el *dataframe newyork.df*
- **Un sistema de coordenadas**: utilizaremos el eje X para la columna de longitud y el eje Y para las coordenadas de latitud. Para indicar los ejes, *ggplot* dispone de un campo denominado *aes* en el que podremos asociar columnas con cada uno de los ejes.
- **Un conjunto de herramientas para representar visualmente los datos**: en este caso, utilizaremos **puntos** para representar cada uno de los datos. *ggplot* dispone del campo *geom_point* con el que podremos representar cada fila del *dataframe* original como un punto en el espacio.

Como queremos representar cada fila en función del conjunto al que pertenece, añadimos el campo *color* en el que representará cada fila con un color **en función del *cluster* al que pertenezca**:

```
> # Datos a representar: newyork.df
> # Coordenadas:
> # -Eje X ==> LON
> # -Eje Y ==> LAT
> # Representacion de los datos: por puntos
> ggplot(newyork.df, aes(x=newyork.LON, y=newyork.LAT)) +
+ geom_point(aes(color = newyork.df$clasificacion.ns.cluster)) +
+ scale_color_discrete(name = "Regiones") + labs(x = "Longitud", y = "Latitud")
```

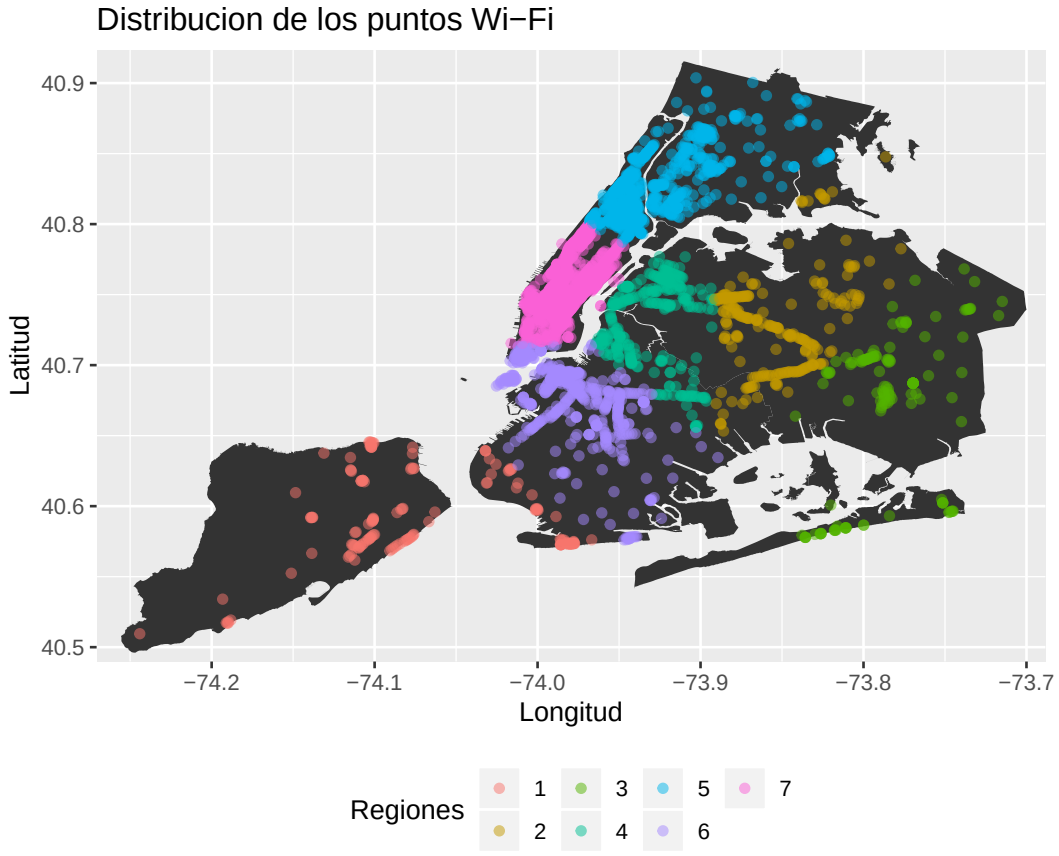



Sin embargo, una mejor representación sería con el mapa de la ciudad de Nueva York. Para ello *GitHub* [3] dispone de una librería de la comunidad que permite representar gráficamente el mapa de la ciudad de Nueva York, utilizando el paquete *maps* [4] de R, que permite proyectar diferentes regiones del globo. Para proyectar el mapa de la ciudad, debemos instalar dicho paquete. Una vez descargada, la añadimos y mediante la función *map_data* seleccionamos

