

Fundamentos de la Ciencia de Datos. Prueba de Laboratorio 1

Ana Cortés Cercadillo¹, Carlos Javier Hellín Asensio², Daniel Ferreiro Rodríguez³, and Francisco Calles Esteban⁴

University of Alcalá, Ctra. Madrid-Barcelona km 33
28805 Alcalá de Henares, Madrid, Spain

¹a.cortesc@edu.uah.es, ²carlos.hellin@edu.uah.es, ³daniel.ferreiror@edu.uah.es,
⁴francisco.calles@edu.uah.es

November 9, 2021

Abstract

En este documento se encontrará un análisis de descripción de datos y un análisis de asociación, aplicando todos los conceptos teóricos vistos en cada lección. Primero se realizan estos análisis de forma guiada y finalmente, con los conceptos teóricos adquiridos, se plantean diferentes formas de afrontar los análisis.

Keywords: Ciencia de datos, Análisis, R, Estadística descriptiva, Paquetes en R, Algoritmo Apriori

Contents

1	Introducción	2
2	Primera parte	2
2.1	Análisis de descripción de datos	2
2.1.1	Directorio de trabajo	2
2.1.2	Formato del fichero de datos	3
2.1.3	Lectura de un fichero de datos en formato tabla	3
2.1.4	Cálculos con matrices	5
2.1.5	Funciones en R	7
2.1.6	Primer análisis de los datos	8
2.1.7	Segundo análisis de los datos	10
2.1.8	Tercer análisis de los datos	10
2.1.9	Cuarto análisis de los datos	12
2.2	Análisis de asociación	13
2.2.1	Instalación de paquetes	13
2.2.2	Aplicación del algoritmo Apriori	23

3	Segunda parte	26
3.1	Análisis de descripción de datos	26
3.1.1	Formato del fichero de datos en CSV	26
3.1.2	Lectura de un fichero de datos en formato CSV	27
3.1.3	Cálculos con matrices	29
3.1.4	Funciones en R	32
3.1.5	Primer análisis de los datos	32
3.1.6	Segundo análisis de los datos	36
3.1.7	Tercer análisis de los datos	38
3.1.8	Cuarto análisis de los datos	40
3.2	Análisis de asociación	41
3.2.1	Instalación de paquetes	41
3.2.2	Creación de paquetes	48
3.2.3	Aplicación del algoritmo Apriori	50
4	Conclusiones	58

1 Introducción

La Ciencia de los Datos es una nueva disciplina que permite obtener conocimiento a partir de los datos mediante métodos estadísticos y matemáticos. Permite descubrir correlaciones y causas entre los datos muy útiles en toma de decisiones en investigación científica u actividad de negocios.

En este documento se va a abordar uno de los grupos de áreas de conocimiento o *Knowledge Area Groups (KAG)* que pertenecen a la ciencia de datos, el análisis de datos o *Data Science Analytics (DSA)*.

Se centrará el estudio en grupos de datos pequeños aplicando el uso de la estadística para obtener información de ellos, aunque, en primer lugar, se plantearán problemas estadísticos simples para familiarizarse con el entorno de trabajo, R. En segundo lugar se usarán algoritmos para encontrar patrones de aparición conjunta en estos conjuntos de datos.

2 Primera parte

2.1 Análisis de descripción de datos

2.1.1 Directorio de trabajo

Para poder cargar un fichero .txt en un *script* o en el área de trabajo de la consola de R es imprescindible saber en qué directorio se encuentra el archivo que se quiere cargar, para ello primero habrá que averiguar cuál es el directorio de trabajo actual del proceso R.

El directorio de trabajo de un proceso R es el directorio o localización en el sistema donde se trabajará con los archivos del proceso (el historial de comandos y el espacio de trabajo), los archivos de funciones creadas y los archivos externos que se cargarán en el proceso (en este caso el archivo de texto “satélites.txt”). Para poder localizar el directorio de trabajo actual se utilizará el comando *getwd()*. Este comando devolverá la ruta absoluta del directorio de trabajo.

En caso de que se quiera cambiar la ruta del directorio de trabajo se utilizará el comando *setwd(dir)*, donde *dir* será la ruta absoluta del directorio que se querrá utilizar en el área de trabajo. Una vez que se ha configurado el nuevo directorio de trabajo se podrá comprobar que es así con el comando anterior *getwd()* y se podrá visualizar el contenido de dicho directorio con *list.files()* que sin ningún argumento devolverá

un vector con los nombres de los archivos y directorios contenidos en el directorio de trabajo. También se podrá utilizar `dir()` [7]. Ambos comandos cuentan con los siguientes argumentos: *path* que es la ruta del directorio del que se quiere listar los archivos (`getwd()` por defecto); *pattern*, una expresión regular opcional; *all.files* que es un valor lógico que indica si se quiere ver también los archivos no visibles; *full.names* que es un valor lógico que indica si los archivos que se van a listar se muestran con su ruta en el directorio; *recursive* que indica si el listado recurre a directorios; *ignore.case* indica si los patrones son *case sensitive*; *include.dirs* indica si los nombres de los subdirectorios están incluidos en el listado recursivo; y *no..* que indica si `".."` y `"."` son excluidos del listado no recursivo.

Un ejemplo del listado del directorio de trabajo sería el siguiente:

```
> list.files()
[1] "frecrel.R"      "historico"      "rango.R"      "satelites.txt"
```

2.1.2 Formato del fichero de datos

El fichero de texto que se leerá será `satelites.txt` y se creará de la siguiente manera:

El archivo contiene una primera fila en la que se establecen los tipos de datos que se van a mostrar en cada columna, en este caso la primera columna no tendrá nombre pues solo se indicará el número de fila del satélite indicado; la segunda columna será "Nombre" (el nombre del satélite) es decir un dato cualitativo; y la tercera y última columna será "Radio" (el radio del satélite) es decir un dato cuantitativo continuo. Las demás filas representarán a cada uno de los satélites con los datos que se han descrito anteriormente. La separación de las columnas se hace mediante las tabulaciones. También es importante que el archivo sea guardado en codificación ANSI para que a la hora de ser cargado por R se puedan mostrar los caracteres con tilde.

Para una mejor visualización, en la Figura 1 se puede observar el uso de las tabulaciones, que se marcan con una flecha, y los saltos de línea, que aparecen en forma de etiqueta con los caracteres `"CRLF"`.

2.1.3 Lectura de un fichero de datos en formato tabla

Para poder leer un fichero `txt` R ofrece la función `read.table` [7] cuyos argumentos más importantes son *file* que es el nombre del archivo en formato string o su ruta absoluta; *header* que se establecerá al booleano `TRUE` o `FALSE` dependiendo de si se quiere mostrar el encabezado del archivo; *sep* para indicar el carácter que separe las columnas del archivo; o *dec* para indicar el carácter utilizado para los puntos decimales, entre otros argumentos más que permite utilizar la función. Esta función leerá el archivo en formato de tabla y creará un *data frame* de los datos del archivo. Estos *data frames* son una clase de objetos R que representan estructuras de datos de dos dimensiones cuyos datos pueden ser de diferentes tipos.

Para poder guardar el *data frame* creado se puede utilizar una variable, para ello se asociará la variable nueva con el comando de `read.table` mediante el uso de `"="` o de `"←"`. Si se utiliza el comando con solo el nombre del archivo como argumento, el contenido de la variable será el siguiente.

```
> s<-read.table("satelites.txt")
> s
```

	Nombre	Radio
1	Cordelia	13
2	Ofelia	16
3	Bianca	22
4	Crésida	33
5	Desdémona	29
6	Julietta	42

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

Figure 1: Formato del fichero satélites.txt.

```

7      Rosalinda    27
8      Belinda     34
9 Luna-1986U10     20
10     Calíbano     30
11     Luna-999U1   20
12 Luna-1999U2     15

```

Se puede observar que el *data frame* creado muestra el contenido del *txt* lo más parecido al formato del archivo de texto ya que al pasar solo como argumento el nombre del archivo se han dejado por defecto los demás como el *header* que al ser *TRUE* por defecto se muestra la primera fila con los nombres de los datos de cada columna y que actuarán como variables para poder trabajar con los datos de forma separada. El argumento *sep* al ser por defecto " " es decir el espacio en blanco pues se establece que el separador de las columnas es más de un espacio en blanco por lo que en caso de que por ejemplo un nombre de un satélite contenga dos palabras pues se hubieran separado en dos columnas lo que resultaría en una fila con cuatro columnas y no se cumpliría el formato establecido.

Se pueden realizar unos ejemplos con los argumentos de *read.table* para ver que cambios o errores se producen en la carga del fichero.

Si el *header* se establece a *FALSE* se produce un error en el escaneado pues la primera fila al no tener tres elementos como las demás ya se establece como el encabezado del archivo por lo que indicar que no hay encabezado se quiere dejar a entender que todas las filas tienen el mismo número de elementos, afirmación que es falsa.

```

> s<-read.table("satelites.txt",header = FALSE)
Error in scan(file = file, what = what, sep = sep, quote = quote,
  dec = dec,: line 1 did not have 3 elements

```

Si el separador de columnas se establece al tabulador el formato del *data frame* cambia y se muestra una nueva columna "X" cuyos valores son también el número de fila.

```

> s<-read.table("satelites.txt",header = TRUE,sep="\t")
> s
      X      Nombre Radio
1  1      Cordelia    13
2  2      Ofelia     16
3  3      Bianca     22
4  4      Crésida    33
5  5      Desdémona  29
6  6      Julieta    42
7  7      Rosalinda  27
8  8      Belinda    34
9  9 Luna-1986U10    20
10 10     Calibano    30
11 11     Luna-999U1  20
12 12 Luna-1999U2    15

```

2.1.4 Cálculos con matrices

El *data frame* creado anteriormente con la función *read.table()* y guardado en la variable *s* se organiza en una matriz de datos con la que se podrá trabajar utilizando diferentes funciones que nos ofrece R.

Como ejemplo se han utilizado cuatro funciones que influyen en todos los datos de la matriz, estas son *dim()*, *order()*, *rev()* y *length()* [7].

La función `dim()` devuelve las dimensiones de un objeto R como una matriz, un *array* o un *data frame*. Recibe como argumento el objeto R y el valor que devuelve puede ser guardado en una variable. Dicho valor puede ser un objeto *NULL*, un entero o un vector de enteros, dependiendo del tipo y contenido del objeto R del argumento. En caso de los *data frame* y matrices se mostrará el número de filas y columnas de estos.

Aplicando la función en la variable `s` se mostrará lo que se puede ver a continuación:

```
> dim(s)
[1] 12 2
```

Como se puede observar los datos del *data frame* se distribuyen en doce filas y dos columnas.

La función `order()` ordena un vector de datos de manera ascendente o descendente mediante un método de ordenación. Sus argumentos son el objeto R que se quiere ordenar; *na.last* que controla el orden de los datos perdidos o *NAs*, *TRUE* si se quieren colocar últimos y *FALSE* si se quieren colocar los primeros; *decreasing* si el orden va a ser descendente estableciéndolo a *TRUE*; y *method* que indica el método de ordenación que se va a utilizar (*auto*, *shell*, *quick* y *radix*). Si sólo se le pasa como argumento el vector lo ordena de manera ascendente y con el método automático por defecto (*radix* en caso de valores enteros). La función devuelve un vector con los índices del vector pasado como argumento ordenados.

Con los datos de los satélites se puede utilizar el radio de estos para ordenarlos de menor a mayor y guardarlo en un nuevo *data frame*.

Para ello primero se guardará en una variable los radios de los satélites de la siguiente manera:

```
> radio = s$Radio
> radio
[1] 13 16 22 33 29 42 27 34 20 30 20 15
```

Si se quiere ver los índices de los datos ordenados de forma ascendente se utilizará la función `tal cual` pero para crear el nuevo *data frame* de los satélites ordenados por el radio se realizará de la siguiente manera:

```
> order(radio)
[1] 1 12 2 9 11 3 7 5 10 4 8 6
> so = s[order(radio),]
> so
```

	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	Calibano	30
4	Crésida	33
8	Belinda	34
6	Julietta	42

Tal como se puede ver los satélites ya están ordenados por su radio

La función `rev()` devuelve una versión invertida del argumento que se le pase. Dicho argumento puede ser un vector u otro objeto que permita su versión invertida.

Para el caso de los satélites se puede crear un *data frame* con éstos ordenados por sus radios de forma inversa, es decir, ordenados de mayor a menor radio. Para ello se puede realizar lo siguiente:

```
> sor = s[rev(order(radio)),]
> sor
```

	Nombre	Radio
6	Julietta	42
8	Belinda	34
4	Crésida	33
10	Calibano	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

La función *rev()* recibe como argumento el vector con los índices ordenados de menor a mayor. La función devuelve los índices ordenados de mayor a menor que pasado a la variable del *data frame* de los satélites devuelve el nuevo *data frame* que se muestra anteriormente.

La función *length()* devuelve un entero de la longitud de un vector de elementos u objeto R pasado como argumento. A diferencia de la función *dim()*, *length()* sirve para obtener la dimensión de objetos unidimensionales, en caso de que se utilice la función en un objeto de dos dimensiones (*array* o matriz) solo se mostrará la longitud de una de las dimensiones (en el *data frame* de los satélites se muestra el número de columnas).

Por ejemplo la longitud del vector de radios sería la siguiente:

```
> length(radio)
[1] 12
```

2.1.5 Funciones en R

En el anterior apartado vimos las matrices, a continuación, explicaremos tres funciones de R: *function()*, *dump()*, *source()* [7]. Se verá que relación hay entre ellas.

La función *function()* define nuevas funciones en el lenguaje R. La creación de una nueva función tiene la siguiente pinta:

```
variable = function(parámetro/s){código de la función}
```

Ejemplo:

```
> rango = function(x){max(x) - min(x)}
```

La función *range()* en R muestra el valor mínimo y máximo del vector pasado por parámetro, pero no calcula su rango. Con este código se consigue crear una función llamada *rango()* que calcula el rango (*máximo – mínimo*) para un vector dado.

```
> rango(radio)
[1] 29
```

Nota: Si no se guarda la función, solo se podrá usar durante la sesión de creación.

La función *dump()* permite salvar una función o script en un archivo que se crea en el directorio de trabajo si no se especifica una ruta. Algunos de sus argumentos son los siguientes: *list*: vector de caracteres. Los nombres de uno o más R objetos que

se van a volcar; *file*: cadena de caracteres que nombra un archivo o una conexión. `""` indica una salida a la consola; *append*: lógico. Si es *TRUE* y *file* es una cadena de caracteres, la salida se agregará a *file*, de lo contrario, sobrescribirá los contenidos de *file*; *control*: vector de caracteres que indica las opciones de análisis; *envir*: el entorno para buscar objetos.

El próximo código añade la función *rango()* en el fichero *rango.R*:

```
> dump("rango", file = "rango.R")
```

En R hay 3 tipos de extensiones de fichero:

- .R (programas)
- .RDATA (variables)
- .RHISTORY (instrucciones)

La función *source()* permite abrir y ejecutar *scripts* en R pasándole como argumento la ruta, entre comillas, del archivo en el dispositivo. R lee todo el archivo y se analiza las expresiones que se evalúan secuencialmente en el entorno elegido. Algunos argumentos de la función *source()*: *file*: una conexión o una cadena de caracteres que proporciona el nombre de la ruta del archivo o URL para leer. `""` indica la conexión *stdin()*; *local*: *TRUE*, *FALSE* o entorno, determinando dónde están las expresiones evaluadas. La opción *FALSE* (el valor predeterminado) corresponde al espacio de trabajo del usuario y *TRUE* al entorno desde el que se llama al *source*; *echo*: lógico. Si es *TRUE*, cada expresión se imprime después del análisis, antes de la evaluación; *verbose*: Si es *TRUE*, se imprimen más diagnósticos (además de *echo = TRUE*) durante el análisis y evaluación de la entrada, incluida información adicional para cada expresión; *chdir*: lógico. Si es *TRUE* y el archivo es un nombre de ruta, el directorio de trabajo de R está temporalmente cambiado al directorio que contiene el archivo para evaluar; *encoding*: vector de caracteres. Las codificaciones que se deben asumir cuando el archivo es una cadena de caracteres.

El código siguiente permite recuperar la función *rango()* antes creada, en la sesión que se requiera:

```
> source("rango.R")
```

2.1.6 Primer análisis de los datos

En este apartado se empieza con el primer análisis. Primero se explicará los conceptos teóricos para luego entrar en profundidad en las funciones vistas en R, de tal forma que se pueda comprobar que los resultados dados son los mismos que se han visto en clase de teoría.

Empezando con el cálculo de la frecuencia absoluta f_i de los datos, que es el número de apariciones de un dato.

$$f_i = \text{número de apariciones de un dato}$$

Para hacerlo en R se calcula con una función llamada *table()* y se le indica qué variable se quiere calcular la frecuencia absoluta. En este caso se hace con el radio del fichero *satelites.txt* que ya ha sido cargado en secciones anteriores.

Se guarda en la variable *fabsr* y se muestra los resultados como se puede ver a continuación:

```
> fabsr = table(s$Radio)
> fabsr
radio
13 15 16 20 22 27 29 30 33 34 42
 1  1  1  2  1  1  1  1  1  1  1
```


Todos los valores tienen uno como frecuencia, excepto el 20 que son dos. Como argumento para `table()` se le puede pasar uno o más objetos o una lista.

