Análisis exploratorio y visualización de datos con R

Francisco Charte Ojeda

Este libro es un proyecto en curso. La presente versión fue generada el **12 de febrero de 2015**. Para obtener la última versión dirigirse a FCHARTE.COM

Primera edición, Agosto 2014

Contenidos

1. Herramientas de trabajo

En este primer capítulo conoceremos el software necesario para trabajar con R, asi como aplicaciones adicionales que pueden facilitar nuestra tarea de manera considerable, como es el caso de RStudio.

1.1 R

La denominación *R* se utiliza tanto para referirse al lenguaje R como al motor encargado de ejecutar los programas escritos en dicho lenguaje. Podemos interactuar con dicho motor mediante una consola de comandos, asi como a través de una interfaz de usuario básica aportada por el propio R.

R es una herramienta de trabajo para el tratamiento y análisis de datos. Frente a otras herrameintas similares, R nos ofrece:

- R es Open Source (multiplataforma, libre, abierto, etc.)
- Gran número de paquetes disponibles
- Extensa comunidad de usuarios
- Ciclo completo de trabajo: Implementación de algoritmos, preparación de datos, análisis de resultados y generación de documentación

1.1.1 Instalación de R

La versión binaria (ejecutables) de R está disponible para Windows[1], OS X[2] y múltiples distribuciones de GNU/Linux[3]. También es posible obtener el código fuente y compilarlo específicamente para la plataforma en la que vayamos a trabajar. Tanto binarios como fuentes se pueden descargar desde `http://www.r-project.org`.

Si nuestro sistema es Windows u OS X, una vez descargado el paquete de software no tenemos más que abrirlo y seguir las instrucciones para completar el proceso de instalación.

[1] La descarga directa del instalador para Windows se encuentra en CRAN: `http://cran.r-project.org/bin/windows/base/`.

[2] La descarga directa para OS X 10.6 y posterior se encuentra en CRAN: `http://cran.r-project.org/bin/macosx/`.

[3] La descarga directa para Debian, RedHat, Suse y Ubuntu se encuentra en CRAN: `http://cran.r-project.org/bin/linux/`.

En el caso de Linux, también podemos efectuar la instalación de R desde el repositorio de software recurriendo a la herramienta correspondiente. Si usamos Ubuntu, o algún otro derivado de Debian, usaríamos `apt-get` tal y como se aprecia en la Figura 1.1.

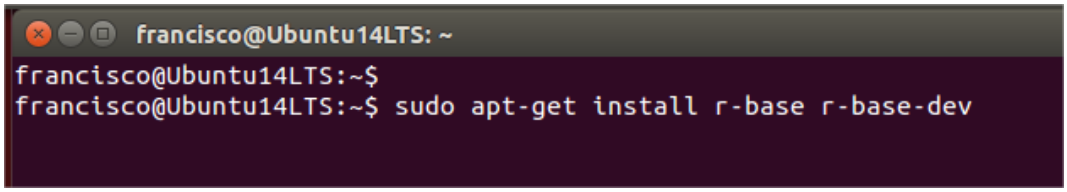


Figura 1.1: INSTALACIÓN DE R EN UBUNTU

Además del paquete adecuado para nuestro sistema operativo, también debemos tener en cuenta si necesitamos la versión de 32 o de 64 bits de R. Para esta última es imprescindible que el sistema sea también de 64 bits. La ventaja es que nos permitirá trabajar con bases de datos mucho mayores, siempre que el equipo donde usemos R tenga memoria suficiente. Para OS X únicamente está disponible la versión de 64 bits.

