Clustering

José Tomás Palma Méndez

Dept. de Ingeniería de la Información y las Comunicaciones. Universidad de Murcia
Contacting author: jtpalma@um.es

1. Introducción

En esta práctica con R vamos diferentes librerías que implementan las diferentes técnicas de clustering que hemos visto en las clases de teoría.

2. El conjunto de datos

Los datos sobre los que vamos a trabajar vamos a utilizar el conjunto de datos `mtcars`.

Lo primero que tenemos que hacer es cargar los datos:

```
> data("mtcars")
```

> Ejercicio 1.
>
> 1.a) Examina el conjunto de datos y determina el número y tipo de los atributos, y el número de instancias.
> 1.b) Según hemos visto en la prácticas anteriores ¿Es un conjunto de datos técnicamente correcto?

3. Clustering jerárquico

Recordemos que el clustering jerárquico trabajaba en base a una medida de distancia entre clusters. Podíamos tener tanto clustering divisivo como aglomerativo. En ambos se tomaba la decisión de dividir un cluster en dos nuevos hijos o de fusionar dos clusters en un cluster padre, respectivamente, en base a dichas distancias. Para trabajar con clustering jerárquico aglomerativo, en R tenemos la función `hclust()`. Como sabemos, el clustering de este tipo va a necesitar una matriz de distancias, con lo que hay que calcular primeramente la distancia. Para eso tenemos la función `dist()`. Una vez tenemos la matriz de distancias, podemos realizar el clustering jerárquico. Su resultado podrá visualizarse en forma de dendrograma.

El primer paso que hay que realizar es escalar los datos, dividir los valores de cada columna por su desviación estándar. Esta operación la podemos hacer con la función `preProcess()` del paquete `caret`:

```
> mtcars.pre <- preProcess(mtcars,method="scale")
> mtcars.scaled <- predict(mtcars.pre, mtcars)
```

Ejercicio 2. Comprueba el resultado de la operación anterior de forma gráfica. Utiliza los gráficos de dispersión y de cajas, para comparar los resultados antes y después del escalado.

Para realizar un clustering aglomerativo, una posible secuencia de comandos es (cuyo resultado se puede ver en la figura 1):

```
> mtcars.dist <- dist(mtcars.scaled,method="euclidian")
> mtcars.hclust.average <- hclust(mtcars.dist, method="average")
> plot(mtcars.hclust.average,hang=-1)
```

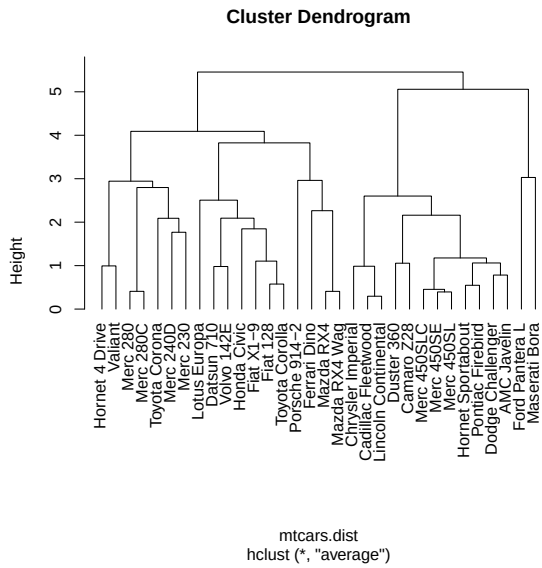

Figura 1. Dendrograma obtenido mediante el método `average`.

Consejo 1. Es recomendable utilizar la función `hclust()` ofrecida por el paquete `fastcluster`, ya que ofrece una implementación más eficiente de la misma.

Ejercicio 3. Como hemos visto en clase, existen varias formas de realizar el clustering aglomerativo en función de cómo se defina la distancia entre cluster.

3.a) Cambia el parámetro `method` de la función `hclust()` y obtén los distintos dendrogramas. Compara los resultados.
3.b) Prueba a cambiar también la forma de calcular la distancia y comprueba como afecta al resultado del clustering.
3.c) Selecciona alguna de las combinaciones anteriores e interpreta los resultados obtenidos.
3.d) ¿Cómo podríamos realizar un clustering jerárquicos por atributos? ¿Qué información podría relevar este tipo de clustering?

De la misma forma que hemos utilizado la distancia euclídea (u otro tipo de función de distancia disponible) para calcular la matriz de distancias para nuestro algoritmo, también podemos usar el coeficiente de correlación de Pearson (recomendable en aplicaciones relacionadas con niveles de expresión genética). Recordemos que este coeficiente mide la relación lineal que hay entre los atributos, en este caso dos a dos. El siguiente código nos permite realizar el análisis utilizando el coeficiente de correlación de Pearson:

```
> cor.pe <- cor(t(as.matrix(mtcars.scaled)),method=c("pearson"))
> dist.pe <- as.dist(1-cor.pe)
> mtcars.pearson <- hclust(dist.pe,method="average")
> plot(mtcars.pearson ,main="Clustering jerárquico Pearson",hang=-1)
```

Como ya hicimos anteriormente, las gráficas correspondientes se pueden obtener a través de la función `plot()` (ver figura 2):

Otra forma de visualizar los resultados del clustering jerárquico es a través de una gráfico de mapa térmico, **Heat Map**. Esto lo podemos hacer con la función `heatmap()`. En nuestro caso podemos visualizar los valores de los atributos con falso color, junto el dendrograma utilizando el coeficiente de correlación de Pearson con la siguiente instrucción (ver figura 3):

```
> heatmap(as.matrix(mtcars.scaled),
          Rowv=as.dendrogram(mtcars.pearson))
```

En este caso, hemos tenido que convertir el `data.frame` `mtcars.scaled` en una matriz numérica, ya que es el tipo de datos que espera la función `heatmap`. Además, debemos indicar qué dendrogramas queremos dibujar en cada dimensión. Dichos dendrogramas, se obtienen a partir de los objetos obtenidos al ejecutar la función `hclust` mediante la función `as.dendrogram`. El resultado de la anterior instrucción se puede ver en la figura 3. Como podemos apreciar, al no indicar que dendrograma se tiene que visualizar para los atributos, la función `heatmap` realiza el clustering jerárquico de forma automática (modificando ciertos parámetros podemos llamar a la función `hclust` e indicar qué tipo de clustering hay que hacer).

