Fundamentos de la Ciencia de Datos

Práctica 2

Grado en Ingeniería Informática
Universidad de Alcalá

Grupo 9

Pablo García García
Abel López Martínez
Álvaro Jesús Martínez Parra
Raúl Moratilla Núñez

26 de diciembre de 2023

Índice general

Índice de figuras

Introducción

RStudio

RStudio es un entorno de desarrollo integrado (IDE) para R, un lenguaje de programación ampliamente utilizado en estadística, análisis de datos, investigación científica y visualización de datos. Será empleado para el desarrollo de esta práctica. La interfaz del IDE de RStudio está diseñada para facilitar el análisis estadístico y la visualización de datos.

- **File**: Aquí puedes crear nuevos archivos y proyectos, incluidos scripts de Python y documentos Sweave. Trabajar con proyectos es una práctica profesional estándar y permite una mejor organización.
- **Edit**: Esta sección funciona de manera similar a otros entornos de desarrollo, proporcionando herramientas de edición de texto estándar.
- **Code**: Incluye herramientas específicas para escribir y ejecutar código R, como insertar operadores o ejecutar scripts completos.
- **View**: Permite ajustar la apariencia de la interfaz de RStudio, como mostrar u ocultar diferentes paneles.
- **Plots**: Muy utilizado para la visualización de datos, permite generar, ver y manipular gráficos estadísticos.
- **Session**: Controla las sesiones de R, permitiendo iniciar, interrumpir o terminar sesiones.
- **Build**: Herramientas relacionadas con la construcción de paquetes de R.
- **Debug**: Funciones para depurar el código R.
- **Profile**: Herramientas para medir el rendimiento del código R.
- **Tools**: Opciones para configurar el entorno de desarrollo globalmente, como la disposición del panel y opciones de proyecto.
- **Help**: Ofrece documentación y ayuda para usar R y RStudio, un recurso esencial para el aprendizaje y la solución de problemas.

El entorno de RStudio es altamente personalizable y se organiza en cuatro áreas principales. Puedes redactar código en el panel de `scripts`, ejecutar instrucciones en el `console`, y ver los objetos activos y el historial en el área de `environment/history`. El cuarto panel es multifuncional, permitiendo administrar archivos, visualizar datos a través de gráficos, instalar y administrar paquetes y buscar en la documentación en la sección indicada a continuación.

`files/plots/packages/help/viewer`

Este diseño modular facilita la gestión eficiente del flujo de trabajo en análisis de datos y desarrollo estadístico.

Figura 1: Logo del entorno RStudio

Posit

La página web de Posit (`https://posit.co/`) ofrece una amplia gama de productos y soluciones para la ciencia de datos, enfocándose en el uso de R y Python. Presenta opciones de software de código abierto, soluciones empresariales y basadas en la nube. Destaca su compromiso con la ciencia de datos accesible para todos, ofreciendo recursos educativos, historias de clientes y soporte para la comunidad. La empresa busca promover un entorno inclusivo y empoderador, destacando su papel en el avance de la ciencia de datos a través de la colaboración y la innovación tecnológica.

Figura 2: Logo de la compañía Posit

De RStudio a Posit

El artículo del blog de Posit (`https://posit.co/blog/rstudio-is-now-posit/`), anteriormente conocido como RStudio, detalla los cambios y compromisos de la compañía a medida que evoluciona. Fundada en 2009, RStudio se centró en crear software de código abierto de alta calidad para científicos de datos. La compañía ha invertido significativamente en el desarrollo de código abierto, educación y la comunidad, con el objetivo de servir a los creadores de conocimiento durante los próximos 100 años.

La transformación de RStudio a Posit no solo implica un cambio de nombre, sino también una reafirmación de sus valores y objetivos. Como una Corporación de Beneficio Público y una B Corp certificada, Posit se compromete a cumplir con los más altos estándares de desempeño social y ambiental, transparencia y responsabilidad. Los directivos de Posit tienen la responsabilidad fiduciaria de abordar las necesidades sociales, económicas y ambientales, además de supervisar los objetivos comerciales de la empresa. La misión de Posit incluye mantener su independencia a largo plazo para cumplir con estos objetivos.

El compromiso de Posit con el código abierto sigue siendo una prioridad central. La empresa cree firmemente que el progreso significativo se logra poniendo herramientas de ciencia de datos al alcance de todos, independientemente de sus medios económicos. Además, Posit busca expandir su enfoque más allá del lenguaje de programación R, abrazando también a la comunidad de Python y a otras comunidades de programación. Aunque R sigue siendo un componente clave, el objetivo es mejorar la comunicación científica para todos, creando software tanto de código abierto como comercial para R, Python y más.

Finalmente, Posit desea fomentar una comunidad más amplia y diversa. Inspirándose en la comunidad de R, la compañía aspira a ayudar a que otras áreas de la ciencia sean tan abiertas, dinámicas, inclusivas y diversas como la comunidad a la que pertenecen. A pesar del cambio de nombre, la dedicación de Posit a ayudar a los científicos de datos a utilizar software de código abierto para plantear y responder preguntas importantes permanece inalterada.

Parte 1

Ejercicios guiados

En esta primera parte de la práctica, se repetirán los ejercicios explicados y realizados por el profesor en las clases de laboratorio, utilizando los mismos procedimientos vistos y plasmándolos en este documento.

1.1. Clasificación no supervisada

1.1.1. $k-$means

Ejercicio 1.1.1. *El primer conjunto de datos, que se empleará para realizar el análisis de clasificación no supervisada con $k-$means, estará formado por las siguientes 8 calificaciones de estudiantes: 1. {4, 4}; 2. {3, 5}; 3. {1, 2}; 4. {5, 5}; 5.{0, 1}; 6. {2, 2}; 7. {4, 5}; 8. {2, 1}, donde las características de las calificaciones son: {Teoría, Laboratorio}.*

En este ejercicio se va a detallar cómo realizar una clasificación no supervisada en un conjunto de datos en R; concretamente utilizando el algoritmo $k-$means. El objetivo será agrupar subconjuntos de los datos en clústers o grupos, creando de esta forma una clasificación del conjunto total. Estos clústers se determinarán en base a la muestra, obteniéndose durante el mismo proceso de clasificación.

El algoritmo $k-$means ofrece una técnica de clústerización en base a una muestra de datos y una cantidad n de clústers. Para empezar a realizar el algoritmo se necesita la ubicación de los centroides de cada clúster. Un centroide es el punto medio del grupo de sucesos que componen el clúster, por lo que habrá tantos centroides como n clústers haya. La cantidad de clústers así como sus centroides iniciales serán elegidos arbitrariamente por el usuario; estos se irán reubicando a medida que se itere en el algoritmo.

Para empezar, en R se necesita introducir la muestra o conjunto de datos con el que se va a trabajar. Estos datos serán introducidos en una matriz utilizando la función `matrix`.

```
m <- matrix(c(4,4, 3,5, 1,2, 5,5, 0,1, 2,2, 4,5, 2,1),2,8)
(m <- t(m))
```

```
##      [,1] [,2]
## [1,]    4    4
## [2,]    3    5
## [3,]    1    2
## [4,]    5    5
## [5,]    0    1
## [6,]    2    2
## [7,]    4    5
## [8,]    2    1
```

Para poder trabajar debidamente se debe trasponer la matriz, de tal forma que cada columna represente una componente. El primer punto conformará la primera fila, el segundo punto la segunda fila y así sucesivamente con todos los datos.

Además del conjunto de datos, el algoritmo $k-$means necesita unos centroides iniciales. Estos se introducirán de la misma forma que la muestra, teniendo en la primera fila el centroide del primer clúster, en la siguiente fila el centroide del segundo... Al tratarse de pocos puntos se eligen arbitrariamente dos centroides, por lo que la clasificación final será realizada con dos clústers. Se introducen así los centroides:

```
c <- matrix(c(0,1,2,2),2,2)
(c < -t(c))

##       [,1]  [,2]
## [1,] FALSE FALSE
## [2,] FALSE FALSE
```

Con la muestra y los centroides introducidos en el entorno de trabajo, ya se puede realizar el algoritmo. Para ello se utilizará la función `kmeans`, función del paquete cargado por defecto `stats`. La función recibe tres parámetros: la muestra de los datos (`m`), los centroides iniciales de los clústers (`c`) y el número de iteraciones que se desean. En este caso se eligen cuatro iteraciones, las mismas que se necesitaron en clase de teoría. En teoría, el número de iteraciones no se sabe a priori, por lo que habría que ir probando. Sin embargo, como ya se sabe el número que se necesita, se pone directamente 4.

```
(clasificacionns = (kmeans(m,c,4)))

## K-means clustering with 2 clusters of sizes 4, 4
##
## Cluster means:
##   [,1] [,2]
## 1 1.25 1.50
## 2 4.00 4.75
##
## Clustering vector:
```

```
## [1] 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"
```

Entre los parámetros que aparecen en la salida se encuentran `Cluster means` y `Clustering vector`. El primero de ellos proporciona la ubicación de los dos centroides de los clústers que conforman la clasificación final. Como se puede apreciar, salen los dos centroides que salieron en teoría. El segundo parámetro indica a qué clúster pertenece cada suceso de la muestra introducida. Así el punto 1 pertenece al clúster 2, el punto 2 pertenece al clúster 2, el punto 3 al clúster 1... Este último se puede utilizar como entrada para realizar otras tareas. Puede ser muy útil para analizar cada clúster por separado, ver sus caracteristicas comunes e intentar deducir por qué esos datos están relacionados.

El siguiente comando permite añadir una columna a la izquierda de la muestra de datos (`m`) que se había introducido al inicio, indicando a qué clúster pertenece cada suceso.

```
(m = cbind(clasificacionns$clúster,m))
```

```
##      [,1] [,2]
## [1,]    4    4
## [2,]    3    5
## [3,]    1    2
## [4,]    5    5
## [5,]    0    1
## [6,]    2    2
## [7,]    4    5
## [8,]    2    1
```

Esto se realiza para poder dividir la muestra según la clasificación que se ha conseguido. Llamando a la función subset se puede extraer la porción de la muestra que pertenece al clúster 1 y la porción restante que pertenece al clúster 2.

```
(mc1 = subset(m,m[,1]==1))
```

```
##      [,1] [,2]
## [1,]    1    2
```

```
(mc2 = subset(m,m[,1]==2))
```

```
##      [,1] [,2]
## [1,]    2    2
## [2,]    2    1
```

Una vez se tienen los datos separados se puede eliminar la columna que indica el clúster al que pertenece cada dato, ya que se sobreentiende que todos pertenecen al mismo clúster porque ya han pasado por un proceso de separación en función de la clasificación. Por ejemplo, para los datos pertenecientes al primer clúster, se haría de la siguiente forma:

```
(mc1 = mc1[,-1])

## [1] 2
```

Con todo esto se consigue, partiendo de una muestra de datos, una clasificación no supervisada utilizando la técnica de clústerización basada en el algoritmo $k-$means. A partir de ella se podrán intentar deducir ciertas conclusiones.

1.1.2. Clusterización jerárquica algomerativa

Ejercicio 1.1.2. *El segundo conjunto de datos, que se empleará para realizar el análisis de clasificación no supervisada con Clusterización Jerárquica Aglomerativa, estará formado por 6 calificaciones de estudiantes: 1. {0.89, 2.94}; 2.{4.36, 5.21}; 3. {3.75, 1.12}; 4. {6.25, 3.14}; 5. {4.1, 1.8}; 6. {3.9, 4.27}.*

La Clusterización Jerárquica Aglomerativa es un método de análisis de datos que comienza tratando cada punto como un clúster individual y luego, iterativamente, combina los clústers más cercanos hasta formar un único clúster. Este método es útil para identificar grupos naturales en los datos. En clase se implementó mediante el paquete `LearnClust` del CRAN.

Inicialmente, se crea una matriz con la muestra del problema, y se transpone la matriz para que tenga el formato deseado.

```
m <- matrix(
        c(0.89,2.94, 4.36,5.21, 3.75,1.12, 6.25,3.14, 4.1,1.8, 3.9,4.27),2,6)
(m <- t(m))

##      [,1] [,2]
## [1,] 0.89 2.94
## [2,] 4.36 5.21
## [3,] 3.75 1.12
## [4,] 6.25 3.14
## [5,] 4.10 1.80
## [6,] 3.90 4.27
```

Para la ejecución del algoritmo utilizamos la función `agglomerativeHC`. Esta función recibe como parámetros la matriz de datos, la métrica de distancia y el criterio de enlace.

```
agglomerativeHC(m, 'EUC', 'MIN')
```

```
## $dendrogram
## Number of objects: 6
##
##
## $clusters
## $clusters[[1]]
##     X1   X2
## 1 0.89 2.94
##
## $clusters[[2]]
##     X1   X2
## 1 4.36 5.21
##
## $clusters[[3]]
##     X1   X2
## 1 3.75 1.12
##
## $clusters[[4]]
##     X1   X2
## 1 6.25 3.14
##
## $clusters[[5]]
##    X1  X2
## 1 4.1 1.8
##
## $clusters[[6]]
##    X1   X2
## 1 3.9 4.27
##
## $clusters[[7]]
##     X1   X2
## 1 3.75 1.12
## 2 4.10 1.80
##
## $clusters[[8]]
##     X1   X2
## 1 4.36 5.21
## 2 3.90 4.27
##
## $clusters[[9]]
```

```
##     X1   X2
## 1 3.75 1.12
## 2 4.10 1.80
## 3 4.36 5.21
## 4 3.90 4.27
##
## $clusters[[10]]
##     X1   X2
## 1 6.25 3.14
## 2 3.75 1.12
## 3 4.10 1.80
## 4 4.36 5.21
## 5 3.90 4.27
##
## $clusters[[11]]
##     X1   X2
## 1 0.89 2.94
## 2 6.25 3.14
## 3 3.75 1.12
## 4 4.10 1.80
## 5 4.36 5.21
## 6 3.90 4.27
##
##
## $groupedClusters
##   cluster1 cluster2
## 1        3        5
## 2        2        6
## 3        7        8
## 4        4        9
## 5        1       10
```

En este código, `EUC` representa la métrica de distancia euclidiana y `MIN` el criterio para la agrupación de clústers. En la salida se pueden observar cuatro apartados.

- El primero es una figura los datos de la matriz que hemos introducido impresos en una gráfica.
- El segundo es el número de puntos que se han introducido, mediante el atributo `$dendrogram`, en este caso 6.
- El tercer apartado es una lista que muestra las coordenadas de todos los clústers al finalizar la ejecución, los 6 primeros son los puntos introducidos y los 5 siguientes son los formados por cada una de las iteraciones uniendo dos clústers anteriores hasta tener todos unidos en el mismo clúster, momento en el que finaliza el algoritmo. Se puede ver en el atributo `$clústers`.

- El cuarto apartado es el proceso que se realiza en cada iteración, se pueden ver en el apartado `$groupedClusters`.

  1. Se unen los clústers 3 y 5, para formar el clúster 7.
  2. Se unen los clústers 2 y 6, para formar el clúster 8.
  3. Se unen los clústers 4 y 8, para formar el clúster 9.
  4. Se unen los clústers 7 y 9, para formar el clúster 10.
  5. Se unen los clústers 1 y 10, para formar el clúster 11.

Para obtener una explicación detallada paso a paso tal y como se ha explicado en clase, usamos la función `agglomerativeHC.details`.

```
agglomerativeHC.details(m, 'EUC', 'MIN')

##   Agglomerative hierarchical clustering is a classification technique that initializes
##  a cluster for each data.
##
##   It calculates the distance between datas depending on the approach type given and
##
##  it creates a new cluster joining the most similar clusters until getting only one.
##   'toList' creates a list initializing datas by creating clusters with each one
##
##   These are the clusters with only one element:

## [[1]]
##      [,1] [,2] [,3]
## [1,] 0.89 2.94    1
##
## [[2]]
##      [,1] [,2] [,3]
## [1,] 4.36 5.21    1
##
## [[3]]
##      [,1] [,2] [,3]
## [1,] 3.75 1.12    1
##
## [[4]]
##      [,1] [,2] [,3]
## [1,] 6.25 3.14    1
##
## [[5]]
##      [,1] [,2] [,3]
## [1,]  4.1  1.8    1
##
## [[6]]
##      [,1] [,2] [,3]
## [1,]  3.9 4.27    1

##
##   In each step:
```

```
##    - It calculates a matrix distance between active clusters depending on the approach and
distance type.
##    - It gets the minimum distance value from the matrix.
##    - It creates a new cluster joining the minimum distance clusters.
##    - It repeats these steps while final clusters do not include all datas.
##
## ------------------------------------------------------------------------------
## STEP =>1
##
##  Matrix Distance (distance type = EUC, approach type = MIN):









##
##  The minimum distance is: 0.764787552199956
##
##  The closest clusters are: 3, 5
##
##  The grouped clusters are added to the solution.
##
##  Grouping clusters 3 and cluster 5, it is created a new cluster:




##
##  The new cluster is added to the solution.
##
## ------------------------------------------------------------------------------
## STEP =>2
##
##  Matrix Distance (distance type = EUC, approach type = MIN):









##
##  The minimum distance is: 1.04651803615609
##
##  The closest clusters are: 2, 6
##
##  The grouped clusters are added to the solution.
##
##  Grouping clusters 2 and cluster 6, it is created a new cluster:
```

```
##     X1   X2
## 1 4.36 5.21
## 2 3.90 4.27

##
##  The new cluster is added to the solution.
##
## ------------------------------------------------------------------------------
## STEP =>3
##
##  Matrix Distance (distance type = EUC, approach type = MIN):

##          [,1] [,2] [,3]     [,4] [,5] [,6]     [,7]     [,8]
## [1,] 0.000000    0    0 5.363730    0    0 3.389985 3.290745
## [2,] 0.000000    0    0 0.000000    0    0 0.000000 0.000000
## [3,] 0.000000    0    0 0.000000    0    0 0.000000 0.000000
## [4,] 5.363730    0    0 0.000000    0    0 2.533397 2.607566
## [5,] 0.000000    0    0 0.000000    0    0 0.000000 0.000000
## [6,] 0.000000    0    0 0.000000    0    0 0.000000 0.000000
## [7,] 3.389985    0    0 2.533397    0    0 0.000000 2.478084
## [8,] 3.290745    0    0 2.607566    0    0 2.478084 0.000000

##
##  The minimum distance is: 2.47808393723861
##
##  The closest clusters are: 7, 8
##
##  The grouped clusters are added to the solution.
##
##  Grouping clusters 7 and cluster 8, it is created a new cluster:

##     X1   X2
## 1 3.75 1.12
## 2 4.10 1.80
## 3 4.36 5.21
## 4 3.90 4.27

##
##  The new cluster is added to the solution.
##
## ------------------------------------------------------------------------------
## STEP =>4
##
##  Matrix Distance (distance type = EUC, approach type = MIN):

##           [,1] [,2] [,3]     [,4] [,5] [,6] [,7] [,8]     [,9]
##  [1,] 0.000000    0    0 5.363730    0    0    0    0 3.290745
##  [2,] 0.000000    0    0 0.000000    0    0    0    0 0.000000
##  [3,] 0.000000    0    0 0.000000    0    0    0    0 0.000000
##  [4,] 5.363730    0    0 0.000000    0    0    0    0 2.533397
##  [5,] 0.000000    0    0 0.000000    0    0    0    0 0.000000
##  [6,] 0.000000    0    0 0.000000    0    0    0    0 0.000000
##  [7,] 0.000000    0    0 0.000000    0    0    0    0 0.000000
##  [8,] 0.000000    0    0 0.000000    0    0    0    0 0.000000
##  [9,] 3.290745    0    0 2.533397    0    0    0    0 0.000000
```

```
##
##  The minimum distance is: 2.53339692902632
##
##  The closest clusters are: 4, 9
##
##  The grouped clusters are added to the solution.
##
##  Grouping clusters 4 and cluster 9, it is created a new cluster:
##     X1   X2
## 1 6.25 3.14
## 2 3.75 1.12
## 3 4.10 1.80
## 4 4.36 5.21
## 5 3.90 4.27
##
##  The new cluster is added to the solution.
##
## ------------------------------------------------------------------------------
## STEP =>5
##
##  Matrix Distance (distance type = EUC, approach type = MIN):
##           [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]    [,10]
##  [1,] 0.000000    0    0    0    0    0    0    0    0 3.290745
##  [2,] 0.000000    0    0    0    0    0    0    0    0 0.000000
##  [3,] 0.000000    0    0    0    0    0    0    0    0 0.000000
##  [4,] 0.000000    0    0    0    0    0    0    0    0 0.000000
##  [5,] 0.000000    0    0    0    0    0    0    0    0 0.000000
##  [6,] 0.000000    0    0    0    0    0    0    0    0 0.000000
##  [7,] 0.000000    0    0    0    0    0    0    0    0 0.000000
##  [8,] 0.000000    0    0    0    0    0    0    0    0 0.000000
##  [9,] 0.000000    0    0    0    0    0    0    0    0 0.000000
## [10,] 3.290745    0    0    0    0    0    0    0    0 0.000000
##
##  The minimum distance is: 3.29074459659208
##
##  The closest clusters are: 1, 10
##
##  The grouped clusters are added to the solution.
##
##  Grouping clusters 1 and cluster 10, it is created a new cluster:
##     X1   X2
## 1 0.89 2.94
## 2 6.25 3.14
## 3 3.75 1.12
## 4 4.10 1.80
## 5 4.36 5.21
## 6 3.90 4.27
##
##  The new cluster is added to the solution.
##
##  This loop has been repeated until the last cluster contained every single clusters.
```

