Fundamentos de la Ciencia de Datos

Práctica 1

Grado en Ingeniería Informática
Universidad de Alcalá

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

14 de noviembre de 2023

Índice general

Índice de figuras

Introducción

El lenguaje R

El lenguaje R, es un software de uso gratuito comúnmente usado en tareas relacionadas con la estadística, como el análisis o visualización de datos; o en general la propia Ciencia de Datos. Para ello cuenta con una gran cantidad de paquetes y herramientas que facilitan el trabajo.

Figura 1: Logo del lenguaje R

El CRAN (Comprehensive R Archive Network, `https://cran.r-project.org/`) es un repositorio de recursos en línea que se utiliza para facilitar la distribución, el intercambio y el acceso a una amplia gama de software y paquetes relacionados con el lenguaje de programación R. La página web de CRAN sirve como el portal central para acceder a estos recursos y ofrece una variedad de apartados y enlaces útiles para los usuarios de R. A continuación, proporcionamos una descripción de los distintos enlaces a los que se puede acceder desde la página principal del CRAN:

- Mirrors: Esta sección permite a los usuarios seleccionar un espejo (mirror) cercano para descargar paquetes y recursos. Los espejos son servidores que almacenan copias de los paquetes y datos de CRAN, lo que ayuda a mejorar la velocidad de descarga y la disponibilidad de los recursos.
- What's new?: En esta sección, los usuarios pueden encontrar información sobre las últimas actualizaciones y novedades en el mundo de R y los paquetes disponibles en CRAN. Esto es útil para estar al tanto de las últimas características y mejoras.
- Search: El enlace "Search" permite a los usuarios buscar paquetes y recursos específicos en el repositorio de CRAN. Además, se puede utilizar la función de búsqueda avanzada del motor de búsqueda de Google.

- CRAN Team: Aquí se puede encontrar información sobre las personas y equipos que trabajan en el mantenimiento y desarrollo de CRAN. Es útil para conocer a las personas detrás de esta valiosa fuente de recursos.

- About R: Esta sección proporciona información sobre el lenguaje de programación R en general. Incluye enlaces a la página de inicio de R y a "The R Journal", una publicación académica relacionada con R.

- Software: Esta sección ofrece acceso a diversas fuentes y binarios relacionados con R, lo que permite a los usuarios descargar e instalar R en su sistema. También proporciona acceso a paquetes, Task Views y otros recursos.

- Documentation: Aquí los usuarios pueden encontrar documentación esencial relacionada con R. Esto incluye manuales, preguntas frecuentes (FAQs) y contribuciones de la comunidad para ayudar a los usuarios a comprender y utilizar R de manera efectiva.

En R, los paquetes son extensiones de software que contienen funciones, datos y documentación para realizar tareas específicas. Antes de utilizar un paquete, debes instalarlo y cargarlo en tu sesión de R. Algunas de las funciones más útiles para preparar los paquetes de un proyecto son:

- **Paquetes por defecto**:
  Mediante `getOption("defaultPackages")` se muestra una lista de los paquetes que se cargan automáticamente cuando inicias una sesión de R. Son los paquetes básicos que R carga por defecto. Para cambiar la lista de archivos que R carga por defecto podemos acceder a la siguiente ubicación (instalación de R por defecto):
  `C:/Program Files/R/R-4.3.1/library/base/R/RProfile`, y modificar el archivo como se observa en el Código 1, añadiendo al vector `dp` los paquetes que deseemos.

```
46 local({dp <- Sys.getenv("R_DEFAULT_PACKAGES")
47        if(identical(dp, "")) ## it fact methods is done first
48            dp <- c("datasets", "utils", "grDevices", "graphics",
49                    "stats", "methods", "arules")
50        else if(identical(dp, "NULL")) dp <- character(0)
51        else dp <- strsplit(dp, ",")[[1]]
52        dp <- sub("[[:blank:]]*([[:alnum:]]+)", "\\1", dp) # strip
              whitespace
53        options(defaultPackages = dp)
54    })
```

Código 1: Modificación en fichero `Rprofile`

- **Instalación de paquetes**:
  La instalación de paquetes puede ser realizada de tres formas distintas:

  1. `install.packages("nombre_del_paquete")`
     A esta función se le debe pasar por parámetro el nombre del paquete que se desea instalar.

2. `install.packages(übicacion_del_paquete", rep=NULL)`
   A la función también se le puede pasar por parámetros la ubicación del archivo, que recomendablemente debe estar en una carpeta temporal en "c:/", este archivo lo descargamos desde:
   `https://cran.r-project.org/` > `Packages` > `Table of available packages, sorted by name` > Elegimos el paquete y descargamos la versión `r-release` de la sección `Windows binaries`.
3. `utils:::menuInstallPkgs()`
   Tras la ejecución de este comando aparecerá una ventana donde se podrá elegir el mirror desde el que se va a descargar el paquete, y tras elegir el mirror (Spain (Madrid) en nuestro caso), aparece otra ventana donde se puede elegir el paquete que se quiere instalar, tras hacer doble click, este se instalará automáticamente.

- **Información de un paquete**:
  Cuando ejecutas `library(help="nombre_del_paquete")`, R te mostrará información detallada sobre el paquete especificado. Esto incluye una descripción del paquete y una lista de las funciones que contiene, junto con sus descripciones.

- **Carga de paquetes**:
  Si ejecutas `library(nombre_del_paquete)` con el nombre de un paquete, R cargará el paquete en tu sesión para que puedas utilizar sus funciones y objetos.

- **Lista de paquetes instalados**:
  Al ejecutar `library()` sin argumentos, R te mostrará una lista de los paquetes que están actualmente cargados en tu sesión de R. Esto te permite verificar qué paquetes están disponibles para su uso.

- **Lista de paquetes cargados**:
  Mediante `search()` podemos ver un listado completo de los paquetes actualmente cargados en memoria.

A parte, es recomendable descargar y conocer a conciencia el manual y las viñetas de todos los paquetes que usemos en nuestros proyectos (disponible en la página web del CRAN).

El lenguaje LaTeX

Para la realización de esta práctica, se empleará el concepto de **programación literaria**, que consiste en crear un documento en el que se combine texto con código, de manera que este se pueda explicar y entender de una manera mucho más sencilla. Una forma de realizar esto con código R, es el uso del lenguaje LaTeX, que es un sistema de composición de documentos enfocado al ámbito científico. Es algo similar a un lenguaje de marcas con el que poder definir la estructura de un documento, pero cuenta con la particularidad de que es un lenguaje Turing–completo, por lo que cualquier algoritmo puede ser implementado dando una mayor flexibilidad, aunque no sea su objetivo principal. Veremos ahora los pasos seguidos para su instalación. Para poder trabajar, lo mínimo que necesitaremos es un

compilador de LaTeX, en este caso se ha optado por la distribución MiKTeX que lo incluye, ya que estamos trabajando en Windows. Además, para una mayor comodidad trabajando con el código, se ha optado por el IDE TeXstudio, uno de los más conocidos en la comunidad.

Una vez hemos tratado ambos lenguajes, necesitamos entender con qué tipos de extensiones se suelen trabajar para ver el proceso de integración con R (sin entrar en profundidad). Estas dependen de cómo queremos almacenar nuestro documento, o cómo están almacenadas dependencias de estos, como por ejemplo, imágenes. Esta tarea se realiza usando un compilador u otro.

Para ello nos fijaremos en la Figura 2. Por ahora nos quedaremos con las extensiones que trabajaremos más a menudo, que serán `.Rnw`, `.tex`, y `.pdf`. La primera de ellas representan los archivos que tienen código LaTeX y R "mezclado", la segunda aquellos que contienen código LaTeX puro, y la última nuestro documento final.

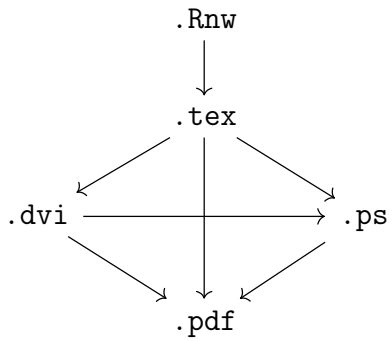

Figura 2: Esquema de extensiones en LaTeX

Existen dos herramientas que nos permiten trabajar con archivos `.Rnw`, estas son Sweave y Knitr. A pesar de que en la asignatura ha sido propuesta la primera de ellas, optaremos por la segunda, pues existieron diversos errores al compilar archivos con esta, y al ser más antigua, los documentos finales tenían menos calidad. Knitr nos ofrece mayor calidad y un mejor formato en el código fuente R mostrado. Para instalarla basta con escribir `install.packages('knitr')` en una consola de R. Sweave viene ya por defecto con `utils`.