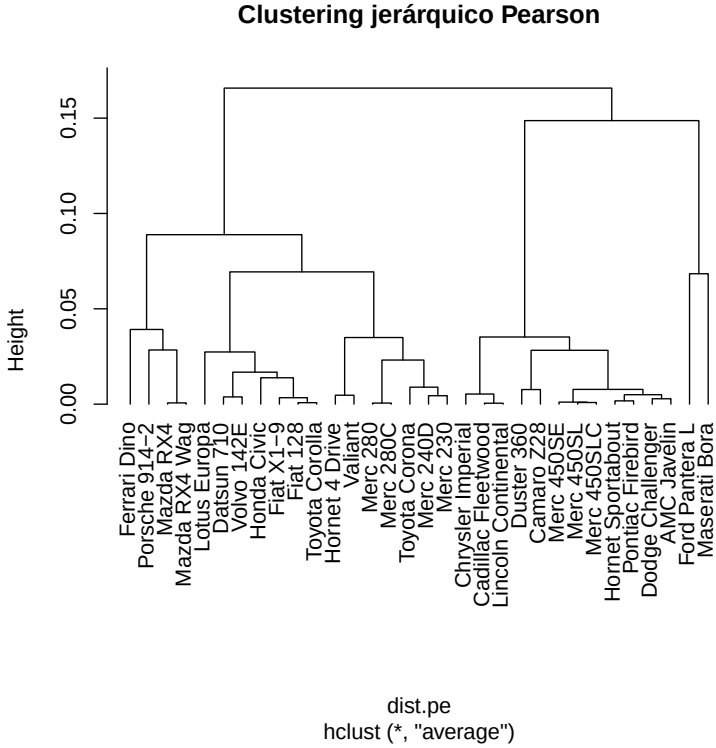

Figura 2. Dendrograma utilizando el coeficiente de correlación de Pearson.

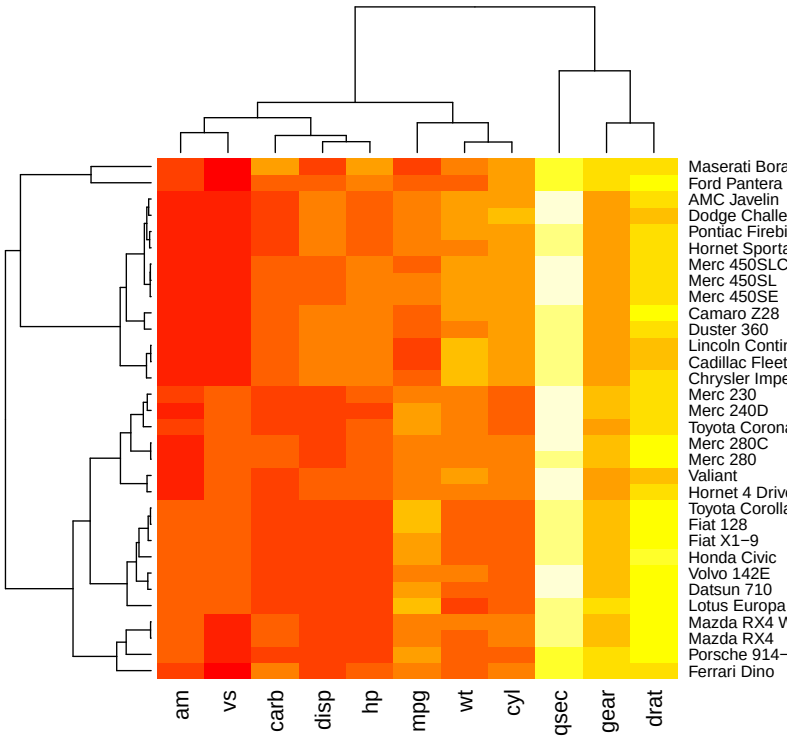

Figura 3. Resultado del clustering en cada dimensión mostrado junto a la visualización de los datos.

Una vez obtenido el dendrograma correspondiente, podemos utilizar la función `cutree()` para cortar un dendrograma a una determinada altura o en un número concreto de clusters. Por ejemplo, si el dendrograma obtenido a partir del índice de correlación para las filas lo queremos cortar en la altura en la que se generan 5 clusters deberemos escribir:

```
> mtcars.hclust.5 <- cutree(mtcars.pearson,k=5)
> mtcars.hclust.5

          Mazda RX4       Mazda RX4 Wag
                  1                   1
         Datsun 710      Hornet 4 Drive
                  2                   3
  Hornet Sportabout             Valiant
                  4                   3
         Duster 360           Merc 240D
                  4                   3
           Merc 230            Merc 280
                  3                   3
          Merc 280C          Merc 450SE
                  3                   4
         Merc 450SL         Merc 450SLC
                  4                   4
 Cadillac Fleetwood Lincoln Continental
                  4                   4
  Chrysler Imperial            Fiat 128
                  4                   2
       Honda Civic      Toyota Corolla
                  2                   2
     Toyota Corona    Dodge Challenger
                  3                   4
       AMC Javelin          Camaro Z28
                  4                   4
   Pontiac Firebird           Fiat X1-9
                  4                   2
      Porsche 914-2        Lotus Europa
                  1                   2
     Ford Pantera L        Ferrari Dino
                  5                   1
      Maserati Bora          Volvo 142E
                  5                   2
```

También podemos realizar simultáneamente el corte para un conjunto determinado de grupos o alturas. Sólo bastaría que en vez de un único valor, utilicemos un array con las alturas o grupos deseados.