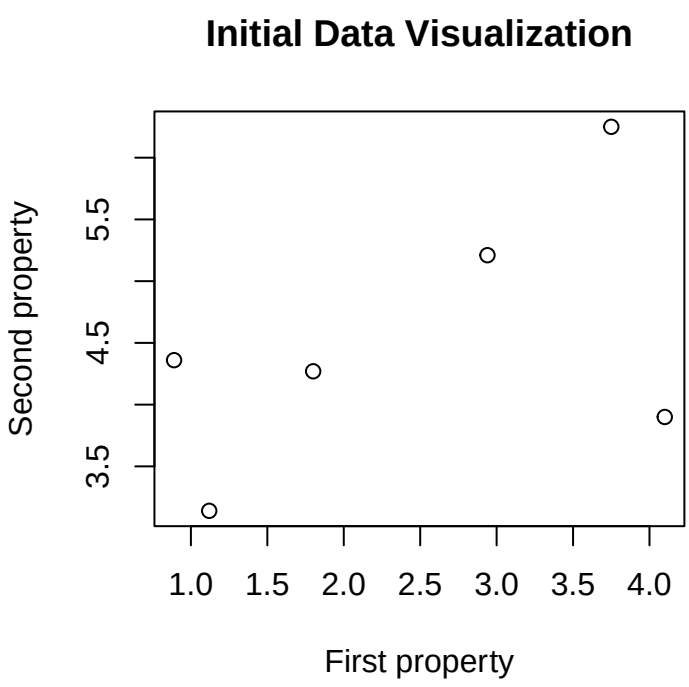


Figura 1.1: Diagrama de LearnClust

La función también se puede ejecutar con diferentes criterios de unión, como `MAX` y `AVG`, para observar cómo cambian los clústers:

```
agglomerativeHC.details(m, 'EUC', 'MAX')
agglomerativeHC.details(m, 'EUC', 'AVG')
```

1.2. Clasificación supervisada

1.2.1. Árboles de decisión

Ejercicio 1.2.1. *El tercer conjunto de datos, que se empleará para realizar el análisis de clasificación supervisada utilizando árboles de decisión, estará formado por las siguientes 9 calificaciones de estudiantes: 1. {A, A, B, Ap}; 2. {A, B, D, Ss}; 3. {D, D, C, Ss}; 4. {D, D, A, Ss}; 5. {B, C, B, Ss}; 6. {C, B, B, Ap}; 7. {B, B, A, Ap}; 8. {C, D, C, Ss}; 9. {B, A, C, Ss}, donde las características de las calificaciones son: {Teoría, Laboratorio, Prácticas, Calificación Global).*

El primer paso a ejecutar para poder realizar la clasificación supervisada con árboles de decisión es introducir los datos en un archivo de texto (`.txt`), que será posteriormente leído con el lenguaje R. Para ello, es necesario que el archivo de texto cumpla con un formato en concreto:

- La primera fila contendrá los nombres de las columnas, iniciando la fila con un tabulado o espacio e introduciendo otro entre cada una de las columnas.
- La primera columna contendrá los nombres de cada una de las filas del dataframe.
- Cada elemento de una fila estará separado del resto con un tabulado o espacio.
- El archivo de texto debe finalizar con una fila vacía.

Este archivo de texto se llamará `aprobados.txt` y será guardado en la carpeta `data`. El archivo es cargado en R de la siguiente forma:

```
(calificaciones=read.table("data/aprobados.txt"))

##    T L P CG
## 1. A A B Ap
## 2. A B D Ss
## 3. D D C Ss
## 4. D D A Ss
## 5. B C B Ss
## 6. C B B Ap
## 7. B B A Ap
## 8. C D C Ss
## 9. B A C Ss
```

Para la implementación del algoritmo, se emplea el paquete `rpart`, el cual se encuentra disponible en el repositorio `CRAN`. Este paquete ofrece un método homónimo que recibe por parámetro, entre otros, los datos en forma de dataframe. Por ello, es necesario poseer los datos en esta forma en lugar de en una tabla:

```
(muestra=data.frame(calificaciones))

##    T L P CG
## 1. A A B Ap
## 2. A B D Ss
## 3. D D C Ss
## 4. D D A Ss
## 5. B C B Ss
## 6. C B B Ap
## 7. B B A Ap
## 8. C D C Ss
## 9. B A C Ss

clasificacion = rpart(CG~T+L+P, data=muestra, method="class", minsplit=1)
```

El primer argumento que se le pasa es `CG T+L+P`, que contiene como primer elemento (a la izquierda de `~`) el criterio que se desea clasificar, y a la derecha de la virgulilla, aparecen

las columnas sobre las que se desea clasificar. También es posible poner un punto (`.`) a la derecha, lo cual significa que se clasificaría en base a todas las columnas que no fueran el criterio de clasificación. El segundo parámetro es, como ya se ha mencionado, la muestra en un dataframe; el tercer parámetro `method='class'` indica que se está abordando un problema de clasificación; y el último parámetro `minsplit=1` es necesario ya que se está trabajando sobre un conjunto de datos pequeño. La salida de la función es la siguiente:

```
clasificacion

## n= 9
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
## 1) root 9 3 Ss (0.3333333 0.6666667)
##   2) L=A,B 5 2 Ap (0.6000000 0.4000000)
##     4) P=A,B 3 0 Ap (1.0000000 0.0000000) *
##     5) P=C,D 2 0 Ss (0.0000000 1.0000000) *
##   3) L=C,D 4 0 Ss (0.0000000 1.0000000) *
```

Se puede apreciar que el árbol consta de dos niveles además de la raíz. El primero de ellos clasifica según la nota en laboratorio. En caso de ser `C` o `D`, la calificación final es `Ss`, en caso de ser `A` o `B` es necesario observar el siguiente nivel. Este segundo nivel clasifica según la nota de prácticas, si la nota es `A` o `B`, la calificación es `Ap` y si es `C` o `D`, la calificación final es `Ss`.

1.2.2. Regresión lineal

Ejercicio 1.2.2. *El cuarto conjunto de datos, que se empleará para realizar el análisis de clasificación supervisada utilizando regresión, estará formado por los siguientes 4 radios ecuatoriales y densidades de los planetas interiores:* {*Mercurio, 2.4, 5.4; Venus, 6.1, 5.2; Tierra, 6.4, 5.5; Marte, 3.4, 3.9*}.

Para empezar este ejercicio se deberán introducir los datos expuestos en el enunciado en un archivo de texto (`.txt`), con el fin de ser posteriormente leídos. La inserción de los datos en este fichero se hará atendiendo a las normas relacionadas con este tipo de archivos que ya se vieron en la primera práctica. Estas normas son las siguientes:

- Existirá una tabulación entre dato y dato.
- La primera columna numera las filas, y en la primera fila se introduce un espacio y el nombre de las variables.
- Se introducirá un salto de línea en la última fila.
- Para los números decimales se utilizarán puntos.

- Al escribir nombres, no se deberán introducir espacios.

Obedeciendo estas normas, se copian los datos en un fichero llamado `planetas.txt`, y se carga en R de la siguiente manera:

```
(muestra = read.table("data/planetas.txt"))

##      R   D
## 1. 2.4 5.4
## 2. 6.1 5.2
## 3. 6.4 5.5
## 4. 3.4 3.9
```

Una vez se tienen los datos en R, se procede a hacer uso de la función `lm` contenida en los paquetes básicos (concretamente en el paquete `stats`). Esta función recibe dos parámetros. El primero de ellos es la fórmula en donde se va a decir en función de qué parámetro se quiere otro parámetro. En este caso se tiene la columna `R` que representa el radio y la columna `D` que representa la densidad. Como se quiere la densidad en función del radio, el primer parámetro tendrá que ser formula=D∼R. Se indica de esta forma que se pretende obtener la columna `D` en función de la columna `R`; o lo que es lo mismo, la densidad en función del radio.

El segundo parámetro que entra a la función es la estructura que contiene los datos que se pretenden estudiar. En este caso, el parámetro `data` será la variable `muestra`, confeccionada previamente. Con los parámetros de la función `lm` claros, se invoca a la misma de la siguiente forma:

```
(regresion=lm(D~R, data=muestra))

##
## Call:
## lm(formula = D ~ R, data = muestra)
##
## Coefficients:
## (Intercept)            R
##      4.3624       0.1394
```

En la salida de la función se observan los coeficientes que conforman la recta de regresión que mejor se adapta a los datos introducidos. El método de obtención o ajuste de la función es el de mínimos cuadrados, pudiendo comprobar que el resultado es el mismo que el que se ha visto en clase. En este método, los coeficientes de la recta $\hat{y} = a + bx$ se calculan de la siguiente forma, siendo x el radio e y la densidad:

$$b = \frac{S_{xy}}{S_x^2}$$
$$a = \bar{y} - b\bar{x}$$

La salida proporciona directamente los valores de a y b, siendo estos el primero y el segundo respectivamente. Con estos valores se puede decir que la densidad de un planeta se puede obtener atendiendo a la siguiente fórmula:

$$D = 4{,}3624 + 0{,}1394R$$

Con el comando `summary` aplicado a la regresión que se acaba de hacer se pueden ver parámetros que detallan esta recta de regresión con respecto a los datos.

```
(summary(regresion))

##
## Call:
## lm(formula = D ~ R, data = muestra)
##
## Residuals:
##       1.       2.       3.       4.
##  0.70312 -0.01253  0.24566 -0.93624
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   4.3624     1.2050   3.620   0.0685 .
## R             0.1394     0.2466   0.565   0.6289
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.846 on 2 degrees of freedom
## Multiple R-squared:  0.1377,Adjusted R-squared:  -0.2935
## F-statistic: 0.3193 on 1 and 2 DF,  p-value: 0.6289
```

Entre todos los detalles que aparecen, se observa el parámetro `Residuals`. Este parámetro devuelve un vector con los residuos, o dicho de otra forma, la distancia entre el valor real y el valor que se obtiene por medio de la recta calculada. También se puede apreciar el parámetro `Multiple R-squared`, el cual sirve como medida de cuán bueno ha sido el ajuste. Este valor podrá tomar valores entre 0 y 1, siendo 0 un ajuste muy malo y 1 un ajuste perfecto. Como se observa, el valor de este parámetro es 0,1377, lo cual quiere decir que el ajuste es malo; y con ello se puede concluir que el radio no explica la densidad de los planetas.

Una vez visto en detalle el cálculo y valoración de la recta de regresión se va a ver cómo se pueden identificar sucesos anómalos mediante la técnica de regresión. El proceso por lo general sigue 5 pasos:

1. Determinar el grado de outlier `d`
2. Obtener la ecuación de la recta de regresión

3. Obtener el error estándar s_r del vector de residuos

4. Calcular el límite para los valores típicos como $d \cdot s_r$

5. Identificar como outliers los residuos (en valor absoluto) que superen ese límite

Hasta ahora se tiene la recta de regresión y el vector de residuos que está en `summary`. Para extraerlo y poder operar con él se hace de la siguiente forma:

```
(res=summary(regresion)$residuals)

##          1.          2.          3.          4.
##  0.70312301 -0.01253452  0.24565541 -0.93624389
```

Una vez se tiene el vector de residuos se calcula el error estándar del mismo:

```
(sr = sqrt(sum(res^2)/4))

## [1] 0.5982136
```

Con ello se puede plantear un bucle el cual compruebe qué elemento o elementos se presentan como anomalías. Se va a elegir como grado de outlier $d = 3$

```
for (i in 1:length(res)){
        if(res[i]>3*sr){
                print("el suceso");
                print(res[i]);
                print("es un suceso anómalo o outlier")
                                }
}
```

Este bucle comprueba por cada elemento del vector de residuos si supera o no el límite establecido para outliers; en caso afirmativo lo imprime por pantalla. En la salida anterior se puede observar que no hay ningún outlier, ya que ningún residuo supera el umbral establecido.

Se va a probar con otro conjunto de datos para demostrar que por medio de este método podemos identificar las anomalías. En primer lugar, se vuelve a confeccionar un archivo de texto (`.txt`) atendiendo a las normas previamente expuestas. Este archivo de texto se llamará `planetas2.txt` para evitar problemas de sobrescritura.

```
(muestra = read.table("data/planetas2.txt"))

##       R    D
## 1.  3.0  2.0
## 2.  3.5 12.0
## 3.  4.7  4.1
```

```
## 4.  5.2  4.9
## 5.  7.1  6.1
## 6.  6.2  5.2
## 7. 14.0  5.3
```

Una vez hecho esto se invoca a la función `lm` para sacar la regresión y se guarda el vector de residuos tal y como se ha hecho previamente.

```
(dfr=lm(D~R, data=muestra))

##
## Call:
## lm(formula = D ~ R, data = muestra)
##
## Coefficients:
## (Intercept)            R
##     6.01445     -0.05723

(res=summary(dfr)$residuals)

##         1.         2.         3.         4.         5.         6.         7.
## -3.8427477  6.1858698 -1.6454482 -0.8168308  0.4919157 -0.4595958  0.0868370
```

Se vuelve a calcular el error estándar de los residuos. Atendiendo a la fórmula hay que dividir entre 7, ya que se tienen 7 entradas en el conjunto de datos del fichero.

```
(sr = sqrt(sum(res^2)/7))

## [1] 2.850242
```

Una vez se tienen todos los datos necesarios, se pone a prueba el código visto previamente para detectar anomalías. Esta vez se cambiará el grado de outlier $d = 2$.

```
for (i in 1:length(res)){
        if(res[i]>2*sr){
                print("el suceso");
                print(res[i]);
                print("es un suceso anómalo o outlier");
                                }
}

## [1] "el suceso"
##      2.
## 6.18587
## [1] "es un suceso anómalo o outlier"
```

Se observa en la salida del código que el suceso 2 es un outlier. Atendiendo a los parámetros que se han ido obteniendo, se observa a simple vista que el suceso 2 supera el límite establecido.

Parte 2

Ejercicios autónomos

En esta segunda parte de la práctica se implementarán los algoritmos vistos anteriormente, plasmando en el documento la explicación del código realizado, así como su aplicación a los enunciados propuestos para esta parte de la práctica.

```
source("data/aux_functions.R")
```

2.1. Clasificación no supervisada

2.1.1. $k-$means

Ejercicio 2.1.1. *El primer conjunto de datos, que se empleará para realizar el análisis de clasificación no supervisada con $k-$means, estará formado por los siguientes 15 valores de velocidades de respuesta y temperaturas normalizadas de un microprocesador {Velocidad, Temperatura}: 1.{3.5, 4.5}; 2.{0.75, 3.25}; 3.{0, 3}; 4.{1.75, 0.75}; 5.{3, 3.75}; 6.{3.75, 4.5}; 7.{1.25, 0.75}; 8.{0.25, 3}; 9.{3.5, 4.25}; 10.{1.5, 0.5}; 11.{1, 1}; 12.{3, 4}; 13.{0.5, 3}; 14.{2, 0.25}; 15.{0, 2.5}. Del análisis visual de los datos se ha concluido que hay una alta probabilidad que sean tres clústers.*

Los datos que se proponen en el enunciado del ejercicio van a ser introducidos en un fichero Excel (`.xlsx`), de tal forma que estos se van a obtener leyendo este fichero desde R. Para ello, se hará uso de la función `read.xlsx` contenida en el paquete `openxlsx` que se ha visto previamente. Con esta función se puede pasar como parámetro la ruta del archivo que contiene los datos, devolviendo los mismos en un dataframe.

