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.

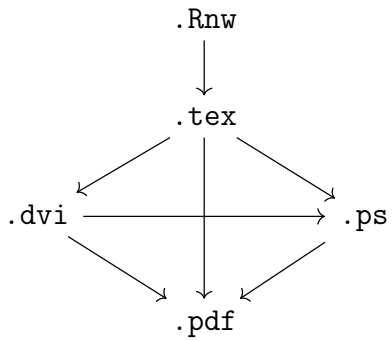

Figura 2: Esquema de extensiones en LaTeX

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.

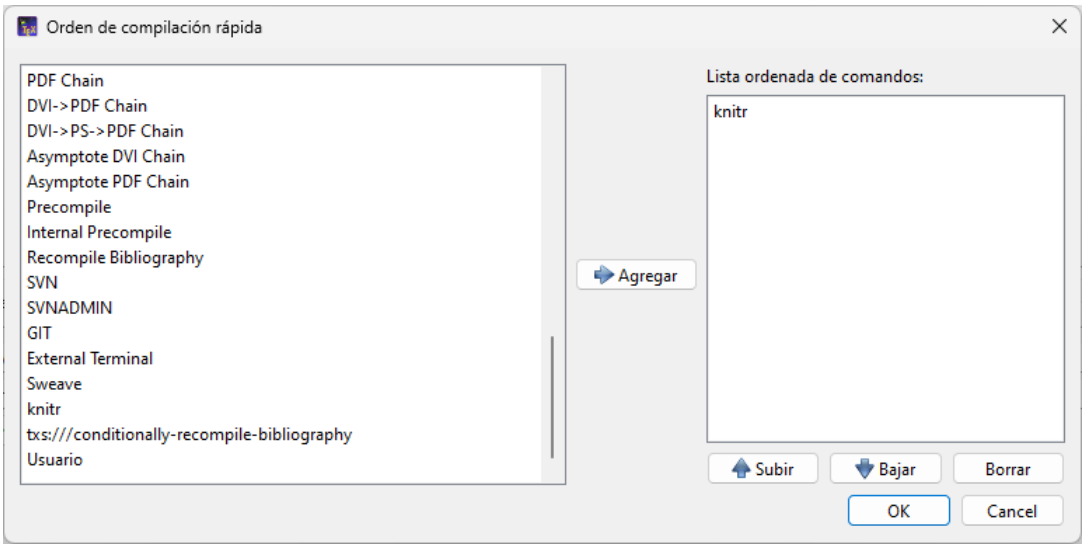

Figura 4: Modificación de compilación

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

Ejercicio 1.1.1. *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

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

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":

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

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

Ejercicio 1.2.1. *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. Técnicas estadísticas

Ejercicio 1.3.1. *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 pasó 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 `$out`, se obtiene que el suceso outlier es el 14, que como se vio en el ejercicio de la clase de teoría es correcto, pero, si nos fijamos en `$conf`, se pueden ver los límites del intervalo, pero estos no coinciden con los vistos en clase. Esto se debe a que al igual que ocurría al usar la función `sd` para el cálculo de la desviación, R utiliza un factor de $n-1$ en vez de n, por lo que no son los valores exactos.

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

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

```
(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, se iteran los elementos de la muestra y se comprueba 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 se vio 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
```

Se puede ver que los valores inferior y superior del intervalo no son iguales que en el ejercicio que realizado en clase, esto se debe a que, como ya se vio, la desviación típica se calcula con un factor de $n - 1$ en vez de n, 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, se obtienen los valores del intervalo correctos y se puede ver que el suceso anómalo es el que se encuentra en la posición 2, es decir el elemento 12.

1.3.2. Técnicas de proximidad y densidad

Ejercicio 1.3.2. *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).*

Algoritmo $k-$vecinos

El algoritmo $k-$vecinos utilizado 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 se añadieron todos los datos de la nueva muestra a una matriz, que luego fue transpuesta para tener los datos dispuestos como requería el cálculo de distancias. Se calcularon las distancias euclídeas mediante la función `dist` y se guardó 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.

$$\text{dist}(p, q) = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2}$$

```
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, se itera cada punto y ordena 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
```

Por último, se itera 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 se obtiene que el punto 4 es un suceso anómalo o outlier, al igual que ocurría en el ejercicio resuelto en teoría.

Algoritmo LOF

Como último ejercicio realizado en clase, se ha empleado el algoritmo LOF (Local Outlier Factor) 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.

Se obtiene una matriz de distancias haciendo uso de la función `dist`, pero en este caso se pasa como parámetro el método que queremos que sea usado para calcular la distancia (`method="manhattan"`), en este caso el método *Manhattan*, que se calcula de la siguiente manera.

$$\text{dist}(x_i, x_j) = |x_{i_1} - x_{j_1}| + |x_{i_2} - x_{j_2}|$$

```
(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 en el segundo ejercicio autónomo de detección de datos anómalos, se implementará un código propio, que realice las simplificaciones oportunas.

Parte 2

Ejercicios autónomos

2.1. Descripción de los datos

Ejercicio 2.1.1. *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 comenzar, se han introducido todos los datos en un fichero CSV. Para realizar este fichero se ha abierto un fichero Excel y se han ido introduciendo los datos en la primera columna. Cabe destacar que la primera fila no corresponde a ningún valor ya que se ha puesto `Distancia(km)` para mantener la estructura con respecto al ejercicio guiado. Una vez han sido introducidos todos los datos, guardamos el fichero con extensión `.csv`. Para leer este fichero dentro de R se hace 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 dataframe haciendo corresponder las filas y las columnas de la tabla. La función empleada ha sido `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 se debería haber usado `read.csv2`. Se observa en la siguiente línea de código cómo se ha creado un archivo `distancia_universitarios.csv` siguiendo la estructura mencionada 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 dicho 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 se puede observar, se ha leído correctamente el fichero, creando el dataframe con los datos 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 se separa por comas, se leerán dos datos en vez de uno.

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

```
distancias = fichero$Distancia
```

Para conocer la longitud de `distancias`, se ha decidido elaborar una función propia, la cual nos devolverá un escalar con el número de elementos que contenga nuestra lista (distancias en este caso).

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

La 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. Veamos cuántas distancias se tienen:

```
(longitud = len(distancias))

## [1] 73
```

Observamos se tienen 73 elementos, es decir, 73 distancias. La próxima utilidad que se necesita es la ordenación de la lista de distancias. Para ello se ha elaborado una función que ordenará la lista en sentido ascendente o descendente utilizando 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. Esto se consigue en un tiempo $\mathcal{O}(n^2)$. La función que realiza este método de ordenación es la siguiente:

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

A esta función se le deberá pasar 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 de manera ascendente, y en caso de `FALSE` se ordenará de forma descendente. Por defecto, si solo se pasa como parámetro la lista, se ordenará de forma ascendente.

Además, el algoritmo de ordenación de la burbuja hace uso de la variable `direccion`, la cual adquiere los valores de 1 (en sentido ascendente) y -1 (en sentido descendente). Esto se consigue con la función `ifelse` del paquete `base`, a la cual se le pasa un valor booleano y dos valores. En función del valor booleano, si este es `TRUE` se coge el primer valor y si es `FALSE` el segundo. El valor escogido es asignado a la variable`direccion`. Esta cambiará o no el signo de los elementos de la lista que se está ordenando. En los números positivos, cuanto mayor sea el valor mayor será este, y en los negativos al revés. Si se ordena una lista de números negativos quedará a la derecha del todo el menor valor al ser este el número mayor. Esto se puede generalizar al sentido de ordenación de la lista.

Hagamos la prueba con la 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, se tenía 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 resulta muy fácil crear una función rango.

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

Primeramente, se ha 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. Para calcular el rango se resta al elemento del último índice, el del primero. Se comprueba la función y se observa que se obtiene un rango de 45.