Lo siguiente es calcular la frecuencia acumulada absoluta f_{ai} , que es la suma de las frecuencias absolutas de todos los datos inferiores al dato más la del dato.

$$f_{ai} = \sum_{i=1}^m f_i$$

En R se usa la función `cumsum()` y se va a aplicar a la frecuencia absoluta de radio cuya variable se ha definido anteriormente como `fabsr`. El único argumento que se le puede pasar es un objeto numérico o complejo, o también un objeto que pueda ser convertido a uno de estos.

```
> faacumr = cumsum(fabsr)
> faacumr
13 15 16 20 22 27 29 30 33 34 42
 1  2  3  5  6  7  8  9 10 11 12
```

Como se ve en los resultados, la primera sale uno, el segundo dos, etc... pero si se observa con el 20, al tener dos valores, aparece un cinco. También importante que el último valor indica el número total de valores, que en este caso es doce.

Para la frecuencia relativa f_{ri} es la frecuencia absoluta de un dato dividida entre el número de datos.

$$f_{ri} = \frac{f_i}{\text{número total de elementos}}$$

En R no hay una función para realizar el cálculo, pero con lo visto en los apartados anteriores, es fácil definir una propia función usando `table()` y `length()` para que calcule la frecuencia relativa.

```
> frel = function(x){table(x) / length(x)}
> dump("frel", file = "frecrel.R")
```

La función `frel()` tiene como entrada el argumento x que es pasado a `table()` siendo el numerador para obtener la frecuencia absoluta y en el denominador también es pasado el argumento a `length()` que es el número total de datos de x . Por último, se guarda en un fichero llamando a la función `dump()` que ha sido ya explicada en la sección *Funciones en R*.

Para probar la nueva función, se le pasa a x como el radio de satélites y se aplica la función.

```
> frelr = frel(s$Radio)
> frelr
x
      13      15      16      20      22      27
0.08333333 0.08333333 0.08333333 0.16666667 0.08333333 0.08333333
      29      30      33      34      42
0.08333333 0.08333333 0.08333333 0.08333333 0.08333333
```

Si se comprueba los resultados, se ve que cada uno da 0.083, excepto el 20 que es el doble y da 0.16.

Para el cálculo de la frecuencia acumulada relativa f_{rai} hacemos la suma de las frecuencias relativas.

$$f_{rai} = \sum_{i=1}^m f_{ri}$$

Esto en R es tan sencillo como volver a usar `cumsum()` pero le pasamos como parámetro la variable `frelr` de haber calculado antes la frecuencia relativa del radio de satélites.

```
> fracumr = cumsum(frelr)
> fracumr
      13      15      16      20      22      27
0.08333333 0.16666667 0.25000000 0.41666667 0.50000000 0.58333333
      29      30      33      34      42
0.66666667 0.75000000 0.83333333 0.91666667 1.00000000
```

Como se puede observar, la acumulada relativa va subiendo y al igual que ocurre en la absoluta, que el último número más alto debería darnos el número total de valores, aquí en la relativa debe dar uno.

2.1.7 Segundo análisis de los datos

El segundo análisis de los datos consiste en el cálculo de la media aritmética.

Como en el primer análisis, primero se explicará el concepto teórico para verificar que los resultados coinciden con los que ofrecen las funciones en R a utilizar.

La media aritmética consiste en el valor numérico que se obtiene al sumar los valores de todos los datos de una muestra y dividirlo entre el número total de datos.

$$\bar{x} = \frac{\sum_{i=1}^n x_i}{n}$$

Existe una segunda fórmula en la que se hace referencia a la frecuencia de clase f_i , es decir, el número de veces que aparece un dato en la muestra.

$$\bar{x} = \frac{\sum_{i=1}^n f_i \cdot x_i}{n}$$

En R se utiliza la función `mean()` que realiza una media aritmética ajustada. Los argumentos de esta función son un objeto R que suele ser un vector de datos numéricos; el `trim` o valor de ajuste que está comprendido entre 0 y 0.5; `na.rm` que indica en un valor lógico si se van a eliminar los datos desaparecidos; y otros argumentos pasados de otros métodos.

Por ejemplo, si utilizamos la función con un solo argumento, el radio de los satélites, se realizará una media aritmética sin ajuste de estos datos.

```
> mean(radio)
[1] 25.08333
```

Al realizar el cálculo manualmente con una de las fórmulas descritas anteriormente se puede comprobar que el resultado coincide con el que muestra la función R.

2.1.8 Tercer análisis de los datos

El tercer análisis de los datos consiste en el cálculo de las medidas de dispersión. En este caso, se explicará el uso de la desviación típica, la varianza y el rango. Todas ellas se usan para el cálculo de la dispersión absoluta, la desviación y la varianza para la media y el rango para las medidas de ordenación.

En primer lugar, la desviación más utilizada es la típica. Para obtener su valor, se usa la siguiente fórmula matemática:

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

En R, la desviación típica se calcula con la función `sd()` (*Standard Deviation*). Esta función calcula la desviación estándar de los valores que se le pasa por parámetro. Puede ser un vector numérico o un objeto R.

Un ejemplo aplicado a los datos obtenidos del fichero anterior es este:

```
> sdr = sd(radius)
> sdr
[1] 8.857029
```

Por otro lado, la varianza es la media de las desviaciones cuadráticas de una variable aleatoria, referidas al valor medio de estas. Se calcula mediante la siguiente fórmula:

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

La función *var()* es la que se usa en R para calcular la varianza. Los argumentos pueden ser un vector o matriz compleja. Su ejecución es la siguiente:

```
> varr = var(radius)
> varr
[1] 78.44697
```

Si se aplica la fórmula teórica descrita anteriormente en ambos casos, ya sea la desviación típica o la varianza, el resultado no saldrá el mismo valor que el obtenido a través de las funciones de R. Esto se debe a que R en vez de dividir la fracción entre n , la divide entre $n - 1$. Quiere decir que se está tomando la definición de la desviación típica y de la varianza para poder ser empleadas con la población y hacer inferencia estadística. Las formulas que se usan en inferencia son estas:

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

Figure 2: Desviación típica para inferencia

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

Figure 3: Varianza para inferencia

Cuando se usa en estos casos, las formulas con el denominador $n - 1$ son menos cerradas, dan mejores resultados, que las del denominador n . Como en el ejemplo de este documento se hace estadística descriptiva y no inferencia, es más adecuado hacerlo entre n , que es más correcto. Entonces las ejecuciones de R anteriores se tienen que ajustar de forma que el resultado sea el mismo que usando la formula estadística. Para ello se hacen los siguientes cambios:

```
> sdr = sqrt((sdr^2) * 11 / 12)
> sdr
[1] 8.47996

> varr = varr * 11 / 12
> varr
[1] 71.90972
```

Por último, el rango es otro cálculo para obtener la dispersión absoluta. El rango es la diferencia entre el mayor valor y el menor valor de los que se disponen.

$$v_{sup} - v_{inf}$$

En R no hay ninguna función que lo calcule, entonces se obtiene haciendo operaciones.

```
> rangor = max(radio) - min(radio)
> rangor
[1] 29
```

Como no hay una función que calcule el rango, se puede definir una función que lo calcule. Esta función sería la siguiente:

```
> rango = function(x){max(x) - min(x)}
```

La llamada a esta función es:

```
> rango(radio)
[1] 29
```

2.1.9 Cuarto análisis de los datos

Las medidas de ordenación son la mediana y los cuantiles.

El objetivo de este análisis es ordenar los datos por orden de magnitud, esto es aplicable a medidas cuantitativas. La ordenación permite hacer un nuevo análisis de los datos.

La mediana es el elemento central de una serie ordenada de valores crecientes de forma que la divide en dos partes iguales, superiores e inferiores a él.

Hay dos métodos para calcular la mediana: sin agrupar los datos y agrupando los datos en clases. Nos centraremos en el primer método porque fue el que se vio en clase. Este tiene en cuenta si el total de datos es par o impar.

- Si n es par: $\tilde{x} = \frac{x_{n/2} + x_{(n/2)+1}}{2}$
- Si n es impar, es el valor central: $\tilde{x} = x_{(n+1)/2}$

En R se utiliza la función `median()` que realiza la mediana del vector o objeto de R pasado como parámetro. El parámetro x : indica la variable sobre la cual se desea hacer el cálculo; $na.rm$: lógico. Indica si los valores *NA* (no disponibles) deben eliminarse antes del cálculo producto.

A continuación, se muestra la mediana del vector `radio`:

```
> medr = median(radio)
> medr
[1] 24.5
```

La función `median()` en este caso recibe como argumento el vector numérico `radio` y determina el elemento central de este.

Los cuantiles son los elementos que permiten dividir un conjunto ordenados de datos en un conjunto de partes de igual tamaño. Pueden ser cuantiles, deciles y percentiles. La observación más pequeña corresponde a una probabilidad de 0 y la más grande a una probabilidad de 1.

Los cuantiles se calculan de la siguiente manera:

- Si $nc \notin \mathbb{N}$: $\tilde{x}_{c,d,p} = x_{[nc+1]}$ $[nc]$ parte entera de nc
- Si $nc \in \mathbb{N}$: $\tilde{x}_{c,d,p} = \frac{x_{nc} + x_{nc+1}}{2}$

En R se utiliza la función `quantile()` [7] que realiza la mediana del vector o objeto de R pasado como parámetro. El parámetro x : vector numérico cuyos cuantiles de muestra se deseen o un objeto de una clase para el que se ha definido un método; $na.rm$: lógico. Si es verdadero, cualquier *NA* y *NaN* se eliminan de x antes de que se calculen los cuantiles; $probs$: vector numérico de probabilidades con valores entre 0 y 1.

Los cuartiles son los elementos de una serie ordenada de valores crecientes de forma que la dividen en cuatro partes iguales ($c = \frac{1}{4}, \frac{2}{4}, \frac{3}{4}$). El valor $\frac{2}{4}$ es igual a la mediana. La función *quantile()* recibe como parámetro el vector *radio* y el valor cuartil que se desea (0.25, 0.5 o 0.75), devolviendo el número que se encuentra en el cuartil correspondiente.

```
> cuar1r = quantile(radio, 0.25)
> cuar1r
25%
19
```

```
> cuar2r = quantile(radio, 0.5)
> cuar2r
50%
24.5
```

```
> cuar3r = quantile(radio, 0.75)
> cuar3r
75%
30.75
```

Los deciles son los elementos de una serie ordenada de valores crecientes de forma que la dividen en diez partes iguales. Un ejemplo de ello sería el cuartil 0.5. Los percentiles son los elementos de una serie ordenada de valores crecientes de forma que la dividen en cien partes iguales.

Se muestra, a continuación, el valor que se encuentra en el percentil 54:

```
> perc54r = quantile(radio, 0.54)
> perc54r
54%
26.7
```

2.2 Análisis de asociación

2.2.1 Instalación de paquetes

Para saber cómo instalar paquetes, lo primero es tener conocimiento de qué paquetes tiene R ya instalados. Para visualizar esto se usa la instrucción *getOption()* con el argumento *"defaultPackages"*.

```
> getOption("defaultPackages")
[1] "datasets" "utils" "grDevices" "graphics" "stats" "methods"
```

Además de los seis paquetes que aparecen, hay uno más que es el paquete *base*. Este es el núcleo de R, el paquete que permite funcionar. Luego se cargan los seis paquetes adicionales para cualquier ejecución de R.

Para saber lo que hay en cada paquete, se puede usar la función *library()* añadiendo en sus argumentos *help* seguido del paquete que se quiere saber las funciones que tiene. Se abre una ventana nueva que indica qué funciones tiene ese paquete del que se está pidiendo la información. Un ejemplo de esta ejecución es la siguiente:

```
> library(help="base")
```

El paquete *datasets* contiene conjuntos de datos que ya están previamente cargados por defecto en R y sobre los que se puede trabajar.

El paquete *utils* contiene una colección de funciones de utilidad. Por ejemplo, la función *help* se encuentra en este paquete.

El paquete *grDevices* admite funciones con gráficos y cuadrículas.



Figure 4: Funciones del paquete *base*.

El paquete *graphics* es el primer paquete que ofrece R sobre gráficos. Cuando se hacen gráficos básicos este paquete es suficiente para trabajar. Si se trabaja más seriamente con gráficos, harán falta otros paquetes como puede ser *ggplot2*, que es muy utilizado.

El paquete *stats* es el primer paquete un poco más amplio pero que viene en el paquete básico sobre estadística que permite hacer análisis estadísticos aún más complicados. Dentro de este paquete se encuentra la función *median*. La función *mean* no aparece en este paquete y se encuentra en el paquete *base* ya que es más conocida y más utilizada.

El paquete *methods* tiene que ver con la programación orientada a objetos ya que R intenta hacer un entorno de lenguaje de programación orientada a objetos y en este paquete se encuentran las herramientas para poder enfocarlo desde esta perspectiva.

Estos son los siete paquetes que se cargan por defecto, el paquete *base* y otros seis. Además de estos, existen los paquetes que están en la librería estándar de datos.

La librería estándar es la librería que desde la Fundación R consideran que es el conjunto de paquetes más importante, más utilizado, que más habitualmente se maneja.

La librería estándar se carga por defecto cuando se carga R para poder trabajar más fácilmente con ella. Para consultar los paquetes que se encuentran en la librería estándar se usa *library()* sin ningún argumento. Ello proporciona un listado de los paquetes que hay en la librería estándar como se puede ver en la Figura 5.

```
> library()
```

Entre estos veintitres paquetes, se puede encontrar el paquete *base*. Dentro de la librería estándar también aparecen los siete paquetes que se cargan por defecto en R.

Se encuentra además el paquete *boot* que contiene funciones de análisis de datos basadas en *resampling*, conocidos como "métodos de *Bootstrap*".

Está el paquete *class* que contiene funciones de clasificación no supervisada.

El siguiente es *cluster* que ofrece funciones de clasificación no supervisada pero basada en clústeres.

El paquete *codetools* ofrece funciones de análisis de código en R.

Los siguientes son *compiler* que es el compilador de R, *foreign* que es un paquete muy famoso y muy utilizado que permite importar datos de otras bases de datos que no

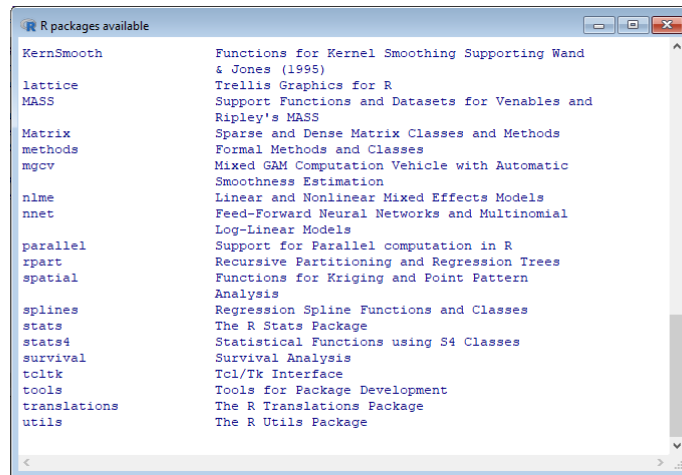


Figure 5: Funciones de la librería estándar.

sean datos básicos, el paquete *grid* tiene funciones para hacer gráficos en cuadrícula y *KernSmooth* que permite funciones de estimadores no paramétricos basados en kernel.

También está *lattice* usado para hacer gráficos. Es un paso más allá del paquete *graphics* aunque el paquete *ggplot2*, que no está en la librería estándar, se usa más que *lattice* en gráficos.

El siguiente es *MASS* que contiene funciones y conjuntos de datos de un libro específico que era el libro básico de S y sobre el que se construyó R. Es un paquete histórico.

Los próximos paquetes son *Matrix* que contiene funciones para controlar matrices, *mgcv* que tiene funciones para tratar modelización realizada con modelos generalizados aditivos (GAM), *nlme* que son modelos lineales y no lineales, *nnet* ofrece funciones para tratar clasificación supervisada y *Feed-Forward Neural Networks* y *parallel* incluye funciones para soportar computación paralela en R, muy útil en *BigData*.

Otros paquetes son *rpart* que contiene funciones igualmente para hacer clasificación supervisada, *spatial* que tiene funciones para realizar análisis basado en patrones de puntos, *splines* que contiene funciones para trabajar con *Regression Spline*, *stats4* con funciones estadísticas utilizando clases de S4 que se usa en estadística pura y *survival* que contiene funciones y bases de datos para tratar estadística aplicada a la salud, investigación y tratamientos.

El siguiente es muy específico, *tcltk*. Tiene interfaces para poder hacer aplicaciones con R, para interfaz web, etcétera.

Finalmente, *tools* para la realización de paquetes, con funciones y herramientas para el desarrollo de paquetes y *translations* que contiene la traducción de los anteriores.