Para este ejercicio en concreto se hará uso de dos ficheros, uno con los datos de velocidad y temperatura (dispuestos cada uno en una columna del Excel), y otro fichero que contenga las coordenadas de los centroides iniciales que se van a utilizar en el algoritmo. En la primera fila de ambos ficheros deberá ir el título de cada columna. Primeramente se leen los datos del ejercicio. A partir de ahora, a cada uno de estos datos se les conocerán como puntos.

```
(data = data.frame(read.xlsx("data/datosKMeans.xlsx")))
```

```
##    Velocidad Temperatura
## 1       3.50        4.50
## 2       0.75        3.25
## 3       0.00        3.00
## 4       1.75        0.75
## 5       3.00        3.75
## 6       3.75        4.50
## 7       1.25        0.75
## 8       0.25        3.00
## 9       3.50        4.25
## 10      1.50        0.50
## 11      1.00        1.00
## 12      3.00        4.00
## 13      0.50        3.00
## 14      2.00        0.25
## 15      0.00        2.50
```

Posteriormente se lee el fichero con los centroides iniciales. En la clasificación final se tendrán tantos clústers como centroides se hayan introducido, ya que al final un centroide representa un clúster.

```
(centroides = data.frame(read.xlsx("data/centroides.xlsx")))
```

```
##     X   Y
## 1 0.5 0.5
## 2 1.0 2.0
## 3 2.0 1.0
```

Una vez se tienen todos los datos en el entorno de trabajo se va a proceder a explicar la implementación del algoritmo. El primer paso del algoritmo consiste en realizar una matriz de distancias. Esta matriz tendrá tantas filas como centroides se hayan declarado y tantas columnas como puntos se hayan introducido. Esta matriz recoge las distancias euclídeas de todos los puntos a todos los centroides. Para realizar esta tabla se ha programado la siguiente función, la cual va rellenando una matriz con las distancias euclídeas de cada punto a cada clúster.

```
create_distance_matrix = function(points, centroids) {
        n_pts = nrow(points)
        n_cent = nrow(centroids)
        distances = data.frame(matrix(ncol = n_pts, nrow = n_cent))
        for (cent in 1:n_cent) {
                for (pt in 1:n_pts) {
                        distances[cent, pt] =
```

```
                                euc_distance(points[pt,], centroids[cent,])
                }
        }
        distances
}
```

Una vez se tiene la matriz de distancias se debe realizar una matriz de asignaciones. Esta matriz será de las mismas dimensiones que la anterior, pero ahora no contendrá las distancias euclídeas. Por cada punto (columna) tendremos un 0 o un 1. Si aparece un 1 en una posición, la distancia del punto a ese clúster que representa la fila es la mínima. Por ejemplo; si el punto 1 esta más cerca del clúster 2 que del resto de clústers, la posición a_{12} tendrá un 1, y el resto de la columna un 0.

Para realizar esta tabla se itera punto por punto, y por cada uno se observa la distancia del mismo a todos los clústers. Se debe poner un 1 en el lugar donde se encuentra la distancia euclídea mínima y un 0 en el resto de celdas de la columna. Es por ello que primero se inicializa a 0 toda la matriz y se va buscando por cada punto el clúster que está a menos distancia, para poner un 1 en su celda correspondiente. El código que recoge esta funcionalidad es el siguiente:

```
create_assigment_matrix = function(dist_matrix) {
        n_pts = ncol(dist_matrix)
        n_cent = nrow(dist_matrix)
        assig_matrix = data.frame(matrix(0, ncol = n_pts, nrow = n_cent))
        for (col in 1:n_pts) {
                minimum_centroid = 1
                for (cent in 2:n_cent) {
                        if (dist_matrix[cent, col] <
                                dist_matrix[minimum_centroid, col]) {
                                minimum_centroid = cent
                        }
                                                }
                assig_matrix[minimum_centroid, col] = 1
        }
        assig_matrix
}
```

Por último, una vez se han asignado los puntos más próximos a cada clúster, se deben recalcular los centroides del mismo. Para recalcular un centroide en específico se deben tener en cuenta todos los puntos asociados al mismo. Las nuevas coordenadas del centroide se calculan realizando la media de los puntos que tiene asignados. El resultado de este algoritmo resulta en un nuevo dataframe donde, por cada fila, se van a tener las nuevas componentes del centroide correspondiente, y que serán usados para continuar con la ejecución del algoritmo.

Antes de ver el código, se ha de considerar que un centroide puede no tener ningún punto asociado, por lo que el cálculo que se propone arriba no sería posible (ya que se tendría que dividir entre cero a la hora de calcular la media). Es por ello que si un centroide no tiene ningún punto asociado, este se quedará igual (sus componentes no se verán alteradas). Esto resultará en que el centroide propuesto no se ha definido muy bien, y desemboca en una clasificación diferente a la esperada.

```
update_cent = function(df_sample, centroids, assig_matrix) {
        n_pts = ncol(assig_matrix)
                                        n_cent = nrow(assig_matrix)
        new_centroids = c()
        for (cent in 1:n_cent) {
                centroid_x = 0
                centroid_y = 0
                count = 0
                for (pt in 1:n_pts) {
                        if (assig_matrix[cent, pt] == 1) {
                                centroid_x = centroid_x + df_sample[pt, 1]
                                centroid_y = centroid_y + df_sample[pt, 2]
                                count = count + 1
                        }
                }
                if (count == 0){
                        new_centroids = append(append(new_centroids,
                         centroids[cent,1]),centroids[cent,2])
                } else {
                        new_centroids = append(append(new_centroids,
                                centroid_x/count), centroid_y/count)
                }
        }
        data.frame(t(matrix(new_centroids, 2, n_cent, dimnames=list(c("X","Y")))))
}
```

El algoritmo se ejecutará hasta que la matriz de asignaciones sea la misma en una iteración con respecto a su anterior. Para visualizar el resultado o clasificación final se propone una función que tomará la matriz de asignaciones final y el valor de los centroides finales. Este método imprimirá las coordenadas de cada centroide, así como los puntos que conforman el clúster representado por ese centroide.

```
printCentroids = function(df_sample, centroids, assig_matrix){
       n_cents = nrow(assig_matrix)
       n_pts = ncol(assig_matrix)
       cat("\n--RESULTADOS--\n")
       for (cent in 1:n_cents){
               cat(paste0("Centroide ", cent, " (",
                       centroids[cent,1], ", ", centroids[cent,2],") \n"))
               for(pt in 1:n_pts){
                       if (assig_matrix[cent,pt] == 1){
                               cat(paste0("  - Punto ", pt, " (",
                                       df_sample[pt, 1], ", ", df_sample[pt,2], ")\n"))
                       }
               }
               cat("\n")
       }
}
```

De todo este algoritmo, se ha considerado que la salida mas útil para el analista es un vector de clústerización. Este consiste en un vector de tantos elementos como puntos se hayan introducido, de tal forma que la posición $i-$ésima del mismo corresponda con el clúster asociado al punto i. Se itera la matriz de asignaciones final por puntos, y por cada uno de ellos se devuelve el valor de la fila que contiene un 1 (el número del centroide al que se ha asociado en la clasificación final)

```
create_clust_vector = function(assig_matrix){
        n_pts = ncol(assig_matrix)
        n_cents = nrow(assig_matrix)
        clust_vector = c()
        for(pt in 1:n_pts){
                for(cent in 1:n_cents){
                        if (assig_matrix[cent,pt] == 1){
                                clust_vector = append(clust_vector, cent)
                        }
                }
        }
        clust_vector
}
```

Todas estas funciones se recogen en la función `fcd_kmeans`. Esta función será la que implemente el algoritmo y a la que se deberá llamar. Recibirá como parámetros un dataframe con los datos de la muestra, otro dataframe con los centroides iniciales y un booleano `details` el cual se usará para recibir una salida más o menos detallada. En caso de que se quiera una salida detallada, se imprimirán paso por paso la matriz de distancias, la matriz de asignaciones y los valores de los nuevos centroides en ese paso concreto. En caso contrario simplemente se imprimirá la clasificación final (se llama a la función `printCentroids`) y se devolverá el vector de clústerización. Por defecto, `details` se encuentra a `FALSE`, ofreciendo una salida no detallada en caso de no indicar este parámetro en la llamada. El usuario podrá asignar a una variable la llamada a la función, recibiendo en la misma el vector de clústerización.

```
fcd_kmeans = function(df_sample, centroids, details = FALSE) {
        n_pts = nrow(df_sample)
        n_cent = nrow(centroids)
        prev_assig_matrix = data.frame(matrix(0, ncol = n_pts, nrow = n_cent))
        step = 1
        dist_matrix = create_distance_matrix(df_sample, centroids)
        assig_matrix = create_assigment_matrix(dist_matrix)
        if(details){
                cat("PASO", step, "\n------- \n")
                cat("Matriz de distancias \n")
                print(dist_matrix)
                cat("\nMatriz de asignaciones \n")
                print(assig_matrix)
```

```
                cat("\n")
        }
        while (!identical(prev_assig_matrix, assig_matrix)) {
                prev_assig_matrix = assig_matrix
                centroids = update_cent(df_sample, centroids, assig_matrix)
                dist_matrix = create_distance_matrix(df_sample, centroids)
                assig_matrix = create_assigment_matrix(dist_matrix)
                if(details) {
                        cat("Los nuevos centroides son:\n")
                                                print(centroids)
                        step = step + 1
                        cat("\nPASO", step, "\n------- \n")
                        cat("Matriz de distancias \n")
                        print(dist_matrix)
                        cat("\nMatriz de asignaciones \n")
                        print(assig_matrix)
                        cat("\n")
                }
        }
        printCentroids(df_sample, centroids, assig_matrix)
        cat("Vector de clusterización\n")
        create_clust_vector(assig_matrix)
}
```

Un aspecto a destacar es el uso de la función `identical` contenida en el paquete `base`. Esta función permite comparar si dos dataframes son iguales. Esto sirve en el algoritmo para ver si la matriz de asignaciones de la anterior iteración es igual que la actual. Esta condición será la que se deba cumplir para que el algoritmo finalice.

Con esto ya estaría el algoritmo implementado. Ahora se va a llamar a la función `fcd_kmeans` con los datos leídos al principio. Se va a optar por una salida detallada.

```
fcd_kmeans(data, centroides, TRUE)
```

```
## PASO 1
## -------
## Matriz de distancias
##         X1       X2       X3        X4       X5       X6        X7       X8
## 1 5.000000 2.761340 2.549510 1.2747549 4.100305 5.153882 0.7905694 2.512469
## 2 3.535534 1.274755 1.414214 1.4577380 2.657536 3.716517 1.2747549 1.250000
## 3 3.807887 2.573908 2.828427 0.3535534 2.926175 3.913119 0.7905694 2.657536
##         X9       X10       X11      X12      X13      X14      X15
## 1 4.802343 1.0000000 0.7071068 4.301163 2.500000 1.520691 2.061553
## 2 3.363406 1.5811388 1.0000000 2.828427 1.118034 2.015564 1.118034
## 3 3.579455 0.7071068 1.0000000 3.162278 2.500000 0.750000 2.500000
##
## Matriz de asignaciones
```

```
##   X1 X2 X3 X4 X5 X6 X7 X8 X9 X10 X11 X12 X13 X14 X15
## 1  0  0  0  0  0  0  1  0  0   0   1   0   0   0   0
## 2  1  1  1  0  1  1  0  1  1   0   0   1   1   0   1
## 3  0  0  0  1  0  0  0  0  0   1   0   0   0   1   0
##
## Los nuevos centroides son:
##       X     Y
## 1 1.125 0.875
## 2 1.825 3.575
## 3 1.750 0.500
##
## PASO 2
## -------
## Matriz de distancias
##         X1       X2       X3        X4       X5       X6        X7       X8
## 1 4.333734 2.404423 2.404423 0.6373774 3.432383 4.475628 0.1767767 2.298097
## 2 1.913439 1.123054 1.913439 2.8259954 1.187960 2.135708 2.8829239 1.676678
## 3 4.366062 2.926175 3.051639 0.2500000 3.482097 4.472136 0.5590170 2.915476
##         X9       X10       X11      X12      X13       X14      X15
## 1 4.126894 0.5303301 0.1767767 3.644345 2.215006 1.0752907 1.976424
## 2 1.805893 3.0921271 2.7039323 1.249500 1.444386 3.3296021 2.118077
## 3 4.138236 0.2500000 0.9013878 3.716517 2.795085 0.3535534 2.657536
##
## Matriz de asignaciones
##   X1 X2 X3 X4 X5 X6 X7 X8 X9 X10 X11 X12 X13 X14 X15
## 1  0  0  0  0  0  0  1  0  0   0   1   0   0   0   1
## 2  1  1  1  0  1  1  0  1  1   0   0   1   1   0   0
## 3  0  0  0  1  0  0  0  0  0   1   0   0   0   1   0
##
## Los nuevos centroides son:
##          X        Y
## 1 0.750000 1.416667
## 2 2.027778 3.694444
## 3 1.750000 0.500000
##
## PASO 3
## -------
## Matriz de distancias
##         X1       X2       X3       X4        X5       X6        X7       X8
## 1 4.131518 1.833333 1.751983 1.201850 3.2414417 4.301970 0.8333333 1.660405
## 2 1.678201 1.352866 2.143394 2.957518 0.9738082 1.901307 3.0454378 1.908598
## 3 4.366062 2.926175 3.051639 0.250000 3.4820971 4.472136 0.5590170 2.915476
##         X9      X10       X11      X12      X13       X14      X15
## 1 3.948453 1.184389 0.4859127 3.425801 1.602949 1.7098570 1.317616
## 2 1.573557 3.237750 2.8838096 1.019108 1.678201 3.4445564 2.353419
## 3 4.138236 0.250000 0.9013878 3.716517 2.795085 0.3535534 2.657536
##
```

```
## Matriz de asignaciones
##   X1 X2 X3 X4 X5 X6 X7 X8 X9 X10 X11 X12 X13 X14 X15
## 1  0  0  1  0  0  0  0  1  0   0   1   0   1   0   1
## 2  1  1  0  0  1  1  0  0  1   0   0   1   0   0   0
## 3  0  0  0  1  0  0  1  0  0   1   0   0   0   1   0
##
## Los nuevos centroides son:
##          X        Y
## 1 0.350000 2.500000
## 2 2.916667 4.041667
## 3 1.625000 0.562500
##
## PASO 4
## -------
## Matriz de distancias
##          X1       X2        X3       X4        X5        X6        X7       X8
## 1 3.7312866 0.850000 0.6103278 2.241093 2.9300171 3.9446166 1.9678669 0.509902
## 2 0.7418539 2.306768 3.0970977 3.492303 0.3033379 0.9510594 3.6895592 2.862897
## 3 4.3611388 2.826355 2.9295104 0.225347 3.4714235 4.4743191 0.4192627 2.798577
##          X9       X10       X11       X12       X13       X14      X15
## 1 3.6034705 2.3070544 1.6347783 3.0450780 0.5220153 2.7901613 0.350000
## 2 0.6194195 3.8144917 3.5951839 0.0931695 2.6316054 3.9008991 3.299042
## 3 4.1368202 0.1397542 0.7629097 3.7023008 2.6845914 0.4881406 2.528741
##
## Matriz de asignaciones
##   X1 X2 X3 X4 X5 X6 X7 X8 X9 X10 X11 X12 X13 X14 X15
## 1  0  1  1  0  0  0  0  1  0   0   0   0   1   0   1
## 2  1  0  0  0  1  1  0  0  1   0   0   1   0   0   0
## 3  0  0  0  1  0  0  1  0  0   1   1   0   0   1   0
##
## Los nuevos centroides son:
##      X    Y
## 1 0.30 2.95
## 2 3.35 4.20
## 3 1.50 0.65
##
## PASO 5
## -------
## Matriz de distancias
##          X1        X2        X3        X4        X5       X6        X7
## 1 3.5556293 0.5408327 0.3041381 2.6348624 2.8160256 3.782195 2.3963514
## 2 0.3354102 2.7681221 3.5584407 3.8029594 0.5700877 0.500000 4.0388736
## 3 4.3384905 2.7060118 2.7879204 0.2692582 3.4438351 4.459260 0.2692582
##           X8        X9       X10       X11       X12       X13       X14
## 1 0.07071068 3.4539832 2.728095 2.0718349 2.8969812 0.2061553 3.1906112
## 2 3.32415403 0.1581139 4.136726 3.9702015 0.4031129 3.0923292 4.1743263
## 3 2.66176633 4.1182521 0.150000 0.6103278 3.6704904 2.5539186 0.6403124
```

```
##         X15
## 1 0.5408327
## 2 3.7566608
## 3 2.3817011
##
## Matriz de asignaciones
##   X1 X2 X3 X4 X5 X6 X7 X8 X9 X10 X11 X12 X13 X14 X15
## 1  0  1  1  0  0  0  0  1  0   0   0   0   1   0   1
## 2  1  0  0  0  1  1  0  0  1   0   0   1   0   0   0
## 3  0  0  0  1  0  0  1  0  0   1   1   0   0   1   0
##
##
## --RESULTADOS--
## Centroide 1 (0.3, 2.95)
##   - Punto 2 (0.75, 3.25)
##   - Punto 3 (0, 3)
##   - Punto 8 (0.25, 3)
##   - Punto 13 (0.5, 3)
##   - Punto 15 (0, 2.5)
##
## Centroide 2 (3.35, 4.2)
##   - Punto 1 (3.5, 4.5)
##   - Punto 5 (3, 3.75)
##   - Punto 6 (3.75, 4.5)
##   - Punto 9 (3.5, 4.25)
##   - Punto 12 (3, 4)
##
## Centroide 3 (1.5, 0.65)
##   - Punto 4 (1.75, 0.75)
##   - Punto 7 (1.25, 0.75)
##   - Punto 10 (1.5, 0.5)
##   - Punto 11 (1, 1)
##   - Punto 14 (2, 0.25)
##
## Vector de clusterización
##  [1] 2 1 1 3 2 2 3 1 2 3 3 2 1 3 1
```

Se ha optado por tres centroides que más o menos se encuentran entre el conjunto total de puntos. Han sido necesarios 4 pasos para llegar a una solución. Se puede ver cómo se han ido calculando las matrices de distancias y asignaciones, así como los nuevos centroides, hasta que se ha llegado a los resultados finales. En estos se aprecian las coordenadas de los tres centroides así como los puntos que contienen cada uno. Por último, se devuelve el vector de clústerización que también se puede ver en la salida de la función realizada.