```
rango_dist = rank(distancias)
rango_dist

## [1] 45
```

Ahora se realizará el cálculo de las frecuencias absoluta y relativa y sus respectivas acumuladas. Para ello, nuevamente se crea una función para cada una. La primera que realizaremos y en la que se basará 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. Se tienen dos listas auxiliares, una de elementos y otra de las frecuencias de esos elementos. Con la lista ordenada, cada vez que se encuentra un elemento distinto del anterior, lo se apunta en la lista de elementos, y mientras se continua iterando la lista, se van apuntando en un contador auxiliar las veces que va apareciendo ese elemento. Nótese que el elemento va a aparecer contiguo y no va a volver a aparecer en el resto de la lista, ya que previamente ha sido ordenada. De esta forma, se encuentra un elemento, se cuentan las veces contiguas que aparece y cuando aparece otro elemento, se apuntan las frecuencias del anterior y se inicia el mismo procedimiento con el siguiente elemento. Una vez se completa la iteración de la lista, se devuelve con `rbind` una

matriz que contiene ambas listas de elementos y sus correspondientes frecuencias.

Para esta función se han 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 realizadas las frecuencias absolutas pasaremos a realizar 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 se disponen de todas las herramientas, ya que se tiene la función `len`, que dice cuántos elementos hay, y la función de frecuencias absolutas que acabamos de realizar. Se puede codificar otra función que devuelva una matriz (con `rbind` al igual que se ha 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))
}
```

Se comprueban 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, elaboraremos dos funciones para las frecuencias acumuladas absoluta y relativa. Con todas las frecuencias de cada elemento y la lista ordenada se puede 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 se tendrá que trabajar con la función de frecuencias absolutas (`absolute_freq`) o con la de frecuencias relativas (`relative_freq`). Se irá iterando cada elemento y se irá 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 se trabajan 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 son correctos, 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, se tiene que obtener la moda del conjunto de datos. Para ello se ha elaborado una función que nos calculará este valor. Ayudándonos de las frecuencias absolutas, se pueden iterar y buscar la mayor frecuencia, cuyo elemento asociado será la moda. Se va iterando la frecuencia de cada elemento diferente de la lista y se comprueba si su frecuencia es la mayor hasta el momento. En caso de serlo, la moda temporal corresponderá 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
}
```

Se extrae la moda del conjunto de datos proporcionado:

```
moda = mode(distancias)
moda

## [1] 30
```

Se observa que la moda del conjunto de distancias es 30. Se puede comprobar que es correcto si observamos 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 de menor a mayor elemento.

El siguiente análisis que se hará es el de la mediana, que se ha visto que es el dato que divide en dos al conjunto total de datos. Para calcular este valor se tendrá que observar la paridad del número total de datos. Para saber si n es par, basta con calcular $n \pmod 2$. Si $n \equiv 0 \pmod 2$ será par, y si $n \pmod 2 \equiv 1$, será impar.

En caso de tener un número de elementos par, se cogerá de 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 $\frac{n+1}{2}$.

Se han englobado estos cálculos en una función, donde se comprueba si se está ante un número de elementos par o impar, y se accede 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
}
```

Comprobación del valor de la mediana del conjunto de datos:

```
mediana = median(distancias)
mediana

## [1] 20
```

Se observa que el valor medio es 20. Esto es correcto, ya que si se observa el conjunto ordenado, se ve que existen 73 datos, luego $\tilde{x}$ será el elemento $x_{\frac{73+1}{2}} = x_{37} = 20$, el valor obtenido como mediana.

Tras haber codificado las medidas de tendencia central, se estudiarán dos medidas de dispersión. La primera de las medidas que se van a calcular es la desviación típica. Para ello se ha 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, $(x_i - \bar{x})$. El bucle `for` de la función es el encargado de realizar esta tarea. Tras ello, se divide el resultado entre el número total de datos n, y se realiza la raíz cuadrada de la medida obtenida. La desviación típica, por tanto, obedece a la fórmula:

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

La desviación típica del conjunto de datos será la siguiente:

```
desviacion = standard_dev(distancias)
desviacion

## [1] 11.23204
```

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

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

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

Con esta función, se calcula la varianza del conjunto de datos:

```
varianza = variance(distancias)
varianza

## [1] 126.1587
```

Terminadas las medidas de dispersión, se procede a calcular los cuantiles. Para calcular los cuantiles se tendrá que definir, además de la lista de la que se quiere 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 se introduce por parámetro un c que no es válido se devolverá `NULL`, ya que no se puede hacer el cálculo. En caso de haber introducido un c válido se observan dos casos.

- Si $nc \in \mathbb{N}$ (siendo n el número total de elementos), se calculará $\frac{x_{nc}+x_{nc+1}}{2}$.
- Si $nc \notin \mathbb{N}$ (siendo n el número total de elementos), se deberá coger el elemento $\lfloor nc \rfloor + 1$ o $\lceil nc \rceil$.

Con estos cálculos se codifica una función que calcula 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 $nc \in \mathbb{N}$, se calcula nc (mód 1), pues $\forall a \in \mathbb{Z}, a \equiv 0$ (mód 1). Además, se garantiza que $nc \in \mathbb{Z} \rightarrow nc \in \mathbb{N}$. Esto es por dos razones fundamentales:

- Siempre se cumplirá que $n > 0$.
- Siempre se cumplirá que $0 < c < 1$.

En caso de que $nc \notin \mathbb{N}$ se trabajará con $\lfloor nc \rfloor$. Para calcularla se ha 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, devolverá 4, pues $\lfloor 4{,}99 \rfloor = 4$.

Con esta función se podrán calcular cuantiles, entre los que se incluyen cuartiles, deciles, percentiles, etc. Para probar la función se calculan 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
```

Se observa que los valores de $\tilde{x}_{\frac{1}{4}}, \tilde{x}_{\frac{2}{4}}, \tilde{x}_{\frac{3}{4}}$ son 8,1, 20 y 28. Además, se ha probado a meter un valor de c no válido y verificamos que efectivamente se devuelve `NULL`, ya que ese cálculo

no sería lógicamente correcto. También se comprueba que $\tilde{x}_{\frac{2}{4}} = \tilde{x}$ que se ha calculado previamente. Ambos valores deben coincidir ya que hacen referencia al mismo elemento que realiza la misma división de los datos.

2.2. Asociación

Ejercicio 2.2.1. *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.*

Para realizar el algoritmo *apriori* es necesario programar previamente algunas funciones auxiliares que ayudarán a trabajar con conjuntos. Las funciones necesarias en este caso son la unión y la diferencia de conjuntos. La unión de dos conjuntos $A \cup B$ resulta en un nuevo conjunto que contiene todos los elementos de A y todos los elementos de B que no están en A. Por otro lado, la diferencia de dos conjuntos $A \backslash B$ da como resultado un nuevo conjunto que contiene todos los elementos de A que no formen parte de B. Como función auxiliar también ser hará uso de la función `len`, explicada en la Sección 2.1.