Llegados a este punto es importante comentar la utilidad del paquete `pvclust`. Dicho paquete permite evaluar un clustering jerárquico a través del cálculo de los *valores-P* de los distintos clusters. El *valor-P* de un cluster, es un número entre 0 y

1 que indica el grado en el que el cluster está soportado por los datos. En [1] podéis analizar en más detalle cómo se calcula este valor.

Siguiendo con nuestro ejemplo, podemos ver qué clusters están mejor soportados por los datos de la siguiente manera (ver Figura 4), utilizando la distancia euclídea y el método de Ward):

```
> library(pvclust)
> hc <- pvclust(t(mtcars.scaled), method.hclust="single",
                method.dist="euclidean")
> plot(hc)
> pvrect(hc, alpha=.95)
```

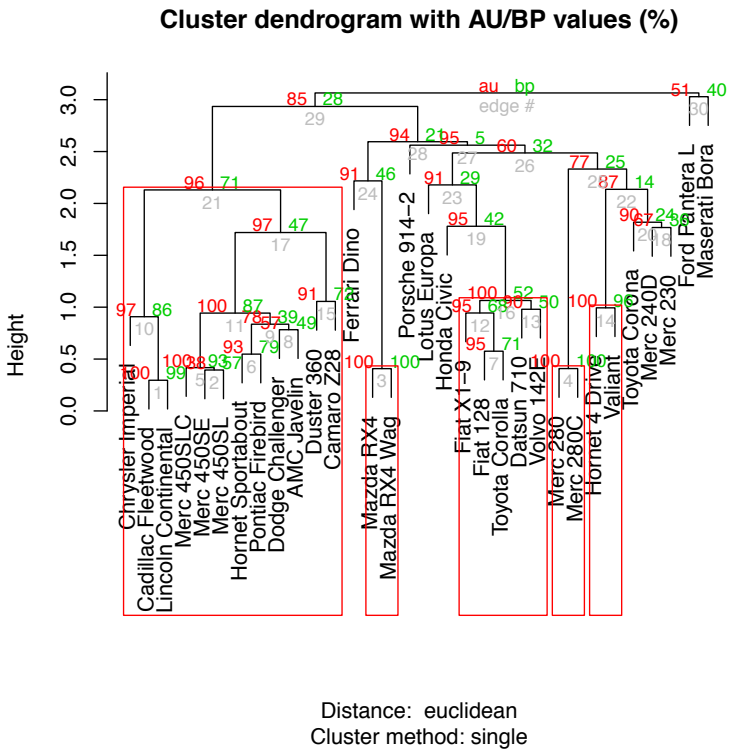

Figura 4. Cálculo del *p-value* asociado a cada cluster.

La función `pvrect()` permiten resaltar aquellos cluster que sobrepasan (o están por debajo) del umbral establecido con el parámetro `alpha`. La función `pvpick()` devuelve dichos clusters.

Ejercicio 4.

4.a) Cambia el parámetro `method` de la función `hclust()` y obtén los distintos dendrogramas utilizando como medida de distancia el índice de correlación de Pearson. Compara los resultados.
4.b) Genera los gráficos correspondientes a los distintos dendrogramas junto a sus heatmaps.
4.c) Prueba a generar distintos dendrogramas a través de la función `pvclust` modificando las funciones de distancia y resalta aquellos clusters con un *p-value* mayor.
4.d) Elige, a tu criterio, la mejor partición e intenta interpretar los resultados.

También podemos realizar clustering jerárquico a través del paquete `cluster`. Este paquete nos ofrece dos funciones para realizar clustering jerárquico a través de las funciones `agnes()` y `diana()`. La función `diana()`, nos permite realizar un clustering jerárquico divisivo y, a través del parámetro `metric`, podemos indicarle diferentes funciones de distancia. Con las siguientes instrucciones podemos realizar el cluster jerárquico divisivo:

```
> library(cluster)
> mtcars.diana <- diana(mtcars.scaled, metric="euclidean")
```

Para mostrar el dendrograma correspondiente podemos utilizar la función `pltree()` (ver Figura 5).

```
> pltree(mtcars.diana)
```

También podemos visualizar un gráfico de barras (banner plot). Con este tipo de gráfico se puede ver las distancias a las que se fusionan las instancias y clusters. Este tipo de gráfico se generan a través de la función `bannerplot()` (ver Figura 6). La función `plot()` aplicada al resultado de la función `agnes()` o `diana()` nos generará las dos gráficas.

```
> bannerplot(mtcars.diana)
```

Ejercicio 5.

5.a) Compara el clustering jerárquico divisivo con distintos clustering aglomerativos utilizando las medidas de distancia entre clusters vistas en clase.
5.b) Muestra los dendrogramas y los gráficos de barras correspondientes a un clustering jerárquico aglomerativo y el divisivo.

5.c) Interpreta los resultados sobre el que, a tu criterio, ofrece la mejor agrupación.

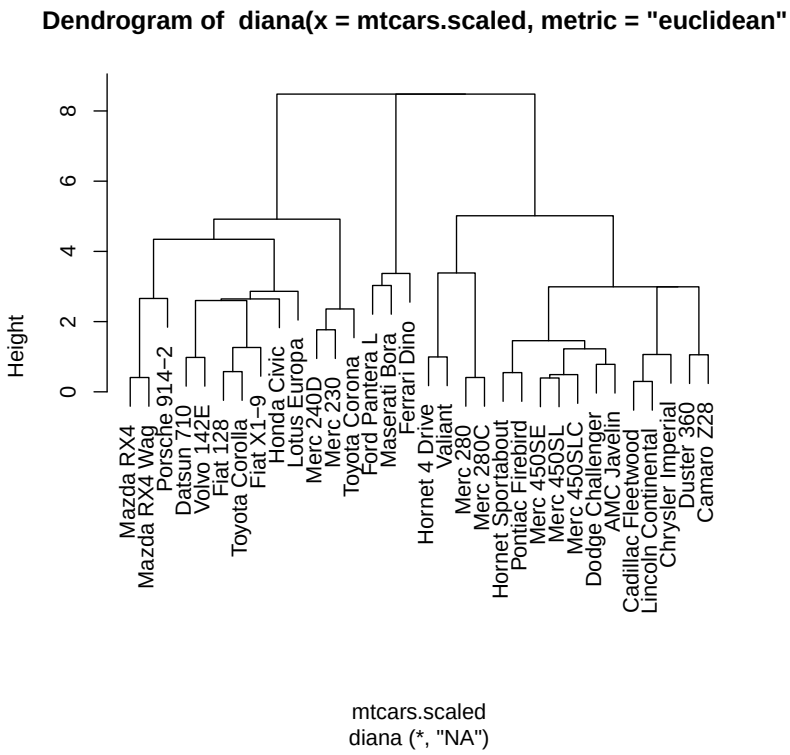

Figura 5. Dendrograma correspondiente al clustering jerárquico divisivo.