Por último, se explicará cómo hemos agilizado el proceso de trabajo con Knitr y TeXStudio. Lo primero será hacer que R cargue por defecto Knitr, para ello modificaremos el archivo `Rprofile` en `libray/base/R` dentro de la carpeta de instalación de R, añadiendo `knitr` al resto de paquetes que carga por defecto. Una vez hecho esto, iremos a la configuración de TeXStudio, y aquí a *Compilar*. En la zona *Órdenes de usuario*, crearemos una nueva tal y como se ve en la Figura 3, de manera que le digamos dónde están los binarios de R, para que pueda crear un fichero `.tex`, y posteriormente invocar a nuestro compilador, para obtener nuestro documento en `.pdf`. TeXStudio se encargará de reemplazar el símbolo % por el nombre del archivo que le ordenamos compilar.

Figura 3: Creación de orden de usuario

Ahora basta modificar el botón verde del IDE para que en vez de invocar al compilador pdfLaTeX, lleve a cabo la instrucción que le hemos dado. Para ello volveremos al menú de compilación en el que nos ubicábamos previamente, y observaremos la sección de *Meta-Órdenes*. Modificaremos el valor del campo *Compilador por defecto*, escribiendo `txs:///knitr` para que se ejecute la orden que previamente hemos creado, o podemos hacerlo de manera gráfica como se observa en la Figura 4. Ahora bastará pulsar el botón verde o F5 para ver a nuestra izquierda el código de nuestro documento, y a la derecha actualizado, el documento PDF final.

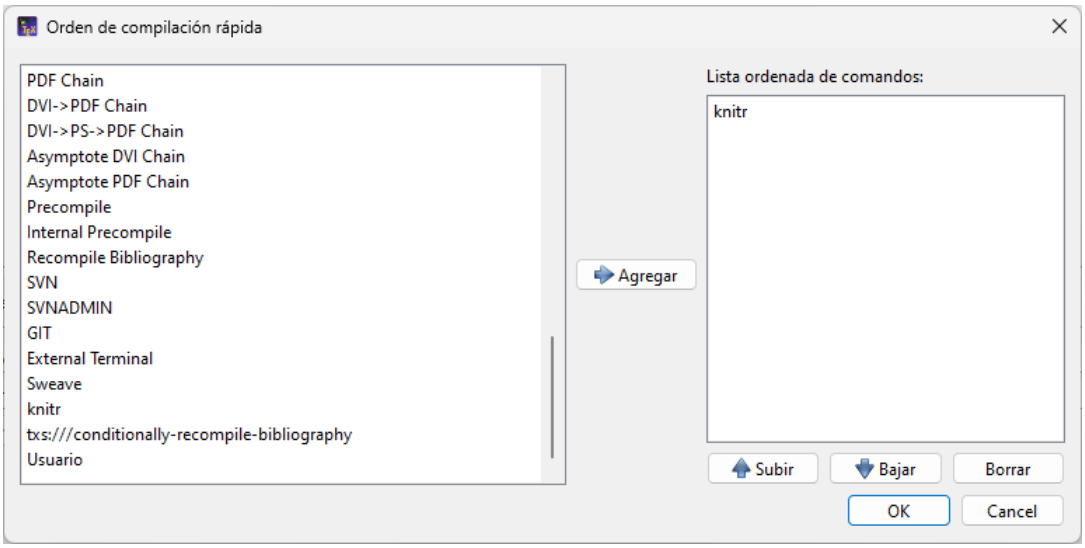

Figura 4: Modificación de compilación

Por último, para llevar un mejor control de versiones del proyecto, y de coordinación entre los miembros del grupo, se usará un repositorio de GitHub. Añadiremos un archivo `.gitignore` para no cargar en el repositorio los archivos temporales generados durante la compilación. Otra alternativa que se podría haber usado, es usar Overleaf (aquí usaríamos la extensión `.Rtex` en vez de `.Rnw`), ya que no sería necesaria la instalación de ningún software, y también trabaja con Knitr. Sin embargo, la integración de GitHub en Overleaf es de pago, por lo que optamos por usar la configuración explicada hasta el momento, para poder tener un mejor control de versiones sin coste alguno.

Por último, mencionar que al igual que R posee su repositorio de paquetes (que ya hemos visto que incluye más cosas) llamado CRAN, LaTeX que en realidad es "un subconjunto" del lenguaje TeX, también tiene su propio portal llamado CTAN o Comprehensive TeX Archive Network (`https://www.ctan.org/`) de donde se descargan los paquetes y otros materiales para el lenguaje.

Figura 5: Mascota de TeX y el CTAN

Parte 1

Ejercicios guiados

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

1.1. Descripción de los datos

"*El primer conjunto de datos, que se empleará para realizar el análisis de descripción de datos, estará formado por datos de una característica cualitativa, nombre, y otra cuantitativa, radio, de los satélites menores de Urano, es decir, aquellos que tienen un radio menor de 50 Km, dichos datos, los primeros cualitativos nominales, y los segundos cuantitativos continuos, son: (Nombre, radio en Km): Cordelia, 13; Ofelia, 16; Bianca, 22; Crésida, 33; Desdémona, 29; Julieta, 42; Rosalinda, 27; Belinda, 34; Luna-1986U10, 20; Calíbano, 30; Luna-999U1, 20; Luna 1999U2, 15.*"

Para comenzar con la resolución de este ejercicio, deberemos escribir los datos en un fichero `.txt`, cumpliendo las siguientes normas:

- 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 a estas normas, copiamos los datos en un fichero llamado `satelites.txt`, y lo cargamos en R de la siguiente manera:

```
s <- read.table("data/satelites.txt")
print(s)

##          nombre radio
## 1      Cordelia    13
## 2        Ofelia    16
## 3        Bianca    22
## 4       Crésida    33
## 5     Desdémona    29
## 6       Julieta    42
## 7     Rosalinda    27
## 8       Belinda    34
## 9  Luna-1986U10    20
## 10     Calíbano    30
## 11   Luna-999U1    20
## 12  Luna-1999U2    15
```

Ahora en la variable `s` tenemos un dataframe con los datos de nuestros satélites. En los dataframes se accede por `[fila, columna]`, y también podemos consultar las dimensiones con la función `dim`. Sería de esperar que nos dijera que tiene 12 filas (los 12 datos), y 2 columnas (`nombre` y `radio`).

```
dim(s)

## [1] 12  2
```

También podemos ordenar el dataframe, en función de una de las magnitudes (columnas), usando la función `order` aplicando recursivamente el concepto de acceder por filas y columnas. Veamos un ejemplo, si en `s` teníamos guardado nuestro dataframe, y queremos ordenar por `radio`, la manera de hacerlo sería la siguiente:

```
s_ordered <- s[order(s$radio), ]
print(s_ordered)

##          nombre radio
## 1      Cordelia    13
## 12  Luna-1999U2    15
## 2        Ofelia    16
## 9  Luna-1986U10    20
## 11   Luna-999U1    20
## 3        Bianca    22
## 7     Rosalinda    27
## 5     Desdémona    29
## 10     Calíbano    30
## 4       Crésida    33
```

```
## 8       Belinda    34
## 6       Julieta    42
```

Podemos introducir nuevos criterios a la ordenación, como por ejemplo, hacerlo en orden descendente. Para esto usaremos la función `rev`.

```
s_ordered_rev <- s[rev(order(s$radio)), ]
print(s_ordered_rev)

##          nombre radio
## 6       Julieta    42
## 8       Belinda    34
## 4       Crésida    33
## 10     Calíbano    30
## 5     Desdémona    29
## 7     Rosalinda    27
## 3        Bianca    22
## 11   Luna-999U1    20
## 9  Luna-1986U10    20
## 2        Ofelia    16
## 12  Luna-1999U2    15
## 1      Cordelia    13
```

También suele ser útil conocer cuántos elementos tiene una columna. Podemos averiguarlo con la función `length`, veamos un ejemplo.

```
length(s$radio)

## [1] 12
```

Otro valor que nos podemos plantear calcular es el rango. Para ello podemos usar las funciones `max` y `min`. Debemos tener cuidado con la función `range` y no confundirnos, pues nos dará los valores máximo y mínimo.

```
r <- max(s$radio) - min(s$radio)
print(r)

## [1] 29

range(s$radio)

## [1] 13 42
```

Para una mejor lectura, podemos cambiar la forma de obtener la columna de los radios:

```
radio <- s$radio
```

La idea de calcular la diferencia de el máximo y el mínimo a mano parece funcionar, sin embargo, para futuros casos sería más ágil tener codificada una función como la siguiente.

```
rango <- function(radio){max(radio) - min(radio)}
rango(radio)

## [1] 29
```

Sin embargo, al salir de R, la definición de la función se pierde, por lo que deberemos guardarla en un fichero, y posteriormente cargarlo en futuras ejecuciones. Lo haremos de la siguiente manera:

```
dump("rango", file = "fn/rango.R")
source("fn/rango.R")
```