```
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 = append(res, element)
                }
        }
        res
}
```

Tras codificar las funciones auxiliares sobre conjuntos, se va a proceder a programar el algoritmo. Para ello, lo primero será trabajar la entrada de datos. Para ello se utilizan las funciones `get_elements` y `get_table`.

```
get_elements = function(data) {
        elements = c()
        for (i in 1:len(data)) {
                elements = union(elements, data[[i]])
        }
        elements
}

get_table = function(data, elements) {
        nCol = len(elements)
        nRow = len(data)
        table = data.frame(matrix(0, ncol = nCol, nrow = nRow,
                dimnames = list(1:nRow, elements)))

        for (i in 1:nRow) {
                for (j in 1:len(data[[i]])) {
                        table[i, data[[i]][j]] = 1
                }
        }
        table
}
```

La primera de ellas, recibe una serie de listas que representa la muestra, por ejemplo $[\{\alpha_1, \alpha_2, \ldots, \alpha_n\}, \{\alpha_{n+1}, \alpha_{n+2}, \ldots, \alpha_{n+m}\}]$, de forma que se calcula $\bigcup_{i=1}^{n+m} \alpha_i$ para calcular el espacio muestral. Supongamos que tenemos {Pan, Agua, Leche}, {Pan, Agua}, y {Pan, Leche}. Calculando la unión se obtiene $E =$ {Pan, Agua, Leche}.

La segunda de ellas crea una matriz (codificado por un dataframe) que representa por filas los sucesos de la muestra, y por columnas los sucesos elementales, encontrándose en la entrada a_{ij} un valor binario representando si dicho suceso elemental se encuentra en el suceso de la muestra. Retomando el ejemplo anterior se tendría la siguiente matriz.

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

```
m = list(c("Pan", "Agua", "Leche"), c("Pan", "Agua"), c("Pan", "Leche"))
eltos = get_elements(m)
(get_table(m, eltos))

##   Pan Leche Agua
## 1   1     1    1
## 2   1     0    1
## 3   1     1    0
```

A continuación se va a realizar una función que cuente las apariciones de una serie de elementos en los sucesos. Se sumará uno al contador cuando todos los elementos aparezcan en el mismo suceso. La función es la siguiente:

```
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
}
```

El objetivo de esta función, es dado un conjunto $\{\alpha_1, \alpha_2, \ldots, \alpha_n\}$, contar cuántas veces es subconjunto de sucesos de la muestra. Para ello, siendo β_{i_m} el elemento $m-$ésimo del suceso de la muestra $i-$ésimo, se realiza la siguiente operación:

$$\sum_{i=1}^{\texttt{nrows}} \bigwedge_{m=1}^{\texttt{ncol}} \beta_{i_m}$$

La siguiente herramienta que se necesita en el algoritmo apriori es el soporte, que queda definido de la siguiente manera.

$$\forall \{A_i\}_{i=1}^{\infty} \subset P(E) \,\text{con}\, A_i \cap A_j \neq \varnothing \,\forall i \neq j,$$
$$s : P(E) \longrightarrow \mathbb{R}^+ \,\text{tal que}\, s(A_i \cup A_j) = \frac{n_{A_i \cup A_j}}{n_T}$$

Como podemos observar, la función `count_appereance` que se había codificado previamente, será de ayuda para calcular el soporte, de forma que ahora basta dividir el resultado de esta por el número de sucesos de la muestra tal y como se hace en la función `support`.

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

Siendo el primer paso del algoritmo el descarte de los sucesos elementales que no superan el umbral de soporte, esto se logra con la función `support_clasif` iterando cada suceso elemental y calculando su soporte mediante las funciones ya explicadas.

```
support_clasif = function(table, ocurrences, s) {
        valid_ocurrences = list()
        for (ocurrence in ocurrences) {
```

```
                support_oc = support(table, ocurrence)
                if (support_oc >= s) {
                        valid_ocurrences = append(
                                valid_ocurrences, list(ocurrence))
                }
        }
        valid_ocurrences
}
```

Una vez calculados aquellos sucesos elementales que superan el umbral del soporte, deberemos aplicar el método $F_{k-1} \times F_{k-1}$ de `apriori-gen` para construir los conjuntos candidatos de dimensiones superiores. En cada iteración del algoritmo, este toma dos sucesos de dimensión $k-1$, $\alpha = \{\alpha_1, \alpha_2, \ldots, \alpha_{k-1}\}$ y $\beta = \{\beta_1, \beta_2, \ldots, \beta_{k-1}\}$ para formar un tercer conjunto de dimensión k de la forma $\{\alpha_1, \alpha_2, \ldots, \alpha_{k-1}, \beta_{k-1}\}$. Este sólo se podrá formar si se cumplen las siguientes condiciones:

- $\forall i \leq k-2, \;\; \alpha_i = \beta_i$
- $\alpha_{k-1} \neq \beta_{k-1}$

Para implementar este algoritmo se necesitarán algunas funciones auxiliares que se presentan a continuación. La primera de ellas será `equals`, que será la encargada de verificar si para dos listas L_1 y L_2, se cumple que $\forall i \, L_{1_i} = L_{2_i}$.

```
equals = function(l1, l2) {
        n = len(l1)

        if (n != len(l2)) return(FALSE)
        if (n==0) return(TRUE)

        for (i in 1:n) if (l1[i] != l2[i]) return(FALSE)
        TRUE
}
```

Las dos siguientes son `front` y `back`. Dada una lista y un valor natural, `front` devuelve otra lista que es una partición de la original hasta el índice del valor indicado. De la misma forma, `back` devuelve la lista que contiene los últimos valores indicados. Ambas funciones hacen uso de las funciones `head` y `tail` (del paquete `utils`) respectivamente, pero devolviendo `NULL` si el numero de elementos a seleccionar es menor o igual a 0.

```
front = function(l, n) {
        if (n <= 0) return(NULL)
        head(l, n)
}
```

```
back = function(l, n) {
        if (n <= 0) return(NULL)
        tail(l, n)
}
```

Como veremos a continuación al explicar el funcionamiento del algoritmo, necesitaremos calcular el número combinatorio $\binom{n}{k}$, que representa el número de subconjuntos de tamaño k que tiene un conjunto de tamaño n. Para realizar su cálculo, se ha programado un algoritmo de programación dinámica que se ejecuta en un tiempo de $\mathcal{O}(n^2)$. En primer lugar, el número combinatorio se define como

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

por lo que fácilmente se ve que $\binom{n}{0} = 1$, y que también $\binom{n}{n} = 1$, pues de la definición se deduce que

$$\binom{n}{k} = \binom{n}{n-k}.$$

Con estas dos identidades y la que se presenta a continuación, se puede calcular cualquier número binomial como suma de otros dos, partiendo del caso en el que vale 1.

$$\begin{aligned}
\binom{n}{k} &= \frac{n(n-1)!}{k!(n-k)!} \\
&= \frac{(n-1)!k}{k!(n-k)!} + \frac{(n-1)!(n-k)}{k!(n-k)!} \\
&= \frac{(n-1)!k}{k(k-1)!(n-k)!} + \frac{(n-1)!(n-k)}{k!(n-k)(n-k-1)!} \\
&= \frac{(n-1)!}{(k-1)!(n-k)!} + \frac{(n-1)!}{k!(n-k-1)!} \\
&= \binom{n-1}{k-1} + \binom{n-1}{k}
\end{aligned}$$

En el código que se muestra, se aplica esta relación de recurrencia rellenando una matriz, de forma que se calcula sin aplicar recursividad, ya que tendría una complejidad superior, lo que empeoraría aún más el tiempo de ejecución del algoritmo apriori.