Después de esta pequeña visión general de la librería estándar, hay que saber cómo cargar un paquete de esta librería. Hay dos caminos:

1. Se puede realizar desde el menú Paquetes > Cargar paquete ... y elegir uno de los paquetes que ya está en la librería estándar.
2. Desde la ventana de comandos, usar `library()` con el nombre del paquete que se pretende cargar.

```
> library(foreign)
```

Para consultar qué paquetes hay cargados, se utiliza `search()`.

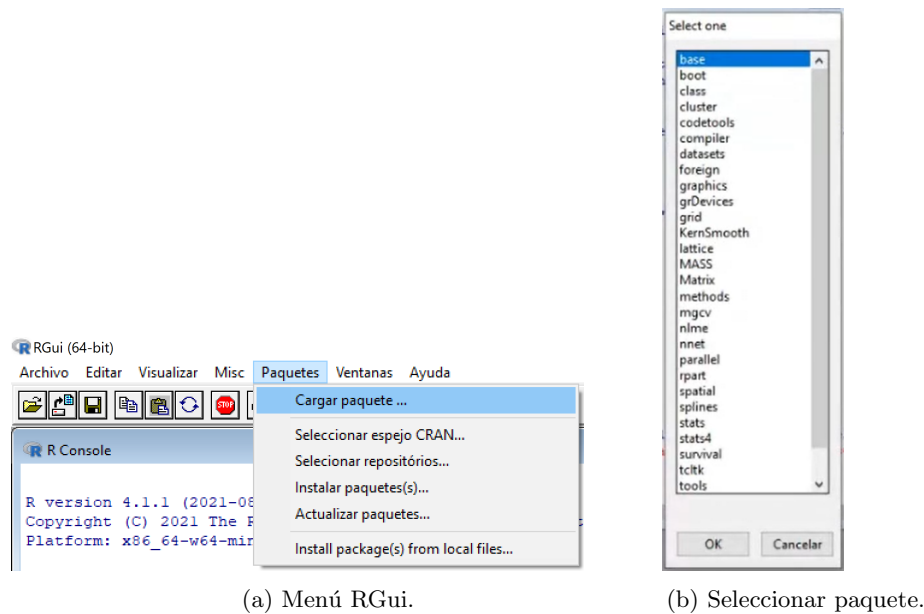


Figure 6

```
> search()
[1] ".GlobalEnv"      "package:foreign"  "package:stats"
[4] "package:graphics" "package:grDevices" "package:utils"
[7] "package:datasets" "package:methods"  "Autoloads"
[10] "package:base"
```

Antes aparecían los paquetes por defecto pero ahora también se ha añadido el paquete cargado. Esta forma de cargar los paquetes solo sirve para los paquetes que ya se encuentran en la librería.

Obviamente, hay muchos más paquetes que se pueden cargar a parte de los 23 que se encuentran ya en la librería estándar. Para cualquier problema que se pretenda resolver se necesita saber cómo cargar paquetes que no se encuentran en la librería estándar. Estos paquetes se encuentran en el CRAN y hay varias formas de realizar esta carga e instalación.

La primera se realiza desde el menú Paquetes de la RGui en la opción de Seleccionar repositorios...

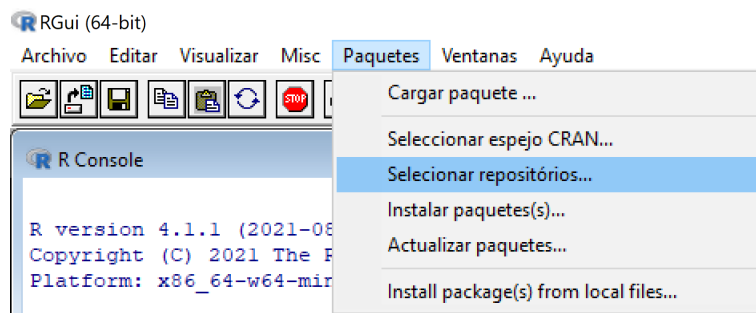


Figure 7

Se elige el repositorio del que se quiere obtener los paquetes. Las opciones que aparecen son CRAN, BioC software, BioC annotation, BioC experiment, CRAN (extras), R-Forge y rforge.net.

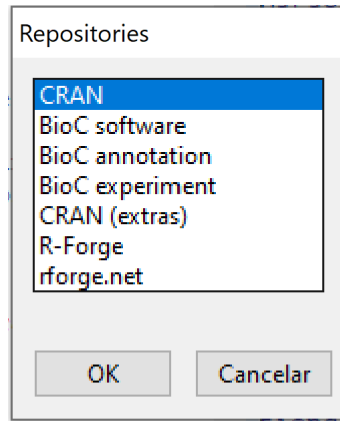
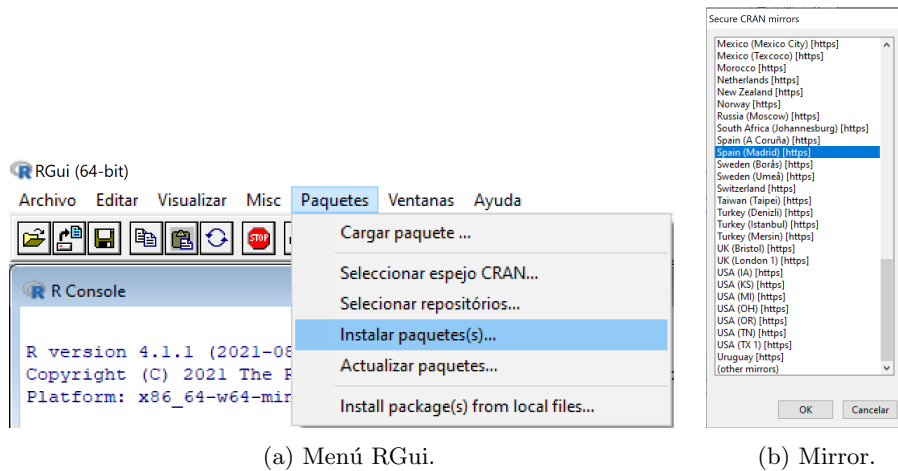


Figure 8

En este caso, para instalar paquetes se usará el repositorio de CRAN. A continuación, en el mismo menú de antes se selecciona la opción Instalar paquete(s)... y se abre una ventana que es un *mirror* donde se elige de dónde se quiere instalar los paquetes. Se elige *Spain (Madrid) [https]* como se muestra en la Figura 9.



(a) Menú RGui.

(b) Mirror.

Figure 9

Luego, aparece otra ventana donde aparecen los paquetes que están en el CRAN a través del mirror seleccionado. Se elige el paquete deseado. Se puede observar los paquetes que se ofrecen en la Figura 10.

Una vez elegido el paquete, éste no se instala en la librería estándar, por lo tanto, puede aparecer una ventana donde se pregunta si se acepta que el paquete se instale en una librería personal y se muestra el directorio de esa librería. Finalmente, se instala el paquete y sus dependencias, es decir, otros paquetes que necesite para funcionar. Las dependencias se suelen instalar automáticamente con el paquete. Entonces, al entrar

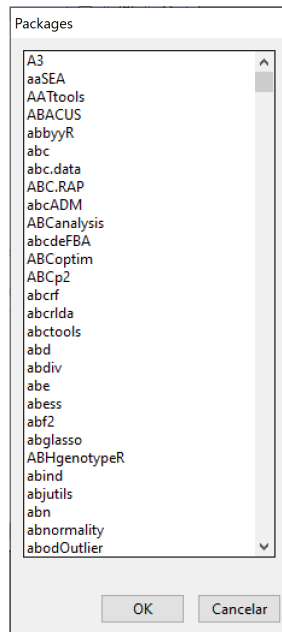


Figure 10

al directorio de instalación, además del paquete que se ha querido instalar aparecerán más paquetes correspondientes a sus dependencias.

Cuando se utilizaban los paquetes de la librería estándar, había que cargarlos antes de usarlos. Hasta ahora solo se ha visto la instalación de un paquete, es decir, descargarlo a un directorio local pero no se encuentra en la sesión de la RGui que se encuentra funcionando. Para cargar el paquete se escribe *library()* con el paquete que se desea cargar. Se comprueba con *search()* que se ha cargado correctamente.

```
> library(LearningRlab)
Loading required package: magick
Linking to ImageMagick 6.9.12.3
Enabled features: cairo, freetype, fftw, ghostscript, heic, lcms, pango,
raw, rsvg, webp
Disabled features: fontconfig, x11
Loading required package: crayon

> search()
[1] ".GlobalEnv"          "package:LearningRlab" "package:crayon"
[4] "package:magick"       "package:foreign"      "package:stats"
[7] "package:graphics"    "package:grDevices"    "package:utils"
[10] "package:datasets"    "package:methods"      "Autoloads"
[13] "package:base"
```

Otra manera de instalar paquetes, es usando la función *install.packages()* con el argumento del paquete que se desea instalar entre comillas, como se observa a continuación:

```
> install.packages("LearningRlab")
```

Otra forma de realizar lo mismo que anteriormente, pero con más control sobre lo que ocurre, es ir a la página web del paquete y descargarlo desde ahí.

Se accede a <https://cran.r-project.org/>, se pulsa *Packages* en el menú situado a la parte izquierda y se busca el paquete deseado ya sea ordenado por fecha de publicación (*Table of available packages, sorted by date of publication*) u ordenado por nombre (*Table of available packages, sorted by name*). Estos pasos se pueden observar en la Figura 11 y 12.

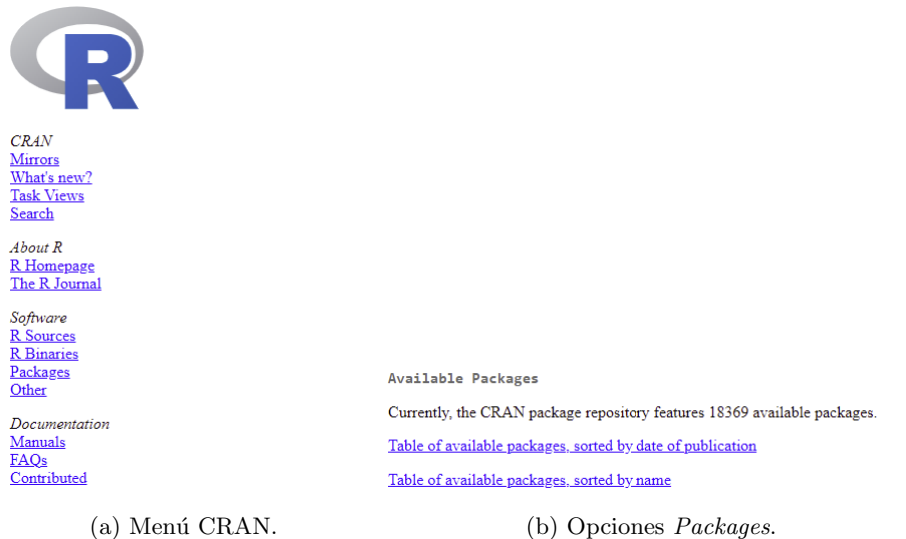


Figure 11

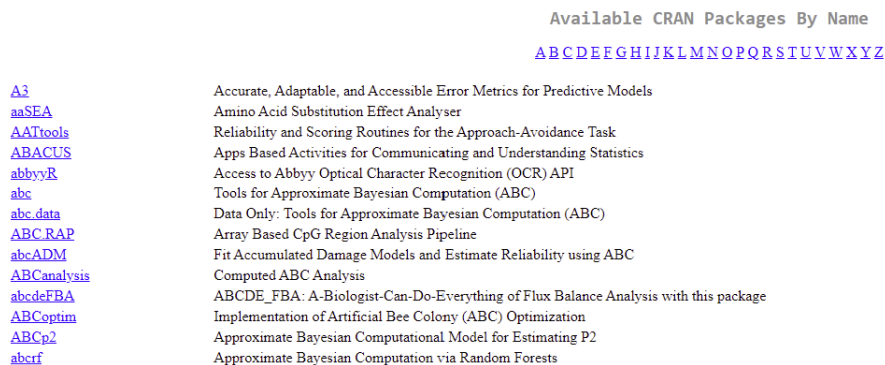


Figure 12: Paquetes ordenados por nombre.

En la página web del paquete debe aparecer toda la información relacionada con él, es decir, la página es el paquete en sí. En esta página está compuesta por la siguiente información (Figura 13):

- **Version:** la última versión que hay del paquete.
- **Depends:** versión de R más reciente que necesita para funcionar.
- **Imports:** otros paquetes que se deben descargar para que el paquete funcione.
- **Suggests:** paquetes que se sugiere su carga para sacar más potencia del paquete.
- **Enhances:** otros paquetes que sirven para mejorar el paquete.

- **Published:** fecha de publicación de la última versión.
- **Author:** autores, personas que lo hacen o mejoran.
- **Maintainer:** persona que tiene el control sobre el paquete.
- **BugReports:** para reforzar en caso de que se encuentren fallos.
- **License:** licencia, ya que es un software libre.
- **URL:** página web del paquete.
- **NeedsCompilation:** si necesita compilación.
- **Citation:** como se tiene que citar.
- **Materials:** documentación que se puede encontrar sobre el paquete.
- **In views:** más información sobre el paquete, agrupada por temas.
- **CRAN checks:** pruebas del paquete.

```
ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics

A system for 'declaratively' creating graphics, based on "The Grammar of Graphics". You provide the data, tell 'ggplot2' how to map variables to
aesthetics, what graphical primitives to use, and it takes care of the details.

Version:      3.3.5
Depends:      R (≥ 3.3)
Imports:      digest, glue, grDevices, grid, gtable (≥ 0.1.1), isoband, MASS, mgcv, rlang (≥ 0.4.10), scales (≥ 0.5.0), stats, tibble, withr (≥ 2.0.0)
Suggests:     covr, ragg, dplyr, ggplot2movies, hexbin, Hmisc, interp, knitr, lattice, mapproj, maps, maptools, multcomp, munsell, nlme, profvis,
quantreg, RColorBrewer, rgeos, rmarkdown, rpart, sf (≥ 0.7-3), syglite (≥ 1.2.0.9001), testthat (≥ 2.1.0), vdiff (≥ 1.0.0), xml2
Enhances:     gr
Published:    2021-06-25
Author:       Hadley Wickham [aut], Winston Chang [aut], Lionel Henry [aut], Thomas Lin Pedersen [aut, cre], Kohske Takahashi
              [aut], Claus Wilke [aut], Kara Woo [aut], Hiroaki Yutani [aut], Dewey Dunnington [aut], RStudio [cph, fnd]
Maintainer:   Thomas Lin Pedersen <thomas.pedersen@rstudio.com>
BugReports:   https://github.com/tidyverse/ggplot2/issues
License:      MIT + file LICENSE
URL:          https://ggplot2.tidyverse.org, https://github.com/tidyverse/ggplot2
NeedsCompilation: no
Citation:     ggplot2 citation info
Materials:    README NEWS
In views:     Graphics, Phylogenetics, TeachingStatistics
CRAN checks:  ggplot2 results
```

Figure 13: Ejemplo de paquete de CRAN (*ggplot2*).

Lo siguiente que aparece en la página web del paquete en CRAN es lo que alguien se puede descargar del paquete. Se divide en la siguiente información (Figura 14):

- **Reference manual:** manual de referencia donde aparecen todas las funciones.
- **Vignettes:** viñetas. Son pequeños manuales de cómo funciona el paquete.
- **Package source:** código fuente del paquete.
- **Windows binaries:** paquete para instalar en Windows.
- **macOS binaries:** paquete para instalar en macOS.
- **Old sources:** antiguas descargas del paquete.
- **Reverse depends:** paquetes que tienen a este paquete como dependencia de uso.
- **Reverse suggests:** paquetes que tienen a este paquete como dependencia de sugerencia.