Volviendo al estudio de nuestros datos, veamos cómo calcular las diferentes frecuencias. Como en R no existe una función para las frecuencias relativas, se definirá y guardará una propia.

```
fabs_radio <- table(radio)
fabsacum_radio <- cumsum(fabs_radio)
frecrel <- function(r){table(r)/length(r)}
dump("frecrel", file = "fn/frecrel.R")

print(fabs_radio)

## radio
## 13 15 16 20 22 27 29 30 33 34 42
##  1  1  1  2  1  1  1  1  1  1  1

print(fabsacum_radio)

## 13 15 16 20 22 27 29 30 33 34 42
##  1  2  3  5  6  7  8  9 10 11 12

print(frecrel(radio))

## r
##         13         15         16         20         22         27         29
## 0.08333333 0.08333333 0.08333333 0.16666667 0.08333333 0.08333333 0.08333333
##         30         33         34         42
## 0.08333333 0.08333333 0.08333333 0.08333333
```

Otro valor que podemos calcular es la media aritmética de los datos, para ello se cuenta con la función `mean`.

```
mr <- mean(radio)
print(mr)

## [1] 25.08333
```

Ahora calcularemos la desviación típica, para ello se cuenta con la función `sd`.

```
sdr <- sd(radio)
print(sdr)

## [1] 8.857029
```

Sin embargo, el resultado obtenido no es el esperado. Esto se debe a que esta función realiza el siguiente cálculo

$$\sigma = \sqrt{\frac{\sum_{i=0}^{n}(x_i - \bar{x})^2}{n-1}}$$

que es más utilizado en inferencia estadística, porque hace que se parezca más a una campana de Gauss (menos sesgo), mientras que la fórmula vista en clase utiliza un factor de n en vez de $n-1$ en el denominador (dentro de la raíz). Para ello, el profesor lo corrigió de la siguiente manera:

```
sdr2 <- sqrt((sdr^2)*(length(radio)-1)/length(radio))
print(sdr2)

## [1] 8.47996
```

En realidad lo que se está realizando es el siguiente "ajuste":

$$\sigma' = \sqrt{\sigma^2 \cdot \frac{n-1}{n}}$$

Una vez hemos visto cómo se calcula la desviación típica, podremos ver cómo calcular la varianza. Como sabemos que es el cuadrado de la desviación típica, bastaría con elevar al cuadrado si no fuera por el "fallo" de $n-1$ visto previamente. En este caso, el profesor lo arregló para el caso particular de la siguiente manera:

```
varr <- var(radio)
varr <- 11/12 * varr
print(varr)

## [1] 71.90972
```

Otro de los valores que se ha enseñado cómo calcular, es la mediana. Para este caso existe la función `median`.

```
medianr <- median(radio)
print(medianr)

## [1] 24.5
```

En último lugar, el profesor enseñó cómo calcular cuantiles, y para ello mostró la función `quantile`, pero se mencionó que se obtienen resultados diferentes a los esperados debido a la forma que tiene de calcularlos, y se deberá programar. Aquí se muestra un ejemplo de cómo se calcularía el primer cuartil.

```
cuar1 <- quantile(radio, 0.25)
print(cuar1)

## 25%
##  19
```

Como añadido, el profesor explicó cómo abrir un ejemplo de Sweave, cómo pasarlo a un fichero que LaTeX pudiese leer, y cómo compilarlo a PDF. Las instrucciones son las que se verán a continuación, aunque como ya se explicó en la introducción, usaremos otra forma de trabajar con estos archivos a lo largo de la práctica.

```
rnwfile<-system.file("Sweave", "example-1.Rnw", package="utils")
Sweave(rnwfile)
tools::texi2pdf("example-1.tex")
```

1.2. Asociación

"*El segundo conjunto de datos, que se empleará para realizar el análisis de asociación, estará formado por las siguientes 6 cestas de la compra:* {*Pan, Agua, Leche, Naranjas*}, {*Pan, Agua, Café, Leche*}, {*Pan, Agua, Leche*}, {*Pan, Café, Leche*}, {*Pan, Agua*}, {*Leche*}."

Lo primero explicado por el profesor fue la preparación de los datos proporcionados para que `arules` fuese capaz de tratar con ellos. Para ello se introduce una matriz de ceros y unos mediante el paquete `Matrix`, que indique en cada suceso, qué elementos contiene. La matriz es la siguiente:

$$\begin{pmatrix} 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Además, deberemos indicar las dimensiones de esta matriz (6×5), que estamos introduciendo los datos por filas (`byrow=TRUE`), y con `dimnames` ponemos los nombres a las filas y las columnas. El código es el siguiente.

```
muestra <- Matrix(c(1, 1, 0, 1, 1,
1, 1, 1, 1, 0,
1, 1, 0, 1, 0,
1, 0, 1, 1, 0,
1, 1, 0, 0, 0,
0, 0, 0, 1, 0), 6, 5, byrow = TRUE, dimnames = list(
c("suceso1", "suceso2", "suceso3", "suceso4", "suceso5", "suceso6"),
c("Pan", "Agua", "Café", "Leche", "Naranjas")), sparse=TRUE)
muestra

## 6 x 5 sparse Matrix of class "dgCMatrix"
##         Pan Agua Café Leche Naranjas
## suceso1   1    1    .     1        1
## suceso2   1    1    1     1        .
## suceso3   1    1    .     1        .
## suceso4   1    .    1     1        .
## suceso5   1    1    .     .        .
## suceso6   .    .    .     1        .
```

A continuación, se ha enseñado cómo mostrar la matriz con puntos y barras, en vez de con unos y ceros. Se consigue con la función `as` y el parámetro `nsparseMatrix`.

```
muestrangCMatrix <- as(muestra, "nsparseMatrix")
muestrangCMatrix

## 6 x 5 sparse Matrix of class "ngCMatrix"
##         Pan Agua Café Leche Naranjas
## suceso1   |    |    .     |        |
## suceso2   |    |    |     |        .
## suceso3   |    |    .     |        .
## suceso4   |    .    |     |        .
## suceso5   |    |    .     .        .
## suceso6   .    .    .     |        .
```

Sin embargo, para el algoritmo debemos pasarle justo la transpuesta de la matriz con la que trabajamos, por ello se utiliza la función `t`.

```
transpmuestrangCMatrix <- t(muestrangCMatrix)
transpmuestrangCMatrix

## 5 x 6 sparse Matrix of class "ngCMatrix"
##          suceso1 suceso2 suceso3 suceso4 suceso5 suceso6
## Pan            |       |       |       |       .
## Agua           |       |       .       |       .
## Café           .       |       .       |       .       .
```

```
## Leche          |       |       |       |       .       |
## Naranjas       |       .       .       .       .       .
```

Podemos consultar algunos datos acerca de los datos de nuestra matriz podemos usar la función `as` con el parámetro `transactions`. Además, con `summary` podemos ver un resumen de algunos parámetros básicos de los datos que contiene la matriz.

```
transacciones = as(transpmuestrangCMatrix, "transactions")
transacciones

## transactions in sparse format with
##  6 transactions (rows) and
##  5 items (columns)

summary(transacciones)

## transactions as itemMatrix in sparse format with
##  6 rows (elements/itemsets/transactions) and
##  5 columns (items) and a density of 0.5666667
##
## most frequent items:
##      Pan    Leche     Agua     Café Naranjas  (Other)
##        5        5        4        2        1        0
##
## element (itemset/transaction) length distribution:
## sizes
## 1 2 3 4
## 1 1 2 2
##
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   1.000   2.250   3.000   2.833   3.750   4.000
##
## includes extended item information - examples:
##   labels
## 1    Pan
## 2   Agua
## 3   Café
##
## includes extended transaction information - examples:
##   itemsetID
## 1   suceso1
## 2   suceso2
## 3   suceso3
```

Finalmente, podemos ejecutar el algoritmo apriori llamando a la función `apriori` del paquete `arules`. Definimos el soporte con un valor del 50 %, y la confianza con 80 %.

```
asociaciones = apriori(transacciones, parameter =
list(support = 0.5, confidence = 0.8))

## Apriori
##
## Parameter specification:
##  confidence minval smax arem  aval originalSupport maxtime support minlen
##         0.8    0.1    1 none FALSE            TRUE       5     0.5      1
##  maxlen target  ext
##      10  rules TRUE
##
## Algorithmic control:
##  filter tree heap memopt load sort verbose
##     0.1 TRUE TRUE  FALSE TRUE    2    TRUE
##
## Absolute minimum support count: 3
##
## set item appearances ...[0 item(s)] done [0.00s].
## set transactions ...[5 item(s), 6 transaction(s)] done [0.00s].
## sorting and recoding items ... [3 item(s)] done [0.00s].
## creating transaction tree ... done [0.00s].
## checking subsets of size 1 2 3 done [0.00s].
## writing ... [7 rule(s)] done [0.00s].
## creating S4 object  ... done [0.00s].
```

Podemos ver el resultado del algoritmo con la función `inspect`.

```
inspect(asociaciones)

##     lhs              rhs     support   confidence coverage  lift count
## [1] {}            => {Leche} 0.8333333 0.8333333  1.0000000 1.00 5
## [2] {}            => {Pan}   0.8333333 0.8333333  1.0000000 1.00 5
## [3] {Agua}        => {Pan}   0.6666667 1.0000000  0.6666667 1.20 4
## [4] {Pan}         => {Agua}  0.6666667 0.8000000  0.8333333 1.20 4
## [5] {Leche}       => {Pan}   0.6666667 0.8000000  0.8333333 0.96 4
## [6] {Pan}         => {Leche} 0.6666667 0.8000000  0.8333333 0.96 4
## [7] {Agua, Leche} => {Pan}   0.5000000 1.0000000  0.5000000 1.20 3
```

Aquí observamos el resultado del algoritmo. Debemos ignorar las dos primeras filas, pues no tiene sentido alguno que $\varnothing \to \{\text{Leche}\}$, o que $\varnothing \to \{\text{Pan}\}$, aparecen por cómo el autor del paquete codificó el algoritmo. En el resto de casos $A \to B$, nos indica cómo de probable es comprar B cuando se compra A (en función del soporte y la confianza provistas).