```
binom = function(n, k) {
        temp = matrix(0, nrow = n + 1, ncol = k + 1)
        for (i in 0:n) {
                for (j in 0:min(i, k)) {
                        if (j == 0 || j == i) {
                                temp[i + 1, j + 1] = 1
```

```
                        } else {
                                temp[i + 1, j + 1] =
                                        temp[i, j] + temp[i, j + 1]
                        }
                }
        }
        temp[n + 1, k + 1]
}
```

Finalmente, la función `fk_1` representa el algoritmo al completo. El algoritmo funciona alrededor de ir añadiendo subconjuntos candidatos calculados en iteraciones previas a un vector. El algoritmo recibe una lista de los sucesos elementales que han superado el umbral de soporte establecido en el paso previo del algoritmo, y con `lapply` añade al vector mencionado previamente los que serían los conjuntos candidatos de dimensión 1. El algoritmo sigue con las sucesivas dimensiones, y por ejemplo, en la dimensión k combina los sucesos candidatos de dimensión $k-1$ dos a dos, obtenidos en la iteración previa. Para cada posible pareja, se comprueban las dos reglas explicadas previamente con ayuda de las funciones `cabeza` y `cola`. Aquellas que verifican ambas reglas, se añaden al vector comentado como candidatos de dimensión k.

En cada iteración será necesario saber en qué posición del vector inicia y finalizan los sucesos candidatos de dimensión $k-1$. Suponiendo que se tienen n sucesos elementales, la posición del primer suceso de dimensión k, viene dada por sumar a la posición del primer suceso de dimensión $k-1$, el número combinatorio $\binom{n}{k}$, pues $F_{k-1} \times F_{k-1}$ irá añadiendo los posibles subconjuntos.

Finalmente, se devuelve el vector sobre el que se ha trabajado pero recortándolo de forma que no aparezcan los sucesos de dimensión 1, pues al formar asociaciones con estos, uno de los dos sucesos que la formarían sería $\varnothing$, cosa que no tiene sentido.

```
fk_1 = function(clasif) {
        comb = lapply(clasif, c)
        n = len(clasif)
        n_comb = n
        start = 1

        for (k in 2:n) {
                for (i in start:(n_comb-1)) {
                        for (j in (i+1):n_comb) {

                                hd1 = front(comb[[i]], len(comb[[i]])-1)
                                hd2 = front(comb[[j]], len(comb[[j]])-1)

                                tl1 = back(comb[[i]], 1)
```

```
                                tl2 = back(comb[[j]], 1)

                                if (equals(hd1, hd2) & !equals(tl1, tl2)) {
                                        new = c(hd1, tl1, tl2)
                                        comb = c(comb, list(new))
                                }
                        }
                }
                start = n_comb + 1
                n_comb = n_comb + binom(n, k)
        }
        back(comb, n_comb-n)
}
```

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

De la misma manera que se ha necesitado calcular el soporte, también se necesitará calcular la confianza, definida como se muestra en la siguiente ecuación, se implementa con la función previamente mostrada `confidence`.

$$\forall \{A_i\}_{i=1}^{\infty} \subset P(E) \text{ con } A_i \cap A_j \neq \varnothing \, \forall i \neq j,$$
$$c : P(E) \longrightarrow \mathbb{R}^+ \text{ tal que } c(A_i \cup A_j) = \frac{n_{A_i \cup A_j}}{n_{A_i}}$$

```
get_associations = function(table, candidates) {
        associations = data.frame()
        final_associations = lapply(candidates, function(x){
                k = len(x)
                for (dim in 1:(k - 1)){
                        left_sides = combn(x, m=dim, simplify=TRUE)
                        for (col in 1:len(left_sides[1,])){
                                left_side = left_sides[, col]
                                right_side = dif(x, left_side)
                                new_asoc = data.frame(
                                        left = I(list(left_side)),
                                        right = I(list(right_side)))
                                associations <<- rbind(associations, new_asoc)
                        }
                }
        })
```

```
        associations
}
```

EXPLICAR GET ASSOCIATIONS

```
print_associations = function(left_list, right_list){
        left_side = paste(left_list[[1]], collapse = ",")
        right_side = paste(right_list[[1]], collapse = ",")
        cat("{")
        cat(left_side)
        cat("} --> {")
        cat(right_side)
        cat("}\n")
}
```

Mediante la función `print_associations` se le pasan la parte izquierda y derecha de la asociación y esta imprime todos los elementos entre llaves y con una flecha en medio.

```
ap_genrules = function(table, asoc, c) {
        asoc = cbind(asoc, data.frame(matrix(1, ncol = 1,
                nrow = len(asoc[,1]), dimnames = list(1:len(asoc[,1]), "valid"))))
        for (i in len(asoc,"left"]):1) {
                A = asoc[i, "left"][[1]]
                right = asoc[i, "right"][[1]]
                B = union(A, right)

                if(asoc[i, "valid"]) {
                        conf = confidence(table, A, right)
                        if (conf < c) {
                                for (j in 1:len(A)) {
                                        A_primes = unlist(lapply(j, function(m) {
                                                        combn(A, m=m, simplify=TRUE)
                                                        }
                                                ), recursive=FALSE)
                                        for (k in seq(1, len(A_primes), by=j)) {
                                                new_left = A_primes[k:(k+j-1)]
                                                new_right = dif(B, new_left)
                                                index = indexOf_asoc(
                                                        asoc, new_left, new_right)
                                                asoc[index, "valid"] = 0
                                        }
                                }
                        }
                }
        }
        subset(asoc, asoc$valid == 1)
}
```

```
indexOf_asoc = function(df, left_side, right_side) {
        found = FALSE
        i = 1
        while(!found & i <= len(df[,1])) {
                found = equals(df[i, "left"][[1]], left_side) &
                        equals(df[i, "right"][[1]], right_side)
                i = i + 1
        }
        i-1
}
```

Finalmente, combinamos todas las funciones explicadas anteriormente en una nueva función llamada `apriori` que ejecuta el propio algoritmo dados los datos de la muestra, el umbral de soporte, y el de confianza. Construye la matriz de elementos, deshecha aquellos sucesos elementales que no superan el umbral de soporte, muestra todas las combinaciones y asociaciones posibles, y finalmente filtra aquellas que superan el umbral de confianza.