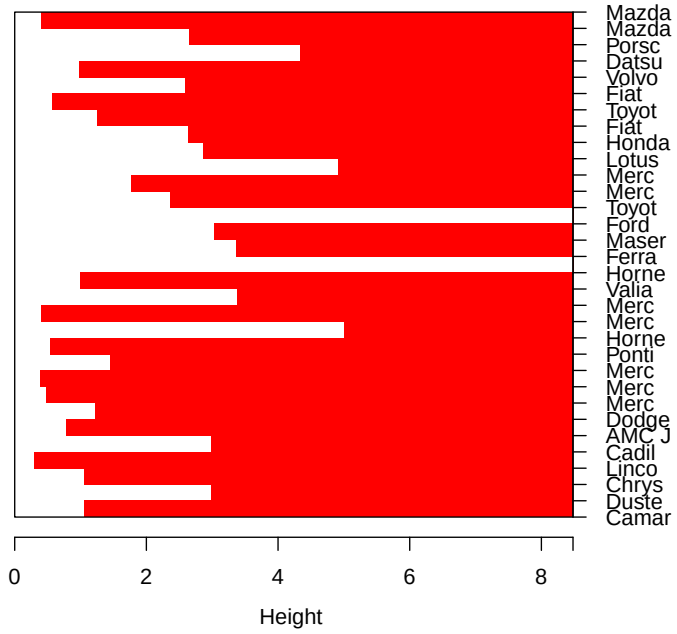

Figura 6. Gráfico de barras correspondiente al clustering jerárquico divisivo.

Para determinar el número apropiado de clusters podemos utilizar el *índice silueta* (silhoutte). Este índice nos mide la validez y consistencia de una partición. Este índice le asigna un valor entre -1 y 1 a cada elemento. Un valor cercano a -1 indica que posiblemente el elemento debe ser asignado a un cluster vecino. Un valor cercano a 1 indicaría que el elemento está bien agrupado y un valor cercano a 0 indica que el elemento está situado en la frontera de dos clusters. Para calcular este índice tenemos la función `silhoutte()`, que para el caso de clustering jerárquico se invoca de la siguiente manera:

```
> diana.sil <- silhouette(cutree(mtcars.diana,2),mtcars.dist)
> diana.sil

      cluster neighbor    sil_width
 [1,]       1        2  0.149743334
 [2,]       1        2  0.148180083
 [3,]       1        2  0.539622261
 [4,]       2        1  0.007977752
 [5,]       2        1  0.487680491
 [6,]       2        1  0.064699758
 [7,]       2        1  0.520312749
 [8,]       1        2  0.361099931
 [9,]       1        2  0.314603254
[10,]       2        1 -0.085590166
[11,]       2        1 -0.079750951
[12,]       2        1  0.531420574
[13,]       2        1  0.517403107
[14,]       2        1  0.519401240
[15,]       2        1  0.509004922
[16,]       2        1  0.508444813
[17,]       2        1  0.513367905
[18,]       1        2  0.585612189
[19,]       1        2  0.538062073
[20,]       1        2  0.572439641
[21,]       1        2  0.302229299
[22,]       2        1  0.485367759
[23,]       2        1  0.479320428
[24,]       2        1  0.479528637
[25,]       2        1  0.498139334
[26,]       1        2  0.599076809
[27,]       1        2  0.417937030
[28,]       1        2  0.481596626
[29,]       2        1  0.182180714
[30,]       2        1 -0.029561109
[31,]       2        1  0.243623277
[32,]       1        2  0.500406554
attr(,"Ordered")
[1] FALSE
attr(,"call")
silhouette.default(x = cutree(mtcars.diana, 2), dist = mtcars.dist)
attr(,"class")
[1] "silhouette"
```

Mediante la función `plot()` podemos ver el gráfico de silueta que representa el objeto anteriormente generado (ver Figura 7). Para determinar cuál es el número óptimo de clusters, se debe calcular el índice silueta para diferentes niveles del clustering jerárquico y seleccionar la partición que produce el índice silueta medio más alto. Por ejemplo, para acceder al índice silueta medio para una partición para dos clusters podemos utilizar el siguiente código:

```
> diana.summ <- summary(diana.sil)
> diana.summ$si.summary[4]

  Mean
0.3707
```

Ejercicio 6. Genera un gráfico con los índices siluetas para los clusterings obtenidos en el ejercicio anterior para particiones entre 1 y 6 clusters.