Para la tercera vía de instalación de un paquete, la vía manual, hay que descargar lo que se necesita para instalar el paquete. Es recomendable descargarse el manual y las viñetas para poder consultar cualquier información de interés sobre ese paquete cuando se necesite. Luego se descarga el programa del apartado *r-release* del sistema operativo correspondiente. Una vez descargado, ya se encuentra el paquete guardado

Documentation:

Reference manual: [ggplot2.pdf](#)

Vignettes: [Extending ggplot2](#)
[Using ggplot2 in packages](#)
[Aesthetic specifications](#)

Downloads:

Package source: [ggplot2_3.3.5.tar.gz](#)

Windows binaries: r-devel: [ggplot2_3.3.5.zip](#), r-release: [ggplot2_3.3.5.zip](#), r-oldrel: [ggplot2_3.3.5.zip](#)

macOS binaries: r-release (arm64): [ggplot2_3.3.5.tgz](#), r-release (x86_64): [ggplot2_3.3.5.tgz](#), r-oldrel: [ggplot2_3.3.5.tgz](#)

Old sources: [ggplot2 archive](#)

Figure 14: Ejemplo de paquete de CRAN (*ggplot2*).

de forma local, pero no instalado. Para instalarlo, se usa, como anteriormente, la función `install.packages()` pero ahora añadiendo la ruta del archivo comprimido que se ha descargado previamente. También hay que añadir a los argumentos la instrucción de que no busque ese paquete en ningún repositorio; esto se hace añadiendo `repos=NULL` despues de la ruta.

```
>install.packages("C:/tmp/ggplot2_3.3.5.zip", repos=NULL)
Installing package into 'C:/Users/Alumno/Documents/R/win-library/4.0'
(as 'lib' is unspecified)
package 'ggplot2' successfully unpacked and MD5 sums checked
```

El paquete se instala en la librería personal. Se carga el paquete en la ejecución de RGui que se esté usando con `library()`.

```
> library(ggplot2)
Error: package or namespace load failed for 'ggplot2' in loadNamespace(i,
  c(lib.loc, .libPaths()), versionCheck = vI[[i]]):
  there is no package called 'gtable'
```

Al intentar cargar el paquete, pueden salir errores debido a que no están cargados los paquetes con los que tiene dependencias. Como ahora se ha elegido la forma manual de instalación de paquetes, no se han instalado por defecto sus dependencias y hay que instalar estos paquetes uno a uno de cualquiera de las 3 formas ya explicadas. Finalmente, una vez instaladas todas las dependencias, se cargará el paquete como en los casos anteriores.

```
> library(ggplot2)
```

Attaching package: 'ggplot2'

The following object is masked from 'package:crayon':

```
%+%
```

Por último, se debe saber que se pueden añadir paquetes para que se carguen por defecto cada vez que se abra una sesión nueva de RGui. Se debe recordar que para ver los paquetes que se cargan por defecto, se usaba el comando `getOption()` con el argumento `defaultPackages`. Lo que se pretende conseguir es que, al ejecutar este comando, aparezcan más paquetes cargados por defecto directamente.

```
> getOption("defaultPackages")
[1] "datasets" "utils" "grDevices" "graphics" "stats" "methods"
```

Para conseguir esta meta, hay que modificar el *profile* de R. Este fichero se encuentra en la carpeta de instalación de la versión de R que se esté utilizando. A la hora de la instalación de R puede haberse elegido otra ruta pero, por defecto, en la

versión 4.1.1 de R bajo *Windows* suele encontrarse en la ruta "C:/Program Files/R/R-4.1.1/library/base/R". En la carpeta *library* es donde está la librería estándar de R con los paquetes que lo hacen funcionar. Ahí aparece un listado de 23 carpetas con los paquetes que se tienen en la biblioteca estándar de R. Se accede a la carpeta correspondiente al paquete *base* y a la subcarpeta llamada *R*. Aquí ya se encuentra el fichero *Rprofile* donde está el perfil de R que se está utilizando y donde están las instrucciones de carga. Este fichero es el que va a ser modificado pero en la carpeta donde se encuentra no se van a poder guardar los cambios porque no se poseen los permisos necesarios para hacerlo, por lo tanto, se copia este fichero y se pega en otra localización donde sí se pueda modificar y guardar. Una vez allí, se abre el fichero con cualquier bloc de notas y ahí se encuentran todas las instrucciones que contiene *Rprofile*. El siguiente paso es localizar la zona del código donde aparecen los paquetes que se cargan por defecto. El código que se pretende encontrar es el siguiente:

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

En la tercera línea de esta zona de código, aparece la variable *dp*, con la sintaxis de R, que es un conjunto de datos correspondiente a los paquetes que se cargan por defecto al principio. Entonces, en esta variable, se añade el paquete deseado que tiene que estar previamente instalado. Por ejemplo, se añadirá el paquete *foreign* para que se cargue por defecto que, al pertenecer a la librería estándar, ya se encuentra instalado. Se añaden los paquetes deseados entre comillas y separados por comas. Una vez añadido el nuevo paquete, el código queda así:

```
local({dp <- Sys.getenv("R_DEFAULT_PACKAGES")
  if(identical(dp, "")) ## it fact methods is done first
    dp <- c("datasets", "utils", "grDevices", "graphics",
           "stats", "methods", "foreign")
  else if(identical(dp, "NULL")) dp <- character(0)
  else dp <- strsplit(dp, ",")[1]
  dp <- sub("[[:blank:]]*([[:alnum:]]+)", "\\1", dp)
  options(defaultPackages = dp)
})
```

Se guarda el fichero con la modificación, se cierra y se vuelve a copiar y pegar en el directorio de origen, reemplazando el anterior fichero por este nuevo. Luego, hay que ejecutar una nueva sesión de R para que los cambios se vean reflejados en la RGui. Para comprobar los cambios, se ejecuta otra vez la instrucción *getOption()* y se observa como aparece el nuevo paquete que se ha cargado por defecto.

```
> getOption("defaultPackages")
[1] "datasets" "utils" "grDevices" "graphics" "stats" "methods"
[7] "foreign"
```

Ahora se ha cargado por defecto el paquete *foreign*. Siempre que se ejecute ahora R, se carga *foreign* además de los que cargaba hasta este momento por defecto. Por lo tanto, modificando el fichero *Rprofile*, se puede definir qué paquetes se desea que se carguen por defecto al inicio de cualquier ejecución, tanto añadir como eliminar.

2.2.2 Aplicación del algoritmo Apriori

Una vez visto la instalación de paquetes en R, se explica la teoría del algoritmo Apriori y se aplica usando el paquete *arules* en R.

El algoritmo Apriori sigue un proceso que se divide en dos pasos:

1. Identificación de las asociaciones frecuentes con el cálculo del Soporte. Cuando el algoritmo calcule el soporte se aprovecha que la medida es antimonótona, es decir, que si un suceso es frecuente y su soporte supera el umbral, todos los subconjuntos de dicho conjunto son también frecuentes.
2. Identificación de las asociaciones de confianza con el cálculo de la Confianza. El algoritmo usa una función llamada *ap-genrules*, que se basa en el siguiente teorema: si se tiene dos conjuntos A y B, si la asociación $A \rightarrow B - A$ no supera el umbral de confianza, entonces cualquier asociación $A' \rightarrow B - A'$, donde A' es cualquier subconjunto de A tampoco supera el umbral de confianza.

A continuación, y ya visto la parte de teoría del algoritmo, se carga el paquete *arules* usando la función *library()*, que como argumento se le pasa el nombre de un paquete, ya sea dando el nombre directamente o como cadena de caracteres.

```
> library(arules)
Loading required package: Matrix
```

```
Attaching package: 'arules'
```

The following objects are masked from 'package:base':

```
abbreviate, write
```

Si se observa, cuando se carga *arules*, también se avisa en la primera línea de la salida que se carga el paquete requerido *Matrix*.

Mediante *search()* se comprueba si los paquetes *arules* y *Matrix* se han instalado de forma correcta.

```
> search()
[1] ".GlobalEnv"          "package:arules"      "package:Matrix"
[4] "package:ggplot2"      "package:LearningRlab" "package:crayon"
[7] "package:magick"       "package:foreign"     "package:stats"
[10] "package:graphics"    "package:grDevices"   "package:utils"
[13] "package:datasets"    "package:methods"     "Autoloads"
[16] "package:base"
```

Ahora ya se puede empezar a resolver el ejercicio. Primeramente se introduce los valores de los sucesos de la muestra para trabajar con ello en R. Esto se puede hacer usando una matriz con valores binarios. Como se puede ver en la Tabla 1 en cada columna se representa los valores de los diferentes sucesos y en cada fila los valores de un suceso determinado, siendo un 1, si se da el suceso elemental, o en caso contrario, un 0, si no se da.

Se introduce esta matriz en R usando la función *Matrix()* de la siguiente manera:

```
> 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", "Cafe", "Leche", "Naranjas")), sparse=TRUE)
> muestra
6 x 5 sparse Matrix of class "dgCMatrix"
      Pan Agua Cafe Leche Naranjas
suceso1 1    1    .    1    1
```

	Pan	Agua	Café	Leche	Naranjas
Suceso 1	1	1	0	1	1
Suceso 2	1	1	1	1	0
Suceso 3	1	1	0	1	0
Suceso 4	1	0	1	1	0
Suceso 5	1	1	0	0	0
Suceso 6	0	0	0	1	0

Table 1: Matriz correspondiente a la muestra observada

```

suceso2  1    1    1    1    .
suceso3  1    1    .    1    .
suceso4  1    .    1    1    .
suceso5  1    1    .    .    .
suceso6  .    .    .    1    .

```

Se almacena la matriz en la variable *muestra* y en la función se introduce como valores una matriz multidimensional usando la función *c()* insertando las filas y dejando un espacio para que visualmente se vea las distintas filas. Después se le indica que la matriz tiene 6 filas y 5 columnas. El *byrow=TRUE* se indica para rellenar la matriz por filas, en caso contrario, se rellenaría por columnas. A continuación, se inserta el *dimnames* para ponerle los nombres, siendo una lista que se crea con el uso de *list()* que esta formada por dos vectores, siendo el primero los sucesos y el segundo son las cosas que se encuentran en la cesta de la compra. Todos estos valores se ponen entre comillas *"* al tratarse de valores de tipo carácter. Por último, se inserta *sparse=TRUE* porque se busca que la matriz sea del tipo *sparse*, ya que *arules* trabaja con una matriz dispersa de posiciones no cero, por lo que es necesario este atributo para luego convertir la matriz muestra a una *ngCMatrix* como se muestra a continuación usando la función *as()*.

```

> muestrangCMatrix<-as (muestra, "nsparseMatrix")
> muestrangCMatrix
6 x 5 sparse Matrix of class "ngCMatrix"
      Pan Agua Cafe Leche Naranjas
sucoso1 |   |   .   |   |
sucoso2 |   |   |   |   |
sucoso3 |   |   .   |   |
sucoso4 |   .   |   |   |
sucoso5 |   |   .   .   |
sucoso6 .   .   .   |   |

```

Como se puede ver en el resultado, se trata de una matriz con rayas y puntos. Para esta función se ha pasado como argumentos el objeto R de muestra y la clase que se quiere convertir.

Ahora se debe transponer la matriz *muestrangCMatrix* usando la función *t()* para analizar los conjuntos de forma correcta.

```

> transpmuestrangCMatrix<-t(muestrangCMatrix)
> transpmuestrangCMatrix
5 x 6 sparse Matrix of class "ngCMatrix"
      suceso1 suceso2 suceso3 suceso4 suceso5 suceso6
Pan          |       |       |       |       |
Agua          |       |       .       |       |
Cafe          .       |       .       |       .

```



```

Leche      |      |      |      |      .      |
Naranjas   |      .      .      .      .      .

```

Quedando los sucesos como columnas y las cosas de la cesta en las filas. De tal forma que ya se puede determinar las posibles transacciones usando la función *as()* con el atributo *transactions* aplicado a la variable *transpmuestrangCMatrix*:

```

> transacciones<-as(transpmuestrangCMatrix, "transactions")
> transacciones
transactions in sparse format with
  6 transactions (rows) and
  5 items (columns)

```

Lo que devuelve es que tiene 6 transacciones (sucesos) de 5 eventos. Si se quiere, se puede hacer un resumen de las transacciones usando la función *summary()*.

```

> 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	Cafe	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   Cafe

```

includes extended transaction information - examples:

```

itemsetID
1   suceso1
2   suceso2
3   suceso3

```

Se indica que hay 6 filas y 5 columnas, los objetos más frecuentes que es algo necesario cuando se hace el cálculo de Soporte y, a continuación, más información de las transacciones.

Ahora se busca las asociaciones con el algoritmo a priori mediante la función *apriori* del paquete *arules* aplicándolo a la variable *transacciones* y poniendo los parámetros que necesita: una lista con el soporte de 0.5 y la confianza de 0.8.

```

> 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
  10 rules
ext
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].

```

Al realizar el análisis no queda tan claro qué ocurre, para ello se usa otra función, la *inspect()* para obtener un resultado más claro aplicado a la variable *asociaciones*:

```

> inspect(asociaciones)
      lhs      rhs      support  confidence coverage lift count
[1] {}      => {Leche} 0.83333333 0.83333333 1.0000000 1.00 5
[2] {}      => {Pan}   0.83333333 0.83333333 1.0000000 1.00 5
[3] {Agua}   => {Pan}   0.66666667 1.00000000 0.66666667 1.20 4
[4] {Pan}    => {Agua}   0.66666667 0.80000000 0.83333333 1.20 4
[5] {Leche}  => {Pan}   0.66666667 0.80000000 0.83333333 0.96 4
[6] {Pan}    => {Leche} 0.66666667 0.80000000 0.83333333 0.96 4
[7] {Agua,Leche} => {Pan} 0.50000000 1.00000000 0.50000000 1.20 3

```

Y ya se obtiene las asociaciones con su soporte y confianza que serían: *Agua* → *Pan*, *Pan* → *Agua*, *Leche* → *Pan*, *Pan* → *Leche* y *{Agua, Leche}* → *Pan*. Revisando la parte vista en clase de teoría se dan los mismos resultados.

3 Segunda parte

3.1 Análisis de descripción de datos

3.1.1 Formato del fichero de datos en CSV

El fichero de texto que se leerá será *distancias.csv* y se creará de la siguiente manera:

El archivo contiene una primera fila, la cabecera, en la que se establecen el nombre de los datos que se van a mostrar en cada columna. En este caso la primera columna será “Distancia”, una característica cuantitativa, que designa la distancia desde el domicilio de cada estudiantes hasta la Universidad, es decir, datos cuantitativos continuos. Las demás filas representarán a cada una de las distancias con datos numéricos. En el caso de que se desee añadir más datos desde el mismo archivo csv, se añaden más columnas. La separación de cada una de las columnas se hace mediante las comas [4].

Para una mejor visualización, en la Figura 15b se puede observar el uso de las comas. El fichero que se utilizará a lo largo de la *Segunda Parte* se muestra en la Figura 15a.

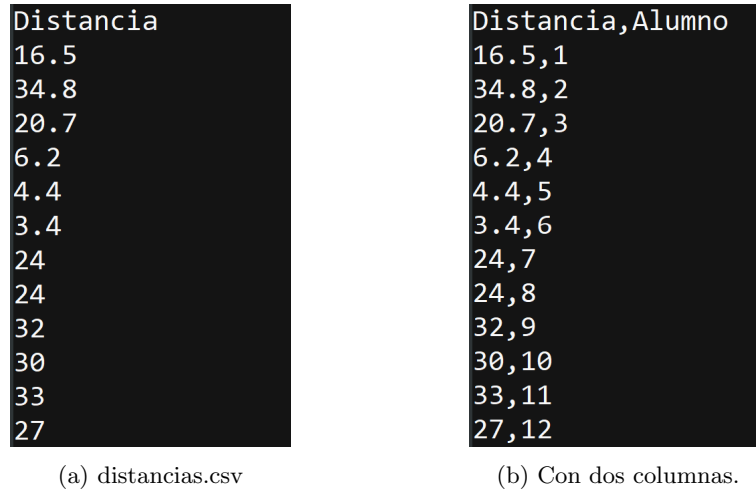


Figure 15

3.1.2 Lectura de un fichero de datos en formato CSV

Para poder leer un fichero en formato CSV y transformarlo en un *data frame* R dispone de dos funciones llamadas *read.csv* y *read.csv2* [7]. Ambas funciones comparten los argumentos más importantes que utilizaba *read.table*; éstos eran *file* que indica el nombre del archivo CSV o de la ruta absoluta a éste entre comillas dobles " " o simples ' ' teniendo en cuenta que las barras de la ruta absoluta pueden ser inclinadas / o dos invertidas \\\; *header* que es un valor lógico que indica si la primera fila del archivo contiene los nombres de las variables; *sep* que indica el caracter separador de las columnas; y *dec* que hace referencia al caracter usado para los puntos decimales. Otros argumentos que se pueden destacar de las dos funciones son *quote* para los caracteres que hagan referencia a citas; *fill* para indicar si las filas tienen distinta longitud pues rellenarlas con espacios en blanco para que todas tengan la misma longitud; y *comment.char* para indicar el caracter usado para los comentarios.

La principal diferencia entre *read.csv* y *read.csv2* son los argumentos por defecto, ya que la primera función utiliza como separador la coma , y decimal el punto .; y la segunda función utiliza como separador el punto y coma ; y decimal la coma ,.

A continuación se guardará en la variable *d* el *data frame* producto de la lectura del CSV de las distancias con *read.csv* y con los argumentos de *file* y *header*.

```
> d <- read.csv("distancias.csv", header = TRUE)
> d
  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 el *data frame* está bien estructurado con la primera columna que indica el número de fila y la segunda los datos numéricos de las distancias.

3.1.3 Cálculos con matrices

El *data frame* creado anteriormente con la función *read.csv()* y guardado en la variable *d* está listo para poder operar con las distintas funciones que ofrece R para matrices, *arrays* y *data frames* pero en este caso se crearán manualmente en *scripts* independientes.

Como ejemplo se crearán y se trabajará con las mismas funciones que en la primera parte, estas son *length()*, *dim()*, *order()* y *rev()*.

La función *length()* se creará de la siguiente manera:

```

longitud <- function(v) {
  if(esta_vacio(unname(v))) return(0)
  else return(1 + longitud(v[-c(1)]))
}

```

La manera más eficiente que se ha visto es mediante una función recursiva que vaya vaciando el vector e incrementado en uno por cada elemento que tenga hasta que se quede completamente vacío y devolver la suma incremental que se ha estado haciendo o 0 en caso de que el vector inicialmente ya estuviera vacío.

Para comprobar si el vector está vacío se crea la función *esta_vacio()* de la siguiente manera:

```

esta_vacio <- function(x) {
  return(identical(x,numeric(0)) ||
         identical(x,integer(0)) ||
         identical(x,character(0)) ||
         identical(x,complex(0)) ||
         identical(x,logical(0)) ||
         is.null(x))
}

```

En esta función se comprueba si el vector o lista pasado como argumento está vacío comparándolo con todos los tipos de datos vacíos y con el objeto *NULL*. Un ejemplo de tipo de dato vacío sería el carácter vacío representado por *character(0)*.

Tras cargar dicha función junto con la de *longitud()* ya se podrá comprobar la longitud de un vector o lista. Para ello se probará con los datos las distancias usando la nueva función y la de R para ver que la salida coincide.