```
apriori = function(data, s, c) {
        elements = get_elements(data)
        table = get_table(data, elements)
        soporte_clasif = support_clasif(table, elements, s)
        combinations = fk_1(soporte_clasif)
        valid_support = support_clasif(table, combinations, s)

        if (len(valid_support) > 0) {

                conf = get_associations(table, valid_support)
                valid_asoc = ap_genrules(table, conf, c)

                for (i in 1:len(valid_asoc[,1])){
                        print_associations(valid_asoc[i,1], valid_asoc[i,2])
                }

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

```
data = list(
c("X", "C", "N", "B"),
c("X", "T", "B", "C"),
c("N", "C", "X"),
c("N", "T", "X", "B"),
```

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

apriori(data, 0.4, 0.9)

## {B} --> {X}
## {C} --> {X}
```

Finalmente, se observan una lista de las asociaciones que han superado los umbrales de soporte y confianza.

2.3. Detección de datos anómalos

2.3.1. Técnicas estadísticas

Ejercicio 2.3.1. *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.*

Como se ha visto previamente, el cálculo de *outliers* por medio de técnicas con base estadística no proporcionaban los mismos resultados que los vistos en clase, ya que hay ciertos cálculos (como el de los cuartiles o la desviación) están realizados de forma diferente a como se ha estudiado; y esto implica un error que se propaga al cálculo de los intervalos para valores atípicos y por consecuente al resultado final. Si recordamos, en la Sección 2.1, se han realizado nuestras propias funciones que proporcionan los resultados correctos a los cálculos. Es por ello que, en base a estas funciones, se van a realizar los algoritmos correspondientes:

En primer lugar se van a introducir los datos. Para ello se hará tal y se ha visto previamente, con la función `matrix`, de tal forma que esta tenga dos columnas (una para `velocidad` y otra para `temperatura`) y tantas filas como entradas de datos se tengan. Tras ello, se pasa esta matriz a un `dataframe` para operar sobre los datos de una forma más cómoda. El resultado es el siguiente:

```
muestra = t(matrix(c(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),
2,11,dimnames=list(c("velocidad","temperatura"))))
muestra=data.frame(muestra)

muestra
```

```
##    velocidad temperatura
## 1         10        7.46
## 2          8        6.77
## 3         13       12.74
## 4          9        7.11
## 5         11        7.81
## 6         14        8.84
## 7          6        6.08
## 8          4        5.39
## 9         12        8.15
## 10         7        6.42
## 11         5        5.73
```

Se observa que se tienen los datos dispuestos en dos columnas, una para `velocidad` y otra para `temperatura`. También se tienen 11 filas, que corresponde con el número total de parejas (`velocidad`, `temperatura`) entrantes al conjunto de datos.

Caja y Bigotes

Con los datos organizados se puede empezar a trabajar. Lo primero de todo será aplicar las medidas de ordenación a la velocidad. Esto quiere decir aplicar el algoritmo de caja y bigotes que hemos visto en la Sección 1.3.1. Primeramente, centraremos nuestra atención en la columna `velocidad`, que es la que interesa ahora:

```
muestra_velocidad = muestra$velocidad
```

Con esta línea de código se guarda la columna de velocidad, y se podrá llamar a la función que implementa el algoritmo directamente con `muestra_velocidad`. Una vez tenemos está todo listo veamos el código del algoritmo.

```
boxNmustaches = function(sample, d, details = FALSE){
	outliers = c()
	if(details){
		print("->PASO 1: DETERMINACIÓN DEL GRADO DE OUTLIER")
		cat("Se ha introducido un grado de outlier d = ",d,"\n\n")
	}
	cuart1 = quant(sample, 0.25)
	cuart3 = quant(sample, 0.75)
	if (details){
		print("->PASO 2: CÁLCULO DE LOS CUARTILES 1 Y 3")
		cat("El cuartil 1 es ",cuart1, " y el cuartil 3 es ", cuart3,"\n\n")
	}
	lim_inf = cuart1 - (d*(cuart3-cuart1))
	lim_sup = cuart3 + (d*(cuart3-cuart1))
	if (details){
		print("->PASO 3: CÁLCULO DE LOS LÍMITES DEL INTERVALO PARA VALORES ATÍPICOS")
		cat("Fórmula -> (Q1 - d*(Q3-Q1), Q3 + d*(Q3-Q1))\n")
		cat("El intervalo para los valores atípicos es: (",lim_inf,", ",lim_sup,")\n\n")
	}
	if(details){
		print("->PASO 4: IDENTIFICACIÓN DE OUTLIERS")
		cat("Los outliers serán aquellos valores que no están en el intervalo anterior\n")
	}
	for (i in 1:len(sample)){
		if(sample[i] < lim_inf || sample[i] > lim_sup){
			outliers = append(outliers, i)
			if(details){
				cat("El punto ",i," con valor ",sample[i]," es un outlier\n")
			}
		}
	}
```

```
        if(len(outliers) == 0){
                cat("No hay outliers")
        }
outliers
}
```

Para comenzar, la función recibe los siguientes parámetros:

- `sample`: El data frame con los datos correspondientes. En este caso, `muestra_velocidad`.
- `d`: El grado de *outlier* a partir del cual un dato se considera anómalo. Servirá para calcular el intervalo para valores atípicos.
- `details`: Es un parámetro que por defecto estará a `FALSE`, y que servirá para que el usuario indique si quiere una salida detallada o sencilla. Con salida sencilla nos referimos a una lista con los índices de los puntos que se han evaluado como *outliers*, mientras que con salida detallada se devolverá esto mismo además de una serie de mensajes por pantalla donde se detalla cada paso y lo que se hace en él, así como los resultados parciales que vamos obteniendo.

Vamos a centrarnos en la salida detallada para ir explicando poco a poco el algoritmo.

- `Paso 1`: En este paso determinamos el grado de outlier el cual es elegido arbitrariamente por el usuario. Esta parte del algoritmo no es más que reescribir el parámetro `d` que ha introducido a la hora de llamar a la función. Además, hacemos uso de `cat`, función contenida en el paquete `base` y que permite realizar salidas por pantalla con argumentos. En nuestro caso, queremos imprimir por pantalla, además de un mensaje, el valor de `d` en la misma línea. Para evitar hacer dos `print`, usamos esta función que pasará el valor de `d` a la cadena de texto que se imprimirá.
- `Paso 2`: En este paso vamos a calcular los cuartiles 1 y 3 (0.25 y 0.75) para posteriormente utilizarlos en el cálculo del intervalo para valores atípicos. Observamos que hacemos uso de la función `quant` que habíamos definido en la sección 2.1. Además, si el usuario lo desea, se imprime por pantalla el valor de ambos.
- `Paso 3`: Una vez calculados los cuartiles vamos a calcular el intervalo que nos permitirá clasificar los *outliers*. Para ello calculamos `lim_inf` y `lim_sup` que serán los límites inferior y superior respectivamente, ambos mediante la fórmula que hemos visto en la sección 1.3. El hecho de haber utilizado nuestra función `quant` para el cálculo de los cuartiles permite que el intervalo sea ahora el que nos salió en clase de teoría.

  `Paso 4`: Por último, recorremos todos los datos y buscamos aquellos que estén por debajo del límite inferior o por encima del límite superior, ya que al no estar comprendido en el intervalo se consideran datos anómalos. Además, añadimos a una lista `outliers` (previamente vacía) estas anomalías para posteriormente devolverlas.

Con este algoritmo vamos a resolver el ejercicio. Vamos a llamar a la función `boxNmustaches` con `muestra_velocidad` y un valor $d = 1{,}5$.