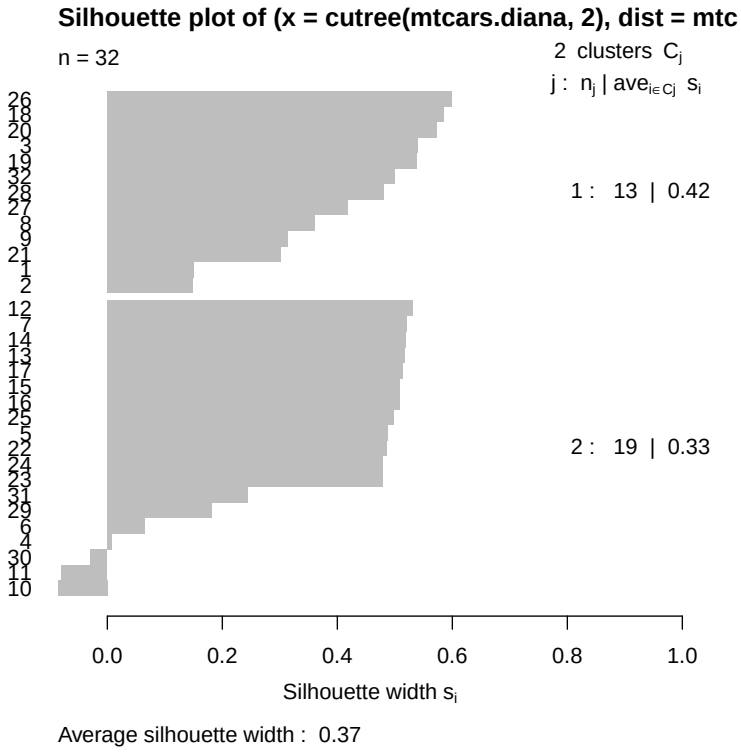

Figura 7. Gráfico de silueta correspondiente al clustering jerárquico divisivo.

4. Clustering Particional

El clustering jerárquico es interesante ya que nos permite no especificar el número de clusters de antemano. Sin embargo, algunas veces es más conveniente usar una técnica de clustering particional. Esto es debido a que los dendrogramas de conjuntos de datos de muchas instancias tienden a ser farragosos aunque pueden proporcionar pistas interesantes.

La primera técnica que hemos visto en clase es el clustering **k-means**, que tiene el inconveniente de tener que especificar el de sobra conocido parámetro `k`. Este método está accesible en `R` a través del comando `kmeans()`. Como ya hemos visto, el algoritmo **k-means** trata de minimizar la suma de las diferencias cuadráticas entre los puntos del cluster y el centroide. También vale la pena recordar que el resultado final del agrupamiento depende de la inicialización de los centroides. En este sentido, al método puede pasársele como argumento el conjunto de centroides inicial determinado de antemano o simplemente el valor **k** a través del parámetro **centers**. Si **centers** es un número en lugar de la lista de centroides, tiene sentido el llamar al algoritmo varias veces con diferentes inicializaciones aleatorias de centroides. El número de veces lo indicamos con el parámetro **nstart**.

Siguiendo con nuestro ejemplo, veamos como hacer un clustering **k-means** usando cuatro grupos, y luego vamos a representar cada individuo y su pertenencia al cluster correspondiente como código de color. La secuencia de comandos es:

```
> mtcars.kmeans <- kmeans(mtcars.scaled,centers=4,iter.max=1000)
```

Ejercicio 7. Analiza cada uno de los campos del objeto `mtcars.kmeans` e indica qué representa cada uno.

Para visualizar el resultado del clustering podemos ver cómo ha quedado la distribución de los cluster y los centroides comparando los atributos por pares. Para ello podemos utilizar la información sobre a qué cluster pertenece cada coche cómo código de color. Por ejemplo, el siguiente código nos permite ver cómo queda la distribución de los coches respecto a los dos primeros atributos, así como están distribuidos los centroides (ver figura 8).

```
> cl <-mtcars.kmeans$cluster
> plot(mtcars.scaled$mpg,mtcars.scaled$wt,col=cl,
     main=paste("Clusters creados respecto a los atributos ",
                names(mtcars.scaled[1]),
                " y",
                names(mtcars.scaled[2]),sep=""),
     xlab=names(mtcars.scaled[1]),
     ylab=names(mtcars.scaled[2]))
> points(mtcars.kmeans$centers ,col=1:5,pch=8)
```

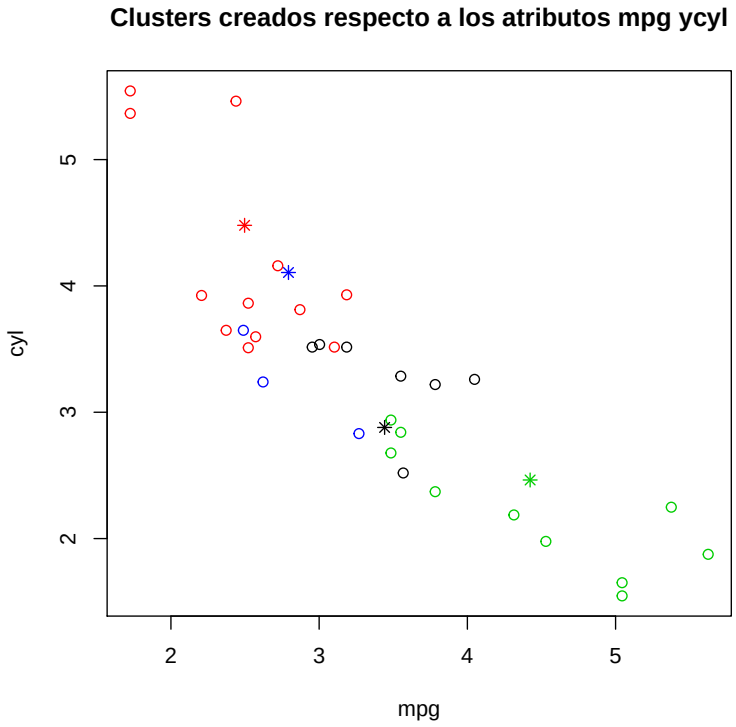

Figura 8. Resultado del clustering **k-means** respecto a los 2 primeros atributos.