[3] https://github.com/zachcp/nycmaps
[4] https://cran.r-project.org/web/packages/maps/maps.pdf

de la base de datos geográfica la región de Nueva York *nyc*. Finalmente, mediante la función *ggplot* representamos gráficamente los *clusters* sobre el mapa de la región (para ello, *ggplot* dispone de la función *geom_map* que permite proyectar un conjunto de datos sobre un mapa):

```
> # Instalamos el paquete
> if(!require(nycmaps)){
+   install.packages("C:/tmp/nycmaps.zip")
+   # Lo importamos
+   library(nycmaps)
+ }
> # Importamos la plantilla de la ciudad de
> # Nueva York
> nyc <- map_data("nyc")
> ## ggtitle para incluir un titulo al grafico
> ## theme para situar la leyenda en la parte inferior
> ggplot() + geom_map(data=nyc, map=nyc, aes(map_id=region)) +
+ geom_point(data = newyork.df, aes(x = newyork.LON,
+ y = newyork.LAT, colour = newyork.df$clasificacion.ns.cluster),
+ alpha = .5) + scale_fill_continuous(guide = guide_legend()) +
+ ggtitle("Distribucion de los puntos Wi-Fi") +
+ theme(legend.position = "bottom") +
+ scale_color_discrete(name = "Regiones") +
+ labs(x = "Longitud", y = "Latitud")
```



El mapa anterior nos muestra la distribución de los puntos *Wi-Fi* situados a lo largo de la ciudad. Por ejemplo, vemos como existen regiones como la 7 (**Manhattan**), la 5 (**Bronx**), donde existe un mayor número de puntos de acceso *Wi-Fi* gratuitos, debido a que se trata de zonas con de mayor actividad, mientras que hay regiones como la 1 (**Staten Island**) donde existe un menor número de puntos de acceso.

### 2.2.2. Hierarchical Clustering

Una de las principales diferencias del agrupamiento jerárquico (*Hierarchical Clustering*) con respecto a *K-Means* es que no se agrupan los datos en torno a un determinado número de centroides, sino que incialmente cada elemento es considerado como un **centroide**. Posteriormente, se identifica aquella pareja de centroides con la menor distancia, combinándolos y formando un único *cluster*, repitiendo este proceso hasta que todos los datos estén contenidos dentro de un mismo *cluster*.
Para el agrupamiento jerárquico, `R` dispone de la función *hclust*, disponible en el paquete *stats*. [5] Según la definición que utilicemos para medir la proximidad entre *clusters*, existen distintos tipos de clasificación jerárquica:

- **MIN o *Single***: define la proximidad entre dos clusters como **la distancia existente entre los dos puntos más cercanos de los dos *clusters***, generando *clusters* contiguos en los que cada punto está más cerca al menos a un punto en su *cluster* que a cualquier otro punto en otro *cluster*
- **MAX o *Complete***: define la proximidad entre dos *clusters* como la **distancia que hay entre los dos puntos más lejanos de los dos *clusters***
- **Group Average**: define la proximidad entre dos *clusters* como la **media de distancias entre todas las parejas que se puedan formar, con puntos de los dos *clusters***.

Para indicar el tipo de algoritmo de clasificación, *hclust* dispone de un campo denominado *method* con el cual podremos modificar el tipo de algoritmo. Por defecto, utiliza el método *complete*, es decir, el algoritmo **MAX**. Sin embargo, vamos a realizar un pequeño estudio comparativo de los tres algoritmos anteriores.
En primer lugar, eliminamos del dataframe *newyork.df* la columna *clasificacion.ns.cluster*, utilizada previamente para el *K-Means*:

```
> newyork.df.2 <- newyork.df[,-3]
```

Para ejecutar la función *hclust* debemos pasar como parámetro las **distancias entre todos los puntos del *dataframe***. Para ello utilizaremos la función *dist*, la cual nos devuelve una matriz con las distancias euclídeas, por defecto:

```
> # Ejecutamos el algoritmo de agrupacion jerarquica
> clustering <- hclust(dist(newyork.df.2))
> summary(clustering)

            Length Class  Mode
merge       6636   -none- numeric
height      3318   -none- numeric
order       3319   -none- numeric
labels         0   -none- NULL
method         1   -none- character
call           2   -none- call
dist.method    1   -none- character
```