```
anomalias_vel = boxNmustaches(muestra_velocidad, 1.5, TRUE)

## [1] "->PASO 1: DETERMINACIÓN DEL GRADO DE OUTLIER"
## Se ha introducido un grado de outlier d =  1.5
##
## [1] "->PASO 2: CÁLCULO DE LOS CUARTILES 1 Y 3"
## El cuartil 1 es  6  y el cuartil 3 es  12
##
## [1] "->PASO 3: CÁLCULO DE LOS LÍMITES DEL INTERVALO PARA VALORES ATÍPICOS"
## Fórmula -> (Q1 - d*(Q3-Q1), Q3 + d*(Q3-Q1))
## El intervalo para los valores atípicos es: ( -3 ,  21 )
##
## [1] "->PASO 4: IDENTIFICACIÓN DE OUTLIERS"
## Los outliers serán aquellos valores que no están en el intervalo anterior
## No hay outliers
```

Observamos cómo se informa al usuario del grado de *outlier* que ha introducido, de los valores de los cuartiles 1 y 3, de cómo se calcula el intervalo para valores atípicos y de todos aquellos *outliers* que se han encontrado. En este ejemplo concreto no hay *outliers* ya que todos los valores se encuentran dentro del intervalo establecido. Si se quisiera acceder a la lista que contiene todos los *outliers* bastaría con acceder en este ejemplo a `anomalias_vel`.

Media y Desviación

En este segundo apartado del ejercicio vamos a detectar anomalías, esta vez sobre la columna de temperatura y aplicando medidas de dispersión. El cálculo del intervalo se realiza esta vez con la media aritmética y la desviación típica. Como hemos ido viendo, el cálculo de la desviación típica mediante la función `sd` es diferente que el que hemos visto en teoría. Por ello, podemos aplicar la función `standard_dev` que hemos elaborado en la sección 1.3 y que nos da el valor que hemos visto en clase de teoría. Primeramente vamos a guardarnos la columna de temperaturas para luego trabajar mejor sobre ella.

```
muestra_temp = muestra$temperatura
```

Ya tenemos los datos de temperatura, así que solo queda llamar a la función que implementa este algoritmo. Es bastante similar a la función anterior de `boxNmustaches`, solo que ahora en vez de calcular cuartiles calculamos la media aritmética y la desviación típica; y la fórmula del intervalo para valores atípicos también cambia, tal y como hemos visto en la sección 1.3. La función que implementa el algoritmo es la siguiente:

```
dispersed_outliers = function(sample, d, details = FALSE){
        outliers = c()
        if(details){
                print("->PASO 1: DETERMINACIÓN DEL GRADO DE OUTLIER")
                cat("Se ha introducido un grado de outlier d = ",d,"\n\n")
        }
        mean_sample = mean(sample)
        if (details){
```

```
                print("->PASO 2: CÁLCULO DE LA MEDIA ARITMÉTICA")
                cat("La media aritmética es ", mean_sample, "\n\n")
        }
        dev_sample = standard_dev(sample)
        if (details){
                print("->PASO 3: CÁLCULO DE LA DESVIACIÓN TÍPICA")
                cat("La desviación típica es ", dev_sample, "\n\n")
        }
        lim_inf = mean_sample - (d*dev_sample)
        lim_sup = mean_sample + (d*dev_sample)
        if (details){
                print("->PASO 4: CÁLCULO DE LOS LÍMITES DEL INTERVALO PARA VALORES ATÍPICOS")
                cat("Fórmula -> (Media - d*DesviaciónTípica, Media + d*DesviaciónTípica)\n")
                cat("El intervalo para los valores atípicos es: (",lim_inf,", ",lim_sup,")\n\n")
        }
        if(details){
                print("->PASO 5: IDENTIFICACIÓN DE OUTLIERS")
                cat("Los outliers serán aquellos valores que no están en el intervalo anterior\n")
        }
        for (i in 1:len(sample)){
                if(sample[i] < lim_inf || sample[i] > lim_sup){
                        outliers = append(outliers, i)
                        if(details){
                                cat("El punto ",i," con valor ",sample[i]," es un outlier\n")
                        }
                }
        }
        if(len(outliers) == 0){
                cat("No hay outliers")
        }
        outliers
}
```

Como podemos ver el código es prácticamente igual. Se pasan los mismos parámetros que en el algoritmo anterior. El primer paso recuerda al usuario el grado de dispersión que ha introducido y el segundo y tercer paso calculan la media y la desviación típica con nuestras funciones que lo hacen como hemos visto en teoría. El cuarto paso calcula los límites del intervalo para valores atípicos haciendo uso de la media y la desviación típica previamente calculadas y con la fórmula que vimos en la sección 1.3, y finalmente se recorren todos los datos en busca de los valores que no están dentro del intervalo calculado.

Al igual que en el algoritmo anterior, se devolverá una lista con los puntos que son *outliers* y se podrá elegir entre una salida detallada con mensajes por pantalla del proceso o una salida limitada a la devolución de la lista de *outliers*.

Vamos a probar nuestro algoritmo con la temperatura y un grado de dispersión $d = 2$:

```
anomalias_temp = dispersed_outliers(muestra_temp,2,TRUE)

## [1] "->PASO 1: DETERMINACIÓN DEL GRADO DE OUTLIER"
## Se ha introducido un grado de outlier d =  2
##
## [1] "->PASO 2: CÁLCULO DE LA MEDIA ARITMÉTICA"
## La media aritmética es  7.5
##
## [1] "->PASO 3: CÁLCULO DE LA DESVIACIÓN TÍPICA"
## La desviación típica es  1.935933
##
## [1] "->PASO 4: CÁLCULO DE LOS LÍMITES DEL INTERVALO PARA VALORES ATÍPICOS"
## Fórmula -> (Media - d*DesviaciónTípica, Media + d*DesviaciónTípica)
## El intervalo para los valores atípicos es: ( 3.628134 ,  11.37187 )
```

```
##
## [1] "->PASO 5: IDENTIFICACIÓN DE OUTLIERS"
## Los outliers serán aquellos valores que no están en el intervalo anterior
## El punto  3  con valor  12.74  es un outlier
```

Observamos cómo se ha ido haciendo todo el proceso (porque hemos puesto que queremos una salida detallada) y vemos que al final se ha encontrado una anomalía en el punto 3 cuyo valor es 12.74, el cual se encuentra fuera del intervalo, por lo que es considerado un *outlier*.

Si quisiéramos acceder a la lista de los puntos que son *outliers* bastaría con acceder a la variable a la que hemos igualado la llamada de la función. En este caso habría que acceder a `anomalias_temp`.

```
anomalias_temp

## [1] 3
```

Y efectivamente tenemos una lista con un 3, el punto que hemos visto que era un *outlier*.

2.3.2. Técnicas de proximidad y densidad

Ejercicio 2.3.2. *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*}*.*

Algoritmo k-vecinos

En primer lugar se va a resolver el ejercicio con técnicas basadas en proximidad (algoritmo `k-vecinos`). Para ello se programará una función principal y funciones auxiliares que vaya necesitando esta primera, con el objetivo de devolver aquellos datos que se consideran anomalías. El funcionamiento del algoritmo es el siguiente:

Para empezar, a partir de ahora cada suceso de la muestra se considerará como un punto en el plano, cuya componente x será el número de mujeres y cuya componente y será el número de hombres. Así se deberá determinar el grado de *outlier* o distancia `d` a partir de la cual un punto es considerado como un dato anómalo. Además se deberá determinar el número de orden o k-vecino más próximo. Ambos parámetros se eligen arbitrariamente, por lo que quedarán a juicio del analista. La programación del algoritmo facilitará la posibilidad de introducir ambos parámetros.

Una vez determinados `d` y `k` se deberán calcular las distancias euclídeas de cada punto al resto de puntos. La distancia euclídea en nuestro problema es definida mediante la siguiente fórmula:

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

Es por ello que se necesita una función que permita obtener el valor de la distancia euclídea entre dos puntos. Se propone la siguiente:

```
euc_distance = function(p1,p2){
        if(len(p1) == len(p2)){
                add = 0
                for(i in 1:len(p1)){
                        add = add + ((p1[i] - p2[i])^2)
                }
                sqrt(add)
        }
        else {
                print("No se puede calcular la distancia euclídea")
        }
}
```

Los parámetros `p1` y `p2` que recibe la función representan dos listas; ambas contienen los valores de las componentes de cada punto. Ambos puntos deberán tener el mismo número de componentes para poder calcularse la distancia euclídea. De esta forma se restan los valores de cada componente y se eleva al cuadrado, añadiéndose este valor a un acumulador de todas las diferencias cuadráticas de las componentes, al cual posteriormente se le aplicará la raíz cuadrada. La distancia euclídea generalizada se puede expresar de esta forma:

$$\text{dist}(p_i, p_j) = \sqrt{\sum_{i=1}^{n}(p_i - p_j)^2}$$

Una vez se tiene una función capaz de calcular las distancias euclídeas se ha de elaborar una estructura que pueda recoger la distancia de cada punto al resto de puntos. Es por ello que se propone una matriz de distancias, donde la posición (i, j) representará la distancia euclídea del punto i al punto j. Para ello se ha realizado la siguiente función:

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

```
}
```

La función `create_distance_matrix` recibe un dataframe (que como se verá más adelante corresponde con la entrada del algoritmo). A partir de ello crea otro dataframe con tantas filas y columnas como puntos haya en la muestra. Así se recorren todos los puntos y se calcula para cada uno su distancia al resto. Teniendo un punto i y un punto j, se rellenan las posiciones $[i, j]$ y $[j, i]$ del nuevo dataframe. La razón que motiva esta doble asignación es que puede optimizar el algoritmo, ya que la distancia de dos puntos es la misma en un sentido o en otro, por lo que no hace falta volver a calcular la misma distancia en el sentido opuesto.

Con las distancias de todos los puntos ya calculadas el siguiente paso es ordenar y detectar si la `k` distancia más próxima sobrepasa o no el grado de *outlier* `d`, ambos definidos por el analista. En caso de sobrepasar `d`, el punto es un *outlier* y viceversa. La función que detecta las anomalías de esta forma es la siguiente:

```
detect_outliers = function(sample, distance_matrix, k, d, details){
        if(details){
                print("->PASO 3: IDENTIFICACIÓN DE LOS OUTLIERS")
        }
        outliers = c()
        for (column in 1:len(distance_matrix[1,])){
                ordered_column = bubble(distance_matrix[column,])
                if (ordered_column[k+1] > d){
                        outliers = append(outliers,column)
                        if(details){
                                cat("El punto", column, "(",
                                paste(sample[column,], collapse = ","),
                                ") es un outlier\n")
                        }
                }
        }
        outliers
}
```

En el dataframe que había quedado antes se tiene en cada columna i-ésima las distancias del punto i-ésimo, por lo que se irá ordenando columna por columna gracias a la función `bubble` que realizada en ejercicios anteriores. De esa ordenación bastará con fijarse en la posición $k + 1$ (ya que la primera posición corresponde a la distancia del punto a sí mismo que es 0 y no es considerada un k-vecino). Si esta es mayor que d se clasifica al punto como anomalía, en caso contrario no lo será. Se devuelve de esta forma una lista con los puntos que son considerados anomalías.

Una característica de esta función es que si el parámetro `details` (que se verá ahora en más detalle) está a `TRUE`, imprime con `cat` que el punto x es un *outlier*. Además de imprimir

el número x del punto imprime las componentes que lo forman con la función `paste` que hemos visto en el ejercicio autónomo de asociación, y que pasa la lista con las componentes del punto a cadena, separando cada una con comas gracias al parámetro `collapse = ","`; de tal forma que también se puedan imprimir las componentes del punto.

Por último, se han de unir todos los módulos en una función que encapsule toda esta funcionalidad. Para ello se ha creado la siguiente función, que será a la que llame el usuario cuando quiera ejecutar el algoritmo `k-vecinos`:

```
k_neighbors = function(sample, k, d, details = FALSE){
        if(details){
                print("->PASO 1: DETERMINACIÓN DE d Y k")
                cat("Grado de outlier: d =",d,"\n")
                cat("K-Vecino más próximo: k =",k,"\n\n")
        }
        d_matrix = create_distance_matrix(sample)
        if(details){
                print("->PASO 2: MATRIZ DE DISTANCIAS ENTRE PUNTOS:\n")
                print(d_matrix)
                cat("\n")
        }
        detect_outliers(sample, d_matrix, k, d, details)
}
```

La función recibirá cuatro parámetros que se detallan a continuación:

- `sample`: La entrada del algoritmo en formato dataframe. Al introducirse en este formato el algoritmo se asegura de que está trabajando con una muestra bien construida, ya que el número de filas y columnas tiene que ser el mismo; en otras palabras, tendremos el mismo número de componentes para cada suceso de la muestra.
- `k`: K-vecino más próximo elegido para la ejecución del algoritmo.
- `d`: Distancia o grado de outlier elegido para la ejecución del algoritmo.
- `details`: Por defecto a `FALSE`. Permite mostrar una ejecución detallada de los pasos del algoritmo. Si se pone a `TRUE` imprime por pantalla el valor de los dos parámetros anteriores, la matriz de distancias y todos los puntos que son considerados *outliers* (por eso `details` también se encuentra presente en la función `detect_outliers`. Además devuelve una lista con todos estos puntos. En caso de estar a `FALSE`, se entiende que el usuario no quiere una ejecución detallada, así que solo se devuelve la lista de *outliers*.

La función llama de esta forma a `create_distance_matrix` para crear la matriz de distancias y a `detect_outliers` para clasificar según el algoritmo `k-vecinos`.

Se procede a probar la función `k_neighbors` con la muestra del ejercicio. En este caso se opta por realizar una ejecución detallada de la misma.

```
muestra = data.frame("mujeres" = c(9,9,11,2,11), "hombres" = c(9,7,11,1,9))
k_neighbors(muestra,3,3.5,TRUE)

## [1] "->PASO 1: DETERMINACIÓN DE d Y k"
## Grado de outlier: d = 3.5
## K-Vecino más próximo: k = 3
##
## [1] "->PASO 2: MATRIZ DE DISTANCIAS ENTRE PUNTOS:\n"
##          X1       X2        X3        X4        X5
## 1  0.000000 2.000000  2.828427 10.630146  2.000000
## 2  2.000000 0.000000  4.472136  9.219544  2.828427
## 3  2.828427 4.472136  0.000000 13.453624  2.000000
## 4 10.630146 9.219544 13.453624  0.000000 12.041595
## 5  2.000000 2.828427  2.000000 12.041595  0.000000
##
## [1] "->PASO 3: IDENTIFICACIÓN DE LOS OUTLIERS"
## El punto 2 ( 9,7 ) es un outlier
## El punto 3 ( 11,11 ) es un outlier
## El punto 4 ( 2,1 ) es un outlier
## [1] 2 3 4
```

Algoritmo LOF

Para resolver este ejercicio de otra forma se empleará el algoritmo LOF o Local Outlier Factor. Este se basa en la densidades y la distancia Manhattan. Esta se define de la siguiente forma para los puntos de la muestra:

$$\text{dist}(p_i, p_j) = |p_{i_1} - p_{j_1}| + |p_{i_2} - p_{j_2}|$$

Esta distancia se calcula en R mediante la función observada a continuación. Para ello se itera sobre las diferentes componentes de los puntos y se calcula su diferencia en valor absoluto, es decir, se generaliza de la siguiente manera:

$$\text{dist}(p_i, p_j) = \sum_{i=1}^{n} |p_i - p_j|$$

```
manhattan_distance = function(p1, p2) {
        add = 0
        for(i in 1:len(p1)){
                add = add + abs(p1[i] - p2[i])
        }
        add
}
```

Como se necesita calcular la distancia entre cada posible par de puntos, se organizan estos datos en una matriz (representada por un dataframe) haciendo uso de la función anterior. Se tiene en cuenta que $\text{dist}(x_i, x_j) = \text{dist}(x_j, x_i)$ y se recoge en la siguiente función.