2.1.2. Clusterización jerárquica algomerativa

Ejercicio 2.1.2. *El conjunto de datos de datos que se empleará para realizar el análisis de clasificación no supervisada con Clusterización Jerárquica Aglomerativa será el mismo que el utilizado en el Ejercicio 2.1.1, pero el este ejercicio le denominaremos segundo conjunto de datos.*

Para la resolución de este ejercicio se implementará una variación del algoritmo visto en clase, que ayudará a simplificar la codificación y el entendimiento del algoritmo. Se utilizarán algunas funciones ya explicadas en otros ejercicios y prácticas, como `len`, la media, la distancia euclídea, la desviación típica, y la covarianza. El primer paso del algoritmo consiste en calcular la distancia entre cada posible pareja de puntos.

```
create_distance_matrix = function(df) {
        n = len(df[,1])
        empty_matrix = matrix(0, ncol = n, nrow = n)
        distances = data.frame(empty_matrix)
        for (i in 1:n) {
                for (j in 1:i) {
                        distances[i, j] = euc_distance(df[i,], df[j,])
                        if (i != j) distances[j, i] = NA
                }
        }
        distances
}
```

Esto se logra mediante la función `create_distance_matrix` donde la entrada a_{ij} representa el valor de $\text{dist}(x_i, x_j)$. Nótese que $\text{dist}(x_i, x_j) = \text{dist}(x_j, x_i)$, por lo que $a_{ij} = a_{ji}$, y por tanto la matriz es simétrica. Para ahorrar cálculos y simplificar el problema, se ignorarán las entradas a_{ij} con $j > i$. Durante la explicación de este ejercicio se ejemplificará con los datos del ejercicio visto en teoría, pues los datos de laboratorio son mucho más extensos. De acuerdo a esto, la matriz de distancias se vería como la siguiente.

$$\begin{array}{c} \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{array} \begin{array}{c} \begin{array}{cccccc} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \end{array} \\ \begin{pmatrix} 0 & \times & \times & \times & \times & \times \\ 4{,}15 & 0 & \times & \times & \times & \times \\ 3{,}39 & 4{,}13 & 0 & \times & \times & \times \\ 5{,}36 & 2{,}8 & 3{,}21 & 0 & \times & \times \\ 3{,}41 & 3{,}42 & 0{,}76 & 2{,}53 & 0 & \times \\ 3{,}29 & 1{,}05 & 3{,}15 & 2{,}61 & 2{,}48 & 0 \end{pmatrix} \end{array}$$

Además, en cada iteración del algoritmo se elegirán una/s serie de distancias de esta última matriz mostrada, necesitando saber cuáles han sido ya elegidas en iteraciones previas. Para ello se emplea una matriz de las mismas dimensiones que almacena ceros y unos indicando si dicha entrada ha sido seleccionada previamente. La creación de esta matriz inicial queda recogida en la función `create_chosen_matrix`, y un ejemplo de esta matriz sería el siguiente.

$$
\begin{array}{c} \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{array}
\begin{array}{c}
\begin{array}{cccccc} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \end{array} \\
\begin{pmatrix}
0 & \times & \times & \times & \times & \times \\
4{,}15 & 0 & \times & \times & \times & \times \\
3{,}39 & 4{,}13 & 0 & \times & \times & \times \\
5{,}36 & 2{,}8 & 3{,}21 & 0 & \times & \times \\
3{,}41 & 3{,}42 & \boxed{0{,}76} & 2{,}53 & 0 & \times \\
3{,}29 & 1{,}05 & 3{,}15 & 2{,}61 & 2{,}48 & 0
\end{pmatrix}
\end{array}
\qquad
\begin{array}{c} \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{array}
\begin{array}{c}
\begin{array}{cccccc} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \end{array} \\
\begin{pmatrix}
\times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times \\
0 & 0 & \times & \times & \times & \times \\
0 & 0 & 0 & \times & \times & \times \\
0 & 0 & 1 & 0 & \times & \times \\
0 & 0 & 0 & 0 & 0 & \times
\end{pmatrix}
\end{array}
$$

```
create_chosen_matrix = function(df) {
        n = len(df[, 1])
        empty_matrix = matrix(0, ncol = n, nrow = n)
        chosen = data.frame(empty_matrix)
        for (i in 1:n) {
                for (j in 1:i) {
                        chosen[j, i] = NA
                }
        }
        chosen
}
```

Finalmente, el último elemento que falta para poder dar comienzo al algoritmo es buscar una forma de llevar un control de la estructura de los clústers, mostrado en el siguiente ejemplo como etiquetas en las filas y las columnas de las matrices.

$$
\begin{array}{c} \\ x_1 \\ C_3 \supset C_2 \supset x_2 \\ C_1 \supset x_3 \\ C_3 \supset x_4 \\ C_1 \supset x_5 \\ C_3 \supset C_2 \supset x_6 \end{array}
\begin{array}{c}
\begin{array}{ccccccc} x_1 & C_3 \supset C_2 \supset x_2 & C_1 \supset x_3 & C_3 \supset x_4 & C_1 \supset x_5 & C_3 \supset C_2 \supset x_6 \end{array} \\
\begin{pmatrix}
0 & \times & \times & \times & \times & \times \\
4{,}27 & 0 & \times & \times & \times & \times \\
3{,}4 & 3{,}15 & 0 & \times & \times & \times \\
4{,}27 & \boxed{2{,}71} & 3{,}15 & 0 & \times & \times \\
3{,}4 & 3{,}15 & \boxed{0{,}76} & 3{,}15 & 0 & \times \\
4{,}27 & \boxed{1{,}05} & 3{,}15 & \boxed{2{,}71} & 3{,}15 & 0
\end{pmatrix}
\end{array}
\tag{2.1}
$$

Se puede observar que aunque los clústers se comporten como conjuntos, la estructura de datos más adecuada para almacenar cada una de las etiquetas es un árbol. Como se tiene que trabajar con un conjunto de n árboles, se estará trabajando con un bosque (vector de árboles). Para hacer esto posible se hará uso del paquete `data.tree`. Al ser la matriz simétrica, el bosque de las filas será igual al de las columnas. Para el último ejemplo mostrado, el bosque tendría el aspecto del mostrado a continuación. Obsérvese que al finalizar el algoritmo, todos los árboles del bosque deberán ser el mismo.

$$\texttt{forest} = \left[x_1, \; \begin{array}{c} C_3 \\ x_4 \quad C_2 \\ \quad\; x_2 \quad x_6 \end{array}, \; \begin{array}{c} C_1 \\ x_3 \quad x_5 \end{array}, \; \begin{array}{c} C_3 \\ x_4 \quad C_2 \\ \quad\; x_2 \quad x_6 \end{array}, \; \begin{array}{c} C_1 \\ x_3 \quad x_5 \end{array}, \; \begin{array}{c} C_3 \\ x_4 \quad C_2 \\ \quad\; x_2 \quad x_6 \end{array} \right] \tag{2.2}$$

Una vez se han presentado las estructuras necesarias para realizar el algoritmo, se explicará el funcionamiento de este. El flujo principal queda recogido en la función `fcd_ahc`.

```
fcd_ahc = function(data, criteria, details = FALSE) {
        dist_matrix = create_distance_matrix(data)
        chosen_matrix = create_chosen_matrix(data)
        first_dist_matrix = dist_matrix
        next_cluster = 1
        forest = sapply(colnames(dist_matrix), function(root){Node$new(root)})
        while(sum(chosen_matrix, na.rm = TRUE) !=
                (len(dist_matrix) * (len(dist_matrix) - 1) / 2)){
                min_index = min_ahc(dist_matrix, chosen_matrix)
                chosen_matrix = choose_distance(dist_matrix, chosen_matrix,
                         dist_matrix[min_index[1], min_index[2]])
                new_tree_tag = Node$new(paste0("C", next_cluster))
                new_tree_tag$AddChildNode(forest[[min_index[1]]])
                new_tree_tag$AddChildNode(forest[[min_index[2]]])
                trees_2_update = sapply(Traverse(new_tree_tag,
                filterFun = isLeaf), function(x){
                        as.integer(substring(x$name, 2))
                        })
                for(i in 1:len(forest)){
                        if(i %in% trees_2_update){
                                forest[[i]] = new_tree_tag
                        }
                }
                next_cluster = next_cluster + 1
                criteria_name = switch(
                        criteria, "MIN" = min, "MAX" = max, "AVG" = fcd_mean)
                dist_matrix = update_distance_matrix(first_dist_matrix,
                        dist_matrix, forest, trees_2_update, criteria_name)
                if (details) {
                        cat("\nIteración", next_cluster - 1)
                        cat("============\n")
                        cat("\n\nMatriz de distancias\n")
```

```
                        print(dist_matrix)
                        cat("\n\nMatriz de distancias elegidas\n")
                        print(chosen_matrix)
                        cat("\n\nDendrogramas\n")
                        print(forest)
                }
        }
        cat("\n\nMatriz cofenética\n")
        print(dist_matrix)
        cat("\nDendrograma final\n")
        print(forest[[1]])
        cpcc = calculate_cpcc(first_dist_matrix, dist_matrix)
        cat("\n\nCoeficiente de CPCC: ", cpcc)
        plot(forest[[1]])
        plot(as.dendrogram(forest[[1]]), center = TRUE, yaxt='n')
}
```

A continuación se irá explicando esta función paso a paso de manera que se entienda cómo funciona el algoritmo. En primer lugar se inician las estructuras de datos comentadas anteriormente. Se observa que el instante inicial, el bosque debe ser de la forma $[x_1, x_2, \ldots, x_n]$. Además, en la variable `next_cluster` se almacena el índice del siguiente clúster que se creará.

El cuerpo del algoritmo transcurre en un bucle. El algoritmo finaliza cuando todos los elementos de la triangular inferior de la matriz han sido seleccionados. Como tanto la matriz de distancias, como la de elegidos son matrices $n \times n$ y solo se trabaja con la triangular inferior, una forma sencilla de comprobar si el algoritmo se encuentra en este punto es comprobar si la suma de la triangular inferior de la matriz de elegidos es igual al número de elementos de la triangular inferior de la matriz de distancias. Como las matrices son cuadradas, la primera columna tendrá n elementos, la segunda $n-1$, y así hasta la última que tendrá un único elemento. Haciendo uso de la igualdad

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2},$$

se puede deducir que el algoritmo finaliza cuando la suma de los elementos de la triangular inferior de la matriz de elegidos sea igual a $\frac{n(n-1)}{2}$, pues se ignoran los elementos de la diagonal principal.

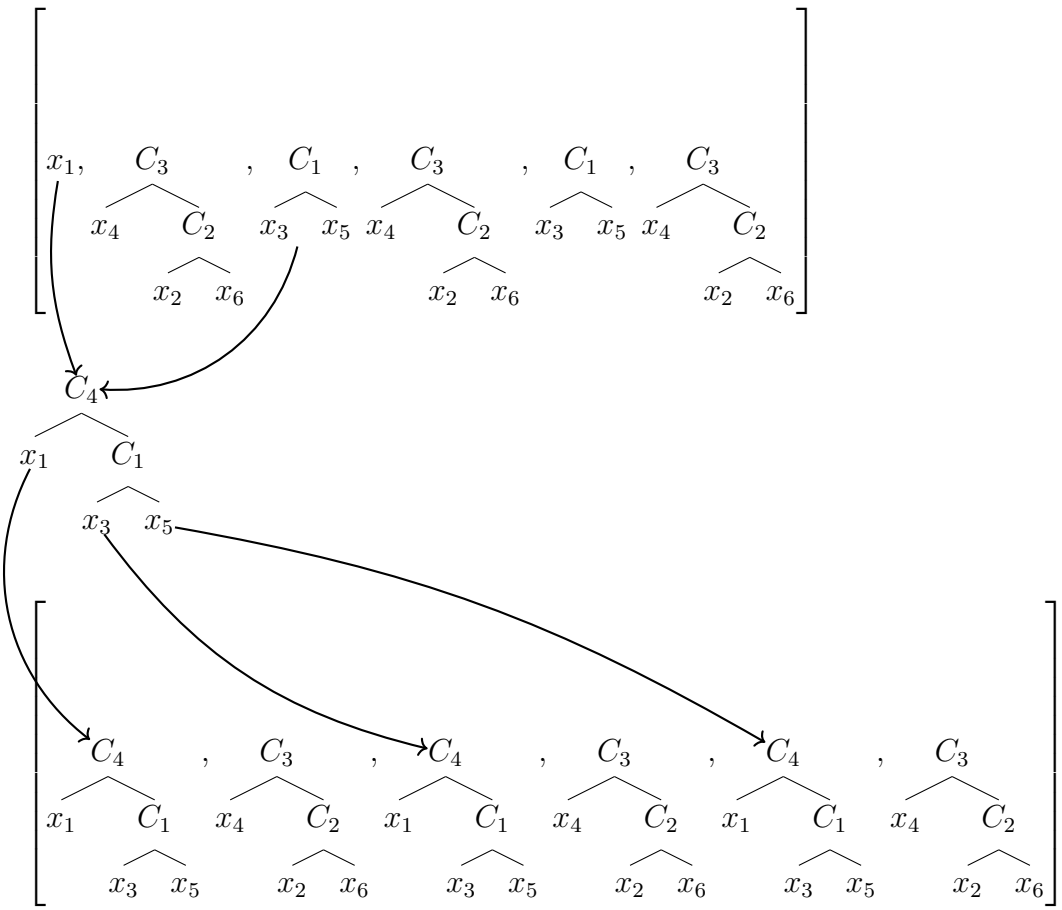
En cada iteración del algoritmo se elige la menor de las distancias que no haya sido elegida previamente y se marca. Si la misma distancia aparece más de una vez, se marcan todas sus apariciones en una misma iteración. Las funciones `min_ahc` y `choose_distance` realizan esta tarea.

```
min_ahc = function(distances, chosen) {
        min_d = Inf
        min_c = c()
        for(i in 1:nrow(distances)) {
                for(j in 1:i) {
                        if (distances[i, j] < min_d
                         && distances[i, j] != 0 && chosen[i, j] == 0) {
                                min_d = distances[i, j]
                                min_c = c(i, j)
                        }
                }
        }
        min_c
}

choose_distance = function(distances, chosen, distance){
        for(i in 1:nrow(distances)) {
                for(j in 1:i) {
                        if (distances[i, j] == distance) {
                                chosen[i, j] = 1
                        }
                }
        }
        chosen
}
```

La primera de las funciones devuelve el índice de fila y columna de la menor de las distancias, y la siguiente, marca en la matriz de elegidos todas las apariciones de esta distancia. Que se haya seleccionado el valor a_{ij} indica que el nuevo clúster C_i, contendrá a las raíces de los árboles A_i y A_j, sustituyéndose los árboles A_i y A_j por este.

Sin embargo, esto sólo sucede así en el caso base de que A_i y A_j son árboles que solo contienen a un único x_i. En caso contrario, se puede observar que deberán ser actualizadas todas las etiquetas de la matriz cuya columna esté en el árbol de la entrada seleccionada (es decir, si se elige a_{35} pero x_2 estaba involucrado, también deberá actualizarse la etiqueta de la segunda columna además de la de la tercera y la quinta). Veamos en el ejemplo de la Ecuación (2.2) qué sucedería si se selecciona la entrada a_{13} (ignorando si esto es correcto). En primer lugar, los árboles A_1 y A_3 se sustituyen por aquel formado por ambos. Sin embargo, también deberá asignarse este nuevo árbol a A_5, pues x_5 está también en este nuevo árbol. En general, para saber qué etiquetas han de actualizarse, si se marca a_{ij}, se crea un nuevo árbol con A_i y A_j, se recorre, y para cada una de los nodos n_k si n_k es hoja y contiene a x_m, el nuevo árbol es asignado a A_m.



Una vez definida la estructura de los clústers en este punto, es momento de actualizar los valores de la matriz de distancias. Se puede observar que los valores marcados nunca deberán cambiar, y que deberán ser actualizadas aquellas entradas cuya etiqueta de fila o columna haya sido actualizada, pues de lo contrario, se tratará de un punto que no ha intervenido en la última fusión de clústers. Esta actualización se realiza en la función `update_distance_matrix`.

```
update_distance_matrix = function(
        initial_dm, actual_dm, forest, new_cluster_points, criteria_name) {
        updated_cols = c(rep(0, times = len(forest)))
        updated_cols[new_cluster_points] = 1

        for (i in 1:len(forest)) {
                if (!updated_cols[i]) {
                        tree_points = sapply(Traverse(forest[[i]],
                                filterFun = isLeaf), function(x){
                                as.integer(substring(x$name, 2))})
                        new_distance = func_criteria(initial_dm,
                                new_cluster_points, tree_points,
```

```
                                criteria_name)

                        for (i in new_cluster_points) {
                                for (j in tree_points) {
                                        minor = min(i, j)
                                        mayor = max(i, j)
                                        actual_dm[mayor, minor] =
                                                new_distance
                                }
                        }
                        updated_cols[tree_points] = 1
                                        }
                                }
        actual_dm
}
```

Esta función calcula las entradas cuyas distancias deberán ser actualizadas, y de entre ellas, va asignando a cada una aquellas cuyas distancias deberán ser iguales. A la hora de calcular las nuevas distancias se puede elegir entre tres criterios:

- `MIN`:
$$\text{dist}(C_i, C_j) = \min_{(x,y)\in C_i \times C_j} \{\text{dist}(x, y)\}$$
- `MAX`:
$$\text{dist}(C_i, C_j) = \max_{(x,y)\in C_i \times C_j} \{\text{dist}(x, y)\}$$
- `AVG`:
$$\text{dist}(C_i, C_j) = \frac{\sum_{x\in C_i} \sum_{y\in C_j} \text{dist}(x, y)}{|C_i \times C_j|}$$

El cálculo según diferentes criterios se realiza en la siguiente función.

```
func_criteria = function(initial_dm, new_cluster_points,
        tree_points, criteria_name) {
        distances = c()
        for (i in new_cluster_points) {
                for (j in tree_points) {
                        minor = min(i, j)
                        mayor = max(i, j)
                        distances = c(distances, initial_dm[mayor, minor])
                }
        }
        criteria_name(distances)
}
```

Adicionalmente, en caso de que el usuario lo desee, se va mostrando la traza del algoritmo para ver paso a paso qué sucede. Cuando acaba el bucle `while` en la variable que almacenaba la matriz de distancias, queda la matriz cofenética, muy útil para calcular el CPCC, que indica cómo de buena ha sido la clusterización. Se calcula mediante las ecuaciones mostradas a continuación y dichos cálculos se realizan en la función `calculate_cpcc`.