[5] https://www.rdocumentation.org/packages/stats/versions/3.6.1/topics/hclust

De los valores resultantes, debemos destacar el campo *merge*:

```
> head(clustering$merge,10)
```

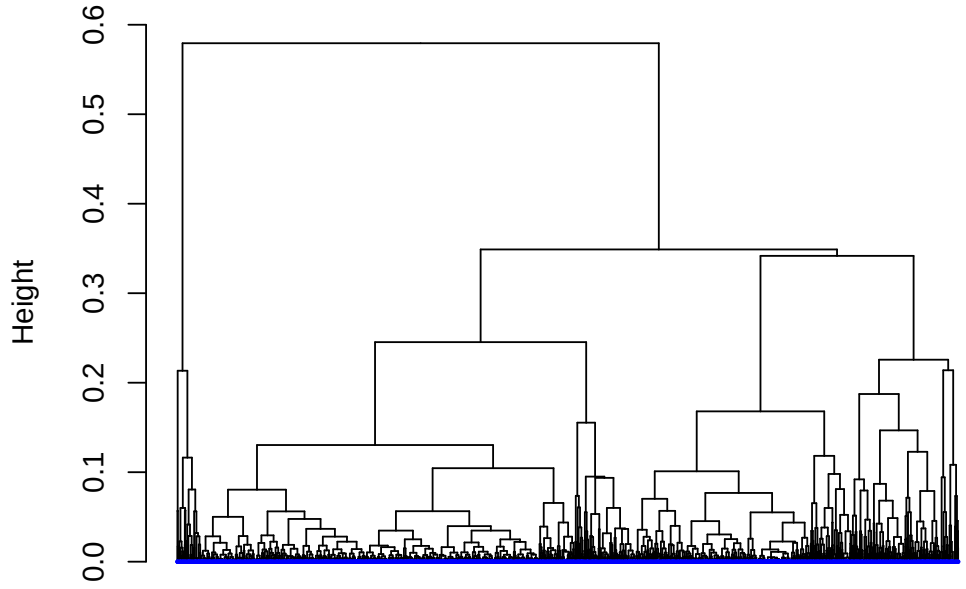
```
      [,1] [,2]
 [1,]  -57  -67
 [2,]  -69    1
 [3,]  -58  -70
 [4,]  -99 -100
 [5,] -101    4
 [6,] -102    5
 [7,] -103    6
 [8,] -605    7
 [9,] -606    8
[10,] -108 -109
```

Tras ejecutar el algoritmo, *hclust* nos devuelve, entre otros campos, una matriz denominada *merge*, la cual describe el **proceso de clusterización de los elementos**: cada fila $i$ describe el proceso de *clusterización* durante la iteración $i$ del algoritmo. Por cada fila, si existe un elemento $j$ negativo, significa que dicho valor fue *clusterizado* durante la iteración $j$ del algoritmo, mientras que si el elemento $j$ es positivo, implica que fue *clusterizado* en la fase previa.
Una forma de poder observar el proceso de agrupación sería ejecutando *plot* sobre la función *hclust*, el cual nos muestra un árbol cuyos nodos hoja son cada una de las fials del *dataframe* original. Para una mejor representación del árbol, vamos a sustituir cada pareja de valores **latitud-longitud** por una lista de puntos:

```
> # cex: indica las dimensiones de cada punto (0.2 significa que
> # debe imprimirse un 20% mas pequeño con respecto al tamaño original)
>
> # pch: indica el tipo de figura (19: punto)
> # color: color de cada punto
> listaPuntos <- list(lab.cex = 0.2, pch = c(NA,19), cex = 0.2, col = "blue")
```

Una vez definida la lista de puntos, ejecutamos la función *plot*. Es muy importante redefinir la variable *clustering* al tipo de dato **dendrograma** (*dendrogram*), un diagrama en forma de árbol que organiza los datos por subcategorías que, a su vez, se dividen en otros hasta llegar a un nodo raíz:

```
> plot(as.dendrogram(clustering), ylab = "Height", nodePar = listaPuntos, leaflab = "none")
```




Finalmente, para realizar la representación gráfica mediante *ggplot* necesitamos podar el árbol anterior. Para ello, disponemos de la función *cutree* con la que podremos realizar la poda del árbol, devolviendo una matriz en la que divide los datos en torno a $k$ centroides. En nuestro caso utilizaremos el número óptimo de centroides: 7.

```
> # Podamos el arbol, agrupando los datos
> # en torno a 7 centroides
> clusters <- cutree(clustering, k = 7)
> # Mostramos los 100 primeros elementos
> head(clusters, 100)

  [1] 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 3 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 3 3 3 3 3 3
```