```
create_distance_matrix = function(df){
        empty_matrix = matrix(ncol = len(df[,1]), nrow = len(df[,1]))
        distances = data.frame(empty_matrix)
        for (i in 1:len(df[,1])){
                p_dists = c()
                for (j in i:len(df[,1])){
                        dist = manhattan_distance(df[i,], df[j,])
                        distances[i,j] = dist
                        distances[j,i] = dist
                }
                distances = rbind(distances, p_dists)
        }
        distances
}
```

Ahora se necesitará ordenar las distancias a los puntos para construir el conjunto $N(x_i, k)$. Para ello se hará uso del algoritmo de ordenación de la burbuja explicado en ejercicios anteriores, pero se le realizarán una serie de modificaciones para adaptarlo a este caso. Esto se realiza con la función que se presenta a continuación.

```
bubble_LOF = function(d_list, p_list){
        n = len(d_list)
        for (i in 2:n){
                for (j in 1:(n-1)){
                        if (d_list[j] > d_list[j+1]){
                                temp = d_list[j]
                                d_list[j] = d_list[j+1]
                                d_list[j+1] = temp

                                temp = p_list[j]
                                p_list[j] = p_list[j+1]
                                p_list[j+1] = temp
                        }
                }
        }
        data.frame("dis"=d_list, "poi"=p_list)
}
```

Esta función recibe los parámetros `d_list` y `p_list`, que representan las listas de las distancias y de los punto, respectivamente. En realidad, realiza lo mismo que el algoritmo visto en otros ejercicios, ordenando la lista `d_list`, pero aplica los cambios en las sucesivas

iteraciones también en la lista `p_list`. Se devolverán dos columnas, una con listas de puntos, y otra con listas a las distancias a dichos puntos, ordenadas de forma ascendente. Estas listas serán paralelas (compartirán índices), y representarán al mismo elemento.

Una vez ordenadas las distancias y los puntos, deberán ser calculados los conjuntos $N(x_i, k)$. Por su propia definición debemos mirar (al menos) hasta el $k-$vecino más cercano al punto, y añadir estos, y en caso de que los sucesivos $k + \alpha$ compartan distancia con el k, entonces también deberán ser añadidos. Esto se recoge en la siguiente función:

```
get_n_set = function(df, k) {
        d_list = list()
        p_list = list()

        distance_matrix = create_distance_matrix(df)

        for (column in 1:len(distance_matrix[1,])){
                ordered_column = tail(bubble_LOF(distance_matrix[,column],
                 1:ncol(distance_matrix)), ncol(distance_matrix)-1)

                k_aux = k

                while(k_aux < nrow(ordered_column) &
                ordered_column[k_aux, "dis"] == ordered_column[k_aux+1, "dis"]) {
                        k_aux = k_aux + 1
                }

                d_list = append(d_list, list(head(ordered_column[,"dis"], k_aux)))
                p_list = append(p_list, list(head(ordered_column[,"poi"], k_aux)))
        }
        data.frame(dis = I(d_list), poi = I(p_list))
}
```

En esta función se seleccionan los primeros $k-$vecinos. Obviamente, en primer lugar se eliminan los elementos $\text{dist}(x_i, x_i)$, pues siempre será igual a 0. Esto se logra con la función `tail` del paquete `utils` y que permite conservar el resto de distancias. Una vez seleccionados los k más cercanos, se va avanzado sobre el resto de vecinos y se revisa si el siguiente al último debe añadirse debido a que sean iguales, y así sucesivamente hasta encontrar una pareja de distancias diferentes, o no tener más vecinos. Se utiliza la función `head` del paquete `utils` para obtener el conjunto $N(x_i, k)$ y se devuelven en un dataframe.

El siguiente paso es calcular la densidad de cada punto. Esta se realiza mediante la siguiente expresión:

$$\text{dens}(x_i, K) = \frac{|N(x_i, K)|}{\sum_{x_j \in N(x_i, K)} \text{dist}(x_i, x_j)}$$

En el código queda representado mediante la siguiente función. Basta con acceder a los datos correctos del dataframe devuelto por la función `get_n_set`. Esta función devuelve una columna con las densidades de cada punto, la fila $i-$ésima almacena $\text{dens}(x_i, k)$.

```
add_list = function(l) {
        add = 0
        for (i in 1:len(l)) {
                add = add + l[i]
        }
        add
}
```

```
get_densities = function(n_set) {
        densities = c()
        for (i in 1:len(n_set[,1])) {
                densities = append(densities, len(n_set[i,1][[1]]) /
                 add_list(n_set[i,1][[1]]))
        }
        densities
}
```

Se hace uso de la función auxiliar `add_list`, que devuelve la suma de los elementos de una lista. En este caso $\sum_{j=1}^{|N|} \text{dist}(x_i, x_j)$.

A continuación, deberán calcularse las densidades relativas medias de cada punto, pudiendo identificar mediante estas los outliers. Esta se calcula mediante la siguiente expresión.

$$\text{drm}(x_i, K) = \frac{\text{dens}(x_i, K) \cdot |N(x_i, K)|}{\sum_{x_j \in N(x_i, K)} \text{dens}(x_j, K)}$$

En el código se calcula mediante la siguiente función, que representa la ecuación mostrada, pero adaptada a las estructuras de datos utilizadas durante el resto del algoritmo. Al igual que hacía `get_densities`, se devuelve una columna con los valores de $\text{drm}(x_i, k)$.

```
get_drm = function(n_set) {
        drms = c()
        for (i in 1:len(n_set[,1])) {
                add_densities = 0

                points = n_set[i, "poi"][[1]]
                for (j in 1:len(points)) {
                        add_densities = add_densities +
                                n_set[points[j], "densities"]
```

```
                }

                drm = n_set[i, "densities"] * len(points) / add_densities
                drms = append(drms, drm)
        }
        drms
}
```

Finalmente, se unen todos los pasos comentados en la función `lof`. Esta muestra un dataframe con los parámetros que se han ido calculando. Para ello se emplea la función `cbind` para ir construyendo dicho dataframe.

```
lof = function(muestra, k) {
        n_set = get_n_set(muestra, k)

        densities = get_densities(n_set)
        n_set = cbind(n_set, densities)

        drms = get_drm(n_set)

        n_set = cbind(n_set, drms)
        n_set
}
```

Se comprueban los resultados obtenidos, dejando a juicio del analista el valor de k y la selección de outlier en función de la drm.

```
muestra = data.frame("mujeres" = c(9,9,11,2,11), "hombres" = c(9,7,11,1,9))
(lof(muestra, 3))
```

```
##          dis     poi  densities     drms
## 1    2, 2, 4 2, 5, 3 0.37500000 1.285714
## 2    2, 4, 6 1, 5, 3 0.25000000 0.750000
## 3    2, 4, 6 5, 1, 2 0.25000000 0.750000
## 4 13, 15, 17 2, 1, 5 0.06666667 0.200000
## 5    2, 2, 4 1, 3, 2 0.37500000 1.285714
```