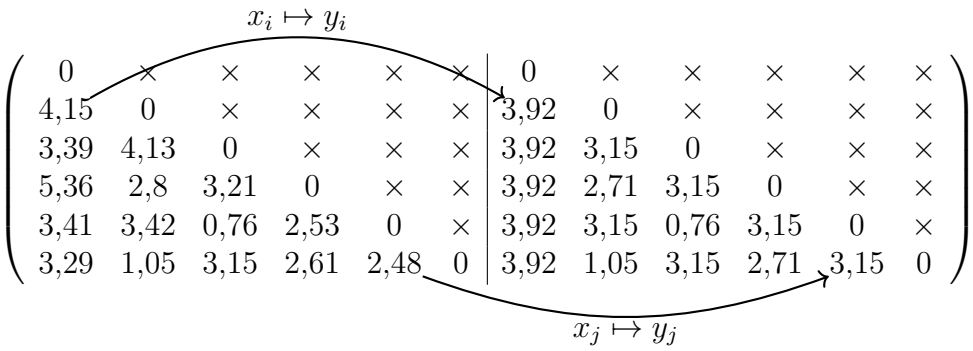
$$CPCC = \frac{S_{xy}}{S_x S_y}$$

$$S_{xy} = \sum_{i=1}^{n} \frac{x_i y_i}{n} - \bar{x}\bar{y}$$

$$S_x = \sqrt{\sum_{i=1}^{n} \frac{(x_i - \bar{x})^2}{n}}$$

$$S_y = \sqrt{\sum_{i=1}^{n} \frac{(y_i - \bar{y})^2}{n}}$$

Queda por definir quiénes son las variables x e y. Estas representan los valores de la matriz de distancias inicial y final. Cada x_i tiene un y_i asociado, por ejemplo, supongamos que estas son la matriz inicial y final de una ejecución del algoritmo:



$$\left(\begin{array}{cccccc|cccccc}
0 & \times & \times & \times & \times & \times & 0 & \times & \times & \times & \times & \times \\
4{,}15 & 0 & \times & \times & \times & \times & 3{,}92 & 0 & \times & \times & \times & \times \\
3{,}39 & 4{,}13 & 0 & \times & \times & \times & 3{,}92 & 3{,}15 & 0 & \times & \times & \times \\
5{,}36 & 2{,}8 & 3{,}21 & 0 & \times & \times & 3{,}92 & 2{,}71 & 3{,}15 & 0 & \times & \times \\
3{,}41 & 3{,}42 & 0{,}76 & 2{,}53 & 0 & \times & 3{,}92 & 3{,}15 & 0{,}76 & 3{,}15 & 0 & \times \\
3{,}29 & 1{,}05 & 3{,}15 & 2{,}61 & 2{,}48 & 0 & 3{,}92 & 1{,}05 & 3{,}15 & 2{,}71 & 3{,}15 & 0
\end{array}\right)$$

```
calculate_cpcc = function(first_dist_matrix, dist_matrix) {
        x = unlist(first_dist_matrix)
        x = x[!is.na(x) & x != 0]

        y = unlist(dist_matrix)
        y = y[!is.na(y) & y != 0]

        sx = standard_dev(x)
        sy = standard_dev(y)
        sxy = covariance(x, y)

        sxy / (sx * sy)
}
```

Para finalizar, se muestra un gráfico del árbol final (los n árboles acaban igual) y un dendrograma de la clusterización resultante.

```
(sample = read.xlsx("data/datosKMeans.xlsx"))
```

```
##    Velocidad Temperatura
## 1       3.50        4.50
## 2       0.75        3.25
## 3       0.00        3.00
## 4       1.75        0.75
## 5       3.00        3.75
## 6       3.75        4.50
## 7       1.25        0.75
## 8       0.25        3.00
## 9       3.50        4.25
## 10      1.50        0.50
## 11      1.00        1.00
## 12      3.00        4.00
## 13      0.50        3.00
## 14      2.00        0.25
## 15      0.00        2.50
```

```
fcd_ahc(sample, "MIN")
```

```
##
##
## Matriz cofenética
##          X1        X2       X3        X4       X5       X6        X7       X8
## 1  0.000000        NA       NA        NA       NA       NA        NA       NA
## 2  2.304886 0.0000000       NA        NA       NA       NA        NA       NA
## 3  2.304886 0.3535534 0.000000        NA       NA       NA        NA       NA
## 4  2.304886 1.8027756 1.802776 0.0000000       NA       NA        NA       NA
## 5  0.250000 2.3048861 2.304886 2.3048861 0.000000       NA        NA       NA
## 6  0.250000 2.3048861 2.304886 2.3048861 0.250000 0.000000        NA       NA
## 7  2.304886 1.8027756 1.802776 0.3535534 2.304886 2.304886 0.0000000       NA
## 8  2.304886 0.2500000 0.250000 1.8027756 2.304886 2.304886 1.8027756 0.000000
## 9  0.250000 2.3048861 2.304886 2.3048861 0.250000 0.250000 2.3048861 2.304886
## 10 2.304886 1.8027756 1.802776 0.3535534 2.304886 2.304886 0.3535534 1.802776
## 11 2.304886 1.8027756 1.802776 0.3535534 2.304886 2.304886 0.3535534 1.802776
## 12 0.559017 2.3048861 2.304886 2.3048861 0.250000 0.559017 2.3048861 2.304886
## 13 2.304886 0.3535534 0.500000 1.8027756 2.304886 2.304886 1.8027756 0.250000
## 14 2.304886 0.5590170 0.559017 0.5590170 2.304886 2.304886 0.5590170 0.559017
## 15 2.304886 0.5000000 0.500000 1.8027756 2.304886 2.304886 1.8027756 0.500000
##          X9       X10      X11      X12      X13      X14 X15
## 1        NA        NA       NA       NA       NA       NA  NA
## 2        NA        NA       NA       NA       NA       NA  NA
## 3        NA        NA       NA       NA       NA       NA  NA
## 4        NA        NA       NA       NA       NA       NA  NA
```

```
## 5        NA        NA       NA       NA       NA       NA  NA
## 6        NA        NA       NA       NA       NA       NA  NA
## 7        NA        NA       NA       NA       NA       NA  NA
## 8        NA        NA       NA       NA       NA       NA  NA
## 9  0.000000        NA       NA       NA       NA       NA  NA
## 10 2.304886 0.0000000       NA       NA       NA       NA  NA
## 11 2.304886 0.3535534 0.000000       NA       NA       NA  NA
## 12 0.559017 2.3048861 2.304886 0.000000       NA       NA  NA
## 13 2.304886 1.8027756 1.802776 2.304886 0.000000       NA  NA
## 14 2.304886 0.5590170 0.559017 2.304886 0.559017 0.000000  NA
## 15 2.304886 1.8027756 1.802776 2.304886 0.500000 0.559017   0
##
## Dendrograma final
##                          levelName
## 1  C14
## 2   ¦--C13
## 3   ¦   ¦--X14
## 4   ¦   °--C12
## 5   ¦       ¦--C5
## 6   ¦       ¦   ¦--X11
## 7   ¦       ¦   °--C4
## 8   ¦       ¦       ¦--X7
## 9   ¦       ¦       °--C3
## 10  ¦       ¦           ¦--X10
## 11  ¦       ¦           °--X4
## 12  ¦       °--C9
## 13  ¦           ¦--X15
## 14  ¦           °--C8
## 15  ¦               ¦--X8
## 16  ¦               °--C7
## 17  ¦                   ¦--C6
## 18  ¦                   ¦   ¦--X13
## 19  ¦                   ¦   °--X3
## 20  ¦                   °--X2
## 21  °--C11
## 22      ¦--X5
## 23      °--C10
## 24          ¦--X12
## 25          °--C2
## 26              ¦--X9
## 27              °--C1
## 28                  ¦--X6
## 29                  °--X1
##
##
## Coeficiente de CPCC:  0.8701407
```

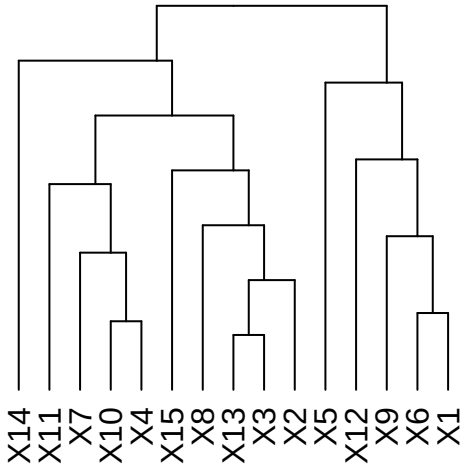


Figura 2.1: Clusterización con `MIN`

```
fcd_ahc(sample, "MAX")
```

```
##
##
## Matriz cofenética
##           X1        X2        X3       X4       X5        X6       X7       X8
## 1  0.0000000        NA        NA       NA       NA        NA       NA       NA
## 2  4.5961941 0.0000000        NA       NA       NA        NA       NA       NA
## 3  4.5961941 0.7905694 0.0000000       NA       NA        NA       NA       NA
## 4  4.5961941 2.8504386 2.8504386 0.000000       NA        NA       NA       NA
## 5  1.0606602 4.5961941 4.5961941 4.596194 0.000000        NA       NA       NA
## 6  0.2500000 4.5961941 4.5961941 4.596194 1.060660 0.0000000       NA       NA
## 7  4.5961941 2.8504386 2.8504386 0.500000 4.596194 4.5961941 0.000000       NA
## 8  4.5961941 0.5590170 0.7905694 2.850439 4.596194 4.5961941 2.850439 0.000000
## 9  0.3535534 4.5961941 4.5961941 4.596194 1.060660 0.3535534 4.596194 4.596194
## 10 4.5961941 3.4003676 3.4003676 3.400368 4.596194 4.5961941 3.400368 3.400368
## 11 4.5961941 3.4003676 3.4003676 3.400368 4.596194 4.5961941 3.400368 3.400368
## 12 0.9013878 4.5961941 4.5961941 4.596194 1.060660 0.9013878 4.596194 4.596194
## 13 4.5961941 2.8504386 2.8504386 2.573908 4.596194 4.5961941 2.573908 2.850439
## 14 4.5961941 3.4003676 3.4003676 3.400368 4.596194 4.5961941 3.400368 3.400368
## 15 4.5961941 2.8504386 2.8504386 2.474874 4.596194 4.5961941 2.474874 2.850439
##           X9       X10      X11      X12      X13      X14 X15
## 1         NA        NA       NA       NA       NA       NA  NA
```

```
## 2         NA        NA       NA       NA       NA       NA  NA
## 3         NA        NA       NA       NA       NA       NA  NA
## 4         NA        NA       NA       NA       NA       NA  NA
## 5         NA        NA       NA       NA       NA       NA  NA
## 6         NA        NA       NA       NA       NA       NA  NA
## 7         NA        NA       NA       NA       NA       NA  NA
## 8         NA        NA       NA       NA       NA       NA  NA
## 9  0.0000000        NA       NA       NA       NA       NA  NA
## 10 4.5961941 0.0000000       NA       NA       NA       NA  NA
## 11 4.5961941 0.7071068 0.000000       NA       NA       NA  NA
## 12 0.9013878 4.5961941 4.596194 0.000000       NA       NA  NA
## 13 4.5961941 3.4003676 3.400368 4.596194 0.000000       NA  NA
## 14 4.5961941 1.2500000 1.250000 4.596194 3.400368 0.000000  NA
## 15 4.5961941 3.4003676 3.400368 4.596194 2.573908 3.400368   0
##
## Dendrograma final
##                    levelName
## 1  C14
## 2   ¦--C13
## 3   ¦   ¦--C9
## 4   ¦   ¦   ¦--X14
## 5   ¦   ¦   °--C5
## 6   ¦   ¦       ¦--X11
## 7   ¦   ¦       °--X10
## 8   ¦   °--C12
## 9   ¦       ¦--C11
## 10  ¦       ¦   ¦--X13
## 11  ¦       ¦   °--C10
## 12  ¦       ¦       ¦--X15
## 13  ¦       ¦       °--C3
## 14  ¦       ¦           ¦--X7
## 15  ¦       ¦           °--X4
## 16  ¦       °--C6
## 17  ¦           ¦--X3
## 18  ¦           °--C4
## 19  ¦               ¦--X8
## 20  ¦               °--X2
## 21  °--C8
## 22      ¦--X5
## 23      °--C7
## 24          ¦--X12
## 25          °--C2
## 26              ¦--X9
## 27              °--C1
## 28                  ¦--X6
## 29                  °--X1
##
```

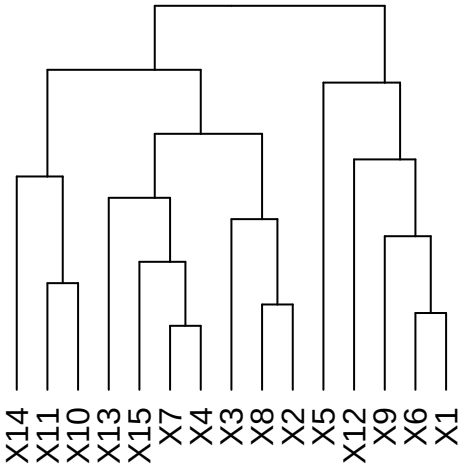


Figura 2.2: Clusterización con `MAX`

```
##
## Coeficiente de CPCC:  0.8157924
```

```
fcd_ahc(sample, "AVG")

##
##
## Matriz cofenética
##           X1        X2        X3        X4        X5        X6        X7
## 1  0.0000000        NA        NA        NA        NA        NA        NA
## 2  3.6662879 0.0000000        NA        NA        NA        NA        NA
## 3  3.6662879 0.4110307 0.0000000        NA        NA        NA        NA
## 4  3.6662879 2.6284228 2.6284228 0.0000000        NA        NA        NA
## 5  0.7297887 3.6662879 3.6662879 3.6662879 0.0000000        NA        NA
## 6  0.2500000 3.6662879 3.6662879 3.6662879 0.7297887 0.0000000        NA
## 7  3.6662879 2.6284228 2.6284228 0.4267767 3.6662879 3.6662879 0.0000000
## 8  3.6662879 0.5590170 0.4110307 2.6284228 3.6662879 3.6662879 2.6284228
## 9  0.3017767 3.6662879 3.6662879 3.6662879 0.7297887 0.3017767 3.6662879
## 10 3.6662879 2.6284228 2.6284228 0.3535534 3.6662879 3.6662879 0.4267767
## 11 3.6662879 2.6284228 2.6284228 0.6170765 3.6662879 3.6662879 0.6170765
## 12 0.7225039 3.6662879 3.6662879 3.6662879 0.7297887 0.7225039 3.6662879
## 13 3.6662879 0.4110307 0.5000000 2.6284228 3.6662879 3.6662879 2.6284228
```

```
## 14 3.6662879 2.6284228 2.6284228 0.8173555 3.6662879 3.6662879 0.8173555
## 15 3.6662879 0.7066960 0.7066960 2.6284228 3.6662879 3.6662879 2.6284228
##           X8        X9       X10       X11      X12      X13      X14 X15
## 1         NA        NA        NA        NA       NA       NA       NA  NA
## 2         NA        NA        NA        NA       NA       NA       NA  NA
## 3         NA        NA        NA        NA       NA       NA       NA  NA
## 4         NA        NA        NA        NA       NA       NA       NA  NA
## 5         NA        NA        NA        NA       NA       NA       NA  NA
## 6         NA        NA        NA        NA       NA       NA       NA  NA
## 7         NA        NA        NA        NA       NA       NA       NA  NA
## 8  0.0000000        NA        NA        NA       NA       NA       NA  NA
## 9  3.6662879 0.0000000        NA        NA       NA       NA       NA  NA
## 10 2.6284228 3.6662879 0.0000000        NA       NA       NA       NA  NA
## 11 2.6284228 3.6662879 0.6170765 0.0000000       NA       NA       NA  NA
## 12 3.6662879 0.7225039 3.6662879 3.6662879 0.000000       NA       NA  NA
## 13 0.4110307 3.6662879 2.6284228 2.6284228 3.666288 0.000000       NA  NA
## 14 2.6284228 3.6662879 0.8173555 0.8173555 3.666288 2.628423 0.000000  NA
## 15 0.7066960 3.6662879 2.6284228 2.6284228 3.666288 0.706696 2.628423   0
##
## Dendrograma final
##                      levelName
## 1  C14
## 2   ¦--C13
## 3   ¦   ¦--C12
## 4   ¦   ¦   ¦--X14
## 5   ¦   ¦   °--C8
## 6   ¦   ¦       ¦--X11
## 7   ¦   ¦       °--C4
## 8   ¦   ¦           ¦--X7
## 9   ¦   ¦           °--C3
## 10  ¦   ¦               ¦--X10
## 11  ¦   ¦               °--X4
## 12  ¦   °--C9
## 13  ¦       ¦--X15
## 14  ¦       °--C7
## 15  ¦           ¦--C5
## 16  ¦           ¦   ¦--X13
## 17  ¦           ¦   °--X3
## 18  ¦           °--C6
## 19  ¦               ¦--X8
## 20  ¦               °--X2
## 21  °--C11
## 22      ¦--X5
## 23      °--C10
## 24          ¦--X12
## 25          °--C2
## 26              ¦--X9
```

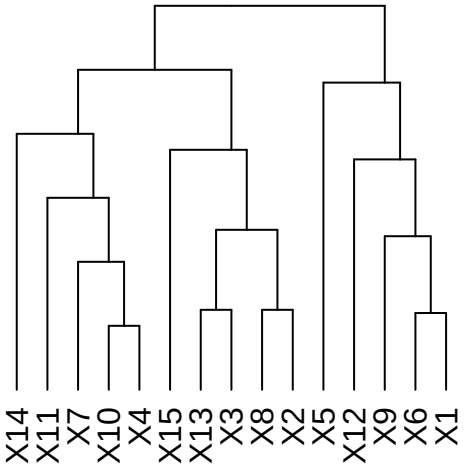


Figura 2.3: Clusterización con `AVG`

```
## 27              °--C1
## 28                 ¦--X6
## 29                 °--X1
##
##
## Coeficiente de CPCC:  0.9423605
```

2.2. Clasificación supervisada

2.2.1. Árboles de decisión

Ejercicio 2.2.1. *El tercer conjunto de datos, que se empleará para realizar el análisis de clasificación supervisada utilizando árboles de decisión, estará formado por el siguiente conjunto de 10 sucesos constituidos por los valores de cuatro características de vehículos: 1. {B, 4, 5, Coche}; 2. {A, 2, 2, Moto}; 3. {N, 2, 1, Bicicleta}; 4. {B, 6, 4, Camión}; 5. {B, 4, 6, Coche}; 6. {B, 4, 4, Coche}; 7.{N, 2, 2, Bicicleta}; 8. {B, 2, 1, Moto}; 9. {B, 6, 2, Camión}; 10. {N, 2, 1, Bicicleta}, donde las características de cada suceso son: {TipoCarnet, NúmeroRuedas, NúmeroPasajeros, TipoVehículo}. Se debe clasificar el tipo de vehículo en función del resto de características. TipoCarnet, es el tipo de carnet necesario para conducir el vehículo.*

Para este conjunto de datos se va a plantear un estudio de clasificación supervisada utilizando como técnica los árboles de decisión. Se buscará definir, a partir de la muestra, un modelo de clasificación que permita obtener el valor de una característica a partir del valor de otras características que componen un suceso de la muestra.

Como se necesita una muestra, lo primero que se va a realizar es la lectura del fichero Excel (`.xlsx`) por medio de la función `read.xlsx` del paquete `openxlsx`. Al igual que en el resto de ejercicios, se deberá pasar la ruta del fichero. En él se encuentra contenido el conjunto de datos propuesto para el ejercicio.

```
(sample = read.xlsx("data/vehiculos.xlsx"))
```

```
##    tCarnet nRuedas nPasajeros tVehículo
## 1        B       4          5     Coche
## 2        A       2          2      Moto
## 3        N       2          1 Bicicleta
## 4        B       6          4    Camión
## 5        B       4          6     Coche
## 6        B       4          4     Coche
## 7        N       2          2 Bicicleta
## 8        B       2          1      Moto
## 9        B       6          2    Camión
## 10       N       2          1 Bicicleta
```

Una vez tenemos los datos leídos hay que plantear el problema. En este caso, se pretende sacar la característica que define el tipo de vehículo (tVehiculo) a partir del resto de características (`tCarnet`, `nRuedas` y `nPasajeros`). Al primero le llamaremos a partir de ahora clasificador. Se va a hacer uso del algoritmo de Hunt. Este algoritmo de clasificación supervisada basada en árboles de decisión se compone de dos pasos fundamentales:

1. **Selección de nodo**: Para el primer paso se elige una característica, que compondrá la raíz del árbol. En el resto de pasos, esta selección formará los nodos intermedios del mismo. La característica que forme el nodo nunca podrá ser el clasificador (`tVehiculo` en este caso).