```

> longitud(d$Distancia)
[1] 73
> length(d$Distancia)
[1] 73

```

La función *dim()* se creará de la siguiente manera:

```
dimension = function(x) {
  return (c(nrow(x),ncol(x)))
}
```

dimension() recibirá como argumento un vector, *array* o *data frame* y devolverá un vector con el número de filas y el número de columnas del argumento gracias a las funciones *nrow()* y *ncol()*.

Se carga en R la función creada y se utiliza con el *data frame* de las distancias. También se utiliza la función de R para ver que los resultados coinciden:

```
> dimension(d)
[1] 73  1
> dim(d)
[1] 73  1
```

Para la función *order()* se la reduce de tal manera que siga un solo método de ordenación, el algoritmo de ordenación de burbuja, que funcione para vectores unidimensionales y listas, y que el orden sea de menor a mayor.

Para ello la función *ordenar()* es definida tal como se muestra a continuación:

```
ordenar <- function(x){
  idx <- 1:longitud(x)
  for(i in 1:(longitud(x)-1))
  {

    for(j in 1:(longitud(x)-1))
    {
      if(x[j+1] <= x[j]){
        aux <- x[j+1]
        x[j+1] <- x[j]
        x[j] <- aux

        auxIdx <- idx[j+1]
        idx[j+1] <- idx[j]
        idx[j] <- auxIdx
      }
    }
  }
  return(idx)
}
```

La función recibe un solo argumento que es un vector o una lista, implementa el algoritmo de ordenación de burbuja mediante dos bucles *for* y devuelve un nuevo vector con los índices de los elementos ordenados de menor a mayor.

Con los datos de las distancias se puede aplicar la función de tal manera comparándola con la que ya viene por defecto en R:

```
> do <- d[ordenar(d$Distancia),]
> do
  [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 30.0 31.4 32.0 33.0 33.0 34.0 34.0 34.8 38.0 46.0
> do <- d[order(d$Distancia),]
> do
  [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 30.0 31.4 32.0 33.0 33.0 34.0 34.0 34.8 38.0 46.0

```

La función *rev()* se creará de la siguiente manera:

```

invertir <- function(x) {
  if(longitud(x) == 0) return(x)
  else return(c(invertir(x[-c(1)]),x[1]))
}

```

Se realizan llamadas recursivas que van vaciando el vector del argumento de entrada y se van añadiendo a uno nuevo empezando por el último elemento hasta el primero.

Aplicando la función creada y la función disponible en R a un vector de enteros quedaría de tal manera:

```

> v = 1:12
> invertir(v)
[1] 12 11 10 9 8 7 6 5 4 3 2 1
> rev(v)
[1] 12 11 10 9 8 7 6 5 4 3 2 1

```

Aplicando la función sobre los datos ordenados de las distancias el resultado quedaría de la siguiente forma:

```

> dor <- d[invertir(ordenar(d$Distancia)), ]
> dor
  [1] 46.0 38.0 34.8 34.0 34.0 33.0 33.0 32.0 31.4 30.0 30.0 30.0 30.0
    30.0 30.0
  [16] 30.0 30.0 29.0 28.0 28.0 27.0 27.0 27.0 26.0 26.0 25.0 25.0 24.1
    24.0 24.0
  [31] 24.0 24.0 22.0 21.6 21.0 20.7 20.0 19.0 19.0 17.2 16.5 16.0 15.0
    13.0 12.0
  [46] 12.0 12.0 12.0 12.0 11.0 10.0 9.7 9.4 9.0 8.1 6.2 5.5 5.1
    5.0 4.5
  [61] 4.5 4.4 4.4 4.0 4.0 3.7 3.7 3.4 3.2 3.1 2.7 2.1 1.0
> dor <- d[rev(order(d$Distancia)), ]
> dor
  [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

```

3.1.4 Funciones en R

En el apartado *Funciones de R* de la primera parte de la práctica, se explicó tres funciones de R: *function()*, *dump()*, *source()* y la relación que había entre ellas.

Ahora, utilizando otras funciones, se quiere lograr el mismo resultado. Se explicará que hacen esas funciones, sus principales argumentos y un ejemplo de ejecución.

La función *rango2* se ha creado utilizando el paquete *lambda.r* [8] para evitar emplear *function()*. *lambda.r* es una extensión de lenguaje que admite una programación funcional en R, ofrece una sintaxis funcional para definir tipos y funciones.

Se ejecuta la función *range()* para saber el valor máximo y el mínimo de las distancias:

```
> range(d$Distancia)
[1] 1 46
```

Creación de una función lambda: Importar la librería *lambda.r*, dar nombre a la función acompañado de los argumentos que se necesiten. Las funciones se definen mediante la palabra reservada *%as%* o *%:=%* (símbolo en lugar de *;-*). Entre llaves *{}*, se especifica el cuerpo de la función que en este caso es la resta de la distancia máxima (devuelta por *maximo()*) y la mínima (devuelta por *minimo()*). Por último, para saber el rango, llamar a *rango2* pasándole el vector de distancias.

```
> library(lambda.r)
> rango2(x) %as% {maximo(x)-minimo(x)}
> rango2(d$Distancia)
[1] 45
```

La función *save()* escribe una representación externa de objetos R en el archivo especificado. Esta función tiene un comportamiento similar al *dump()*. Los argumentos más significativos son los siguientes: *list*: un vector de caracteres que contiene los nombres de los objetos que se guardarán; *file*: el nombre del archivo donde se guardarán los datos; *ascii*: si es *TRUE*, se escribe una representación ASCII de los datos. El valor predeterminado de *ascii* es *FALSO*, lo que conduce a la escritura de un archivo binario; *envir*: entorno para buscar objetos para guardar; *compress*: permite comprimir el archivo usando las extensiones *gzip*, *bzip2* o *xz*.

Ejemplo de ejecución usando *save()*:

```
> save(rango2, file="rango2.R")
```

La función *load()* carga los datos escritos con la función *save()*. Esta función tiene un comportamiento similar al *source()*. Los argumentos más significativos son los siguientes: *file*: el nombre del archivo a cargar; *envir*: el entorno donde se deben cargar los datos.

Ejemplo de ejecución *load()*:

```
> library(lambda.r)
> load("rango2.R")
> rango2(d$Distancia)
[1] 45
```

Importante: Se tiene que llamar a la librería *lambda.r* para poder usar la función *rango2*.

Si se desea eliminar el fichero desde R, se le pasa el nombre o la ruta de este a la función *unlink()* [7]:

```
> unlink("rango2.R")
```

3.1.5 Primer análisis de los datos

Para este primer análisis se han programado todas las funciones de frecuencias y, además, se van a explicar uno a uno. En la primera, la frecuencia absoluta, se trata de una función cuyo parámetro de entrada *x* que recibe debe ser *numeric*.


```

freq_abs <- function(x)
{
  if (!is.numeric(x))
  {
    stop("x is not a numeric")
  }

  data <- sort(unique(x))
  counter <- vector(mode="numeric", length=0)
  for (value in data)
  {
    total <- longitud(x[x == value])
    counter <- c(counter, total)
  }

  setNames(counter, data)
}

```

Esto se comprueba al principio de la función usando *is.numeric(x)*, y se da un mensaje de error si ocurre lo contrario con *stop()*. Después la variable *x* es aplicada a la funciones *unique()* [7] de forma que solo existan números sin duplicaciones y, por último, *sort()* para tener los datos ordenados de menor a mayor, almacenando el resultado de aplicar estas dos funciones a la variable *data*. Seguidamente se crea un vector de modo numérico y tamaño cero en la variable *counter* usando la función *vector()*. Para finalizar, y siendo la parte más importante, se trata de realizar un bucle *for* con la variable *data* (que se recuerda que son datos sin repeticiones) para obtener los valores y saber cuáles pertenecen a *x* usando *x[x == value]* cuya longitud es el total que se almacena en el vector *counter*. Se establece los nombres y se devuelve el resultado.

Si se usa la función con los datos de distancia, se obtiene el siguiente resultado:

```

> freqabs <- freq_abs(d$Distancia)
> freqabs
  1  2.1  2.7  3.1  3.2  3.4  3.7   4  4.4  4.5   5  5.1  5.5  6.2  8.1   9
  1  1  1  1  1  1  2   2  2  2   1  1  1  1  1  1
9.4 9.7 10 11 12 13 15 16 16.5 17.2 19 20 20.7 21 21.6 22
  1  1  1  1  5  1  1  1  1  1  2  1  1  1  1  1
24 24.1 25 26 27 28 29 30 31.4 32 33 34 34.8 38 46
  4  1  2  2  3  2  1  8  1  1  2  2  1  1  1

```

Y si se hace con *table()* como en la *Primera parte* se obtiene los mismos resultados.

```

> table(d$Distancia)
  1  2.1  2.7  3.1  3.2  3.4  3.7   4  4.4  4.5   5  5.1  5.5  6.2  8.1   9
  1  1  1  1  1  1  2   2  2  2   1  1  1  1  1  1
9.4 9.7 10 11 12 13 15 16 16.5 17.2 19 20 20.7 21 21.6 22
  1  1  1  1  5  1  1  1  1  1  2  1  1  1  1  1
24 24.1 25 26 27 28 29 30 31.4 32 33 34 34.8 38 46
  4  1  2  2  3  2  1  8  1  1  2  2  1  1  1

```

Para la frecuencia relativa se trata de una división en la que, ya que ha sido programada anteriormente, se llama a la función absoluta *freq_abs()* en el numerador y dejando la longitud en el denominador. Tiene como único argumento *x* que debe ser de tipo *numeric*.

```

freq_rel <- function(x)
{

```

```

    freq_abs(x) / longitud(x)
}

```

Se utiliza con las distancias para obtener la frecuencia relativa y, una vez más, se comprueba los resultados con `table()` y `length()` para asegurar que los resultados son correctos.

```

> freqrel <- freq_rel(d$Distancia)
> freqrel

```

	1	2.1	2.7	3.1	3.2	3.4	3.7
0.01369863	0.01369863	0.01369863	0.01369863	0.01369863	0.01369863	0.01369863	0.02739726
	4	4.4	4.5	5	5.1	5.5	6.2
0.02739726	0.02739726	0.02739726	0.01369863	0.01369863	0.01369863	0.01369863	0.01369863
	8.1	9	9.4	9.7	10	11	12
0.01369863	0.01369863	0.01369863	0.01369863	0.01369863	0.01369863	0.01369863	0.06849315
	13	15	16	16.5	17.2	19	20
0.01369863	0.01369863	0.01369863	0.01369863	0.01369863	0.02739726	0.01369863	0.01369863
	20.7	21	21.6	22	24	24.1	25
0.01369863	0.01369863	0.01369863	0.01369863	0.05479452	0.01369863	0.02739726	0.02739726
	26	27	28	29	30	31.4	32
0.02739726	0.04109589	0.02739726	0.01369863	0.10958904	0.01369863	0.01369863	0.01369863
	33	34	34.8	38	46		
0.02739726	0.02739726	0.01369863	0.01369863	0.01369863			

```

> table(s$Distancia) / length(d$Distancia)

```

	1	2.1	2.7	3.1	3.2	3.4	3.7
0.01369863	0.01369863	0.01369863	0.01369863	0.01369863	0.01369863	0.01369863	0.02739726
	4	4.4	4.5	5	5.1	5.5	6.2
0.02739726	0.02739726	0.02739726	0.01369863	0.01369863	0.01369863	0.01369863	0.01369863
	8.1	9	9.4	9.7	10	11	12
0.01369863	0.01369863	0.01369863	0.01369863	0.01369863	0.01369863	0.01369863	0.06849315
	13	15	16	16.5	17.2	19	20
0.01369863	0.01369863	0.01369863	0.01369863	0.01369863	0.02739726	0.01369863	0.01369863
	20.7	21	21.6	22	24	24.1	25
0.01369863	0.01369863	0.01369863	0.01369863	0.05479452	0.01369863	0.02739726	0.02739726
	26	27	28	29	30	31.4	32
0.02739726	0.04109589	0.02739726	0.01369863	0.10958904	0.01369863	0.01369863	0.01369863
	33	34	34.8	38	46		
0.02739726	0.02739726	0.01369863	0.01369863	0.01369863			

Para la frecuencia acumulada se ha creado una única función que dependiendo de lo que reciba el argumento *type* se realiza la acumulada absoluta o la relativa. Este argumento por defecto tiene el valor "abs" por lo que llamando a la función sin pasarle nada a *type*, realizará la frecuencia acumulada absoluta. En el caso de querer hacer la acumulada relativa, se le pasará "rel". Como en las anteriores, se tiene otro argumento llamado *x* que debe ser del tipo *numeric*.

```

freq_cum <- function(x, type="abs")
{
  f <- list(abs = freq_abs, rel = freq_rel)
  if (type %in% names(f))
  {
    data <- f[[type]](x)
  }
  else

```

```

{
  stop(paste("Unrecognized type:", type))
}

counter <- vector(mode="numeric", length=0)

acumulator <- 0
for (value in 1:longitud(data))
{
  acumulator <- acumulator + data[value]
  counter <- c(counter, acumulator)
}

setNames(counter, sort(unique(x)))
}

```

A partir de una lista que se almacena en la variable *f* se conoce las posibles opciones de *type*, es decir, si se le pasa "*abs*" llamará a la función *freq-abs* o *freq-rel* si es pasado "*rel*". Todo ello se realiza a partir del primer *if* que comprueba si ese tipo está dentro de la lista para posteriormente realizar la llamada a esa función y, en caso contrario, mandar un error con el mensaje de tipo no reconocido. Por último, se realiza el sumatorio de ir acumulando los valores que hay en *data* y se almacena en *counter* que se devuelve estableciendo los nombres.

Al comprobar los resultados, vemos que es correcto, tanto para la absoluta:

```

> freqcumabs <- freq_cum(d$Distancia)
> freqcumabs
  1  2.1  2.7  3.1  3.2  3.4  3.7   4  4.4  4.5   5  5.1  5.5  6.2  8.1   9
  1   2   3   4   5   6   8  10  12  14  15  16  17  18  19  20
9.4  9.7  10  11  12  13  15  16 16.5 17.2  19  20 20.7  21 21.6  22
21  22  23  24  29  30  31  32  33  34  36  37  38  39  40  41
24 24.1  25  26  27  28  29  30 31.4  32  33  34 34.8  38  46
45  46  48  50  53  55  56  64  65  66  68  70  71  72  73

> cumsum(table(d$Distancia))
  1  2.1  2.7  3.1  3.2  3.4  3.7   4  4.4  4.5   5  5.1  5.5  6.2  8.1   9
  1   2   3   4   5   6   8  10  12  14  15  16  17  18  19  20
9.4  9.7  10  11  12  13  15  16 16.5 17.2  19  20 20.7  21 21.6  22
21  22  23  24  29  30  31  32  33  34  36  37  38  39  40  41
24 24.1  25  26  27  28  29  30 31.4  32  33  34 34.8  38  46
45  46  48  50  53  55  56  64  65  66  68  70  71  72  73

```

como para la relativa:

```

> freqcumrel <- freq_cum(d$Distancia, "rel")
> freqcumrel
      1      2.1      2.7      3.1      3.2      3.4      3.7
0.01369863 0.02739726 0.04109589 0.05479452 0.06849315 0.08219178 0.10958904
      4      4.4      4.5      5      5.1      5.5      6.2
0.13698630 0.16438356 0.19178082 0.20547945 0.21917808 0.23287671 0.24657534
      8.1      9      9.4      9.7      10      11      12
0.26027397 0.27397260 0.28767123 0.30136986 0.31506849 0.32876712 0.39726027
      13      15      16      16.5      17.2      19      20
0.41095890 0.42465753 0.43835616 0.45205479 0.46575342 0.49315068 0.50684932
      20.7      21      21.6      22      24      24.1      25
0.52054795 0.53424658 0.54794521 0.56164384 0.61643836 0.63013699 0.65753425

```

```

      26      27      28      29      30      31.4      32
0.68493151 0.72602740 0.75342466 0.76712329 0.87671233 0.89041096 0.90410959
      33      34      34.8      38      46
0.93150685 0.95890411 0.97260274 0.98630137 1.00000000