Como podemos observar, cada dato queda asociado con un *cluster*. A continuación, vamos a realizar una comparación del número de elementos agrupados en torno a cada *cluster* entre el *K-Means* y agrupación jerárquica con el algoritmo **MAX**:

```
> for(i in 1:7){
+    cat("Numero de elementos en el centroide ",i," para K-Means: ", sum(newyork.df$clasificacion.ns.cluster == i), "\n")
+    cat("Numero de elementos en el centroide ",i," para agrupación jerárquica con MAX: ", sum(clusters == i), "\n\n")
+ }

Numero de elementos en el centroide  1  para K-Means:  157
Numero de elementos en el centroide  1  para agrupación jerárquica con MAX:  381
```

```
Numero de elementos en el centroide  2  para K-Means:  244
Numero de elementos en el centroide  2  para agrupación jerárquica con MAX:  914

Numero de elementos en el centroide  3  para K-Means:  148
Numero de elementos en el centroide  3  para agrupación jerárquica con MAX:  1589

Numero de elementos en el centroide  4  para K-Means:  298
Numero de elementos en el centroide  4  para agrupación jerárquica con MAX:  100

Numero de elementos en el centroide  5  para K-Means:  761
Numero de elementos en el centroide  5  para agrupación jerárquica con MAX:  260

Numero de elementos en el centroide  6  para K-Means:  605
Numero de elementos en el centroide  6  para agrupación jerárquica con MAX:  38

Numero de elementos en el centroide  7  para K-Means:  1106
Numero de elementos en el centroide  7  para agrupación jerárquica con MAX:  37
```

Como podemos observar, mientras que en el *K-Means* los elementos se concentran en torno a los últimos centroides, la agrupación jerárquica con **MAX** concentra los elementos en torno a los primeros centroides.
Como existen varios algoritmos de clasificación jerárquica, vamos a realizar un estudio comparativo del número de elementos agrupados en torno a cada cluster utilizando los siguientes algoritmos:

- *K-Means*
- Agrupación jerárquica con **MIN**
- Agrupación jerárquica con **MAX**
- Agrupación jerárquica con *average*

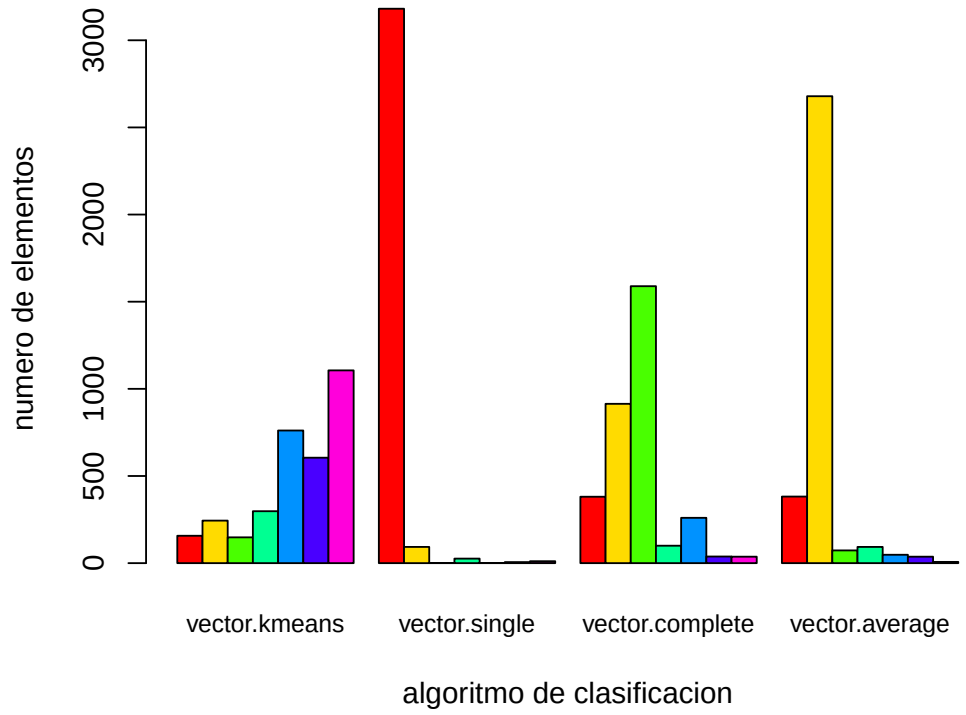
```
> # Paso 1: Aplicamos agrupacion jerarquica
> # para cada algoritmo: MAX, MIN, avg
> clustering.single <- hclust(dist(newyork.df.2), method = "single")
> clustering.complete <- hclust(dist(newyork.df.2))
> clustering.average <- hclust(dist(newyork.df.2), method = "average")
> # Paso 2: Una vez aplicado el algoritmo, realizamos
> # el proceso de poda con 7 centroides con cutree()
> clusters.single <- cutree(clustering.single, k = 7)
> clusters.complete <- cutree(clustering.complete, k = 7)
> clusters.average <- cutree(clustering.average, k = 7)
> # Paso 3: almacenamos el numero de elementos en cada cluster
> # para cada algoritmo
> vector.kmeans <- c()
> vector.single <- c()
> vector.complete <- c()
> vector.average <- c()
```

```
> # Para ello utilizaremos un bucle for que vaya concatenando
> # en un vector el numero de elementos existentes en el cluster i
> # para cada algoritmo
> for(i in 1:7){
+    vector.kmeans <- c(vector.kmeans, sum(newyork.df$clasificacion.ns.cluster == i))
+    vector.single <- c(vector.single, sum(clusters.single == i))
+    vector.complete <- c(vector.complete, sum(clusters.complete == i))
+    vector.average <- c(vector.average, sum(clusters.average == i))
+ }
> vector.kmeans

[1]  157  244  148  298  761  605 1106

> vector.single

[1] 3181   93    1   26    1    6   11

> vector.complete

[1]  381  914 1589  100  260   38   37

> vector.average

[1]  382 2679   73   93   48   37    7
```