   El árbol con el que se va a trabajar en esta implementación será de naturaleza binaria; es decir, cada nodo tendrá dos hijos. En función del valor de la característica del nodo raíz salen dos nodos hijo. El objetivo de esto es que en base a ese valor que diferencia ambos nodos hijo, se pueda determinar el valor del clasificador. En el hijo de la derecha siempre habrá un nodo hoja, y en el izquierdo podrá haber un nodo hoja o un subárbol. En caso de ser un nodo hoja, se podrá determinar el valor de la característica clasificadora (que será el valor que aparezca en ese nodo). En caso de tener un subárbol, significa que el valor de la característica clasificadora no se puede determinar aún, por lo que será necesario fijarse en más criterios además del ya visto para sacar el valor de la característica clasificadora. Un ejemplo de árbol es el siguiente:

En este ejemplo se encuentran dos divisiones. Primeramente se clasifica o divide por número de pasajeros. Aquellos vehículos con un número de pasajeros igual a 2 se clasifican como Moto (valor del clasificador). En caso de tener 4 se clasifica por medio de la característica que define el número de ruedas. Si se tienen 4 ruedas, el valor del clasificador es Coche, y si tiene 6 es Camión.

Con el fin de poder optimizar estos árboles, se ha de encontrar la característica que mejor clasifique en base a la muestra con la que se está trabajando. Esto se va a realizar con la medida de ganancia de información ($\triangle_i$). Esta medida mide de una división, la diferencia de impureza entre el nodo padre y los nodos hijos una vez realizada esa división. A mayor ganancia de información, mejor división se habrá realizado, por lo que el objetivo será añadir a la clasificación, la división con mayor $\triangle_i$. La ganancia de información se calcula de la siguiente forma:

$$\triangle_i = I_{padre} - \sum_{j=1}^{k} \frac{N(n_j)}{N} \cdot I(n_j)$$

- I_{padre} = Impureza del nodo padre
- k = Número total de hijos (en esta implementación siempre será 2)
- $N(n_j)$ = Número de sucesos de la muestra asociados al nodo hijo j
- N = Número total de sucesos de la muestra actual
- $I(n_j)$ = Impureza del nodo hijo j

Como se puede ver, para calcular la ganancia de información, se necesita la impureza I del nodo padre y de sus hijos. Esta puede ser calculada de diferentes formas en función del método que se utilice. Todas ellas utilizan las frecuencias relativas (f_i) de las clases de equivalencia c del clasificador. Se distinguen tres:

- **Entropía**

$$I_{nodo} = -\sum_{i=1}^{c} f_i \cdot log_2(f_i)$$

- **Error**

$$I_{nodo} = 1 - max(f_i)$$

- **Gini**

$$I_{nodo} = 1 - \sum_{i=1}^{c} (f_i)^2$$

2. **Clasificación de los sucesos:** Una vez se ha elegido la característica que va a conformar el nodo raíz, se clasifica. Si se pueden clasificar completamente los sucesos de la muestra (que en cada hijo solo haya un valor del clasificador posible) se tendrán dos nodos hoja y termina la clasificación. Si no lo permite, se trata de un nodo interno,

por lo que hay que volver al primer paso y enlazar al nodo hijo de la izquierda un nuevo árbol. Cada vez que se clasifique con un criterio determinado, todas las filas de la muestra que contengan ese o esos valores en la característica del nodo se deben eliminar (ya que se considera que han sido clasificadas).

Una observación a tener en cuenta es que un nodo ha clasificado completamente cuando la impureza del padre es igual a la ganancia de información ($\triangle_i = I_{padre}$), ya que significa que las impurezas de los hijos son 0.

Puede darse el caso en el que el algoritmo no pueda seguir clasificando porque la ganancia de información sea 0 en todos los escenarios de clasificación posibles. En ese caso, el algoritmo finalizará habiendo clasificado todo lo anterior pero dejando el nodo final abierto, es decir, sin una clasificación concreta. Los valores del nodo izquierdo en la última serán todos los valores del clasificador que quedan en la muestra.

En esta implementación se utilizarán árboles binarios (cada nodo con 2 hijos) tal y como se ha mencionado previamente. Esto condiciona la forma de actuar en el algoritmo, así que la metodología a seguir se resume en los siguientes pasos:

1. **Selección del mejor nodo**: Para determinar el mejor nodo se calcula la ganancia de información para todas las divisiones posibles y se escoge la mejor. Las divisiones posibles son todas las combinaciones que resultan de elegir una característica concreta como nodo raíz, y dentro de cada característica elegir un valor de la misma que se utilizará para clasificar en el nodo de la derecha. Si se tienen n características y m posibles valores para cada característica, se tienen un total de $n \cdot m$ escenarios
2. **Unión de la clasificación actual al resto de clasificación**: Una vez se ha encontrado la mejor clasificación, esta se une a la clasificación previa.
3. **Recorte de la muestra**: Todos los sucesos de la muestra cuyo valor en la característica del nodo raíz sea igual al que define el nodo de la derecha son eliminados de la muestra. Se repiten los tres pasos anteriores con la nueva muestra hasta que el algoritmo finalice.

El algoritmo finalizará si no se tienen más datos en la muestra (es decir, ya se ha clasificado todo), o si la mejor ganancia de información es 0 (esto equivale a decir que no se puede encontrar ningún criterio de clasificación).

Con esta base teórica se procede a explicar la implementación del algoritmo en R. Primeramente, para seleccionar el mejor nodo se necesita calcular la ganancia de información. Esta a su vez sale de las impurezas, obtenidas en base a las frecuencias relativas en cada nodo de las clases de equivalencia del clasificador. Es por ello que lo primero que se plantea es una función `get_frec` que permita obtener las frecuencias relativas para el cálculo de la impureza de cada nodo. A esta función habrá que pasarle los siguientes parámetros:

- **sample**: La muestra de datos pendientes de clasificar.

- **col**: Columna que representa la característica del nodo raíz.
- **elements**: Clases de equivalencia de `col` que se tendrán en cuenta a la hora de calcular las frecuencias relativas. Si un suceso tiene uno de estos valores en la columna `col` se tiene en cuenta para las frecuencias relativas, sino se omite.
- **criteria**: Columna de la característica clasificadora.
- **values**: Todas las clases de equivalencia de la columna clasificadora.

Se empieza con una lista de ceros. Cada índice de esta lista representará las veces que aparece un valor del clasificador (es decir, las veces que aparece Moto, Camión, Coche y Bicicleta en este ejercicio). En el nodo raíz basta con contar las veces que aparece cada uno en toda la muestra, pero en los nodos hijo la muestra se divide. Por ello, es importante pasar el parámetro `elements`, para determinar qué sucesos contabilizarán para las frecuencias en cada caso.

El código es sencillo, se crea la lista con todo 0 y de longitud determinada por los posibles valores del clasificador, se comprueban todos los sucesos de la muestra y, si ese suceso está dentro del nodo que se está analizando, se incrementa en uno la frecuencia que corresponda. Finalmente se deben devolver frecuencias relativas, por lo que se dividen todos los valores entre los sucesos asociados al nodo. Nótese que la suma de esta lista deberá dar 1 en cualquier caso. El código es el siguiente:

```
get_frec = function(sample, col, criteria, elements, values) {
        n = 0
        frecs = setNames(data.frame(matrix(rep(0, times = length(values))
                , nrow = 1)), values)
        for (i in 1:len(sample[, col])) {
                if (sample[i, col] %in% elements) {
                        frecs[1, sample[i, criteria]] =
                                frecs[1, sample[i, criteria]] + 1
                        n = n + 1
                }
        }
        frecs / n
}
```

Una vez se tienen las frecuencias de los nodos, hay que calcular la impureza de los mismos. Para ello, se ha visto previamente que existen tres métodos diferentes (entropía, error y gini). El cálculo de la impureza se realiza en función de las frecuencias en cualquier caso, por lo que será un parámetro obligatorio en las tres funciones. Se tiene una por cada método.

El código de la entropía implementa el cálculo visto previamente. Un matiz importante en esta función es que se controla que, en caso de haber frecuencias con valor 0 o 1, se presentaría un cálculo matemáticamente imposible. Es por ello que en caso de encontrar esos números, el resultado de la operación correspondiente con esa frecuencia será 0.

```
entropy = function(frec) {
        acum = 0
        for (i in 1:len(frec)) {
                f = frec[1, i]
                if (f != 0 && f != 1) {
                        acum = acum - f * log(f, 2)
                }
        }
        acum
}
```

El código del método de error:

```
error = function(frec) {
        1 - max(frec)
}
```

Y el del método Gini:

```
gini = function(frec) {
        acum = 0
        for (i in 1:len(frec)) {
                f = frec[1, i]
                acum = acum + f^2
        }
        1 - acum
}
```

Con estas funciones auxiliares se puede confeccionar una función `get_impurity` que llame adecuadamente a estas anteriores y permita obtener el valor de la impureza de un nodo. Los parámetros que deberá tener esta función son los mismos que la función `get_frec` y un parámetro adicional que indique la medida con la que se quiere calcular la impureza. La función recoge las frecuencias del nodo y llama a una de las tres últimas funciones que se han visto, la cual regresa el valor de la impureza de un nodo.

```
get_impurity = function(sample, col, elements, criteria,
                 equivalence_clases, measure) {
        frec = get_frec(sample, col, criteria,
                elements, equivalence_clases)
        switch(measure,
                        "entropy" = entropy(frec),
                        "error" = error(frec),
                        "gini" = gini(frec))
}
```

Una consideración es que se necesitan todas las clases de equivalencia (`equivalence_classes`) de la característica clasificadora, para que este código pueda funcionar. Al final, la función `get_impurity` será llamada por otra de nivel superior que analice todos los nodos que conforman una división, pero esta necesita una lista con todos los valores que puede adquirir, en este caso, `tVehiculo`. Para ello, se ha hecho una función `get_elements`, la cual recibe la muestra completa de datos y la columna de la que se quieren sacar las clases de equivalencia. El código es muy sencillo, se itera toda la columna y se sacan todos los valores a una lista auxiliar, la cual será devuelta una vez se haya iterado todo.

```
get_elements  = function(sample, col) {
        elements = c()
        for (element in sample[, col]) {
                if (!(element %in% elements)) {
                        elements = c(elements, element)
                }
        }
        elements
}
```

La función de nivel superior de la que se hablaba será evidentemente `get_gain`, la cual será llamada para obtener la ganancia de información de una división concreta. Esta función es crucial en el algoritmo, ya que de ella dependerá la selección del siguiente nodo. La función llama a `get_elements` y a la función `get_impurity` para calcular la impureza del padre y de los hijos. Con todas las impurezas, se puede aplicar la fórmula definida al principio de esta sección. Los parámetros que recibe la función son los siguientes:

- **sample**: Muestra de datos actual.
- **left**: Valores de la característica del nodo raíz que conforman el nodo hijo de la izquierda. Este estará compuesto por sucesos que contengan alguno de estos valores en la característica.
- **right**: Valores de la característica del nodo raíz que conforman el nodo hijo de la derecha. Este estará compuesto por sucesos que contengan alguno de estos valores en la característica.
- **criteria**: Nombre de la columna que representa la característica clasificadora en la muestra.
- **col**: Nombre de la columna que representa la característica del nodo raíz en la muestra.
- **measure**: Medida de la impureza. Este parámetro podrá adquirir los siguientes valores: “entropy”, “error”, “gini”.

```
get_gain = function(sample, left, right, criteria, col, measure) {
        n = len(sample)
        equivalence_clases = get_elements(sample, criteria)
        impurity_father = get_impurity(sample, col, union(left, right), criteria, equivalence_clases, measure)
        impurity_left = get_impurity(sample, col, left, criteria, equivalence_clases, measure)
        impurity_right = get_impurity(sample, col, right, criteria, equivalence_clases, measure)
        impurity_father - (len(left) / n * impurity_left + len(right) / n * impurity_right)
}
```

Una vez se tiene cómo calcular la ganancia de información de una división concreta hay que dar formato a esas divisiones. La pregunta que se plantea es: *¿Cómo se puede representar un árbol con dos nodos hijo y un nodo raíz?*. La estructura de datos que se ha escogido es un `data.frame`. Esta estructura contendrá columnas que internamente tendrán todos los atributos necesarios para definir una división o clasificación concreta. Estos atributos son los siguientes:

- **parent**: Define la característica del nodo raíz.
- **right_element**: El valor del clasificador que va a aparecer en el nodo de la derecha (el nodo de la derecha solo estará formado por sucesos con este valor en el clasificador).
- **left**: Valores de la característica del nodo raíz para los cuales un suceso se clasifica en el nodo de la izquierda.
- **right**: Valores de la característica del nodo raíz para los cuales un suceso se clasifica en el nodo de la derecha.
- **gain**. Ganancia de información ($\triangle_i$) de la división.

Además esta función recibe los siguientes parámetros:

- **sample**: Muestra de datos con la que se trabaja para hacer la clasificación.
- **col**: Característica que conformará el nodo raíz de la nueva clasificación.
- **right_element**: Valor de la columna clasificadora que tendrán los suceso del nodo hijo de la derecha.
- **criteria**: Característica clasificadora.
- **measure**: Medida de impureza que se usará para calcular la ganancia de información.

El código que crea la nueva clasificación es el siguiente:

```
create_classification = function(sample, col, right_element, criteria, measure) {
        left = c()
        for (i in 1:len(sample[, col])) {
                element = sample[i, col]
                if (!(element %in% left) && sample[i, criteria] != right_element) {
                        left = unlist(append(left, element))
                }
        }
```

```
        right = dif(get_elements(sample, col), left)
        if(len(right) > 0 && len(left) > 0) {
                gain = get_gain(sample, left, right, criteria, col, measure)
        } else {
                gain = 0
        }
        data.frame(parent=col, right_element,
                left=I(list(left)), right=I(list(right)), gain)
}
```

La clasificación final va a representarse como un `data.frame` con el conjunto de divisiones realizadas a lo largo del algoritmo, donde cada fila (o clasificación) representa un nivel del árbol. La característica del nodo raíz se saca con la columna `parent`, el valor deducido del clasificador en el nodo de la derecha con `right_element`, y los valores de la característica del nodo raíz que definen los hijos izquierdo y derecho se sacan con `left` y `right`. El nodo de la izquierda estará definido por la siguiente fila del `data.frame` y así sucesivamente hasta llegar al último árbol que contiene dos hojas. El nodo de la izquierda de este árbol estará definido por los valores restantes en la columna clasificadora de la muestra.

En términos prácticos, un `data.frame` como salida que representa un árbol de decisión no es viable, ya que requiere de un esfuerzo por parte del usuario para entender el resultado de la ejecución. Lo que se busca en esta implementación es proporcionar una salida visual que no requiera de ningún esfuerzo por parte del usuario para entenderla. Es por ello que se ha decidido elaborar una función `df_2_tree`, la cual pasará la información del `data.frame` a una estructura de árbol, pudiendo ofrecer una salida visual del árbol de decisión final, fruto de la ejecución realizada sobre la muestra introducida. Esta función se ha basado principalmente en el paquete `data.tree`, explicado en el anexo del documento.

Para empezar, la forma en la que se construye la estructura del árbol es de abajo arriba. Esto evidentemente implica un recorrido del `data.frame` desde la última fila hasta la primera. La función recibe como parámetros el `data.frame` que define el árbol de decisión, la muestra final de datos y el nombre de la característica clasificadora.

Para construir un árbol primero se define el nombre del nodo raíz (con `Node$new`), el cual contendrá el valor de la columna `parent`. Además, se define un atributo `gain` dentro de ese nuevo nodo que almacena la ganancia de información de la división. Posteriormente, se declara el estilo del nodo, indicando que este debe mostrar visualmente el nombre y la ganancia de información. Esto se hace con la función `SetNodeStyle`.

El nodo de la derecha se define indicando en el atributo del nombre el valor del clasificador que representa ese nodo. Para asociarlo al nodo raíz se hace uso de la función `AddChildNode` aplicada al nodo raíz. Además, se define un atributo `label` con los valores de `right`.

El nodo de la izquierda se definirá de dos formas distintas. En la primera iteración, el nombre del mismo serán todos los elementos que quedan en la muestra; mientras que en el

resto de iteraciones, en vez de declarar un nodo nuevo, se define como hijo izquierdo el árbol de la anterior iteración. De esta forma, los árboles que vamos creando se van encadenando con el de la iteración siguiente sucesivamente hasta llegar a la raíz. En ambos casos, se define un atributo `label` con los valores de `left`.

Este atributo `label` permite que a la hora de mostrar el árbol salgan los valores del nodo raíz que clasifican el nodo izquierdo y el nodo derecho en la arista. Para ello hacemos uso de la función `SetEdgeStyle`. Una vez se ha terminado de construir la estructura del árbol, con la función `plot` se muestra todo lo que se ha definido previamente. El código de la función es el siguiente:

```
df_2_tree = function(df, final, criteria) {
        last_tree = Node$new(df[nrow(df), "parent"])
        last_tree$gain = round(df[nrow(df), "gain"], 3)
        SetNodeStyle(last_tree, label = paste(last_tree$name, "\n", last_tree$gain))
        new_left_leaf = Node$new(paste(get_elements(final, criteria)))
        new_left_leaf$label = final[nrow(final), df[nrow(df), "parent"]]
        new_left_leaf$gain = ""
        last_tree$AddChildNode(new_left_leaf)
        new_right_leaf = Node$new(df[nrow(df), "right_element"])
        new_right_leaf$label = df[nrow(df), "right"]
        new_right_leaf$gain = ""
        last_tree$AddChildNode(new_right_leaf)
        SetEdgeStyle(new_left_leaf, label = new_left_leaf$label, fontsize = 11)
        SetEdgeStyle(new_right_leaf, label = new_right_leaf$label, fontsize = 11)
        SetNodeStyle(new_left_leaf, label = paste(new_left_leaf$name))
        SetNodeStyle(new_right_leaf, label = paste(new_right_leaf$name))

        for (row_index in (nrow(df)-1):1) {
                last_tree$label = df[row_index, "left"]
                new_tree = Node$new(df[row_index, "parent"])
                new_tree$AddChildNode(last_tree)
                right_node = Node$new(df[row_index, "right_element"])
                right_node$label = df[row_index, "right"]
                right_node$gain = ""
                new_tree$AddChildNode(right_node)
                SetEdgeStyle(right_node, label = right_node$label, fontsize = 11)
                SetEdgeStyle(last_tree, label = last_tree$label, fontsize = 11)
                last_tree = new_tree
                last_tree$gain = round(df[row_index, "gain"], 3)
                SetNodeStyle(last_tree, label =
                                paste(last_tree$name, "\n", last_tree$gain))
                SetNodeStyle(right_node, label = paste(right_node$name))
        }
        return(last_tree)
}
```

Todas esta funciones van a trabajar conjuntamente para proporcionar la salida de la ejecución del algoritmo. El usuario llamará a una función, pasando la muestra de datos, las características con las que se va a clasificar, el clasificador y el método de cálculo de la impureza de los nodos. Esta función `hunt` será la encargada de la ejecución completa del algoritmo.