> cumsum(table(d$Distancia) / length(d$Distancia))
      1      2.1      2.7      3.1      3.2      3.4      3.7
0.01369863 0.02739726 0.04109589 0.05479452 0.06849315 0.08219178 0.10958904
      4      4.4      4.5      5      5.1      5.5      6.2
0.13698630 0.16438356 0.19178082 0.20547945 0.21917808 0.23287671 0.24657534
      8.1      9      9.4      9.7      10      11      12
0.26027397 0.27397260 0.28767123 0.30136986 0.31506849 0.32876712 0.39726027
      13      15      16      16.5      17.2      19      20
0.41095890 0.42465753 0.43835616 0.45205479 0.46575342 0.49315068 0.50684932
      20.7      21      21.6      22      24      24.1      25
0.52054795 0.53424658 0.54794521 0.56164384 0.61643836 0.63013699 0.65753425
      26      27      28      29      30      31.4      32
0.68493151 0.72602740 0.75342466 0.76712329 0.87671233 0.89041096 0.90410959
      33      34      34.8      38      46
0.93150685 0.95890411 0.97260274 0.98630137 1.00000000

```

3.1.6 Segundo análisis de los datos

El segundo análisis de los datos consiste en el cálculo de la media aritmética, la media geométrica, la media armónica y la moda.

Como en el primer análisis de datos, se crearán funciones que reemplacen a las que ya proporciona R y se compararán los resultados para ver si coinciden. En el caso de la media geométrica, la media armónica y la moda se explicarán también los conceptos teóricos.

Para la media aritmética se creará la siguiente función:

```
mediaAritmetica <- function(x) {sumatorio(x)/longitud(x)}
```

En ella se utiliza la función *sumatorio()* que lo que hace es emular el comportamiento de la función en R *sum()*, dicha función se explicará más tarde. Esta función suma todos los datos del vector pasado como argumento, y luego se divide el resultado de esa función entre la longitud del vector.

Con los datos de las distancias el resultado sería el siguiente:

```

> mean(d$Distancia)
[1] 18.53425
> mediaAritmetica(d$Distancia)
[1] 18.53425

```

La media geométrica consiste en la raíz del producto de un conjunto de datos numéricos enteros. Este tipo de media se utiliza sobre todo para calcular medias sobre porcentajes.

$$\bar{x}_g = \left(\prod_{i=1}^n x_i \right)^{1/n}$$

No existe una función en R como tal pero se puede adaptar con *exp*, *mean* o *mediaAritmetica* creada anteriormente y *log* de la siguiente manera:

```

> exp(mean(log(d$Distancia)))
[1] 13.89035
> exp(mediaAritmetica(log(d$Distancia)))
[1] 13.89035

```

También se podrá adaptar con la formula teórica como se muestra a continuación:

```
> mediaGeometrica <- function(x) {return(prod(x)^(1/longitud(x)))}
> mediaGeometrica(d$Distancia)
[1] 13.89035
```

La media armónica consiste en el número de datos de una muestra entre la suma de los inversos de esos datos. Los datos deben ser numéricos enteros y distintos de cero. Se utiliza sobre todo para las velocidades o los tiempos.

$$1/\bar{x}_h = \frac{\sum_{i=1}^n 1/x_i}{n}$$

Tampoco existe una función en R que la realice por lo que se creará una que se adapte a la fórmula teórica y se aplicará a las distancias.

```
> mediaArmonica <- function(x) {return(1/mediaAritmetica(1/x))}
> mediaArmonica(d$Distancia)
[1] 8.814335
```

La moda consiste en el valor que aparece con más frecuencia en un conjunto de datos.

Para utilizar la moda en R se puede crear una función que utilice la frecuencia absoluta de los datos o mediante el uso de un paquete.

La función que se creará será la siguiente:

```
moda <- function(x) {return(as.integer(names(which.max(freq_abs(x))))))}
```

Esta función utiliza la función de la frecuencia absoluta de los datos para saber cuantas veces se repiten los valores, después averigua el dato que se repite más con la función *which.max* que en el caso de la tabla de frecuencias absolutas lo que hace es devolver el valor que más se repite junto con el índice de este en la tabla. Luego de esos dos números se extrae el del valor del dato mediante *names* que lo devuelve en formato caracter y por último este se convierte en entero para mostrarlo por pantalla.

La otra opción es mediante la instalación del paquete *modeest*. Dicho paquete contiene una función, *mlv*, que sirve para realizar diferentes tipos de modas dados unos datos. Esta función puede recibir como argumento el método con el que se quiere trabajar, en este caso el método **mfv** sería el adecuado ya que permite devolver el dato más frecuente.

Moda de las distancias con la función creada:

```
> moda(d$Distancia)
[1] 30
```

Moda de las distancias con el paquete *modeest* [6]:

```
> library("modeest")
> mlv(d$Distancia, method = "mfv")
[1] 30
```

Hay que destacar que en el caso de que haya más de un dato que sea el más frecuente, la función creada solo devolverá uno de los datos mientras que la función del paquete *modeest* devolverá todos los datos.

```
> v = c(1,1,1,2,2,2)
> moda(v)
[1] 1
> mlv(v, method = "mfv")
[1] 1 2
```

3.1.7 Tercer análisis de los datos

El tercer análisis de los datos consiste en el cálculo de la desviación típica, la varianza y el rango.

Se ha conseguido programar en R nuevas funciones que producen el mismo comportamiento que las instrucciones que se usaban anteriormente para realizar el cálculo. A continuación, se compararán las funciones de R y las funciones nuevas.

La función creada para calcular la desviación típica es la siguiente:

```
desTip <- function(v)
{
  sqrt(sumatorio(v, -mediaAritmetica(v), 2)/longitud(v))
}
```

La función *longitud()* es la que se ha explicado anteriormente que simula *length()* y *mediaAritmetica()* es lo mismo que *mean()*. Otra nueva función que se puede ver en el código de la función *desTip()* es *sumatorio()*. El código de esta función es el siguiente:

```
sumatorio <- function(v, valorSuma = 0, valorPotencia = 1)
{
  if (longitud(v) == 1) return((v[1] + valorSuma)^valorPotencia)
  else return((v[1] + valorSuma)^valorPotencia +
    sumatorio(v[-c(1)], valorSuma, valorPotencia))
}
```

En ella se utiliza la función *sumatorio()* que lo que hace es emular el comportamiento de la función en R *sum()*, dicha función se explicará más tarde. Esta función suma todos los datos del vector pasado como argumento, y luego se divide el resultado de esa función entre la longitud del vector. En ella se realiza la suma de todos los elementos del vector pasado como argumento. También tiene otros dos argumentos opcionales: *valorSuma* y *valorPotencia*. Estos tienen por defecto los valores 0 y 1, respectivamente. Si al hacer la llamada a esta función se le pasa como parámetro otros valores en esas posiciones, se usarán los valores pasados como ocurre en la llamada de *sumatorio()* en la función *desTip()*.

```
sumatorio(v, -mediaAritmetica(v), 2)
```

Lo que se realiza con estos valores es que en cada llamada recursiva, además de sumar el valor de la posición del vector correspondiente al resultado de la siguiente llamada, se puede sumar un número añadido (*valorSuma*) y este resultado elevarlo a otro número (*valorPotencia*). En el caso de la llamada anterior, cada valor de la posición correspondiente del vector se le resta la media aritmética y se eleva al cuadrado. Para comprobar el correcto funcionamiento, se compara la ejecución con la de *sum()*:

```
> sum(d$Distancia)
[1] 1353
> sumatorio(d$Distancia)
[1] 1353
```

Volviendo al cálculo de la desviación típica, comprobamos que el resultado es el mismo que *sd()* con el ajuste explicado en la sección *Tercer análisis de los datos* de la *Primera parte* para la estadística descriptiva:

```
> sqrt((sd(d$Distancia)^2) * (length(d$Distancia)-1) / length(d$Distancia))
[1] 11.23204
> desTip(d$Distancia)
[1] 11.23204
```

El siguiente cálculo que se realiza es el de la varianza. La función creada es:

```

varianza <- function(v)
{
  sumatorio(v, -mediaAritmetica(v), 2)/longitud(v)
}

```

Su equivalente de las funciones de R es *var()*. También se realiza la comprobación de su correcto funcionamiento con el ajuste de la estadística descriptiva:

```

> var(d$Distancia) * (length(d$Distancia)-1) / length(d$Distancia)
[1] 126.1587
> varianza(d$Distancia)
[1] 126.1587

```

Finalmente, se calcula el rango. Como se explicó anteriormente, el rango no tiene una función concreta en R que lo calcule, sino que se usa el máximo y el mínimo para obtener su valor, por lo tanto, se programarán estas dos funciones. Primero se muestra la función *maximo()* que es igual que *max()* de R:

```

maximo <- function(v)
{
  m <- v[1]
  if (longitud(v) == 1) return(m)
  else if (m < v[2]) return(maximo(v[-c(1)]))
  else return(maximo(v[-c(2)]))
}

```

En este código va recorriendo recursivamente el vector y compara los valores cogiendo el valor máximo en cada caso. Un ejemplo de este código es el siguiente:

```

> max(d$Distancia)
[1] 46
> maximo(d$Distancia)
[1] 46

```

Para la función *minimo()* se tiene este código:

```

minimo <- function(v)
{
  m <- v[1]
  if (longitud(v) == 1) return(m)
  else if (m > v[2]) return(minimo(v[-c(1)]))
  else return(minimo(v[-c(2)]))
}

```

Su función equivalente en R es *min()*. En este caso, igual que con *maximo()*, se recorre de forma recursiva el vector y busca el valor mínimo. La comprobación de su funcionamiento es el siguiente:

```

> min(d$Distancia)
[1] 1
> minimo(d$Distancia)
[1] 1

```

Por último, ya se puede crear la función para el rango con las nuevas funciones auxiliares:

```

rango <- function(x)
{
  maximo(x) - minimo(x)
}

```

Se realiza la comprobación del código anterior:

```
> max(d$Distancia)-min(d$Distancia)
[1] 45
> rango(d$Distancia)
[1] 45
```

3.1.8 Cuarto análisis de los datos

En el cuarto análisis de los datos consiste en el cálculo de la mediana y los cuantiles (cuartiles, deciles y percentiles).

En este apartado se seguirá la línea de los anteriores, se han creado funciones que simulan el comportamiento de las que brinda R. Se verá también si el resultado de las funciones de R es igual al que retorna las nuevas funciones.

Para la mediana se creará la siguiente función:

```
es_par <- function(num)
{
  if(num %% 2 == 0) return(1)
  else return(0)
}

mediana <- function(x)
{
  xOrdenado <- x[ordenar(x)]
  if(es_par(longitud(xOrdenado)))
  {
    i <- (longitud(xOrdenado)/2)
    return((xOrdenado[i] + xOrdenado[i+1])/2)
  }
  else return(xOrdenado[(longitud(xOrdenado)+1)/2])
}
```

La función *mediana()* ordena el vector que recibe como argumento. Se comprueba, con ayuda de *es_par()*, si la longitud de este es par o impar:

- Si es par devuelve la media de los dos elementos centrales del vector.
- Si es impar devuelve el elemento central del vector.

La función *es_par()* recibe un número como argumento, devolviendo 1 si es par y 0 si es impar.

Con los datos de las distancias el resultado sería el siguiente:

```
> median(d$Distancia)
[1] 20
> mediana(d$Distancia)
[1] 20
```

Como se puede comprobar el resultado es el mismo con ambas funciones.

Para los cuantiles se creará la siguiente función:

```
es_entero <- function(num)
{
  if(num %% 1 == 0) return(1)
  else return(0)
}

cuantil <- function(x, valor)
{
  xOrdenado <- x[ordenar(x)]
```



```

    i <- (valor * longitud(xOrdenado))
    if(es_entero(i)) return((xOrdenado[i] + xOrdenado[i+1])/2)
    else return(xOrdenado[trunc(i)+1])
}

```

La programación de la función *cuantil()* se inspira en las fórmulas aprendidas en clases. La función recibe dos argumentos, *x*: el vector y *valor*: número entre $[0,1]$ que representa el cuartil, decil o percentil que se desea. Se ordena el vector, se obtiene un *i* (posición) resultado de multiplicar el *valor* y la longitud del vector. Se comprueba si *i* es entero o no:

- Si *i* es entero devuelve la media de los elementos que se encuentran en la posición *i* e (*i*+1), respectivamente.
- Si *i* no es entero devuelve el elemento que está en la posición superior luego de truncar *i*.

La función *es_entero()* devolverá 1 si el número es entero y 0 si no lo es.

El resultado sería el siguiente usando la función *quantile()*:

```

> quantile(d$Distancia,0.25)
25%
8.1
> quantile(d$Distancia,0.5)
50%
20
> quantile(d$Distancia,0.75)
75%
28
> quantile(d$Distancia,0.54)
54%
21.528

```

El resultado sería el siguiente usando la función *cuantil()*:

```

> cuantil(d$Distancia,0.25)
[1] 8.1
> cuantil(d$Distancia,0.5)
[1] 20
> cuantil(d$Distancia,0.75)
[1] 28
> cuantil(d$Distancia,0.54)
[1] 21.6

```

Los resultados tanto de la fórmula *quantile()* y *cuantil()* son iguales excepto el valor del percentil 54. Se usó la fórmula *cuantil()* con los datos de la primera parte y se observa una mayor disparidad en los resultados esto es porque R calcula los cuantiles de una forma distinta a como lo hacemos en clases de teoría.

3.2 Análisis de asociación

3.2.1 Instalación de paquetes

Además de lo explicado anteriormente sobre la instalación de paquetes, es también interesante saber que se puede instalar varios paquetes a la vez sin escribir la misma función varias veces utilizando la función *c* dentro de la función *install.packages()*. Hay que tener en cuenta que se necesita comillas para especificar el nombre de los paquetes.

```

> install.packages(c("LearningRlab", "ggplot2"))

```

Los anteriores paquetes se han instalado desde el repositorio de CRAN, pero cada repositorio tiene su forma particular de instalar un paquete. Por ejemplo, en el caso de Bioconductor, el modo estándar de instalar un paquete es ejecutar el siguiente script:

```
source("https://bioconductor.org/biocLite.R")
```

La instrucción anterior instalará de forma local las funciones necesarias para la instalación de paquetes bioconductor. Si se desea instalar los paquetes básicos de Bioconductor, ahora se puede realizar con la función *biocLite()*. La función *biocLite()* también admite que se especifique el nombre de los paquetes de la misma forma que *install.packages()*.

Como se ha podido observar, en el caso de que regularmente se necesite utilizar diferentes fuentes en la instalación de paquetes, que cada repositorio instale sus paquetes de forma diferente puede llegar a convertirse en una labor un poco pesada.

El uso del paquete *devtools* [10] simplificará esta tarea, puesto que contiene funciones específicas para cada repositorio. Para el uso de este paquete es posible que se necesite instalar las *Rtools*.

Para instalar las *Rtools* se accede a la página web <https://cran.r-project.org/bin/windows/Rtools/> en el caso de que el sistema operativo sea *Windows* (Figura 16), se elige el instalador adecuado para cada máquina y se ejecuta siguiendo las instrucciones de instalación. Si es otro sistema operativo, en el menú de la derecha de la página de CRAN está la opción *R Binaries* (Figura 11a) y aparecen los diferentes sistemas operativos disponibles (Figura 17).

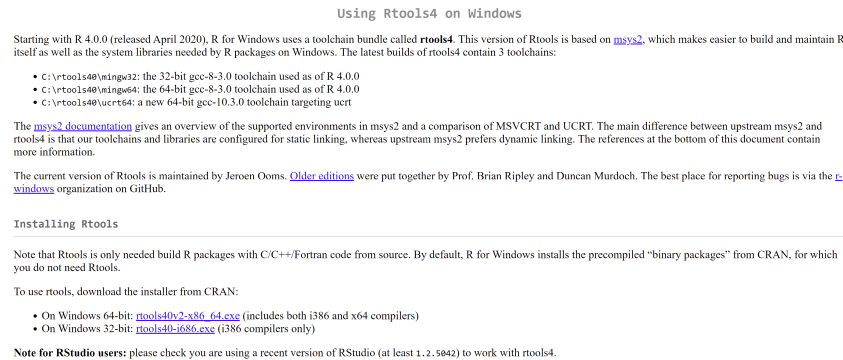







Figure 16: Página web de instalación de *Rtools*.

Una vez instaladas las *Rtools*, el paquete *devtools* no debe de dar ningún error. Las funciones que incluye el paquete *devtools* para la instalación de otros paquetes son las siguientes:

- *install_bioc()* desde Bioconductor.
- *install_bitbucket()* desde Bitbucket.
- *install_cran()* desde CRAN.
- *install_git()* desde un repositorio git.
- *install_github()* desde GitHub.
- *install_local()* desde un archivo alojado de forma local.
- *install_svn()* desde un repositorio SVN.
- *install_url()* desde una URL.
- *install_version()* para una versión específica de un paquete de CRAN.

Index of /bin

<u>Name</u>	<u>Last modified</u>	<u>Size</u>	<u>Description</u>
 Parent Directory		-	
 linux/	2020-07-07 11:45	-	
 macos/	2005-04-19 09:45	-	
 macosx/	2021-11-02 05:20	-	
 windows/	2017-09-29 11:35	-	

Apache Server at cran.r-project.org Port 443

Figure 17: Opción *R Binaries*.

De forma usual, los paquetes que se instalan desde algún repositorio como CRAN o Bioconductor son paquetes binarios que ya se encuentran compilados previamente.

En algunas ocasiones se necesitará instalar paquetes que no se encuentran compilados, por ejemplo:

- Paquetes en desarrollo.
- Versiones anteriores de paquetes.
- Paquetes que no se encuentran depositados en CRAN o Bioconductor, sino en repositorios personales como GitHub.
- Paquetes que se están desarrollando de forma local.

Existen algunas funciones que permiten instalar paquetes desde código fuente. Anteriormente, se solían utilizar las funciones nombradas anteriormente del paquete *devtools*; sin embargo, recientemente se creó el paquete *remotes* [5] que contiene las mismas funciones pero está específicamente diseñado para ayudar a trabajar con paquetes desde código fuente.

El código fuente de un paquete se encuentra en diferentes lugares dependiendo del repositorio en el que se localice. En CRAN se puede encontrar en las secciones *URL* y *Package source* como se puede ver en las Figuras 13 y 14. Si el paquete se encuentra disponible en Bioconductor (Página web <https://www.bioconductor.org/packages/release/bioc/> y Figura 18), el link al código fuente se puede encontrar en la sección *Package Archives* (Figura 19). Si el paquete se encuentra en GitHub o GitLab, hace falta conocer el nombre de usuario del autor y el nombre del paquete que normalmente será también el nombre del repositorio a instalar.

Para instalar la última versión en desarrollo del repositorio CRAN, se usa la función *install_dev()* del paquete *remotes*. Se puede escribir la instrucción de esta forma:

```
> remotes::install_dev("ggplot2")
```

El operador `::` permite llamar funciones desde un paquete sin la necesidad de cargarlo, es decir, sin usar previamente la función *library()*.

The screenshot shows the Bioconductor website interface. At the top, there is a navigation bar with links for Home, Install, Help, Developers, and About. A search bar is located on the right. Below the navigation bar, the page title is "Bioconductor Software Packages". The main content area displays a table of software packages, categorized by Bioconductor version (Release 3.14). The table has three columns: Package, Maintainer, and Title. The packages listed include a4, a4Base, a4Classif, a4Core, a4Preproc, a4Reporting, ABAEnrichment, and ABarray. To the right of the table, there are two sidebars. The first sidebar, titled "Packages", lists various package types: Analysis software packages, Annotation packages, Illustrative experiment data packages, Workflow packages, and Online books. It also mentions that Bioconductor is available via Docker and Amazon Machine Images. The second sidebar, titled "Development Version", lists packages under development: Analysis software packages and Illustrative experiment data packages. It also mentions Developer Resources.

Package	Maintainer	Title
a4	Laure Cougnaud	Automated Affymetrix Array Analysis Umbrella Package
a4Base	Laure Cougnaud	Automated Affymetrix Array Analysis Base Package
a4Classif	Laure Cougnaud	Automated Affymetrix Array Analysis Classification Package
a4Core	Laure Cougnaud	Automated Affymetrix Array Analysis Core Package
a4Preproc	Laure Cougnaud	Automated Affymetrix Array Analysis Preprocessing Package
a4Reporting	Laure Cougnaud	Automated Affymetrix Array Analysis Reporting Package
ABAEnrichment	Steffi Grote	Gene expression enrichment in human brain regions
ABarray	Yongming Andrew Sun	Microarray QA and statistical data analysis for Applied Biosystems Genome Survey Microarray (AB1700) gene expression data.

Figure 18: Paquetes de Bioconductor.

Package Archives

Follow [Installation](#) instructions to use this package in your R session.

Source Package	bioCancer 1.22.0.tar.gz
Windows Binary	bioCancer 1.22.0.zip
macOS 10.13 (High Sierra)	bioCancer 1.22.0.tgz
Source Repository	git clone https://git.bioconductor.org/packages/bioCancer
Source Repository (Developer Access)	git clone git@git.bioconductor.org:packages/bioCancer
Package Short Url	https://bioconductor.org/packages/bioCancer/
Package Downloads Report	Download Stats
Old Source Packages for BioC 3.14	Source Archive

Figure 19: Sección *Package Archives*.

Si el paquete de la última versión en desarrollo se encuentra en Bioconductor, se usa la función *install_bioc()* de la misma forma que la anterior.

Independientemente de donde se encuentre el paquete, podemos instalar un paquete depositado en una cuenta de GitHub. Para poder instalar un paquete desde GitHub se necesita conocer el usuario del creador y el nombre del repositorio donde se encuentra depositado el paquete. Esta información se usa en la función *install_github()* de la siguiente manera:

```
remotes::install\_github("usuario/repositorio")
```

Para instalar versiones anteriores de los paquetes de CRAN se usa la función *install_version()* como se muestra a continuación:

```
remotes::install_version("pkgname", version = "version")
```

En los paquetes de Bioconductor, se usa el paquete *BiocManager*, que permite acceder al repositorio de paquetes de proyectos de Bioconductor y es una herramienta conveniente para instalar y actualizar estos paquetes. La función que se utiliza es la siguiente con el nombre del paquete y la versión deseada:

```
BiocManager::install(pkgs = "pkgname", version = "version")
```

La función principal para instalar el paquete de manera local, ya sea de un código fuente de un paquete descargado o de un paquete propio en desarrollo, es:

```
remotes::install_local()
```

Ahora se va a realizar un ejemplo con el paquete *testthat* [9] descargado del repositorio de CRAN. Los pasos a seguir son los siguientes:

1. Se guarda en una variable el directorio donde se desea descargar el código fuente del paquete. En este caso, se usará el mismo que el directorio de trabajo, por lo tanto se usa la función *getwd()*.

```
> dir <- getwd()
```

2. Se usa la función *download.packages()* para descargar el código fuente del paquete *testthat* en forma de directorio comprimido. Se pasan como argumentos el nombre del paquete, el directorio donde se desea descargar y el tipo de código que es, en este caso fuente (*source*). Luego se pide seleccionar el *mirror* (Figura 9b).

```
> pkg <- download.packages("testthat", dir, type = "source")
--- Please select a CRAN mirror for use in this session ---
probando la URL 'https://cran.rediris.es/src/contrib/
testthat_3.1.0.tar.gz'
Content type 'application/x-gzip' length 695232 bytes (678 KB)
downloaded 678 KB
```

En la variable donde se guardan los datos de la descarga, *pkg*, se almacena el nombre del paquete y la ubicación temporal donde se encuentra el archivo comprimido.

```
> pkg
      [,1]
[1,] "testthat"
      [,2]
[1,] "C:/Users/atien/OneDrive - Universidad de Alcala/
Documents/R/testthat_3.1.0.tar.gz"
```

También se puede descargar manualmente como se ha explicado en la *Primera parte* desde la página web de CRAN y guardarlo en el directorio que se ha asignado a la variable anterior *dir*.

3. La ruta de la variable *pkg* se usará para instalar el paquete desde su ubicación local con la instrucción *install_local()* del paquete *remotes*.

```
> remotes::install_local(pkg[, 2])
- checking for file 'C:\Users\atien\AppData\Local\Temp\Rtmp8cVpyS\remotes403c58535e29\testthat\DESCRIPTION'
- preparing 'testthat': (2.1s)
- checking DESCRIPTION meta-information ...
- cleaning src
- checking vignette meta-information ...
- checking for LF line-endings in source and make files and
shell scripts
- checking for empty or unneeded directories
- building 'testthat_3.1.0.tar.gz'

Installing package into 'C:/Users/atien/OneDrive - Universidad
de Alcalá/Documents/R/win-library/4.1'
(as 'lib' is unspecified)
* installing *source* package 'testthat' ...
** using staged installation
** libs
```

Se puede verificar que un paquete se encuentra en la lista de paquetes instalados con el uso de la función *installed.packages()*. Para ello, se debe seguir los siguientes pasos:

1. Se obtiene toda la información de los paquetes instalados.

```
> a<-installed.packages()
> a
```

	Package	LibPath	
cli	"cli"	"C:/Users/atien/OneDrive - Universidad de Alcalá/Documents/R/win-library/4.1"	
ellipsis	"ellipsis"	"C:/Users/atien/OneDrive - Universidad de Alcalá/Documents/R/win-library/4.1"	
evaluate	"evaluate"	"C:/Users/atien/OneDrive - Universidad de Alcalá/Documents/R/win-library/4.1"	
fansi	"fansi"	"C:/Users/atien/OneDrive - Universidad de Alcalá/Documents/R/win-library/4.1"	
...			
	Version	Priority	Depends
cli	"3.1.0"	NA	"R (>= 2.10)"
ellipsis	"0.3.2"	NA	"R (>= 3.2)"
evaluate	"0.14"	NA	"R (>= 3.0.2)"
fansi	"0.5.0"	NA	"R (>= 3.1.0)"
...			
	Imports		
cli	"glue, utils"		
ellipsis	"rlang (>= 0.3.0)"		
evaluate	"methods"		
fansi	"grDevices, utils"		
...			
	Enhances		
cli	NA		
ellipsis	NA		
evaluate	NA		

```
fansi      NA
...
      License
License_is_FOSS License_restricts_use OS_type MD5sum
cli      "MIT + file LICENSE"
NA      NA      NA      NA
ellipsis "MIT + file LICENSE"
NA      NA      NA      NA
evaluate "MIT + file LICENSE"
NA      NA      NA      NA
fansi    "GPL (>= 2)"
NA      NA      NA      NA
...
      NeedsCompilation Built
cli      "yes"      "4.1.1"
ellipsis "yes"      "4.1.1"
evaluate "no"       "4.1.1"
fansi    "yes"      "4.1.1"
...
```

2. A partir de toda la información obtenida anteriormente, se cogen solo los nombres de los paquetes con la siguiente instrucción:

```
> packages<-a[,1]
> packages
      cli      ellipsis      evaluate      fansi
      "cli"      "ellipsis"      "evaluate"      "fans"
      farver      fastmap      fs
      "farver"      "fastmap"      "fs"
      gert      ggplot2      gh      gitcreds
      "gert"      "ggplot2"      "gh"      "gitcreds"
      glue      gtable      highr
      "glue"      "gtable"      "highr"
      httr      ini      isoband      jsonlite
      "httr"      "ini"      "isoband"      "jsonlite"
      knitr      labeling      lifecycle
      "knitr"      "labeling"      "lifecycle"
      magrittr      memoise      mime      munsell
      "magrittr"      "memoise"      "mime"      "munsell"
      openssl      pillar      pkgbuild
      "openssl"      "pillar"      "pkgbuild"
      pkgconfig      pkgload      praise      prettyunits
      "pkgconfig"      "pkgload"      "praise"      "prettyunits"
      processx      ps      purrr
      "processx"      "ps"      "purrr"
      R6      rappdirs      rcmdcheck      RColorBrewer
      "R6"      "rappdirs"      "rcmdcheck"      "RColorBrewer"
      rematch2      remotes      rlang
      "rematch2"      "remotes"      "rlang"
      roxygen2      rprojroot      rstudioapi      rversions
      "roxygen2"      "rprojroot"      "rstudioapi"      "rversions"
      scales      sessioninfo      stringi
      "scales"      "sessioninfo"      "stringi"
      stringr      sys      testthat      tibble
      "stringr"      "sys"      "testthat"      "tibble"
```

usethis	utf8	vctr	
"usethis"	"utf8"	"vctr"	
viridisLite	waldo	whisker	withr
"viridisLite"	"waldo"	"whisker"	"withr"
xfun	xml2	xopen	
"xfun"	"xml2"	"xopen"	
yaml	zip	base	boot
"yaml"	"zip"	"base"	"boot"
class	cluster	codetools	
"class"	"cluster"	"codetools"	
compiler	datasets	foreign	graphics
"compiler"	"datasets"	"foreign"	"graphics"
grDevices	grid	KernSmooth	
"grDevices"	"grid"	"KernSmooth"	
lattice	MASS	Matrix	methods
"lattice"	"MASS"	"Matrix"	"methods"
mgcv	nlme	nnet	
"mgcv"	"nlme"	"nnet"	
parallel	rpart	spatial	splines
"parallel"	"rpart"	"spatial"	"splines"
stats	stats4	survival	
"stats"	"stats4"	"survival"	
tcltk	tools	translations	utils
"tcltk"	"tools"	"translations"	"utils"

- Finalmente, se busca el paquete deseado entre el listado de paquetes anterior. Esto se realiza de la siguiente forma. En este ejemplo, se busca el paquete *boot*.

```
> is.element("boot", packages)
[1] TRUE
```

3.2.2 Creación de paquetes

En esta sección se explicará cómo crear paquetes en R en unos sencillos pasos básicos [3].

Primero es necesario tener instalado el paquete *devtools* y también es recomendable instalar el paquete *roxygen2*. Se realiza con la instrucción:

```
> install.packages("devtools", dependencies=TRUE)
> install.packages("roxygen2", dependencies=TRUE)
```

El siguiente paso es crear la estructura de directorios que tiene un paquete en R. El paquete que se va a crear se llama *paqueteEstadistica* donde se añadirán todas las funciones creadas en la sección *Análisis de descripción de datos* de la *Segunda parte* de este documento. Una vez conocido el nombre del paquete, hay que crear una carpeta que se llame igual que este paquete. Dentro de la carpeta se crea un archivo sin extensión con título *DESCRIPTION*. En su interior hay que añadir, como mínimo, la siguiente información:

```
Package: paqueteR
Version: 0.1
encoding: UTF-8
```

El próximo paso consiste en crear una carpeta llamada *R* para añadir el código de las funciones que se desean que estén en el paquete.

El cuarto paso es documentar el código de estas funciones. Un ejemplo de la documentación se puede observar en el código de la función *freq.abs()*:


```

#' Frecuencia absoluta
#'
#' Función para calcular la frecuencia absoluta.
#'
#' @param x Un vector
#'
#' @return Frecuencia absoluta de los elementos de x
#' @export
#'
#' @examples freq_abs(c(1,2,1,3))
freq_abs <- function(x)
{
  if (!is.numeric(x))
  {
    stop("x is not a numeric")
  }

  data <- sort(unique(x))
  counter <- vector(mode="numeric", length=0)
  for (value in data)
  {
    total <- longitud(unname(x[x == value]))
    counter <- c(counter, total)
  }

  setNames(counter, data)
}

```

La documentación en R empieza cada línea con `#'`. Se hace una breve explicación del nombre de la función y su funcionalidad. Se pueden distinguir los siguientes campos:

1. *@param* donde se explican los argumentos de la función y su tipo.
2. *@return* donde se explica lo que devuelve la función.
3. *@export* indica si la función la puede usar los usuarios del paquete.
4. *@examples* para crear ejemplos de la función.

Una vez documentadas las funciones, hay que situar el directorio de trabajo en esa carpeta.

```
> setwd("./paqueteEstadistica")
```

Luego se procesa la documentación con la función *document()* del paquete *devtools* de la siguiente manera:

```
> devtools::document()
```

Esta instrucción genera una nueva carpeta dentro del directorio del paquete llamada *man* y un archivo llamado *NAMESPACE* que contiene la siguiente información:

```
# Generated by roxygen2: do not edit by hand
```

Finalmente, ya creada la estructura básica del paquete se puede compilar con la instrucción *build()* de esta forma:

```
> devtools::build()
```

Esta función crea el archivo comprimido *paqueteEstadistica_0.1.tar.gz* con el paquete listo para ser compartido.

Ya se puede instalar el paquete como se ha explicado anteriormente.

Además del paquete *paqueteEstadistica* se creará también otro paquete con el código del algoritmo Apriori que se llamará *paqueteApriori*.

3.2.3 Aplicación del algoritmo Apriori

Para la aplicación del algoritmo Apriori en la segunda parte se tiene una nueva muestra que se encuentra en la Tabla 2, siendo la matriz con la que se va a trabajar.

	X	A	T	N	B	C
Suceso 1	1	0	0	1	1	1
Suceso 2	1	0	1	0	1	1
Suceso 3	1	0	0	1	0	1
Suceso 4	1	0	1	1	1	0
Suceso 5	0	0	0	0	1	1
Suceso 6	0	0	0	1	0	0
Suceso 7	1	0	0	0	1	1
Suceso 8	0	1	1	0	0	0

Table 2: Matriz correspondiente a la muestra observada, donde: {X: Faros de Xenon, A: Alarma, T: Techo Solar, N: Navegador, B: Bluetooth, C: Control de Velocidad}

En este caso, la muestra se almacena en el fichero de datos extras.txt siguiendo las mismas pautas vistas en la sección *Formato del fichero de datos* de la *Primera parte*.

	X	A	T	N	B	C
suceso1	1		0	0	1	1
suceso2	1		0	1	0	1
suceso3	1		0	0	1	0
suceso4	1		0	1	1	0
suceso5	1		0	0	0	1
suceso6	0		0	0	1	0
suceso7	1		0	0	0	1
suceso8	0	1	1	0	0	0

De tal forma que pueda ser leído por una función y realizar todas las transformaciones necesarias a la matriz. Para ello, se ha programado la función `read.apriori()` que como argumento de entrada es `file`, es decir, el nombre del fichero que vamos a cargar como matriz. Una vez cargada como matriz, se siguen los mismos pasos que en la *Primera parte* pero todo ello ha sido comprimido en una misma función.