1.3. Detección de datos anómalos

1.3.1. Primer ejercicio

"*El tercer conjunto de datos, que se empleará para realizar el análisis de detección de datos anómalos utilizando técnicas con base estadística, estará formado por los siguientes 7 valores de resistencia y densidad para diferentes tipos de hormigón* {*Resistencia, Densidad*}*:* {*3, 2; 3.5, 12; 4.7, 4.1; 5.2, 4.9; 7.1, 6.1; 6.2, 5.2; 14, 5.3*}*. Aplicar las medidas de ordenación a la resistencia y las de dispersión a la densidad.*"

- **Caja y Bigotes**

  Es una herramienta gráfica utilizada en estadística para representar la distribución de un conjunto de datos y detectar sucesos anómalos o outliers.

  Para realizar el cálculo de los outliers en clase, se introdujeron los datos en una matriz mediante `matrix`, luego se transpuso y se paso a un `dataframe`. Para esta primera técnica vamos a usar la primera columna (*resistencia*).

```
muestra=t(matrix(c(3,2,3.5,12,4.7,4.1,5.2,4.9,7.1,6.1,6.2,5.2,14,5.3),
        2,7,dimnames=list(c("resistencia","densidad"))))

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

```
##   resistencia densidad
## 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 forma de obtener los outliers es mediante la función `boxplot`, pasándole la columna de los datos, el grado de outlier (`range`) o distancia a la que el suceso se considera outlier, y por último `plot=FALSE`, que se usa para no mostrar el gráfico como tal y solo sacar la información por el terminal.

```
(boxplot(muestra$r, range=1.5, plot=FALSE))
```

```
## $stats
##      [,1]
## [1,] 3.00
## [2,] 4.10
## [3,] 5.20
```

```
## [4,] 6.65
## [5,] 7.10
##
## $n
## [1] 7
##
## $conf
##          [,1]
## [1,] 3.677181
## [2,] 6.722819
##
## $out
## [1] 14
##
## $group
## [1] 1
##
## $names
## [1] "1"
```

Como podemos ver en `$out`, saca que el suceso outlier es el 14, lo cual como vimos en el ejercicio de la clase de teoría es correcto, pero, si nos fijamos en `$conf`, podemos ver los límites del intervalo, pero estos no coinciden con los que vimos en clase, esto se debe a que, al igual que ocurría al usar la función `sd` para el cálculo de la desviación, R hace una distribución, ajustándola con unos porcentajes de probabilidad. Por lo que no son los valores exactos.

Ahora vamos a hacerlo mediante el cálculo de los cuartiles como vimos en los ejercicios anteriores. Además, hemos calculado los límites del intervalo mediante la siguiente ecuación.

$$(Q_1 - d \cdot (Q_3 - Q_1),\ Q_3 + d \cdot (Q_3 - Q_1))$$

```
(cuar1r=quantile(muestra$r, 0.25))

## 25%
## 4.1

(cuar3r=quantile(muestra$r, 0.75))

##  75%
## 6.65

(int=c(cuar1r-1.5*(cuar3r-cuar1r), cuar3r+1.5*(cuar3r-cuar1r)))
```

```
##    25%    75%
##  0.275 10.475
```

Por último, iteramos los elementos de la muestra y comprobamos para cada uno si es menor que el límite inferior del intervalo o si es mayor que el límite superior.

```
for (i in 1:length(muestra$r)) {
        if (muestra$r[i] < int[1] || muestra$r[i] > int[2]) {
                print("el suceso")
                print(i)
                print("es un suceso anómalo o outlier")
        }
}

## [1] "el suceso"
## [1] 7
## [1] "es un suceso anómalo o outlier"
```

En este ejercicio, al igual que vimos en clase, el suceso outlier es el que está en la posición número 7, es decir el elemento 14.

- **Media y Desviación**

  Otra manera de buscar sucesos outliers usando técnicas con base estadística es mediante el uso de la media y la desviación típica. Aquí vamos a usar la otra columna del dataframe (*densidad*), mediante el uso de las funciones `mean` y `sd` sacamos ambos valores, que unimos en los límites del intervalo mediante la siguiente ecuación.

$$\bar{X}_a - d \cdot S_a,\ \bar{X}_a + d \cdot S_a$$

```
(media = mean(muestra$d))

## [1] 5.657143

(desv = sd(muestra$d))

## [1] 3.085913

(int=c(media-2*desv, media+2*desv))

## [1] -0.5146825 11.8289682
```

Podemos ver que los valores inferior y superior del intervalo no son iguales que en el ejercicio que calculamos en clase, esto se debe a que, como ya vimos, la desviación típica se calcula mediante una distribución probabilistica, por lo que arreglando el cálculo de la desviación quedaría así.

```
(desv = sqrt(sd(muestra$d)^2 *
        (length(muestra$d)-1) / length(muestra$d)))

## [1] 2.857

(int=c(media-2*desv, media+2*desv))

## [1] -0.05685714 11.37114285

for (i in 1:length(muestra$d)) {
        if (muestra$d[i] < int[1] || muestra$d[i] > int[2]) {
                print("el suceso")
                print(i)
                print("es un suceso anómalo o outlier")
        }
}

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

Como resultado, obtenemos los valores del intervalo correctos y podemos ver que el suceso anómalo es el que se encuentra en la posición 2, es decir el elemento 12.

1.3.2. Segundo ejercicio

"*El cuarto conjunto de datos, que se empleará para realizar el análisis de detección de datos anómalos utilizando técnicas basadas en la proximidad y en la densidad, estará formado por las siguientes 5 calificaciones de estudiantes: 1. {4, 4}; 2. {4, 3}; 3. {5, 5}; 4. {1, 1}; 5. {5, 4} donde las características de las calificaciones son: (Teoría, Laboratorio).*"

- **K-Vecinos**

  El algoritmo k-vecinos para identificar outliers basado en distancias consiste en calcular la distancia de un punto a sus k vecinos más cercanos y considerar los puntos con distancias mayores a un cierto valor (grado de outlier) como anómalos.

  En clase añadimos todos los datos de la nueva muestra a una matriz, que luego transpusimos para tener los datos dispuestos como nos requería el cálculo de distancias. Calculamos las distancias *euclídeas* mediante la función `dist` y lo guardamos en una

matriz de 5×5, para poder visualizar la distancia de cada punto hacia el resto. El cálculo de esta distancia se realiza de la siguiente manera.

$$d(A, B) = \sqrt{(X_B - X_A)^2 + (Y_B - Y_A)^2}$$

```
muestra=matrix(c(4,4,4,3,5,5,1,1,5,4),2,5)

muestra=t(muestra)

distancias=as.matrix(dist(muestra))

(distancias=matrix(distancias,5,5))
```

```
##          [,1]     [,2]     [,3]     [,4]     [,5]
## [1,] 0.000000 1.000000 1.414214 4.242641 1.000000
## [2,] 1.000000 0.000000 2.236068 3.605551 1.414214
## [3,] 1.414214 2.236068 0.000000 5.656854 1.000000
## [4,] 4.242641 3.605551 5.656854 0.000000 5.000000
## [5,] 1.000000 1.414214 1.000000 5.000000 0.000000
```

Mediante el siguiente bucle, iteramos cada punto y ordenamos de menor a mayor las distancias hacia los demás puntos.

```
for (i in 1:5) {
        distancias[,i] = sort(distancias[,i])
}

(distanciasordenadas=distancias)
```

```
##          [,1]     [,2]     [,3]     [,4]     [,5]
## [1,] 0.000000 0.000000 0.000000 0.000000 0.000000
## [2,] 1.000000 1.000000 1.000000 3.605551 1.000000
## [3,] 1.000000 1.414214 1.414214 4.242641 1.000000
## [4,] 1.414214 2.236068 2.236068 5.000000 1.414214
## [5,] 4.242641 3.605551 5.656854 5.656854 5.000000
```

Y por último, iteramos cada punto de la matriz de distancias ordenadas y comprobamos si el elemento k-éismo está a una distancia mayor de 2.5 (elegido arbitrariamente), donde el punto se consideraría outlier.

En el código se puede ver que se elige el elemento 4 en vez del 3 que era la `K` elegida para resolverlo, esto se debe a que la primera fila de valores, son las distancias desde cada punto hacia sí mismo, es decir, siempre 0, por lo que los descartamos en el conteo de los `K` elementos.