Primeramente se va a ver el código:

```
hunt = function(sample, classes, criteria, measure, details = FALSE) {
        final_clasification = data.frame()
        best_gain = -1
        best_clasification = NULL
        while (nrow(sample) > 0 && best_gain != 0) {
                best_gain = -1
                for (col in classes) {
                        for (element in get_elements(sample, criteria)) {
                                clasification = create_classification(sample, col, element, criteria, measure)
                                actual_gain = clasification$gain
                                if (actual_gain > best_gain) {
                                        best_gain = actual_gain
                                        best_clasification = clasification
                                }
                        }
                }
                if (best_gain != 0) {
                        sample = subset(sample, sample[, best_clasification$parent] %in% unlist(best_clasification$left))
                        final_clasification = rbind(final_clasification, best_clasification)
                        if (details) {
                                cat("\nIteración", nrow(final_clasification))
                                cat("\n===========\n")
                                cat("Se escoge el nodo", best_clasification$parent, "con una ganancia de", best_gain)
                                cat("\nLos sucesos que tengan el/los valor/es", paste(best_clasification$right),
                                 "en ese nodo se clasifican como", best_clasification$right_element)
                                cat("\nLa nueva muestra es:\n")
                                print(sample)
                                cat("\n")
                        }
                }
        }

        tree = df_2_tree(final_clasification, sample, criteria)
        cat("\nÁrbol de decisión\n================\n")
        print(tree, "label", "gain")
        tree
}
```

El código selecciona la mejor división en base a la muestra actual de datos. Para ello se hacen todas las divisiones posibles con todas las características, y en cada una de ellas, se hace una división por cada valor que puede adquirir el clasificador, dejando este en el nodo de la derecha. En la muestra del ejercicio se hacen divisiones por `tCarnet`, `nRuedas` y `nPasajeros`. Dentro de cada característica se hacen divisiones dejando como elemento de la derecha Moto, Camión, Coche y Bicicleta. De todas estas posibles divisiones se calcula la ganancia de información de cada una de ellas, y se coge la división que devuelve la mayor ganancia de información. Esta será conocida como `best_clasification`.

Si la mejor clasificación tiene una ganancia de información mayor que 0 (es decir, que clasifique algo), se debe recortar la muestra. Para ello, se hace uso de la función `subset`, la cual devuelve un `data.frame` con los sucesos de la muestra que no han sido aún clasificados (que su valor en la característica del nodo raíz es cualquiera de los del nodo izquierdo). Además, se añade al conjunto de clasificaciones la nueva clasificación.

Este procedimiento se repite hasta que no se tengan más sucesos en la muestra (no hay nada más que clasificar) o hasta que la mejor ganancia de información sea 0 (lo que querrá decir que no se puede clasificar nada más en la muestra restante).

Una vez se tiene el `data.frame` final representando el árbol de decisión, se llama a la función `data_2_tree` para pasar el mismo a una estructura de árbol.

La función cuenta con un parámetro `details` (inicialmente establecido a `FALSE`), el cual si se pone a `TRUE` indica al usuario que división se ha escogido en cada paso (con su Δ_i) y qué muestra se queda después de haber dividido.

Si se ejecuta la función con la muestra del ejercicio se obtiene la siguiente salida:

```
union = function(c1, c2) {
if (len(c1) == 0) {
c2
} else if (is.element(c1[1], c2)) {
union(c1[-1], c2)
} else {
union(c1[-1], append(c2, c1[1]))
}
}

dif = function(c1, c2) {
res = c()
for (element in c1) {
if (!(element %in% c2)) {
res = c(res, element)
}
}
res
}
tree = hunt(sample, c("tCarnet", "nRuedas", "nPasajeros"),
 "tVehículo", "error", TRUE)

##
## Iteración 1
## ============
## Se escoge el nodo tCarnet con una ganancia de 0.4142857
## Los sucesos que tengan el/los valor/es N en ese nodo se clasifican como Bicicleta
## La nueva muestra es:
##   tCarnet nRuedas nPasajeros tVehículo
## 1       B       4          5     Coche
## 2       A       2          2      Moto
## 4       B       6          4    Camión
## 5       B       4          6     Coche
## 6       B       4          4     Coche
## 8       B       2          1      Moto
## 9       B       6          2    Camión
##
```

```
##
## Iteración 2
## ============
## Se escoge el nodo tCarnet con una ganancia de 0.4464286
## Los sucesos que tengan el/los valor/es A en ese nodo se clasifican como Moto
## La nueva muestra es:
##   tCarnet nRuedas nPasajeros tVehículo
## 1       B       4          5     Coche
## 4       B       6          4    Camión
## 5       B       4          6     Coche
## 6       B       4          4     Coche
## 8       B       2          1      Moto
## 9       B       6          2    Camión
##
##
## Iteración 3
## ============
## Se escoge el nodo nRuedas con una ganancia de 0.375
## Los sucesos que tengan el/los valor/es 6 en ese nodo se clasifican como Camión
## La nueva muestra es:
##   tCarnet nRuedas nPasajeros tVehículo
## 1       B       4          5     Coche
## 5       B       4          6     Coche
## 6       B       4          4     Coche
## 8       B       2          1      Moto
##
##
## Iteración 4
## ============
## Se escoge el nodo nRuedas con una ganancia de 0.25
## Los sucesos que tengan el/los valor/es 4 en ese nodo se clasifican como Coche
## La nueva muestra es:
##   tCarnet nRuedas nPasajeros tVehículo
## 8       B       2          1      Moto
##
##
## Árbol de decisión
## =================
##               levelName label  gain
## 1 tCarnet                       0.414
## 2  ¦--tCarnet            B, A 0.446
## 3  ¦   ¦--nRuedas           B 0.375
## 4  ¦   ¦   ¦--nRuedas    4, 2  0.25
## 5  ¦   ¦   ¦   ¦--Moto      2
```

```
## 6  ¦   ¦   ¦   °--Coche     4
## 7  ¦   ¦   °--Camión        6
## 8  ¦   °--Moto              A
## 9  °--Bicicleta             N
```

Como se puede ver, se ha optado por una salida detallada. Se imprimen las iteraciones que se han hecho, indicando el nodo raíz escogido así como su ganancia. Se imprimen las conclusiones de esa división y la muestra restante. Por último, se imprime el árbol final. Para poder visualizarlo de una mejor forma se puede hacer plot de la salida que ofrece la función, obteniendo el siguiente gráfico visual.

```
## Error in loadNamespace(name): there is no package called 'webshot'
```

Por lo tanto, se pueden sacar las siguientes conclusiones del ejercicio:

- Si un vehículo tiene carnet N es una bicicleta.
- En caso contrario, si un vehículo tiene carnet A se trata de una moto.
- Si el tipo de carnet es B y el vehículo tiene 6 ruedas, es un camión.
- En cualquier otro caso, si el vehículo tiene 4 ruedas es un coche y si tiene 2 ruedas es una moto.

2.2.2. Regresión lineal

Ejercicio 2.2.2. *El cuarto conjunto de datos, que se empleará para realizar el análisis de clasificación supervisada utilizando regresión, estará formado por los siguientes 4 subconjuntos de datos: 1.* $\{10, 8.04; 8, 6.95; 13, 7.58; 9, 8.81; 11, 8.33; 14, 9.96; 6, 7.24; 4, 4.26; 12, 10.84; 7, 4.82; 5, 5.68\}$*; 2.* $\{10, 9.14; 8, 8.14; 13, 8.74; 9, 8.77; 11, 9.26; 14, 8.1; 6, 6.13; 4, 3.1; 12, 9.13; 7, 7.26; 5, 4.74\}$*; 3.* $\{10, 7.46; 8, 6.77; 13, 12.74; 9, 7.11; 11, 7.81; 14, 8.84; 6, 6.08; 4, 5.39; 12, 8.15; 7, 6.42; 5, 5.73\}$*; 4.* $\{8, 6.58; 8, 5.76; 8, 7.71; 8, 8.84; 8, 8.47; 8, 7.04; 8, 5.25; 19, 12.5; 8, 5.56; 8, 7.91; 8, 6.89\}$*. Se deben calcular las rectas de regresión de los cuatro subconjuntos y sus parámetros de ajuste.*

Para la resolución de este ejercicio, se hará uso de la recta de regresión lineal de la forma $\hat{y} = a + bx$ donde los parámetros a y b obedecen a las siguiente ecuaciones.

$$\begin{aligned} b &= \frac{S_{xy}}{S_x^2} \\ a &= \bar{y} - b\bar{x} \end{aligned} \tag{2.3}$$

Para el cálculo de estos parámetros se necesita hacer uso de funciones ya implementadas en la práctica anterior como la media aritmética, la desviación típica y la varianza, se recuerda el código a continuación.

```
fcd_mean = function(list) {
        add = 0
        for (i in 1:len(list)) {
                add = add + list[i]
        }
        add / len(list)
}

standard_dev = function(list) {
        mean = fcd_mean(list)
        n = len(list)
        add = 0
        for (i in 1:n) {
                add = add + ((list[i] - mean)^2)
        }
        sqrt(add/n)
}

variance = function(list) {
        dev = standard_dev(list)
        var = dev^2
        var
}
```

El siguiente cálculo será hallar el valor de la covarianza de la siguiente manera

$$S_{xy} = \frac{\sum_{i=1}^{n} x_i y_i}{n} - \bar{x}\bar{y}, \tag{2.4}$$

y quedará reflejado en la función `covariance` que recibe dos vectores que representan los datos y sus valores correspondientes:

$$\mathbf{X} = (x_1, x_2, \ldots, x_n)$$
$$\mathbf{Y} = (y_1, y_2, \ldots, y_n)$$

La función realiza el cálculo de la Ecuación (2.4) usando `%*%`, que en este caso realiza el producto escalar de $\mathbf{X}$ e $\mathbf{Y}$. También se verifica que estos vectores compartan dimensiones, pues de lo contrario no se podrá calcular.

```
covariance = function(x, y) {
        if (len(x) != len(y)) {
                stop("X e Y deben tener la misma dimensión")
        }
        else {
```

```
                sum = x %*% y
                (sum/len(x))-(fcd_mean(x)*fcd_mean(y))
        }
}
```

Finalmente, la función `regression_line` calcula los parámetros a y b mediante la Ecuación (2.3). Devuelve el resultado de forma matricial para más tarde trabajar de forma más cómoda.

$$\mathbf{P} = \begin{pmatrix} a \\ b \end{pmatrix}$$

```
regression_line = function(x, y) {
        b = covariance(x, y) / variance(x)
        a = fcd_mean(y) - b * fcd_mean(x)
        matrix(c(a, b), ncol = 1)
}
```

Una vez se tienen calculados los parámetros de la recta de regresión lineal, es momento de evaluar cómo de buena es la aproximación. Para ello se emplea la métrica R^2. Su valor depende de otros dos, SSR y SSY, que obedecen a las siguientes ecuaciones.

$$SSR = \sum_{i=1}^{n} (\hat{y}_i - \bar{y})^2$$

$$SSY = \sum_{i=1}^{n} (y_i - \bar{y})^2$$

Estos cálculos quedan recogidos en las funciones `ssr` y `ssy`. La primera de ellas calcula el valor requerido de forma matricial, recibiendo $\mathbf{P}$, $\mathbf{X}$, e $\mathbf{Y}$ como parámetros, y realiza las siguientes operaciones, donde el cuadrado se realiza elemento a elemento, en vez de forma matricial. En este caso, `%*%` funciona como el producto de matrices usual.

$$\hat{\mathbf{Y}} = \begin{pmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_n \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix}$$

$$SSR = \sum_{i=1}^{n} (\hat{\mathbf{Y}}_i - \bar{\mathbf{Y}}_i)^2$$

De la manera similar, se calcula el SSY como $\sum_{i=1}^{n} (\mathbf{Y}_i - \bar{\mathbf{Y}}_i)^2$. Además se conoce que la relación entre estos dos parámetros y R^2 es la siguiente.

$$R^2 = \frac{SSR}{SSY}$$

```
ssr = function(p, x, y) {
        X = matrix(c(rep(1, times = len(x)), x), ncol = 2)
        y_hat = X %*% p
        sum((y_hat - rep(fcd_mean(y)))^2)
}

ssy = function(y) {
        sum((y - rep(fcd_mean(y)))^2)
}

r2 = function(sr, sy) {
        sr/sy
}
```

Finalmente se encapsula todo en una única función llamada `fcd_regression` que muestra por consola la ecuación de $\hat{y}$ y el valor de R^2, además de una gráfica con la recta y los puntos.

```
fcd_regression = function(sample) {
        X = sample[, 1]
        Y = sample[, 2]

        param = regression_line(X, Y)
        r = r2(ssr(param, X, Y), ssy(Y))

        a = param[1, 1]
        b = param[2, 1]

        print(sprintf("y = %.3fx + %.3f", b, a))
        print(sprintf("R2 = %.3f", r))

        plot(X, Y, col = "blue", main = "Recta de regresión",
                xlab = "Eje X", ylab = "Eje Y")
        abline(a=a, b=b, col="red")
}
```

De la siguiente forma, se prueba para los casos del profesor. Se observa que las aproximaciones no son buenas, pues los valores de R^2 no se aproximan a 1. En la Figura 2.4 se debe a que los puntos se encuentran de manera un tanto dispersa, dando a entender que no existe relación entre las variables. En la Figura 2.5 se ve claramente que la relación es cuadrática, por lo que es inadecuado utilizar una recta para aproximar esos puntos. En la Figura 2.6 el problema viene dado por un outlier, y en la Figura 2.7 se dan diferentes valores a y para un mismo x lo que hace que la pendiente de esta recta tienda a infinito.

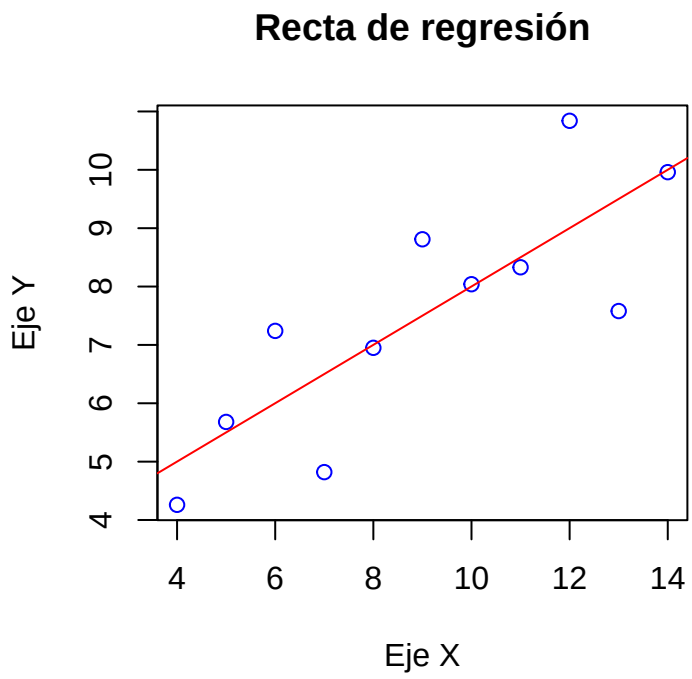


Figura 2.4: Datos no correlacionados

```
sample1 = read.xlsx("../Memoria/data/conj1.xlsx", colNames=FALSE)
fcd_regression(sample1)

## [1] "y = 0.500x + 3.000"
## [1] "R2 = 0.667"
```

```
sample2 = read.xlsx("../Memoria/data/conj2.xlsx", colNames=FALSE)
fcd_regression(sample2)

## [1] "y = 0.500x + 3.001"
## [1] "R2 = 0.666"
```

```
sample3 = read.xlsx("../Memoria/data/conj3.xlsx", colNames=FALSE)
fcd_regression(sample3)

## [1] "y = 0.500x + 3.002"
## [1] "R2 = 0.666"
```

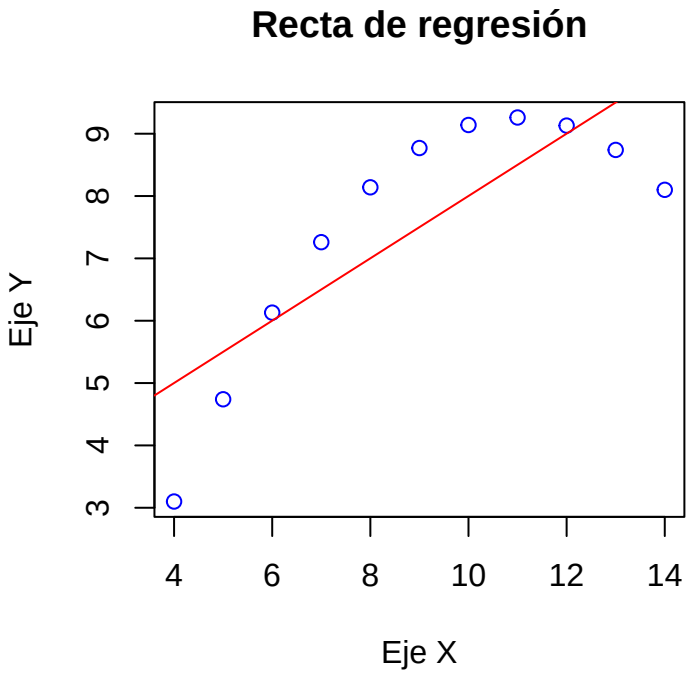


Figura 2.5: Relación cuadrática

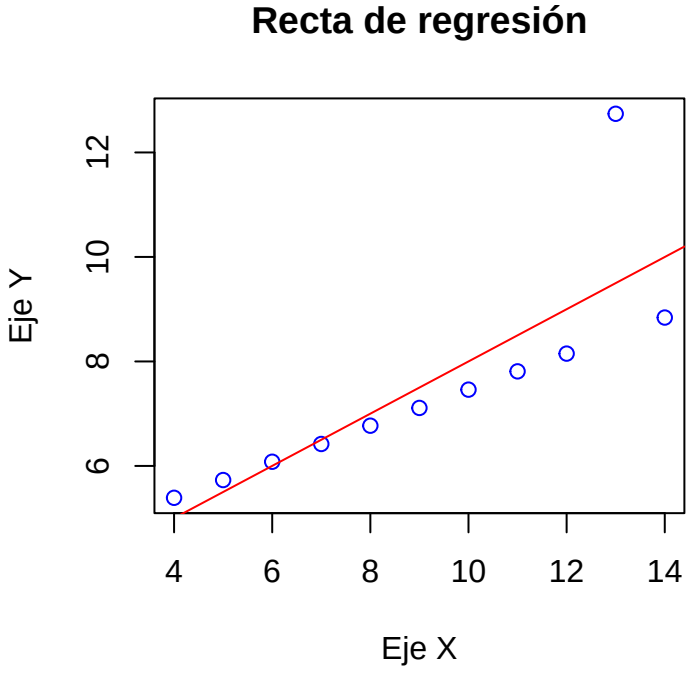


Figura 2.6: Relación lineal con outlier

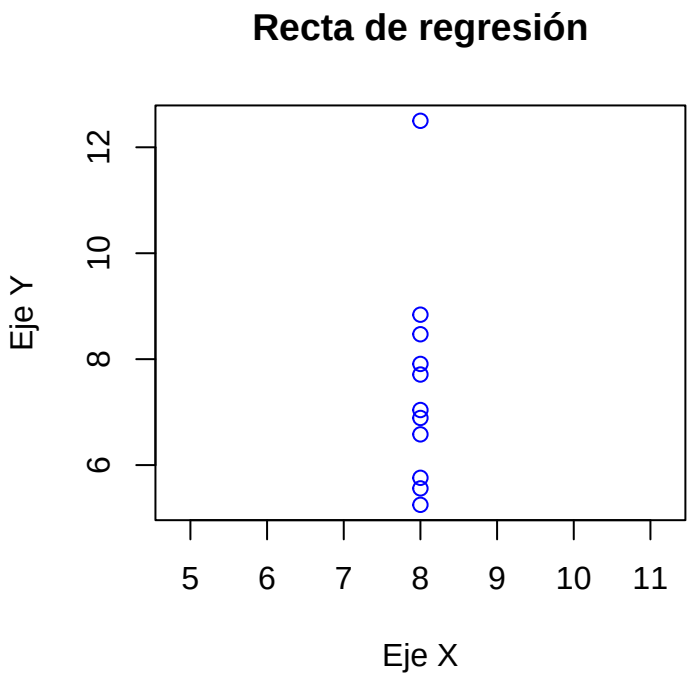


Figura 2.7: Recta con pendiente infinita

```
sample4 = read.xlsx("../Memoria/data/conj4.xlsx", colNames=FALSE)
fcd_regression(sample4)

## [1] "y = NaNx + NaN"
## [1] "R2 = NaN"

## Error in int_abline(a = a, b = b, h = h, v = v, untf = untf, ...): 'a' y 'b'
deben ser finitos
```

Apéndice A

Apéndice. Librería de Funciones Auxiliares

Para finalizar, se explicará la librería externa realizada (`data/aux_functions.R`), en la que se encuentran funciones auxiliares que ya fueron programadas en la PL1 o que no forman parte de un algoritmo en sí, y que además se reutilizan en varios de ellos.