```
read.apriori <- function(file)
{
  if (!is.character(file))
  {
    stop("file is not a character")
  }

  muestra <- Matrix(as.matrix(read.table(file)), sparse=T)
  muestrangCMatrix <- as(muestra, "nsparseMatrix")
  t(muestrangCMatrix)
}
```

Así se puede cargar directamente el fichero extras.txt y ya se tiene la matriz preparada para usar en el algoritmo Apriori.

```
> muestraExtras <- read.apriori("extras.txt")
> muestraExtras
6 x 8 sparse Matrix of class "ngCMatrix"
```

	suceso1	suceso2	suceso3	suceso4	suceso5	suceso6	suceso7	suceso8
X						.		.
A	
T	
N	
B			.			.		.
C				.		.		.

Como prueba de su funcionamiento, se ha hecho lo mismo para las cestas de la parte anterior, para que se pueda comprobar que se obtiene la misma matriz.

```
> muestraCestas <- read.apriori("cestas.txt")
> muestraCestas
5 x 6 sparse Matrix of class "ngCMatrix"
      suceso1 suceso2 suceso3 suceso4 suceso5 suceso6
Pan      |      |      |      |      |      .
Agua     |      |      |      .      |      .
Cafe     .      |      .      |      .      .
Leche    |      |      |      |      .      |
Naranjas |      .      .      .      .      .
> transpmuestrangCMatrix
5 x 6 sparse Matrix of class "ngCMatrix"
      suceso1 suceso2 suceso3 suceso4 suceso5 suceso6
Pan      |      |      |      |      |      .
Agua     |      |      |      .      |      .
Cafe     .      |      .      |      .      .
Leche    |      |      |      |      .      |
Naranjas |      .      .      .      .      .
```

Para aplicar el algoritmo Apriori se ha programado antes en R siguiendo el pseudocódigo de Christian Borgelt [2], el *paper* de Roberto J. Bayardo [1] y lo visto en clase de teoría (sin implementar la optimización del *hash tree*). Primero se va a explicar las distintas funciones del algoritmo, empezando por la principal y descomponiendo las otras funciones que son llamadas. Para finalizar, se demuestra que los resultados son correctos al compararlos con el uso de la función *apriori()* del paquete *arules*.

Se empieza con *apriori_main()* que es la llamada principal para que comience el algoritmo. Tiene como argumentos *matrix* que es la matriz de la muestra (que puede haber sido cargada con *read.apriori()*), el *support* que es el umbral del soporte y, por último, el *confidence* siendo el umbral de la confianza. La función comienza obteniendo los candidatos y sucesos a partir de la matriz que se le pasa, y realiza los siguientes pasos:

Paso A:

Subpaso A.1. Realiza la primera poda de los sucesos elementales llamando a la función *prune()*.

Subpaso A.2.1 A partir de los sucesos elementales se llama a la función *apriori-gen()* empezando por la dimensión dos ($k = 2$) y aumentando en uno hasta que no haya más candidatos.

Subpaso A.2.2 Se realiza la poda de los nuevos candidatos en la dimensión k llamando a la función *prune()* y que, como optimización, se podría implementar en un *hash tree*.

Paso B: Se aplica la función *ap_genrules()* para el cálculo de la confianza y conocer las asociaciones siguiendo el teorema visto en teoría.

Paso C: Se muestra los resultados con *printResults()* al igual que lo hace la función *inspect()* del paquete *arules*.

```
apriori_main <- function(matrix, support, confidence)
{
```

```

candidates <- list()
for (c in seq(length(rownames(matrix))))
{
    candidates <- c(candidates, c)
}
transactions <- seq(length(colnames(matrix)))

frequents = prune(matrix, candidates, transactions, support)

k <- 2
repeat
{
    candidates <- apriori_gen(frequents, k)

    if (length(candidates) == 0)
    {
        break
    }

    frequents_k <- prune(matrix, candidates, transactions, support)
    frequents <- c(frequents, frequents_k)

    k <- k + 1
}

rules <- c()
for (n in length(frequents):2)
{
    for (i in 1:length(frequents[[n]]))
    {
        rules_n <- ap_genrules(matrix, c(), c(), frequents[[n]][[i]], n,
                                transactions, confidence)
        rules <- c(rules, rules_n)
    }
}

printResults(matrix, rules, transactions)
}

```

La función *prune()* que es utilizada por el algoritmo en el **Subpaso A.1** y **Subpaso A.2.2** tiene como argumentos de entrada *matrix* siendo la matriz de la muestra, *candidates* que son los candidatos a podar, *transactions* los sucesos y *support* el umbral de soporte. Básicamente esta función mantiene unos contadores de soportes que almacena el número de apariciones del candidato en la matriz usando la función *countOccurrences()* y seleccionado aquellos que superan el umbral del soporte.

```

prune <- function(matrix, candidates, transactions, support)
{
    supportCounters = c()

    for (c in 1:length(candidates))
    {
        counter <- countOccurrences(matrix, candidates[[c]], transactions)
        supportCounters = c(supportCounters, counter)
    }
}

```

```

frequentes <- list()
for (c in 1:length(candidates))
{
    if ((supportCounters[c] / length(transactions)) >= support)
    {
        frequentes <- c(frequentes, candidates[c])
    }
}

list(unique(frequentes))
}

```

Para conocer las apariciones de un candidato en la matriz se ha programado esta función llamada *countOccurrences()* que tiene como argumentos *matrix* la matriz de la muestra, *candidate* siendo el candidato y *transactions* los sucesos. Aquí es cierto que se puede hacer una mejora optimización con el uso de *hash tree* pero para simplificar la programación del algoritmo, se va recorriendo los sucesos y comprobando si dicho candidato pertenece a la matriz de muestra y se le suma uno al contador.

```

countOccurrences <- function(matrix, candidate, transactions)
{
    counter <- 0

    for (t in transactions)
    {
        contains <- T
        for (i in candidate)
        {
            contains <- matrix[i, t] && contains
        }

        if (contains)
        {
            counter <- counter + 1
        }
    }

    counter
}

```

La función *apriori_gen* es la encargada de identificar los candidatos en cada dimensión k comprobando que se cumplen dos condiciones:

1. $a_i = b_i$ para $i = 1, 2, \dots, k - 2$
2. $a_{k-1} \neq b_{k-1}$

Recibe como argumentos *frequentes* que son los candidatos que han pasado por la poda y la dimensión k .

```

apriori_gen <- function(frequentes, k)
{
    candidates = list()

    for (f in 1:length(frequentes[[k - 1]]))
    {
        for (j in f + 1:length(frequentes[[k - 1]]) - 1)

```

```

    {
      if (j > length(frequents[[k - 1]]))
      {
        next
      }

      a <- frequents[[k - 1]][[f]]
      b <- frequents[[k - 1]][[j]]

      if (k - 2 > 0)
      {
        for (i in 1:(k - 2))
        {
          if (a[i] != b[i])
          {
            next
          }
        }

        if (a[k - 1] != b[k - 1])
        {
          candidates <- c(candidates, list(union(a, b)))
        }
      }
    }

    candidates
  }
}

```

Usando la función *ap_genrules()* que se basa en el teorema visto en clase de si sean dos conjuntos A y B, si la asociación $A \rightarrow B - A$ no supera el umbral de confianza, entonces cualquier asociación $A' \rightarrow B - A'$, donde A' es cualquier subconjunto de A tampoco supera el umbral de confianza. Para ello se realizan llamadas de forma recursiva de la función *ap_genrules()* hasta que no haya más subconjuntos posibles. Tiene como argumentos *matrix* siendo la matriz de muestra, *rules* las asociaciones que se van almacenando en cada llamada recursiva y luego se devuelve, los dos conjuntos A y B, la dimensión a la que se está aplicando la función con *dimensions*, los sucesos en *transactions* y, por último, el umbral de confianza a comprobar con *confidence*.

```

ap_genrules <- function(matrix, rules, A, B, dimensions, transactions, confidence)
{
  if (length(A) == 0)
  {
    combinations <- combn(B, dimensions - 1)
  }
  else
  {
    combinations <- combn(A, dimensions - 1)
  }

  if (length(combinations) == 0)
  {
    return(rules)
  }
}

```

```

for (col in 1:ncol(combinations))
{
  lhs <- c()
  for (row in 1:nrow(combinations))
  {
    lhs <- c(lhs, combinations[row, col])
  }

  rhs <- setdiff(B, lhs)

  nB <- countOccurrences(matrix, B, transactions)
  nA <- countOccurrences(matrix, lhs, transactions)

  if ((nB / nA) >= confidence)
  {
    rules <- c(rules, list(c(lhs = list(lhs), rhs = rhs)))

    ap_genrules(matrix, rules, lhs, B, dimensions - 1, transactions,
                 confidence)
  }
}

rules
}

```

Y la última función, *printResults()* que simplemente calcula de cada asociación obtenida el *support*, *confidence*, *coverage*, *lift* y *count*. Para después almacenarlo en un *data.frame* y que se muestre como lo haría la función *inspect()* del paquete *arules*. Tiene como argumentos la *matrix* de muestra, *rules* que son las asociaciones seleccionados y los sucesos en *transactions*.

```

printResults <- function(matrix, rules, transactions)
{
  rows <- rownames(matrix)
  n <- length(transactions)
  lhs <- c()
  rhs <- c()
  support <- c()
  confidence <- c()
  coverage <- c()
  lift <- c()
  count <- c()
  for (i in length(rules):1)
  {
    count_lhs_rhs <- countOccurrences(matrix, c(rules[[i]]$lhs, rules[[i]]$rhs),
                                       transactions)
    count_lhs <- countOccurrences(matrix, rules[[i]]$lhs, transactions)
    count_rhs <- countOccurrences(matrix, rules[[i]]$rhs, transactions)

    support_i <- count_lhs_rhs / n
    coverage_i <- count_lhs / n

    lhs <- c(lhs, paste("{", paste(rows[rules[[i]]$lhs], collapse=","), "}", sep=""))
    rhs <- c(rhs, paste("{", paste(rows[rules[[i]]$rhs], collapse=","), "}", sep=""))
  }
}

```

```

        support <- c(support, support_i)
        confidence <- c(confidence, (count_lhs_rhs / count_lhs))
        coverage <- c(coverage, coverage_i)
        lift <- c(lift, (support_i / (coverage_i * (count_rhs / n))))
        count <- c(count, count_lhs_rhs)
    }

    results <- data.frame(lhs, "=>", rhs, support, confidence, coverage, lift, count)
    names(results) <- c("lhs", "", "rhs", "support", "confidence", "coverage", "lift", "count")

    results
}

```

Para finalizar, se demuestra que el algoritmo funciona dando los mismos resultados que con el paquete *arules*. En este caso, para las muestras de extras incluidos de coches también se ha elegido como en el ejercicio anterior el umbral de soporte a 0.5 y confianza a 0.8, aunque obviamente esto puede cambiar y obtener otros resultados como por ejemplo con un soporte de 0.4 y una confianza de 0.9.

```

> apriori_main(muestraExtras, 0.5, 0.8)
  lhs  rhs support confidence coverage  lift count
1 {C} => {B}   0.500  0.8000000    0.625 1.280000    4
2 {B} => {C}   0.500  0.8000000    0.625 1.280000    4
3 {C} => {X}   0.625  1.0000000    0.625 1.333333    5
4 {X} => {C}   0.625  0.8333333    0.750 1.333333    5
5 {B} => {X}   0.625  1.0000000    0.625 1.333333    5
6 {X} => {B}   0.625  0.8333333    0.750 1.333333    5
7 {B,C} => {X} 0.500  1.0000000    0.500 1.333333    4
8 {X,C} => {B} 0.500  0.8000000    0.625 1.280000    4
9 {X,B} => {C} 0.500  0.8000000    0.625 1.280000    4

> apriori_main(muestra, 0.4, 0.9)
  lhs  rhs support confidence coverage  lift count
1 {C} => {X}   0.625          1    0.625 1.333333    5
2 {B} => {X}   0.625          1    0.625 1.333333    5
3 {B,C} => {X} 0.500          1    0.500 1.333333    4

```

Si se hace lo mismo con el algoritmo Apriori se puede comprobar que los resultados son iguales:

```

> transacciones <- as(muestraExtras, "transactions")
> inspect(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: 4


```

set item appearances ...[0 item(s)] done [0.00s].
set transactions ...[6 item(s), 8 transaction(s)] done [0.00s].
sorting and recoding items ... [4 item(s)] done [0.00s].
creating transaction tree ... done [0.00s].
checking subsets of size 1 2 3 done [0.00s].
writing ... [9 rule(s)] done [0.00s].
creating S4 object ... done [0.00s].

```

	lhs	rhs	support	confidence	coverage	lift	count
[1]	{C}	=> {B}	0.500	0.8000000	0.625	1.280000	4
[2]	{B}	=> {C}	0.500	0.8000000	0.625	1.280000	4
[3]	{C}	=> {X}	0.625	1.0000000	0.625	1.333333	5
[4]	{X}	=> {C}	0.625	0.8333333	0.750	1.333333	5
[5]	{B}	=> {X}	0.625	1.0000000	0.625	1.333333	5
[6]	{X}	=> {B}	0.625	0.8333333	0.750	1.333333	5
[7]	{B,C}	=> {X}	0.500	1.0000000	0.500	1.333333	4
[8]	{X,C}	=> {B}	0.500	0.8000000	0.625	1.280000	4
[9]	{X,B}	=> {C}	0.500	0.8000000	0.625	1.280000	4

```

> inspect(apriori(transacciones, parameter=list(support=0.4, confidence=0.9)))
Apriori

```

```

Parameter specification:
confidence minval smax arem aval originalSupport maxtime support minlen
          0.9    0.1    1 none FALSE             TRUE         5     0.4     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 ...[6 item(s), 8 transaction(s)] done [0.00s].
sorting and recoding items ... [4 item(s)] done [0.00s].
creating transaction tree ... done [0.00s].
checking subsets of size 1 2 3 done [0.00s].
writing ... [3 rule(s)] done [0.00s].
creating S4 object ... done [0.00s].

```

	lhs	rhs	support	confidence	coverage	lift	count
[1]	{C}	=> {X}	0.625	1	0.625	1.333333	5
[2]	{B}	=> {X}	0.625	1	0.625	1.333333	5
[3]	{B,C}	=> {X}	0.500	1	0.500	1.333333	4

Y lo mismo ocurre con las cestas de la compra de la *Primera parte*:

```

> apriori_main(muestraCestas, 0.5, 0.8)

```

	lhs	rhs	support	confidence	coverage	lift	count	
1	{Leche}	=> {Pan}	0.6666667		0.8	0.8333333	0.96	4
2	{Pan}	=> {Leche}	0.6666667		0.8	0.8333333	0.96	4
3	{Agua}	=> {Pan}	0.6666667		1.0	0.6666667	1.20	4
4	{Pan}	=> {Agua}	0.6666667		0.8	0.8333333	1.20	4
5	{Agua,Leche}	=> {Pan}	0.5000000		1.0	0.5000000	1.20	3

4 Conclusiones

Esta práctica está dividida en dos partes: la segunda parte consistió en investigar como cumplir los objetivos de la primera sin emplear lo enseñado en esta parte. Se ha optado por programar prácticamente todas las funciones, incluido el Algoritmo Apriori, y se ha buscado diferentes funciones con un comportamiento similar al mostrado en las clases de laboratorio.

Se ha completado todos los apartados pedidos. Con ello, se ha podido conocer un poco más en profundidad el lenguaje de R con el estudio de algunas de sus funciones y paquetes. Esta práctica ha sido la antesala para poder sumergirse en el mundo de la Ciencia de datos.

References

- [1] Roberto J. Bayardo. “Mining the most interesting rules”. In: *In Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM Press, 1999, pp. 145–154.
- [2] Christian Borgelt. *Apriori — Finding Association Rules/Hyperedges with the Apriori Algorithm*. URL: <https://borgelt.net/apriori.html>.
- [3] David Callejas and Marcelino de la Cruz. “Cómo crear paquetes en R”. In: *Ecosistemas* 29 (Apr. 2020). DOI: 10.7818/ECOS.1948.
- [4] Paulo Carvalho et al. “Information Visualization for CSV Open Data Files Structure Analysis”. In: Mar. 2015. DOI: 10.5220/0005265301010108.
- [5] Jim Hester et al. *remotes: R Package Installation from Remote Repositories, Including 'GitHub'*. R package version 2.4.1. 2021. URL: <https://CRAN.R-project.org/package=remotes>.
- [6] Paul Poncet. *modeest: Mode Estimation*. R package version 2.4.0. 2019. URL: <https://CRAN.R-project.org/package=modeest>.
- [7] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria, 2021. URL: <https://www.R-project.org/>.
- [8] Brian Lee Yung Rowe. *lambda.r: Modeling Data with Functional Programming*. R package version 1.2.4. 2019. URL: <https://CRAN.R-project.org/package=lambda.r>.
- [9] Hadley Wickham. “testthat: Get Started with Testing”. In: *The R Journal* 3 (2011), pp. 5–10. URL: https://journal.r-project.org/archive/2011-1/RJournal_2011-1_Wickham.pdf.
- [10] Hadley Wickham, Jim Hester, and Winston Chang. *devtools: Tools to Make Developing R Packages Easier*. R package version 2.4.2. 2021. URL: <https://CRAN.R-project.org/package=devtools>.