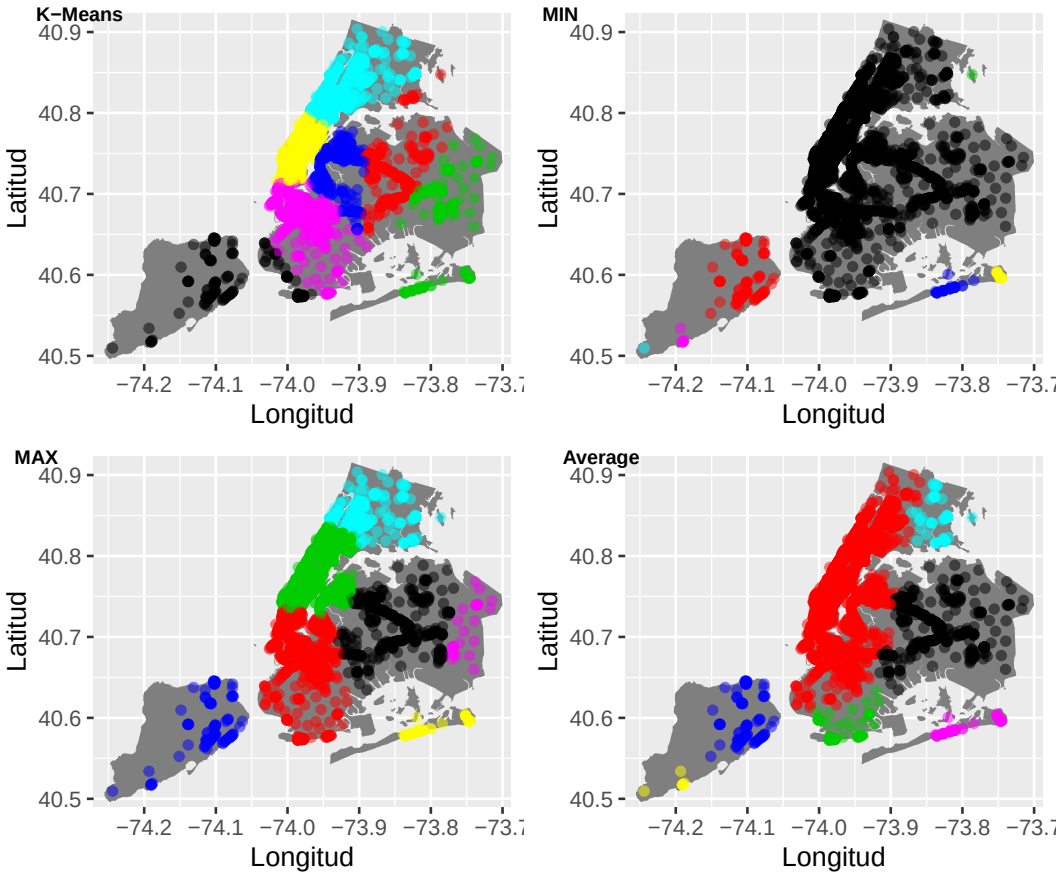
Como podemos observar, obtenemos **el número de elementos para cada cluster en función del tipo de algoritmo**. Para una mejor comparación, vamos a representar gráficamente los vectores mediante la función *barplot*, disponible en el paquete *graphics*:
Analizando la gráfica anterior, podemos extraer las siguientes conclusiones:

- **El algoritmo *K-Means* concentra un mayor porcentaje de sus elementos en torno a los *clusters* centrales y finales**.
- Por el contrario, **el método de agrupación jerárquica con el algoritmo MIN (*single*) concentra prácticamente todos los elementos en torno al primer *cluster*. Esto puede deberse a que durante las primeras etapas del proceso de *clusterización*, los elementos se han ido concentrando en torno al primer conjunto, quedando elementos finales cuya distancia entre dichos nodos es menor que la distancia al primer *cluster*, creando por ello nuevos conjuntos**.
- Por otro lado, **utilizando el algoritmo MAX (*complete*), hay una mayor dispersión de los datos con respecto al anterior algoritmo, situándose en torno a los *clusters* intermedios**.
- Por último, **el algoritmo *average* sitúa la mayoría de los datos en torno al segundo *cluster***.

```
> # Previamente unimos todos los vectores anteriores, creando un dataframe
> barplot(cbind(vector.kmeans,vector.single,vector.complete,vector.average),
+ xlab = "algoritmo de clasificacion", ylab = "numero de elementos", col = rainbow(7),
+ beside = T, cex.names = 0.8)
```




Mediante la función *ggplot*, podemos observar en el mapa la clasificación de los datos en función del algoritmo:




### 2.2.3. DBSCAN

Por último, vamos a tratar con uno de los algoritmos de clasificación más citados y utilizados: **DBSCAN**, de las siglas en inglés *Density-based spatial clustering of applications with noise*. Se trata de un algoritmo de clasificación **basado en la densidad**, es decir, se basa en detectar **áreas en las que existen una mayor concentración de puntos**, así como **áreas vacías o con escasos puntos**. Aquellos puntos que no se concentran en torno a un *cluster* será considerados **ruido**.
Supongamos que tenemos un conjunto de puntos a ser clasificados. Mediante el algoritmo **DBSCAN**, los puntos se pueden clasificar como:

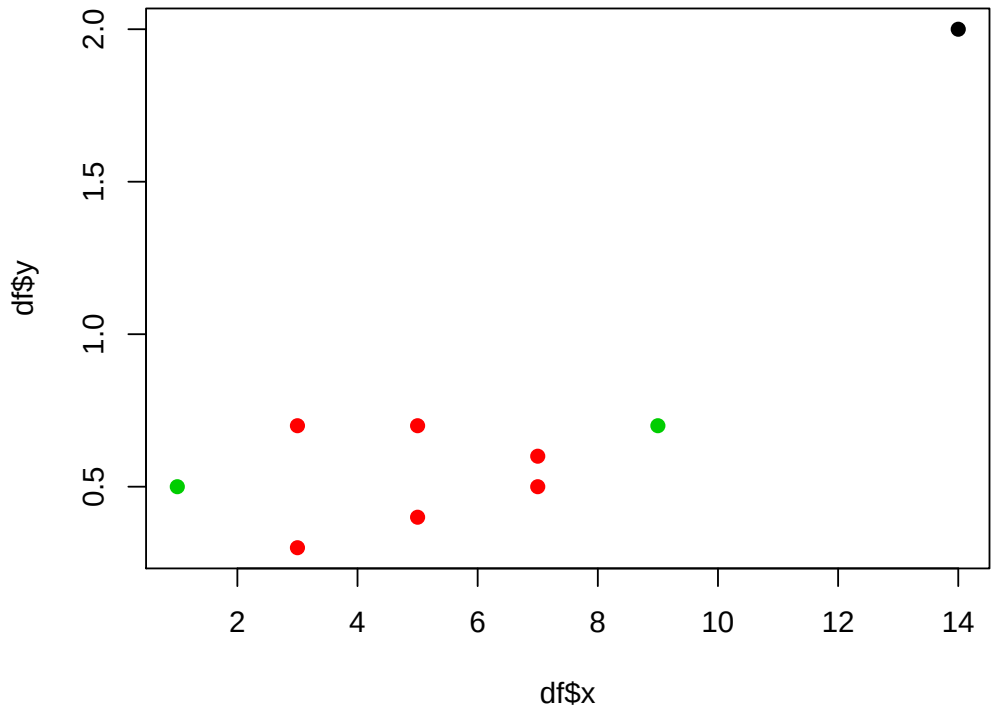
- **Puntos núcleo**
- **Puntos alcanzables**
- **Ruido**