```
for (i in 1:5) {
        if (distanciasordenadas[4,i] > 2.5) {
                print(i)
                print("es un suceso anómalo o outlier")
        }
}
```

```
## [1] 4
## [1] "es un suceso anómalo o outlier"
```

Podemos ver que como salida obtenemos que el punto 4 es un suceso anómalo o outlier, al igual que ocurría en el ejercicio resuelto en teoría.

- **LOF (Local Outlier Factor)**

  Como último ejercicio realizado en clase, se ha empleado el algoritmo LOF para identificar outliers basado en distancias, que evalúa la densidad local de puntos en relación con sus vecinos, identificando valores anómalos en regiones menos densas que el entorno circundante.

  Obtenemos una matriz de distancias haciendo uso igual que antes de la función `dist`, pero en este caso pasándole como parámetro el método que queremos que use para calcular la distancia (`method="manhattan"`), en este caso el método *Manhattan*, que se calcula de la siguiente manera.

$$d(A, B) = |X_B - X_A| + |Y_B - Y_A|$$

```
(distanciasM=as.matrix(dist(muestra, method="manhattan")))
```

```
##   1 2 3 4 5
## 1 0 1 2 6 1
## 2 1 0 3 5 2
## 3 2 3 0 8 1
## 4 6 5 8 0 7
## 5 1 2 1 7 0
```

Para finalizar, el profesor nombró algunos paquetes que realizaban una implementación no simplificada del algoritmo, estos son `RLof`, `DDoutlier` y `DMwR`, pero nosostros en el segundo ejercicio autónomo de detección de datos anómalos, sí vamos a implementar un código propio, que realice las simplificaciones oportunas.

Parte 2

Ejercicios autónomos

2.1. Descripción de los datos

"*El primer conjunto de datos, que se empleará para realizar el análisis de descripción de datos, estará formado por datos de una característica cuantitativa, distancia, desde el domicilio de cada estudiantes hasta la Universidad, dichos datos, cuantitativos continuos, son: 16.5, 34.8, 20.7, 6.2, 4.4, 3.4, 24, 24, 32, 30, 33, 27, 15, 9.4, 2.1, 34, 24, 12, 4.4, 28, 31.4, 21.6, 3.1, 4.5, 5.1, 4, 3.2, 25, 4.5, 20, 34, 12, 12, 12, 12, 5, 19, 30, 5.5, 38, 25, 3.7, 9, 30, 13, 30, 30, 26, 30, 30, 1, 26, 22, 10, 9.7, 11, 24.1, 33, 17.2, 27, 24, 27, 21, 28, 30, 4, 46, 29, 3.7, 2.7, 8.1, 19, 16.*"

Para empezar, hemos introducido todos estos datos en un fichero CSV. Para realizar este fichero se ha abierto un Excel y hemos ido introduciendo los datos en la primera columna. Cabe destacar que la primera fila no corresponde a ningún valor ya que hemos puesto `Distancia(km)` para mantener la estructura con respecto al ejercicio guiado. Una vez hemos introducido todos los datos guardamos el fichero con extensión `.csv`. Para leer este fichero dentro de R hacemos uso de la función `read.csv`, perteneciente al paquete por defecto `utils`. Esta función en realidad es un uso diferente de la función `read.table` orientado a la lectura de ficheros CSV. Esta función lee el archivo en formato tabla y a partir de él crea un data frame haciendo corresponder las filas y las columnas de la tabla. La función que hemos utilizado es `read.csv` primeramente porque es un fichero CSV y además porque está delimitado por comas (en caso de haber estado delimitado por punto y coma habría que haber usado `read.csv2`. Observamos en la siguiente línea de código cómo hemos creado un archivo `distancia_universitarios.csv` siguiendo la estructura que hemos mencionado previamente. Con este fichero creado basta con llamar a la función `read.csv` pasando como parámetro el fichero. Si el archivo estuviera en otro directorio habría que pasar por parámetro la ruta donde se encuentra el archivo.

```
fichero = read.csv("data/distancia_universitarios.csv")
fichero
```

```
##    Distancia
## 1       16.5
```

```
## 2       34.8
## 3       20.7
## 4        6.2
## 5        4.4
## 6        3.4
## 7       24.0
## 8       24.0
## 9       32.0
## 10      30.0
## 11      33.0
## 12      27.0
## 13      15.0
## 14       9.4
## 15       2.1
## 16      34.0
## 17      24.0
## 18      12.0
## 19       4.4
## 20      28.0
## 21      31.4
## 22      21.6
## 23       3.1
## 24       4.5
## 25       5.1
## 26       4.0
## 27       3.2
## 28      25.0
## 29       4.5
## 30      20.0
## 31      34.0
## 32      12.0
## 33      12.0
## 34      12.0
## 35      12.0
## 36       5.0
## 37      19.0
## 38      30.0
## 39       5.5
## 40      38.0
## 41      25.0
## 42       3.7
## 43       9.0
## 44      30.0
## 45      13.0
```

```
## 46      30.0
## 47      30.0
## 48      26.0
## 49      30.0
## 50      30.0
## 51       1.0
## 52      26.0
## 53      22.0
## 54      10.0
## 55       9.7
## 56      11.0
## 57      24.1
## 58      33.0
## 59      17.2
## 60      27.0
## 61      24.0
## 62      27.0
## 63      21.0
## 64      28.0
## 65      30.0
## 66       4.0
## 67      46.0
## 68      29.0
## 69       3.7
## 70       2.7
## 71       8.1
## 72      19.0
## 73      16.0
```

Como podemos observar se nos ha leído correctamente el fichero, creando el dataframe con los datos correctamente leídos. Al ser un CSV delimitado por comas, hay que tener cuidado con los números decimales y separar estos por un punto, ya que si lo separamos por comas se nos leerán dos datos en vez de uno.

Como vamos a trabajar todo el rato con la columna `Distancia` del dataframe, para no tener que estar accediendo todo el rato a esta podemos definirla en una variable nueva a la que llamaremos distancias, de tal forma que no tengamos que estar haciendo `fichero$Distancia` todo el rato.

```
distancias = fichero$Distancia
```

Para saber la longitud de `distancias` hemos decidido elaborar nuestra propia función la cual nos dará un escalar con el número de elementos que haya en nuestra lista (distancias en este caso).

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

Nuestra función len se basa en un contador el cual se inicializa a 0 y, por medio de un bucle, iterar todos los elementos de la lista y aumentar en 1 cada vez que haya un nuevo elemento. Por último, se devolverá el contador. Esta función será fundamental en otras funciones tal y como veremos más adelante. Vamos a ver cuántas distancias tenemos:

```
longitud = len(distancias)
longitud

## [1] 73
```

Observamos que tenemos 73 elementos, es decir, 73 distancias. La próxima utilidad que necesitamos es la ordenación de nuestra lista de distancias. Para ello hemos elaborado una función que ordenará la lista en sentido ascendente o descendente utilizando para ello el método de la burbuja.

El método de la burbuja consiste en ir evaluando por pares todos los elementos de una lista, de tal forma que se vayan reposicionando según el sentido en el que se esté ordenando. En caso de ir ordenando en sentido ascendente, el mayor elemento de la lista se irá reposicionando hasta llegar a la derecha del todo. En caso de hacerlo en sentido descendente, el elemento que quedará a la derecha del todo será el menor. Conforme se vaya avanzando, tendremos por cada iteración un elemento más colocado a la derecha del todo, y así sucesivamente hasta que todos los elementos queden ordenados. La función que realiza este método de ordenación es la siguiente:

```
bubble = function(list, asc = TRUE){
        n = len(list)
        if(asc){
                for (i in 2:n){
                        for (j in 1:(n-1)){
                                if (list[j] > list[j+1]){
                                        temp = list[j]
                                        list[j] = list[j+1]
                                        list[j+1] = temp
                                }
                        }
                }
```

```
        }
        else {
                for (i in 2:n){
                        for (j in 1:(n-1)){
                                if (list[j] < list[j+1]){
                                        temp = list[j]
                                        list[j] = list[j+1]
                                        list[j+1] = temp
                                }
                        }
                }
        }
        list
}
```

A esta función habrá que pasarle la lista que se quiere ordenar y `TRUE` o `FALSE` para indicar en el sentido que se quiere ordenar. En caso de ser `TRUE` se ordenará ascendentemente, y en caso de `FALSE` se ordenará descendentemente. Por defecto, si solo se pasa como parámetro la lista, se ordenará de forma ascendente. Vamos a hacer la prueba con nuestra lista de distancias:

```
distancias_asc = bubble(distancias)
distancias_asc
```

```
##  [1]  1.0  2.1  2.7  3.1  3.2  3.4  3.7  3.7  4.0  4.0  4.4  4.4  4.5  4.5  5.0
## [16]  5.1  5.5  6.2  8.1  9.0  9.4  9.7 10.0 11.0 12.0 12.0 12.0 12.0 12.0 13.0
## [31] 15.0 16.0 16.5 17.2 19.0 19.0 20.0 20.7 21.0 21.6 22.0 24.0 24.0 24.0 24.0
## [46] 24.1 25.0 25.0 26.0 26.0 27.0 27.0 27.0 28.0 28.0 29.0 30.0 30.0 30.0 30.0
## [61] 30.0 30.0 30.0 30.0 31.4 32.0 33.0 33.0 34.0 34.0 34.8 38.0 46.0
```