También podemos visualizar la misma información pero involucrando a más atributos de la siguiente forma (ver figura 9):

```
> cl <-mtcars.kmeans$cluster
> plot(mtcars.scaled[,1:5],col=cl,
      main="Distribución de los cluster respecto los 5 primeros atributos")
> points(mtcars.kmeans$centers ,col=1:5,pch=8)
```

Otra forma de visualizar el resultado del clustering consiste en representar, para cada cluster, los valores de los atributos de cada instancia de dicho cluster. De esta forma, si el proceso de clustering ha ido bien las curvas deberían ser similares. Esto se puede hacer con la función `matplot` (ver figura 10):

```
> par(mfrow=c(2,2))
> for(i in 1:4){
       matplot(t(mtcars.scaled[mtcars.kmeans$cluster == i,]), type = "l",
       main=paste("cluster:",i),ylab="valores",xlab="atributos")
```

```
 }
> par(mfrow=c(1,1))
```

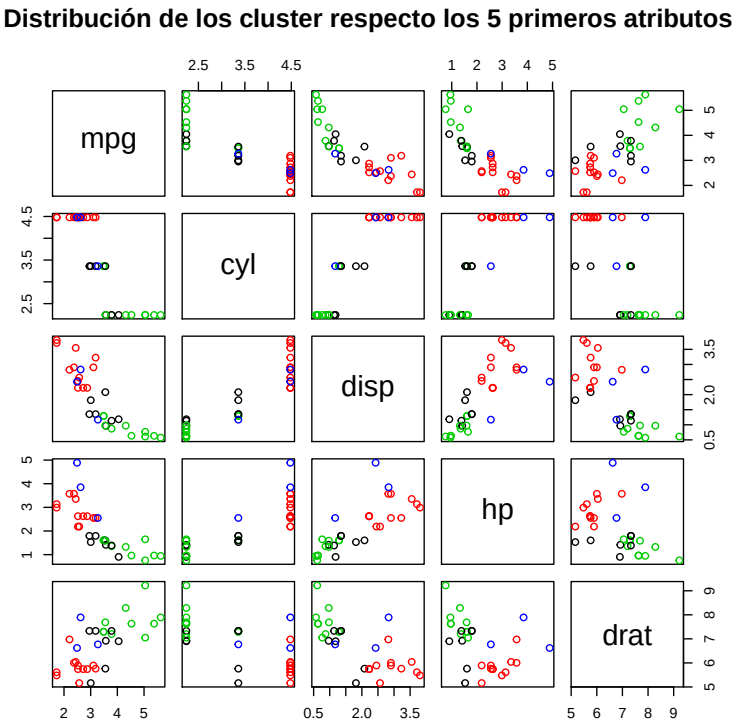

Figura 9. Resultado del clustering iterativo respecto a los 5 primeros atributos.

Ejercicio 8.

8.a) Realizar un procedimiento basado en la variable `tot.withinss` del objeto devuelto por el procedimiento `kmeans()` que muestre, de forma gráfica, los valores de dicha variable generados para 10 llamadas al algoritmo para distintos valores de k.

8.b) ¿Cuál sería el valor de k idóneo?

8.c) ¿Qué valor de k óptimo obtendríamos si lo calculásemos a través del índice silueta?

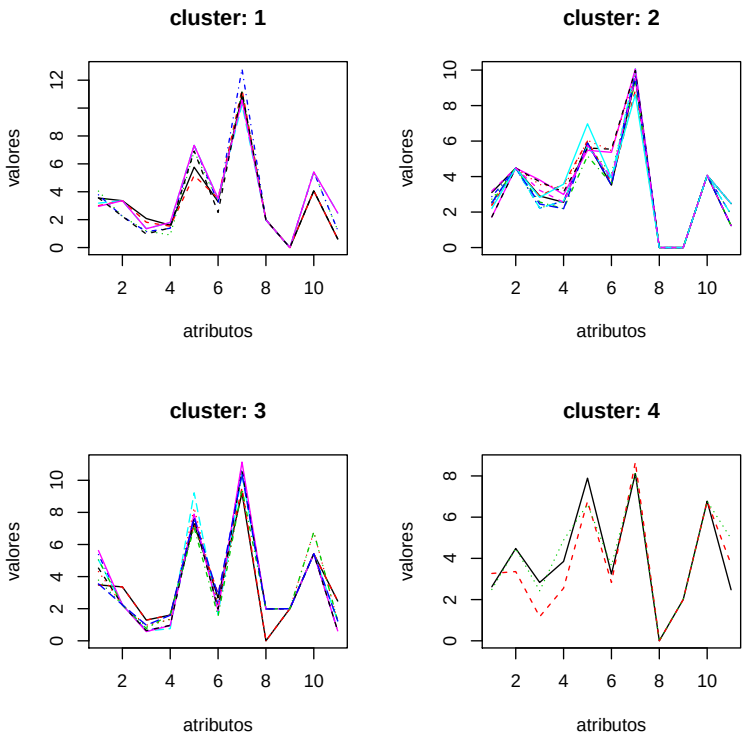

Figura 10. Valores de los atributos agrupados por clusters.

4.1. El paquete `cluster`

Como ya hemos visto, el paquete `cluster` nos ofrece funciones para realizar los dos tipos de clustering jerárquico. Este paquete también nos ofrece funciones para realizar varios tipos de clustering particional. La técnica k-medoides se implementa a través de las funciones `pam()` (partition around medoids) y `clara()` (clustering large applications). Gracias a la función `fanny()` podemos desarrollar un clustering basado en lógica borrosa. Por ejemplo, para aplicar la técnica k-medoides podemos utiliza este código:

```
> mtcars.kmedoid <- pam(mtcars.scaled,k=3)
```

Las funciones `pam()` y `fanny()` permiten ser invocadas, en vez con la matriz de datos, con una matriz con las medidas de disimilitud de las instancias.

Para estas funciones, el paquete `cluster` ofrece la función `clusplot()` que nos genera un gráfico con la distribución de los cluster en función de las dos primeras componentes principales (ver Figura 11).

```
> clusplot(mtcars.kmedoid)
```

El método DBScan lo podemos realizar a través de la función `dbscan()` del paquete del mismo nombre. Los parámetros mínimos que se deben utilizar son: `eps`, para indicar el valor de la ϵ-vecindad y `minPts` número mínimos de puntos en la ϵ-vecindad (por defecto 5). El resto de los parámetros permiten especificar pesos diferentes para cada punto, la posibilidad de que los puntos frontera sean considerados ruidos o no y modificar la forma de procesar los árboles kd. Para determinar el valor k, podemos utilziar el gráfico de k-distancias mediante la función `kNNdistplot()`. Este método de clustering puede ser aplicado utilizando las siguientes instrucciones (ver la Figura 12):

```
> library(dbscan)
> k=3
> kNNdistplot(mtcars.scaled, k)
> abline(h=2.7, col="red")
> mtcars.dbscan <- dbscan(mtcars.scaled,eps=2.7, minPts = 3)
```

Ejercicio 9.