A.1. Funciones de la PL1

Se han reutilizado las siguientes funciones de la PL1:

1. `len`. Esta función calcula la longitud de una lista dada.

```
len = function(list) {
        count = 0
        for (element in list) {
                count = count + 1
        }
        count
}
```

2. `fcd_mean`. Esta función calcula la media aritmética de una lista dada. Es equivalente a la función `mean` del paquete `base` de R. Se define la media aritmética como:

$$fcd\_mean = \frac{\sum_{i=1}^{n}(x_i)}{n}$$

```
fcd_mean = function(list) {
        add = 0
        for (i in 1:len(list)) {
```

```
                add = add + list[i]
        }
        add / len(list)
}
```

3. `union`. Realiza la unión de dos conjuntoss $A \cup B$ que resulta en un nuevo conjunto que contiene todos los elementos de A y todos los elementos de B que no están en A.

```
union = function(c1, c2) {
        if (len(c1) == 0) {
                c2
        } else if (is.element(c1[1], c2)) {
                union(c1[-1], c2)
        } else {
                union(c1[-1], append(c2, c1[1]))
                        }
}
```

4. `dif`. Realiza la diferencia de dos conjuntos $A \backslash B$ que da como resultado un nuevo conjunto que contiene todos los elementos de A que no formen parte de B.

```
dif = function(c1, c2) {
        res = c()
        for (element in c1) {
                if (!(element %in% c2)) {
                        res = c(res, element)
                }
        }
        res
}
```

5. `standard_dev`. Esta función calcula la desviación típica de un conjunto de datos. No es posible utilizar la función `sd` del paquete `base` ya que la fórmula interna que utiliza divide entre `N-1` en lugar de entre `N`, siendo `N` la longitud del conjunto. Se define la desviación típica como:

$$standard\_dev = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n}}$$

```
standard_dev = function(list) {
        mean = fcd_mean(list)
        n = len(list)
        add = 0
        for (i in 1:n) {
                add = add + ((list[i] - mean)^2)
        }
        sqrt(add/n)
}
```

6. `variance`. Calcula la varianza de un conjunto de elementos dado. Nuevamente, no es posible utilizar la función `var` del paquete `base` por la misma razón que la desviación típica. Se define la varianza como el cuadrado de la desviación típica.

$$variance = sd^2$$

```
variance = function(list) {
        dev = standard_dev(list)
        var = dev^2
        var
}
```

A.2. Nuevas funciones

Se han creado dos nuevas funciones que se reutilizan en distintos algoritmos.

1. `euc_distance`. Esta función calcula la distancia euclídea entre dos puntos de dos coordenadas. Se define la distancia euclídea como:

$$euc\_distance(p_i, p_j) = \sqrt{(p_{i_1} - p_{j_1})^2 + (p_{i_2} - p_{j_2})^2}$$

```
euc_distance = function(p1, p2) {
        sqrt(((p1[1] - p2[1])^2) + ((p1[2] - p2[2])^2))
}
```

2. `covariance`. Esta función calcula la covarianza entre dos conjuntos de datos. Se define la covarianza como:

$$covariance(x, y) = \frac{\sum_{i=1}^{n} x_i y_i}{n} - \bar{x}\bar{y},$$

```
covariance = function(x, y) {
        if (len(x) != len(y)) {
                stop("X e Y deben tener la misma dimensión")
        }
        else {
                sum = x %*% y
                (sum/len(x))-(fcd_mean(x)*fcd_mean(y))
        }
}
```

Apéndice B

Paquetes externos

B.1. openxlsx

El paquete `openxlsx` de R es una herramienta para manejar archivos Excel en formato `xlsx`. Proporciona funciones para leer, escribir, y modificar estos archivos sin depender de Java. Una de sus funciones más importantes es `read.xlsx`, que permite leer hojas de cálculo Excel en R. Esta función es útil para importar datos complejos, ofreciendo opciones para manejar diferentes tipos de datos y formatos.

La función `read.xlsx` tiene argumentos clave que determinan cómo se lee una hoja de cálculo de Excel:

- **xlsxFile**: Ruta al archivo Excel, objeto Workbook o URL.
- **sheet**: Especifica la hoja de cálculo a leer, por nombre o índice.
- **startRow**: Define la primera fila de datos a leer, ignorando las anteriores.
- **colNames**: Si es TRUE, la primera fila de datos se utiliza como nombres de columna.
- **rowNames**: Si es TRUE, la primera columna se usa para los nombres de fila.
- **skipEmptyRows / skipEmptyCols**: Determinan si se ignoran filas o columnas vacías.
- **cols**: Un vector numérico que especifica qué columnas leer.
- **rows**: Un vector numérico que especifica qué filas leer.
- **detectDates**: Si es TRUE, intenta convertir cadenas de texto que parecen fechas en fechas de R.
- **na.strings**: Define qué cadenas se interpretarán como `NA`.

Estos argumentos ofrecen flexibilidad para manejar datos de Excel en R.

Además, `openxlsx` incluye funciones para escribir datos en Excel, personalizar el estilo de las celdas, y manejar las hojas de cálculo, entre otras capacidades. La documentación del paquete en CRAN brinda ejemplos prácticos y guías detalladas para su uso eficiente.

B.2. data.tree

El paquete `data.tree` de R es una estructura de datos para manejar datos jerárquicos. Permite a los usuarios de R crear estructuras de árbol a partir de datos jerárquicos y recorrer el árbol en varios órdenes. Es posible agregar y acumular valores, imprimir y trazar el árbol, convertir a y desde `data.frame`, entre otras funcionalidades. El paquete es útil en una variedad de aplicaciones, incluyendo árboles de decisión, aprendizaje automático, finanzas y la conversión de y hacia JSON, por nombrar algunos.

El paquete provee herramientas para convertir estructuras de datos entre diferentes formatos como `data.frame`, listas de listas, dendrogramas, etc. Los usuarios pueden construir árboles de datos de manera programática o mediante conversión de otros formatos. Además, `data.tree` facilita la visualización de datos jerárquicos y puede integrarse con widgets HTML para una presentación más interactiva.

Una característica distintiva de `data.tree` es su capacidad para manejar grandes volúmenes de datos jerárquicos de manera eficiente, lo cual a menudo puede ser complicado con las estructuras de datos básicas en R. Esto lo hace particularmente valioso para prototipos rápidos de algoritmos de búsqueda, pruebas de nuevas ideas de clasificación y más.

Algunas de las funciones que se han utilizado en la realización de la práctica son los siguientes.

B.2.1. Traverse

La función `Traverse` se utiliza para recorrer un árbol o un subárbol de `Node` en un orden específico. Devuelve una lista de objetos `Node` que han sido filtrados y podados por las funciones `filterFun` y `pruneFun`. Los argumentos más importantes son:

- **node**: Es el nodo raíz del árbol o subárbol que se va a recorrer.
- **traversal**: Define el orden en el que se recorre el árbol. Los valores posibles son pre-order, post-order, in-order, level, ancestor, o una función personalizada. Cada uno de estos términos se refiere a un método específico de recorrido del árbol.
    - **pre-order**: Visita primero el nodo, luego recorre cada subárbol de izquierda a derecha.
    - **post-order**: Recorre primero los subárboles de izquierda a derecha y visita el nodo después.
    - **in-order**: Recorre el primer subárbol, visita el nodo, luego recorre el segundo subárbol.
    - **level**: Recorre los nodos nivel por nivel, empezando por la raíz.
    - **ancestor**: Visita un nodo y luego todos sus ancestros hasta la raíz.

- **pruneFun**: Una función que toma un `Node` como entrada y devuelve `TRUE` o `FALSE`. Si devuelve `FALSE` para un nodo, dicho nodo y todo su subárbol se excluyen del recorrido.
- **filterFun**: Similar a `pruneFun`, pero en este caso, si la función devuelve `FALSE` para un nodo, solo ese nodo se excluye del resultado, no así todo el subárbol.

Estas funciones permiten una gran flexibilidad al trabajar con datos jerárquicos, ya que se pueden recoger o excluir nodos específicos basándose en criterios definidos por el usuario. Esto es útil en una variedad de aplicaciones, desde la manipulación de datos hasta la visualización y la creación de algoritmos específicos para recorrer y trabajar con estructuras de árbol.

B.2.2. Node$new

La función `Node$new` se utiliza para crear un nuevo objeto `Node`. Este es a menudo el punto de partida para construir un árbol jerárquico de datos desde cero, típicamente comenzando con la creación del nodo raíz. Aquí están los argumentos más importantes de esta función:

- **name**: El nombre del nodo a crear. Este es un argumento obligatorio y define la identidad del nodo dentro del árbol.
- **check**: Este argumento controla cómo se manejan los nombres de los nodos en cuanto a su conformidad con las reglas de nomenclatura. Los valores posibles incluyen:
- **check**: Es el valor predeterminado. Si se proporciona un nombre que no se ajusta a las normas, se emitirá una advertencia.
- **no-warn**: Se comprobará la conformidad del nombre, pero no se emitirán advertencias si hay un problema.
- **no-check o FALSE**: No se comprobará la conformidad del nombre. Se debe usar si el rendimiento es crítico y se está seguro de que los nombres son conformes. En caso de no conformidad, se pueden esperar errores más difíciles de diagnosticar.
- **...**: Este es un lugar para pasar atributos adicionales al nodo. Pueden ser pares de nombres y valores que definen las propiedades del nodo.

Al llamar a `Node$new`, se crea un nuevo objeto `Node` con el nombre y los atributos especificados. Posteriormente, se pueden agregar hijos a este nodo o establecer más propiedades para construir un árbol de datos completo.

B.2.3. Node$AddChildNode

La función `AddChildNode` se utiliza para añadir un nodo hijo a un nodo existente. Esto es esencial para construir la estructura jerárquica de un árbol.

El único argumento de la función es `child`, el nodo hijo que se va a añadir al nodo actual. Debe ser un objeto `Node` que ya haya sido creado, típicamente con `Node$new`.

Cuando se llama a `AddChildNode(child)`, el nodo especificado en `child` se adjunta al nodo desde el cual se llama al método. El método devuelve el nodo hijo, lo que permite la posibilidad de encadenar llamadas para una construcción más fluida del árbol.

B.2.4. SetNodeStyle

La función `SetNodeStyle` se utiliza para definir el estilo de los nodos en la representación gráfica de un árbol. Esta función es importante para personalizar la apariencia de los nodos cuando se visualizan árboles jerárquicos.

Los argumentos más importantes de la función son:

1. **tree o node**: Se refiere al árbol o nodo al que se aplicará el estilo.
2. **label**: Función que define cómo se generará la etiqueta de texto para cada nodo. Por ejemplo, puede combinar el nombre del nodo y otro atributo del nodo para crear una etiqueta más informativa.
3. **tooltip**: Similar a `label`, pero para definir el texto que aparecerá como un tooltip (información emergente) cuando se pase el cursor sobre el nodo en una visualización interactiva.
4. **fontname**: El nombre de la fuente que se utilizará para el texto del nodo.
5. **inherit**: Un parámetro lógico (`TRUE` o `FALSE`) que determina si el nodo heredará los estilos de su nodo padre.
6. **keepExisting**: Un parámetro lógico (`TRUE` o `FALSE`) que especifica si se deben mantener los estilos existentes que ya se han definido para los nodos.

   Además de estos argumentos, hay otros estilos que se pueden configurar, como:

7. **color**: El color del texto del nodo.
8. **fillcolor**: El color de fondo del nodo.
9. **fontcolor**: El color de la fuente del nodo.
10. **fontsize**: El tamaño de la fuente del nodo.
11. **height / width**: Altura y anchura del nodo.
12. **penwidth**: El ancho del borde del nodo.
13. **shape**: La forma del nodo (por ejemplo, caja, elipse, círculo).

Estos estilos permiten una personalización detallada de cómo se presentan los nodos en una visualización, lo que puede ser útil para resaltar ciertos aspectos de los datos o hacer que la visualización sea más legible y estéticamente agradable.

B.2.5. SetEdgeStyle

La función `SetEdgeStyle` permite personalizar el estilo de las aristas en la representación gráfica de un árbol. Esta personalización es útil para hacer que la visualización de un árbol sea más informativa y estéticamente agradable. Aquí están los argumentos más importantes de esta función:

1. **tree**: Se refiere al árbol completo al que se aplicarán los estilos de las aristas.
2. **arrowhead**: Define el estilo del extremo de la arista que apunta al nodo hijo. Por ejemplo, "none"significa que no habrá flecha.
3. **label**: Una función que toma un nodo como argumento y devuelve el texto que se usará como etiqueta en la arista.
4. **fontname**: El nombre de la fuente que se usará para el texto de las etiquetas de las aristas.
5. **penwidth**: Una función que toma un nodo como argumento y devuelve el ancho de la línea de la arista. Esto puede variar dependiendo de los atributos del nodo, por ejemplo, el tamaño del nodo o el valor de un dato asociado al mismo.
6. **color**: Una función que toma un nodo como argumento y devuelve el color de la arista. Esto permite, por ejemplo, variar el color de las aristas basándose en ciertas propiedades de los nodos o sus datos asociados.
7. **inherit**: Un parámetro lógico (TRUE o FALSE) que determina si las aristas heredarán los estilos de las aristas de su nodo padre.
8. **keepExisting**: Un parámetro lógico (TRUE o FALSE) que especifica si se deben mantener los estilos existentes que ya se han definido para las aristas.

Además de estos argumentos, se pueden establecer otros estilos comunes en las aristas, como:

- **color**: El color de la arista.
- **fontcolor**: El color de la fuente de la etiqueta de la arista.
- **fontsize**: El tamaño de la fuente de la etiqueta de la arista.
- **dir**: La dirección de la arista (por ejemplo, hacia adelante, hacia atrás, ambos, ninguno).

Estos estilos son importantes para controlar la apariencia visual de cómo se conectan los nodos en el árbol y pueden ser útiles para resaltar o diferenciar rutas o relaciones dentro de la estructura del árbol.

B.2.6. Otros métodos

Algunos otros métodos del paquete `data.tree` de R son funciones que permiten manipular y gestionar estructuras de datos jerárquicas. Aquí hay una breve explicación de algunos de los métodos disponibles (estos no han sido utilizados en la práctica).

1. **Node$addChild**: Añade un nodo hijo al nodo actual.
2. **Node$addSibling**: Añade un nuevo nodo hermano al mismo nivel que el nodo actual.
3. **Node$addSiblingNode**: Añade un nodo hermano ya existente como objeto `Node` al mismo nivel que el nodo actual.
4. **Node$removeChild**: Elimina un nodo hijo del nodo actual.
5. **Node$removeAttribute**: Elimina un atributo del nodo actual.
6. **Node$sort**: Ordena los hijos de un nodo basándose en una función o un criterio de ordenación.
7. **Node$revert**: Invierte el orden de los hijos del nodo actual.
8. **Node$prune**: Elimina todos los nodos hijos del nodo actual, simplificando el árbol.
9. **Node$climb**: Se mueve hacia arriba en la jerarquía del árbol, generalmente para llegar al nodo padre o ancestros.
10. **Node$navigate**: Navega a otro nodo utilizando una ruta relativa desde el nodo actual.
11. **Node$get**: Recupera valores o atributos de los nodos del árbol.
12. **Node$do**: Ejecuta una función sobre un conjunto de nodos.
13. **Node$set**: Asigna valores o atributos a los nodos del árbol.
14. **Node$clone**: Crea una copia completa del nodo y su subárbol.

Estos métodos permiten un control detallado y una manipulación flexible de los datos jerárquicos, lo que los hace fundamentales para una amplia gama de tareas analíticas y de visualización de datos.

B.3. DiagrammeR

El paquete `DiagrammeR` de R es una herramienta para la creación, modificación, análisis y visualización de gráficos de redes y estructuras de grafos. Permite la adición y eliminación paso a paso de nodos y aristas, y trabaja con datos en tablas para la adición masiva de nodos, aristas y metadatos asociados. Una selección de algoritmos de gráficos está disponible para el análisis de estas estructuras, y las visualizaciones pueden aprovechar las propiedades

estéticas asignadas a nodos y aristas.

Este paquete se relaciona con `data.tree` en la capacidad de visualizar estructuras de datos jerárquicos. Mientras `data.tree` es eficiente para crear y manipular datos jerárquicos, `DiagrammeR` puede tomar estos datos y representarlos visualmente como gráficos o redes, lo que puede ser particularmente útil para la visualización de árboles de decisiones o estructuras organizativas.

Por ejemplo, `data.tree` podría usarse para estructurar y manipular datos jerárquicos, y luego dicha estructura podría ser convertida a un formato adecuado para `DiagrammeR` para su visualización como un gráfico de nodos y aristas. La capacidad de `DiagrammeR` para utilizar datos de tablas para generar gráficos es especialmente útil, ya que las estructuras de `data.tree` pueden ser transformadas en tablas (nodos y aristas) y luego visualizadas utilizando `DiagrammeR`.