Analicemos cada uno de los datos:

- Se dice que un punto $p$ es un **punto núcleo** si al menos existen *minPts* puntos que están a una distancia $\epsilon$ de él y dichos puntos son **directamente alcanzables** desde $p$.
- Se dice que un punto $q$ es un **punto alcanzable** desde un **punto núcleo** $p$ si existe una secuencia de puntos $p_1$,...,$p_n$, donde $p_1 = p$ y $p_n = q$, de tal modo que cada punto $p_{i+1}$ es directamente alcanzable desde $p_i$. Por tanto, **todos los puntos de la secuencia deben ser puntos núcleos, con la posible excepción de $q$**.

- Por último, cualquier punto que no sea alcanzable desde cualquier otro punto es considerado como **ruido**

Pongamos un ejemplo:




En el imagen anterior, los puntos marcados en rojo son **puntos núcleo**. Por otro lado, los puntos marcados en verde son puntos **alcanzables** desde cualquier **punto núcleo**. Por último, el punto marcado en **negro** representa **ruido**, es decir, puntos que no son núcleo ni alcanzables desde otros puntos.
Una de las características más importantes del algoritmo **DBSCAN** es que puede existir más de un punto núcleo, conformando un *cluster*.
Por tanto, un *cluster* tiene dos propiedades:

1. **Todos los puntos de dicho *cluster* son alcanzables entre sí**
2. **Si un punto A es densamente alcanzable desde otro punto B del *cluster*, A también formará parte de dicho *cluster***.

El algoritmo comienza con dos parámetros de entrada: la ***e* vecindad de cada punto**, es decir, el conjunto de nodos vecinos de cada punto; y el número **mínimo** de puntos para que una región se considere densa y, con ello, un *cluster* (*minPts*). Comenzamos con un punto aleatorio no visitado en iteraciones anteriores:

- Si la región es densa, es decir, si el número de vecinos ($e$ vecindad) es mayor o igual al mínimo de puntos establecido (*minPts*), iniciamos un *cluster* sobre dicho punto.
- En caso contrario, marcamos a ese punto como **ruido**.

Si un punto forma parte de un *cluster*, su conjunto de *e* vecinos pasan a formar parte también del *cluster*, siempre y cuando la *e* vecindad de estos puntos **sea lo suficientemente densa**, es decir, mayor o igual a *minPts*. El proceso se repite hasta **haber construido un *cluster* por completo**. De este modo, cualquier punto no visitado podremos comprobar si se trata de un nuevo *cluster* o ruido.
Tras analizar el comportamiento del algoritmo, vamos a aplicarlo para **clasificación de imágenes**, concretamente la **imagen por satélite del río Nilo** tomada por la **NASA**[6].

Para ejecutar el algoritmo de clasificación, realizamos un total de tres fases:
**En primer lugar**, aplicamos la técnica de **apilamiento de enfoque**, el cual consiste en generar a partir de una misma fotografía múltiples imágenes aplicando diferentes enfoques. Para ello disponemos de la función *stack*, disponible en el paquete *raster*.

```
> if(!require(rgdal)){
+   install.packages("rgdal")
+   require(rgdal)
+ }
> # 1. Apilamiento de enfoque
> # Instalamos la libreria
> if(!require(raster)){
+   install.packages("raster")
+   require(raster)
+ }
> # Realizamos el proceso de rasterizado
> image <- stack("nilo.jpg")
```

**A continuación**, sobre las imagenes apiladas aplicamos el algoritmo *dbscan* mediante la función *dbscan*, disponible en el paquete *dbscan*. En este caso, ejecutaremos el algoritmo con un total de 10 puntos mínimos para considerar para considerar a una región como un *cluster* (**minPts**), y una densidad del número de vecinos de 0.8 para cada punto (**eps**). Analicemos una muestra de los resultados obtenidos:

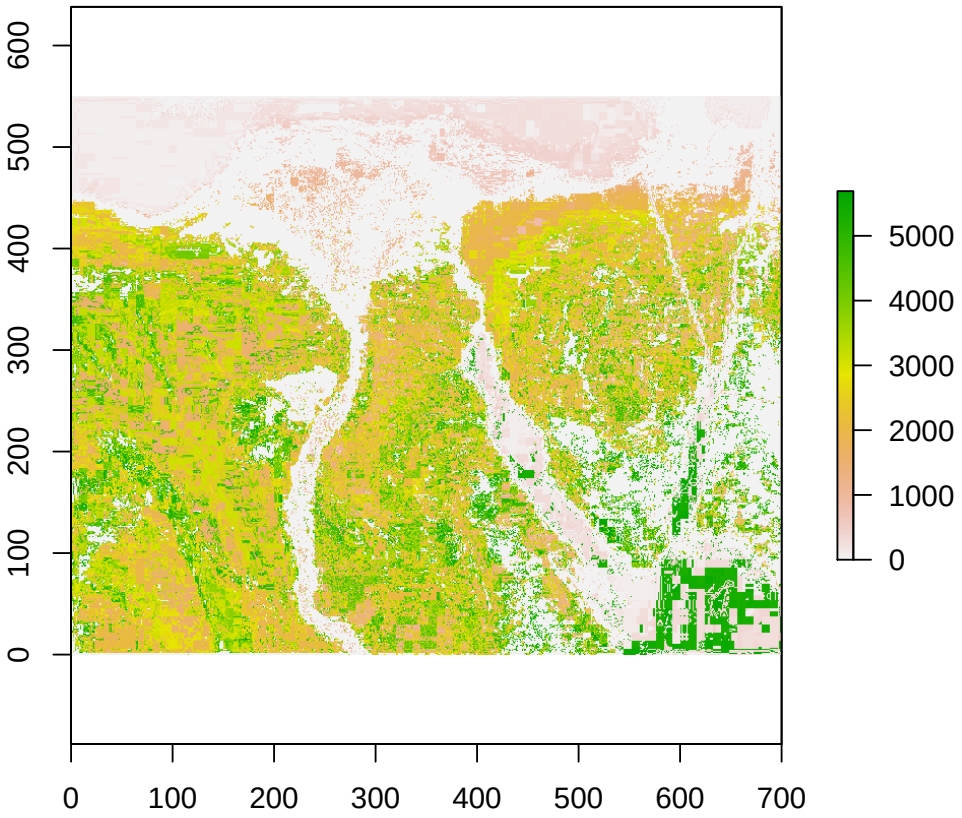
```
> # Incluimos el paquete dbscan
> if(!require(dbscan)){
+   install.packages("dbscan")
+   require(dbscan)
```

[6] https://www.jpl.nasa.gov/spaceimages/details.php?id=PIA02647

```
+ }
> db <- dbscan(image[], eps = 0.8, minPts = 10)
> head(db$cluster, 200)
  [1]  1  2  3  4  5  5  4  4  6  6  6  6  6  6  6  6  7  7  7  7  7  7  7  7  7  7  7  7  7  7  6  6  6  6  6  6  6  6
 [78] 10 10 10 11 11 11 12 12 10 10 10  0  6  0 10 10  0 13  0  0 13 14  0  0  0 13  0  0  0  0  0  0 15  0  0 16 17  0  0  0 10
[155]  0 27 27  0 27 27 12 12 12 12 12 12 12 10  0 10 28 10  0 10  0  0 24  0 10 12  0 24 29 29  0  0  0  0 29 29 30  3  3  3
```

Tras ejecutar la función, *dbscan* nos devuelve un *dataframe* con tres campos: **eps**, **minPts** y **cluster**. Este último muestra a qué *cluster* pertenece cada elemento, salvo las filas a 0, los cuales se tratan de **ruido**.
Para representar los resultados por pantalla *convertimos la imagen original en un conjunto de píxeles*, conocido como **rasterizado**, utilizando la función *raster*. Finalmente, asignamos cada *cluster* al que pertenece cada punto a cada píxel de la imagen, obteniendo la siguiente imagen:

```
> # Proceso de rasterizado
> result <- raster(image[[1]])
> # Asignamos el valor del cluster a cada pixel
> result <- setValues(result, db$cluster)
> plot(result)
```




Analizando los resultados obtenidos, el algoritmo **separa correctamente las áreas desérticas con respecto a las zonas con vegetación**. Sin embargo, **no distingue correctamente entre las zonas de vegetación y el mar**. Esto último puede deberse a que el número de puntos mínimos sea pequeño, por lo que aumentándolo podríamos llegar a distinguir ligeramente zonas costeras.