9.a) Genera los distintos tipos de gráficas comentadas en esta sesión para el objeto `mtcars.dbscan`.

9.b) Intenta encontrar el k óptimo utilizando el índice silueta.

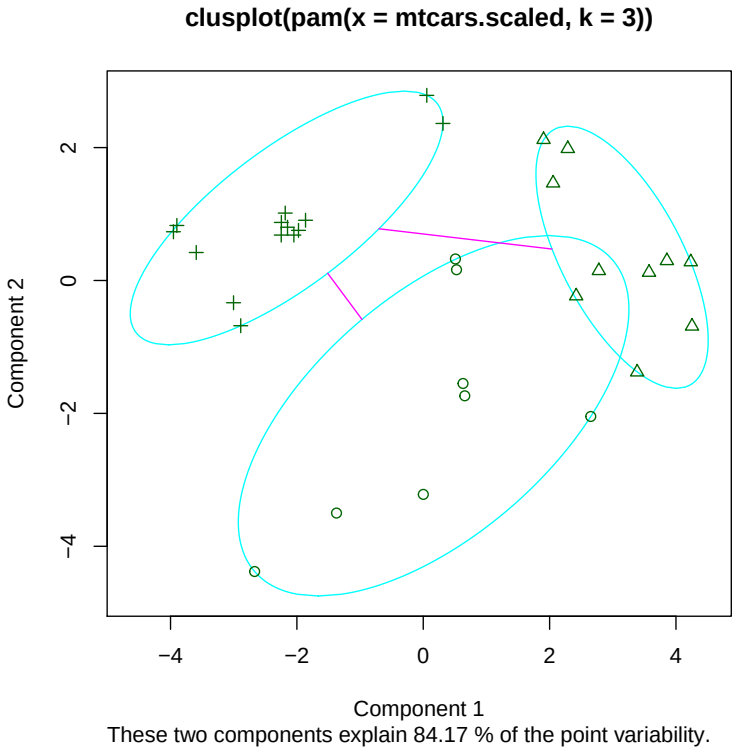

Figura 11. Distribución de cluster en función de las dos primeras componentes.

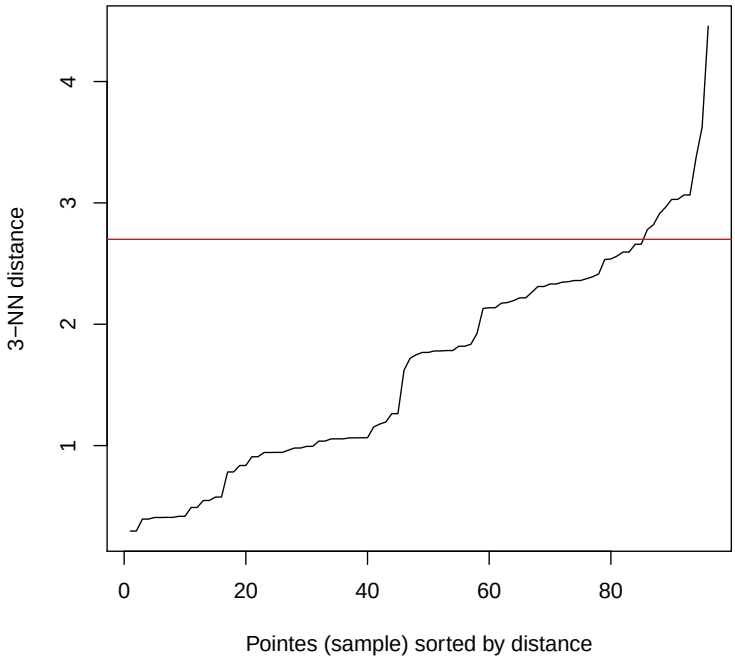

Figura 12. Gráfico de k-distancias.

4.2. Evaluación de los clusters

Una forma de evaluar la bondad del cluster, así como determinar el número óptimo de clusters es mediante el estadístico **gap** [2] a través de la función `clusGap()` del paquete `cluster`. Este estadístico tiene la ventaja de que se hace máximo cuando se evalúa sobre un agrupamiento con el número óptimo de clusters. Por ejemplo, si queremos calcular dicho estadístico para determinar el número óptimo de clusters según el algoritmo kmedias, bastaría con la siguiente instrucción:

```
> mtcars.gap.kmeans <- clusGap(mtcars.scaled, kmeans, K.max = 5)
> mtcars.gap.kmeans

Clustering Gap statistic ["clusGap"].
B=100 simulated reference sets, k = 1..5
 --> Number of clusters (method 'firstSEmax', SE.factor=1): 3
         logW   E.logW        gap     SE.sim
[1,] 3.516331 3.580034 0.06370321 0.04017785
[2,] 3.201365 3.336695 0.13533087 0.04745514
[3,] 2.975832 3.162507 0.18667446 0.04086469
[4,] 2.903370 3.037797 0.13442764 0.04198068
[5,] 2.687571 2.943522 0.25595161 0.04314046
```

El segundo parámetro es el que índica qué función de clustering particional se va a utilizar (se puede utilizar cualquiera de las vistas anteriormente). El último parámetro indica el número de clusters máximo a utilizar.