```
distancias_desc = bubble(distancias, FALSE)
distancias_desc
```

```
##  [1] 46.0 38.0 34.8 34.0 34.0 33.0 33.0 32.0 31.4 30.0 30.0 30.0 30.0 30.0 30.0
## [16] 30.0 30.0 29.0 28.0 28.0 27.0 27.0 27.0 26.0 26.0 25.0 25.0 24.1 24.0 24.0
## [31] 24.0 24.0 22.0 21.6 21.0 20.7 20.0 19.0 19.0 17.2 16.5 16.0 15.0 13.0 12.0
## [46] 12.0 12.0 12.0 12.0 11.0 10.0  9.7  9.4  9.0  8.1  6.2  5.5  5.1  5.0  4.5
## [61]  4.5  4.4  4.4  4.0  4.0  3.7  3.7  3.4  3.2  3.1  2.7  2.1  1.0
```

Con la lista ordenada se puede saber de forma muy sencilla el rango. Previamente, en los ejercicios guiados, teníamos que llamar a las funciones `max` y `min`. Con la lista ordenada, se pueden acceder a estos valores, ya que estarán en el primer y último índice en función del sentido en el que hayamos ordenado la lista. Es por ello que nos resulta muy fácil crear una función rango.

```
rank = function(list){
        ordered_list = bubble(list)
        ordered_list[len(ordered_list)] - ordered_list[1]
}
```

Primeramente hemos ordenado la lista en orden ascendente, de tal forma que el valor máximo se encontrará en el último índice y el valor mínimo en el primero. Lo que hacemos para calcular el rango es restar el elemento del último índice menos el del primero. Comprobamos nuestra función y observamos que nos sale un rango de 45.

```
rango_dist = rank(distancias)
rango_dist

## [1] 45
```

Ahora vamos a realizar el cálculo de las frecuencias absoluta y relativa y sus respectivas acumuladas. Para ello, nuevamente crearemos una función para cada una. La primera que vamos a realizar y en la que se van a basar el resto será la frecuencia absoluta. El código de la función es el siguiente:

```
absolute_freq = function(list){
        ordered_list = bubble(list)
        n = len(ordered_list)
        elements = vector()
        frequencies = vector()
        i = 1
        while (i <= n){
                actual_element = ordered_list[i]
                elements = append(elements, actual_element)
                actual_freq = 0
                j = i
                while(j <= n & actual_element == ordered_list[j]){
                        actual_freq = actual_freq + 1
                        j = j+1
                }
                frequencies = append(frequencies, actual_freq)
                i = j
        }
        rbind(elements, frequencies)
}
```

El algoritmo consiste en iterar toda la lista. Tenemos dos listas auxiliares, una de elementos y otra de las frecuencias de esos elementos. Con la lista ordenada, cada vez que encontremos un elemento distinto del anterior, lo apuntaremos en la lista de elementos, y

mientras continuamos iterando la lista vamos apuntando en un contador auxiliar las veces que va apareciendo es elemento. Nótese que el elemento va a aparecer contiguo y no va a volver a aparecer en l resto de la lista ya que previamente la hemos ordenado. De esta forma, encontramos un elemento, contamos las veces contiguas que aparece y cuando aparece otro elemento apuntamos las frecuencias del anteior e inciamos el mismo procedimiento con el siguiente elemento. Una vez completamos la iteración de la lista devolvemos con rbind una matriz que contiene ambas listas de elementos y sus correspondientes frecuencias.

Para esta función hemos utilizado las funciones de append (paquete base) que devuelve un vector con la lista que se pasa como parámetro y el elemento que se introduce como parámetro concatenado a la derecha del mismo, y la función rbind (paquete base), que devuelva una matriz con los vectores que se pasan como parámetros a modo de fila. Comprobamos las frecuencias absolutas:

```
frecuencia_abs = absolute_freq(distancias)
frecuencia_abs
```

```
##             [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
## elements       1  2.1  2.7  3.1  3.2  3.4  3.7    4  4.4   4.5     5   5.1
## frequencies    1  1.0  1.0  1.0  1.0  1.0  2.0    2  2.0   2.0     1   1.0
##             [,13] [,14] [,15] [,16] [,17] [,18] [,19] [,20] [,21] [,22] [,23]
## elements      5.5   6.2   8.1     9   9.4   9.7    10    11    12    13    15
## frequencies   1.0   1.0   1.0     1   1.0   1.0     1     1     5     1     1
##             [,24] [,25] [,26] [,27] [,28] [,29] [,30] [,31] [,32] [,33] [,34]
## elements       16  16.5  17.2    19    20  20.7    21  21.6    22    24  24.1
## frequencies     1   1.0   1.0     2     1   1.0     1   1.0     1     4   1.0
##             [,35] [,36] [,37] [,38] [,39] [,40] [,41] [,42] [,43] [,44] [,45]
## elements       25    26    27    28    29    30  31.4    32    33    34  34.8
## frequencies     2     2     3     2     1     8   1.0     1     2     2   1.0
##             [,46] [,47]
## elements       38    46
## frequencies     1     1
```

Una vez hemos hecho las frecuencias absolutas pasamos a hacer las frecuencias relativas. La frecuencia relativa no es más que la frecuencia absoluta de un elemento dividido entre el total de elementos. Para realizar este cálculo disponemos de todas las herramientas, ya que tenemos la función `len` que nos dice cuántos elementos hay y la función de frecuencias absolutas que hemos visto ahora mismo. Podemos codificar otra función que devuelva una matriz (con `rbind` al igual que hemos visto en el ejercicio anterior) con los elementos distintos de la lista y su frecuencia absoluta dividida entre la longitud de la misma. Siguiendo esta idea, el código de la función es el siguiente.

```
relative_freq = function(list){
        f_abs = absolute_freq(list)
        elements = f_abs[1,]
        abs_fvalues = f_abs[2,]
```

```
        rbind(elements,abs_fvalues/len(list))
}
```

Y comprobamos las frecuencias relativas de la lista.

```
frecuencia_rel = relative_freq(distancias)
frecuencia_rel
```

```
##                [,1]       [,2]       [,3]       [,4]       [,5]       [,6]
## elements 1.00000000 2.10000000 2.70000000 3.10000000 3.20000000 3.40000000
##          0.01369863 0.01369863 0.01369863 0.01369863 0.01369863 0.01369863
##                [,7]       [,8]       [,9]      [,10]      [,11]      [,12]
## elements 3.70000000 4.00000000 4.40000000 4.50000000 5.00000000 5.10000000
##          0.02739726 0.02739726 0.02739726 0.02739726 0.01369863 0.01369863
##               [,13]      [,14]      [,15]      [,16]      [,17]      [,18]
## elements 5.50000000 6.20000000 8.10000000 9.00000000 9.40000000 9.70000000
##          0.01369863 0.01369863 0.01369863 0.01369863 0.01369863 0.01369863
##                [,19]       [,20]       [,21]       [,22]       [,23]
## elements 10.00000000 11.00000000 12.00000000 13.00000000 15.00000000
##           0.01369863  0.01369863  0.06849315  0.01369863  0.01369863
##                [,24]       [,25]       [,26]       [,27]       [,28]
## elements 16.00000000 16.50000000 17.20000000 19.00000000 20.00000000
##           0.01369863  0.01369863  0.01369863  0.02739726  0.01369863
##                [,29]       [,30]       [,31]       [,32]       [,33]
## elements 20.70000000 21.00000000 21.60000000 22.00000000 24.00000000
##           0.01369863  0.01369863  0.01369863  0.01369863  0.05479452
##                [,34]       [,35]       [,36]       [,37]       [,38]
## elements 24.10000000 25.00000000 26.00000000 27.00000000 28.00000000
##           0.01369863  0.02739726  0.02739726  0.04109589  0.02739726
##                [,39]     [,40]       [,41]       [,42]       [,43]       [,44]
## elements 29.00000000 30.000000 31.40000000 32.00000000 33.00000000 34.00000000
##           0.01369863  0.109589  0.01369863  0.01369863  0.02739726  0.02739726
##                [,45]       [,46]       [,47]
## elements 34.80000000 38.00000000 46.00000000
##           0.01369863  0.01369863  0.01369863
```

Por último, vamos a elaborar dos funciones para las frecuencias acumuladas absoluta y relativa. Con todas las frecuencias de cada elemento y la lista ordenada podemos iterar la lista de frecuencias e ir sumando para cada elemento la suma de las frecuencias de los elementos menores o iguales al elemento del que se está calculando. Dependiendo de qué frecuencia acumulada se esté calculando tendremos que trabajar con la función de frecuencias absolutas (`absolute_freq`) o con la de frecuencias relativas (`relative_freq`). Iremos iterando cada elemento e iremos creando un nuevo vector con la frecuencia acumulada. Los códigos de las funciones que realizan la frecuencia absoluta acuulada y la frecuencia relativa acumulada son respectivamente los siguientes:

```
acum_absolute_freq = function(list){
        f_abs = absolute_freq(list)
        elements = f_abs[1,]
        abs_fvalues = f_abs[2,]
        acum_abs_fvalues = vector()
        acum = 0
        for (i in 1:len(elements)){
                acum = acum + abs_fvalues[i]
                acum_abs_fvalues = append(acum_abs_fvalues, acum)
        }
        rbind(elements, acum_abs_fvalues)
}
```

```
acum_relative_freq = function(list){
        f_rel = relative_freq(list)
        elements = f_rel[1,]
        rel_fvalues = f_rel[2,]
        acum_rel_fvalues = vector()
        acum = 0
        for (i in 1:len(elements)){
                acum = acum + rel_fvalues[i]
                acum_rel_fvalues = append(acum_rel_fvalues, acum)
        }
        rbind(elements, acum_rel_fvalues)
}
```

Como se puede observar es el mismo código solo que en uno trabajamos con las frecuencias absolutas y en el otro con las relativas. Comprobamos su funcionamiento:

```
frecuencia_abs_acum = acum_absolute_freq(distancias)
frecuencia_abs_acum
```

```
##                 [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
## elements           1  2.1  2.7  3.1  3.2  3.4  3.7    4  4.4   4.5     5   5.1
## acum_abs_fvalues   1  2.0  3.0  4.0  5.0  6.0  8.0   10 12.0  14.0    15  16.0
##                 [,13] [,14] [,15] [,16] [,17] [,18] [,19] [,20] [,21] [,22]
## elements          5.5   6.2   8.1     9   9.4   9.7    10    11    12    13
## acum_abs_fvalues 17.0  18.0  19.0    20  21.0  22.0    23    24    29    30
##                 [,23] [,24] [,25] [,26] [,27] [,28] [,29] [,30] [,31] [,32]
## elements           15    16  16.5  17.2    19    20  20.7    21  21.6    22
## acum_abs_fvalues   31    32  33.0  34.0    36    37  38.0    39  40.0    41
##                 [,33] [,34] [,35] [,36] [,37] [,38] [,39] [,40] [,41] [,42]
## elements           24  24.1    25    26    27    28    29    30  31.4    32
## acum_abs_fvalues   45  46.0    48    50    53    55    56    64  65.0    66
```

```
##                   [,43] [,44] [,45] [,46] [,47]
## elements             33    34  34.8    38    46
## acum_abs_fvalues     68    70  71.0    72    73

frecuencia_rel_acum = acum_relative_freq(distancias)
frecuencia_rel_acum

##                         [,1]       [,2]       [,3]       [,4]       [,5]
## elements          1.00000000 2.10000000 2.70000000 3.10000000 3.20000000
## acum_rel_fvalues 0.01369863 0.02739726 0.04109589 0.05479452 0.06849315
##                         [,6]     [,7]      [,8]      [,9]     [,10]     [,11]
## elements          3.40000000 3.700000 4.0000000 4.4000000 4.5000000 5.0000000
## acum_rel_fvalues 0.08219178 0.109589 0.1369863 0.1643836 0.1917808 0.2054795
##                       [,12]     [,13]     [,14]    [,15]     [,16]     [,17]
## elements          5.1000000 5.5000000 6.2000000 8.100000 9.0000000 9.4000000
## acum_rel_fvalues 0.2191781 0.2328767 0.2465753 0.260274 0.2739726 0.2876712
##                       [,18]      [,19]      [,20]      [,21]      [,22]
## elements          9.7000000 10.0000000 11.0000000 12.0000000 13.0000000
## acum_rel_fvalues 0.3013699  0.3150685  0.3287671  0.3972603  0.4109589
##                       [,23]      [,24]      [,25]      [,26]      [,27]
## elements         15.0000000 16.0000000 16.5000000 17.2000000 19.0000000
## acum_rel_fvalues  0.4246575  0.4383562  0.4520548  0.4657534  0.4931507
##                       [,28]      [,29]      [,30]      [,31]      [,32]
## elements         20.0000000 20.7000000 21.0000000 21.6000000 22.0000000
## acum_rel_fvalues  0.5068493  0.5205479  0.5342466  0.5479452  0.5616438
##                       [,33]     [,34]      [,35]      [,36]      [,37]
## elements         24.0000000 24.100000 25.0000000 26.0000000 27.0000000
## acum_rel_fvalues  0.6164384  0.630137  0.6575342  0.6849315  0.7260274
##                       [,38]      [,39]      [,40]     [,41]      [,42]
## elements         28.0000000 29.0000000 30.0000000 31.400000 32.0000000
## acum_rel_fvalues  0.7534247  0.7671233  0.8767123  0.890411  0.9041096
##                       [,43]      [,44]      [,45]      [,46] [,47]
## elements         33.0000000 34.0000000 34.8000000 38.0000000    46
## acum_rel_fvalues  0.9315068  0.9589041  0.9726027  0.9863014     1
```

Se puede comprobar que están bien ya que la frecuencia absoluta acumulada del último elemento es igual al número total de elementos (73) y la frecuencia relativa acumulada del último elemento es 1.

Prosiguiendo con el análisis de datos, tenemos que obtener la moda del conjunto de datos. Para ello hemos elaborado una función que nos calculará este valor. Ayudándonos de las frecuencias absolutas, podemos iterar estas y buscar la mayor frecuencia, cuyo elemento asociado será la moda. Así vamos iterando la frecuencia de cada elemento diferente de la lista y vamos comprobando si su frecuencia es la mayor hasta ahora. En caso de serlo, la moda temporal correspondera a ese elemento. Así observaremos todos los elementos, devolviendo después de terminar la iteración de la lista el elemento con mayor frecuencia, la moda. El código que incluye esta funcionalidad es el siguiente:

```
mode = function(list){
        frequencies = absolute_freq(list)
        elements = frequencies[1,]
        freq_values = frequencies[2,]
        actual_mode = 0
        actual_mode_val = 0
        for (i in 1:len(elements)){
                if (freq_values[i] > actual_mode_val){
                        actual_mode_val = freq_values[i]
                        actual_mode = elements[i]
                }
        }
        actual_mode
}
```

Y extraemos la moda de nuestro conjunto de datos:

```
moda = mode(distancias)
moda

## [1] 30
```

Observamos que la moda del conjunto de distancias es 30. Podemos comprobar que está bien si nos fijamos en las frecuencias absolutas, ya que el elemento más que tiene es el 30 con una frecuencia absoluta de 8. Cabe destacar que en caso de haber dos o más elementos con la misma mayor frecuencia del conjunto de datos el algoritmo devolverá el elemento de menor magnitud al iterar el algoritmo de menor a mayor elemento.

El siguiente análisis que tendremos que hacer es el de la mediana, que ya hemos visto que es el dato que divide en dos al conjunto total de datos. Para calcular este valor tendremos que fijarnos en la paridad del número total de datos. Y es que si este es par o impar se calculará de una forma u otra. Para saber si un número es par o impar bastará con ver su módulo 2 (valor del resto del número dividido entre 2). Si este es 0 estamos ante un número par, en caso contrario ante uno impar.

En caso de tener un número de elementos par tendremos que coger con la lista ordenada el elemento $\frac{n}{2}$ y el elemento $\frac{n}{2}+1$, sumarlos y dividir entre dos. En caso de tener un número de elementos impar directamente tendremos que coger el elemento $\frac{n+1}{2}$.

Hemos englobado estos cálculos en una función, donde comprobamos si estamos ante un número de elementos par o impar y accedemos a los elementos correspondientes. La función es la siguiente:

```
median_value = function(list){
        n = len(list)
```

```
        ordered_list = bubble(list)
        if (n%%2 == 0){
                median = (ordered_list[n/2] + ordered_list[(n/2)+1]) / 2
        }
        else{
                median = ordered_list[(n+1)/2]
        }
        median
}
```

Comprobamos el valor de la mediana de nuestro conjunto de datos:

```
mediana = median(distancias)
mediana

## [1] 20
```

Vemos que el valor medio es 20. Estamos en lo correcto, ya que si nos fijamos en el conjunto ordenado de los datos vemos que tenemos 73 elementos (número impar), luego la mediana será el elemento $\frac{73+1}{2}$; es decir, el elemento 37 que vemos que es el 20, el valor que nos ha salido como mediana.

Tras haber codificado las medidas de tendencia central, vamos a estudiar dos medidas de dispersión. La primera de las medidas que vamos a calcular es la desviación típica. Para ello hemos codificado la siguiente función:

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

Como se puede observar, la función consiste en ir sumando las diferencias entre cada dato y la media de todos ellos elevadas al cuadrado. El `for` de la función es el encargado de realizar esta tarea. Tras ello, se divide el resultado entre el número total de datos y se realiza la raíz cuadrada de la medida obtenida. La desviación típica, por tanto, obedece a la fórmula:

$$\sigma = \sqrt{\frac{\sum_{i=0}^{n}(x_i - \bar{x})^2}{n}}$$

La desviación típica de nuestro conjunto de datos será por tanto la siguiente:

```
desviacion = standard_dev(distancias)
desviacion

## [1] 11.23204
```

La segunda y última medida de dispersión que vamos a estudiar es la varianza, que en el fondo, no es más que elevar la desviación típica al cuadrado.

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

Con esta función, calculamos la varianza en el conjunto de datos:

```
varianza = variance(distancias)
varianza

## [1] 126.1587
```

Terminadas las medidas de dispersión, vamos a proceder a calcular los cuantiles. Para calcular los cuantiles tendremos que definir además de la lista de la que queremos calcular el valor de c (que es ...). Esto último tendrá que corresponder a un valor entre 0 y 1 en función de c. Así, si introducimos por parámetro un c que no es válido se nos devolverá `NULL`, ya que no se puede hacer un cálculo. En caso de haber introducido un c válido tendremos que observar dos casos.

- Si nc (siendo n el número total de elementos) es natural tendremos que coger los elementos nc y $nc + 1$, los sumaremos y dividiremos entre 2.
- Si nc (siendo n el número total de elementos) no es natural, se deberá coger el elemento $\lfloor nc \rfloor + 1$, siendo $\lfloor nc \rfloor$ la parte entera del producto.

Con estos cálculos podemos codificar una función que nos da los cuantiles. La función es la siguiente:

```
quant = function(list, c){
        ordered_list = bubble(list)
        n = len(list)
        if (c < 0 | c > 1){
                quant = NULL
        }
```

```
        else{
                if((n*c)%%1 == 0){
                        quant = (ordered_list[(n*c)] +
                         ordered_list[(n*c) + 1]) / 2
                }
                else {
                        int_prod = floor(n*c)
                        quant = ordered_list[int_prod + 1]
                }
        }
        quant
}
```

Para poder comprobar si $nc \in \mathbb{N}$, calculamos nc (mód 1). Cualquier número entero dividido entre 1 nos va a dar de resto 0; en caso contrario dará otra cosa que no sea 0. Además, aseguramos que el producto, en caso de ser entero, nos va a dar un número natural ya que siempre va a ser positivo. Esto es por dos razones fundamentales:

- Siempre se cumplirá que $n \geq 0$.
- Siempre se cumplirá que $0 \leq c \leq 1$.

En caso de estar ante un número que no es natural tendremos que acceder a la parte entera del producto. Para poder acceder a ella hemos hecho uso de la función `floor` perteneciente al paquete `base`. Esta función recoge como parámetro un número y devuelve su aproximación en número entero redondeado hacia abajo. Por ejemplo, si aplicamos esta función a 4,99 nos devolverá 4, pues $\lfloor 4{,}99 \rfloor = 4$.

Con esta función podremos calcular cuantiles, entre los que se incluyen cuartiles, deciles, percentiles... Para probar la función vamos a calcular los tres cuartiles $\frac{1}{4}$, $\frac{2}{4}$ y $\frac{3}{4}$.

```
cuartil1 = quant(distancias,0.25)
cuartil2 = quant(distancias,0.5)
cuartil3 = quant(distancias,0.75)
cuartilerr = quant(distancias, -0.2)

cuartil1
## [1] 8.1
cuartil2
## [1] 20
cuartil3
## [1] 28
cuartilerr
## NULL
```

Observamos que los valores de los cuartiles 1, 2 y 3 son 8.1, 20 y 28. Además hemos probado a meter un valor de c no válido y verificamos que efectivamente nos devuelve `NULL` ya que ese cálculo no sería lógicamente correcto. También comprobamos que el valor del segundo cuartil es igual al de la mediana que hemos calculado más arriba. Ambos valores deben coincidir ya que hacen referencia al mismo elemento que realiza la misma división de los datos.

2.2. Asociación

"*El segundo conjunto de datos, que se empleará para realizar el análisis de asociación, estará formado por las siguientes conjuntos de extras incluidos en 8 ventas de coches:* {*X, C, N, B*}, {*X, T, B, C*}, {*N, C, X*}, {*N, T, X, B*}, {*X, C, B*}, {*N*}, {*X, B, C*}, {*T, A*}. *Donde:* {*X: Faros de Xenon, A: Alarma, T: Techo Solar, N: Navegador, B: Bluetooth, C: Control de Velocidad*}*, son los extras que se pueden incluir en cada coche.*"

```
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]))
        }
}

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

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

```
count_appearance = function(table, elements){
        count = 0
        for (i in 1:len(table[,1])){
                acum = 1
                for (element in elements){
                        acum = (table[i,element]) & acum
                }
                count = count + acum
        }
        count
}
```

```
support = function(table, elements) {
        count_appearance(table, elements) / len(table[,1])
}

support_clasif = function(table, ocurrences, s){
        valid_ocurrences = c()
        for (ocurrence in ocurrences){
                support_oc = support(table, ocurrence)
                if (support_oc >= s){
                        valid_ocurrences = append(valid_ocurrences, ocurrence)
                }
        }
        valid_ocurrences
}
```

```
create_comb = function(table, clasif, s) {
        combinations = c()
        dim = 2
        next_dim = TRUE
        while (dim <= len(clasif) & next_dim == TRUE) {
                next_dim = FALSE
                comb = unlist(lapply(dim, function(m) {
                        combn(clasif, m=m, simplify=TRUE)
                        }), recursive=FALSE)

                for (j in seq(1, len(comb), by=dim)) {
                        add = c()
                        for (k in j:(j+dim-1)) {
                                add = append(add, comb[k])
                        }
```

```
                        if (support(table, add) >= s) {
                                next_dim = TRUE
                                combinations = append(combinations, list(add))
                        }
                }
                dim = dim+1
        }
        combinations
}
```

```
confidence = function(table, left, right) {
        count_appearance(table, union(left, right)) /
                count_appearance(table, left)
}

get_asotiations = function(table, comb, c) {
        kMax = len(comb[len(comb)][[1]])
        listLeft = list()
        listRight = list()

        for (i in 2:kMax) {

                split = Filter(function(x) length(x)==i, comb)

                for (j in 1:len(split)) {

                        for (k in 1:(i-1)) {

                                leftSides = lapply(k, function(m) {
                                        combn(split[j][[1]], m=m, simplify=TRUE)})

                                df = do.call(rbind.data.frame, leftSides)

                                for (n in 1:len(df[1,])) {

                                        all = split[j][[1]]

                                        left = df[,n]

                                        right = dif(split[j][[1]], df[,n])

                                        if (confidence(table, left, right) >= c) {
```

```
                                                listLeft = append(listLeft , list(left))
                                                listRight = append(listRight , list(righ

                                        }
                                }
                        }
                }
        }
        asoc = data.frame(left = I(listLeft), right = I(listRight))
        asoc
}
```

```
apriori = function(table, elements, s, c) {
        soporte_clasif = support_clasif(table, elements, s)

        combinations = create_comb(table, soporte_clasif, s)

        if (len(combinations) > 0) {

                conf = get_asotiations(table, combinations, c)
                print(conf)

        } else {
                print("No hay ninguna combinación que pase el soporte")
        }
}
```

```
elements = c(c("X"), c("A") ,c("T"), c("N"), c("B"), c("C"))

tabla = matrix(c(
        1,0,0,1,1,1,
        1,0,1,0,1,1,
        1,0,0,1,0,1,
        1,0,1,1,1,0,
        1,0,0,0,1,1,
        0,0,0,1,0,0,
        1,0,0,0,1,1,
        0,1,1,0,0,0),8,6,byrow=TRUE,dimnames=list(
c("occu1","occu2","occu3","occu4","occu5","occu6","occu7","occu8"),
c("X", "A", "T", "N", "B", "C")))

apriori(tabla, elements, 0.5, 0.8)
```

```
##    left right
## 1     X     B
## 2     B     X
## 3     X     C
## 4     C     X
## 5     B     C
## 6     C     B
## 7     B  X, C
## 8     C  X, B
## 9  X, B     C
## 10 X, C     B
## 11 B, C     X
```

2.3. Detección de datos anómalos

2.3.1. Primer ejercicio

"*El tercer conjunto de datos, que se empleará para realizar el análisis de detección de datos anómalos utilizando técnicas con base estadística, estará formado por los siguientes 10 valores de velocidades de respuesta y temperaturas normalizadas de un microprocesador* {*Velocidad, Temperatura*}*:* {*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*}*. Aplicar las medidas de ordenación a la velocidad y las de dispersión a la temperatura.*"

2.3.2. Segundo ejercicio

"*El cuarto conjunto de datos, que se empleará para realizar el análisis de detección de datos anómalos utilizando técnicas basadas en la proximidad y en la densidad, estará formado por el número de Mujeres y Hombres inscritos en una serie de cinco seminarios que se han impartido sobre biología. Los datos son:* {*Mujeres, Hombres*}*: 1.* {*9, 9*}*; 2.* {*9, 7*}*; 3.* {*11, 11*}*; 4.* {*2, 1*}*; 5.* {*11, 9*}*.*"