El paquete `fpc`, además de ofrecernos funciones adicionales para realizar distintos tipos de clustering, nos permite calcular medidas adicionales sobre la bondad del clustering. La función `cluster.stats()` nos calcula una multitud de estadísticos relacionados con la bondad del clustering a partir de la matriz de distancias:

```
> library(fpc)
> mtcars.kmeans.stats <- cluster.stats(mtcars.dist,mtcars.kmeans$cluster)
> mtcars.kmeans.stats

$n
[1] 32

$cluster.number
[1] 4

$cluster.size
[1]  7 12 10  3

$min.cluster.size
[1] 3

$noisen
[1] 0
```

```
$diameter
[1] 3.409096 2.991021 4.344500 3.371960

$average.distance
[1] 2.550664 1.890632 2.641349 3.155185

$median.distance
[1] 2.911401 2.143996 2.659124 3.064841

$separation
[1] 2.486626 2.934650 2.217334 2.217334

$average.toother
[1] 4.500279 5.297273 5.259489 5.508547

$separation.matrix
         [,1]     [,2]     [,3]     [,4]
[1,] 0.000000 2.934650 2.486626 3.895800
[2,] 2.934650 0.000000 3.614074 3.630612
[3,] 2.486626 3.614074 0.000000 2.217334
[4,] 3.895800 3.630612 2.217334 0.000000

$ave.between.matrix
         [,1]     [,2]     [,3]     [,4]
[1,] 0.000000 4.516726 3.973723 6.189678
[2,] 4.516726 0.000000 5.923364 5.031581
[3,] 3.973723 5.923364 0.000000 5.604115
[4,] 6.189678 5.031581 5.604115 0.000000

$average.between
[1] 5.118041

$average.within
[1] 2.271644

$n.between
[1] 361

$n.within
[1] 135

$max.diameter
[1] 4.3445

$min.separation
[1] 2.217334

$within.cluster.ss
[1] 90.00399
```

```
$clus.avg.silwidths
        1         2         3         4
0.3177759 0.5692990 0.3130840 0.3378928

$avg.silwidth
[1] 0.4125168

$g2
NULL

$g3
NULL

$pearsongamma
[1] 0.7154405

$dunn
[1] 0.5103772

$dunn2
[1] 1.259426

$entropy
[1] 1.285675

$wb.ratio
[1] 0.4438503

$ch
[1] 26.02806

$cwidegap
[1] 2.331479 2.130159 2.594859 3.064841

$widestgap
[1] 3.064841

$sindex
[1] 2.248469

$corrected.rand
NULL

$vi
NULL
```

La función `cluster.stats()` también se puede aplicar sobre clustering jerárquicos después de aplicar la función `cutree()`. Otras funciones que podemos utilizar son las funciones `prediction.strength()` y `nselectboot()`. La función `prediction.strength()` determina el número de clusters calculando la capacidad predictiva

de un clustering [3] y eligiendo como número de clusters óptimo como aquél número de clusters más grande que tiene una capacidad predictiva mayor que 0.8 o 0.9. Por ejemplo, para determinar el número óptimo de cluster mediante la capacidad predictiva del agrupamiento generado por el algoritmo *k-means*, debemos de utilizar la siguiente instrucción:

```
> mtcars.kmeans.pred <- prediction.strength(mtcars.scaled, 2,5,
                      clustermethod = kmeansCBI,
                      classification = "average",
                      M=10, cutoff = 0.8)
> mtcars.kmeans.pred

Prediction strength
Clustering method:  kmeans
Maximum number of clusters:  5
Resampled data sets:  10
Mean pred.str. for numbers of clusters:  1 1 1 1 1
Cutoff value:  0.8
Largest number of clusters better than cutoff:  5
```

El segundo y tercer parámetros indican el número mínimo y máximo de clusters entre los que buscar. El parámetro `clustermethod` define el tipo de clustering a realizar (para ver las técnicas que se pueden utilizar analizar la función `clusterboot()` del paquete `fpc`). El parámetro `classification` indica cómo se van a asignar a los clusters lo elementos no clasificados (existe una dependencia con el parámetro anterior que puede ser consultada en la descripción de la función). Por último, Los parámetros `M` y `cutoff` indican el número de veces en el que se va a particionar el conjunto de datos y el valor de corte para seleccionar el número óptimo de clusters.

La función `nselectboot()` determina el número de clusters realizando bootstrap [1]. Por ejemplo, para determinar el número óptimo de clusters para la técnica *k-means* podemos utilizar el siguiente código:

```
> mtcars.kmeans.nsboot <- nselectboot(mtcars.scaled,B=2,
                                      clustermethod=kmeansCBI,
                                      classification="average",krange=3:7)
> mtcars.kmeans.nsboot

$kopt
[1] 6

$stabk
[1]        NA        NA 0.1542969 0.2001953
[5] 0.1367188 0.1337891 0.1669922

$stab
     [,1] [,2]      [,3]      [,4]      [,5]
[1,]    0    0 0.1894531 0.2246094 0.1425781
[2,]    0    0 0.1191406 0.1757812 0.1308594
```

```
          [,6]      [,7]
[1,] 0.1542969 0.1093750
[2,] 0.1132812 0.2246094
```

Ejercicio 10.

10.a) Genera una tabla que recoja los valores para el estadístico gap y el número óptimo de clusters para los métodos analizados en este guión exceptuando el implementado por la función `fanny` y `dbscan`.

10.b) Nombra los estadísticos calculados por la función `cluter.stat()`. Genera una tabla en la que queden los estadísticos analizados en clase para las técnicas del apartado anterior.

10.c) Genera una tabla que recoja el número óptimo de clusters para los métodos los métodos analizados en este guión exceptuando el implementado por la función `fanny` utilizando la función `prediction.strength()`.

10.d) Repite el apartado anterior utilizando la función `nselectboot()`.

Referencias

1. Y. Fang and J. Wang, "Selection of the number of clusters via the bootstrap method," *Computational Statistics & Data Analysis*, vol. 56, no. 3, pp. 468–477, 2012.
2. R. Tibshirani, G. Walther, and T. Hastie, "Estimating the number of clusters in a data set via the gap statistic," *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 63, no. 2, pp. 411–423, 2001.
3. R. Tibshirani and G. Walther, "Cluster validation by prediction strength," *Journal of Computational and Graphical Statistics*, vol. 14, no. 3, pp. 511–528, 2005.