1.2 Herramientas integradas en R

Completada la instalación de R, en la ruta `/usr/bin` (Linux) o `\Archivos de programa\R\versión\bin` (Windows) encontraremos varios ejecutables:

- **R**[4] Es la aplicación principal de R. Con ella accederemos a la consola y ejecutaremos programas escritos en R.
- **Rscript** Motor de *scripting* de R.
- **Rgui.exe** Interfaz de usuario específica para Windows. El acceso directo que se agrega al menú **Inicio** de Windows durante la instalación apunta a este ejecutable.
- **R.app** Interfaz de usuario específica para OS X. El acceso directo a esta aplicación lo encontraremos habitualmente en la carpeta **Aplicaciones** del sistema.
- **Rterm.exe/Rcmd.exe** Ejecutables oboletos específicos de Windows, mantenidos por compatibilidad.

En los apartados de esta sección se introduce someramente cada una de las tres mecánicas de uso de R: como una consola de comandos, mediante una interfaz de usuario básica y como motor de ejecución de *scripts*.

1.2.1 La consola de R

Es la herramienta básica para operar con R de manera interactiva. Es similar a una línea de comandos del sistema operativo, pero el intérprete procesa sentencias en R en lugar de en Bash o PowerShell. La Figura 1.2 muestra la consola de R en Linux y la Figura 1.3 la misma herrameinta en Windows. A pesar de la diferente apariencia, el funcionamiento de la consola es fundamentalmente idéntico en todos los sistemas operativos.

Para ejecutar cualquier comando R, no tenemos más que introducirlo y pulsar **Intro**. Un comando puede extenderse por varias líneas. El indicador (*prompt*) mostrado en la consola cambiará de > a + siempre que se detecte que aún faltan parámetros por facilitar para completar el comando.

[4] En Linux el ejecutable no tiene extensión, en Windows la extensión será `.exe`.

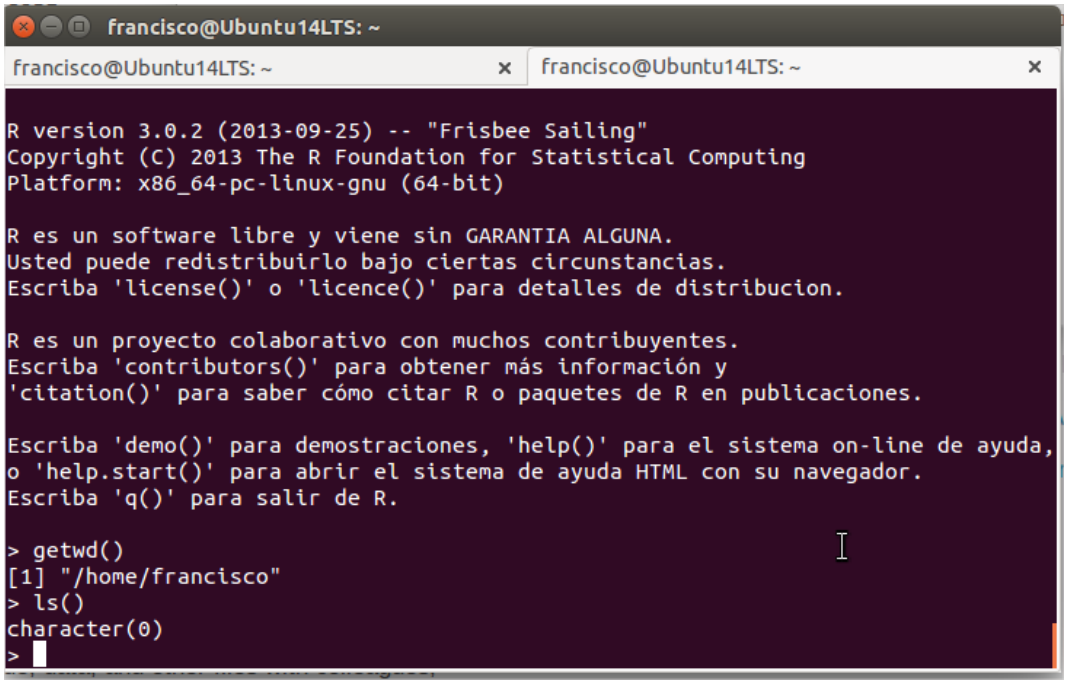


Figura 1.2: CONSOLA DE R EN LINUX

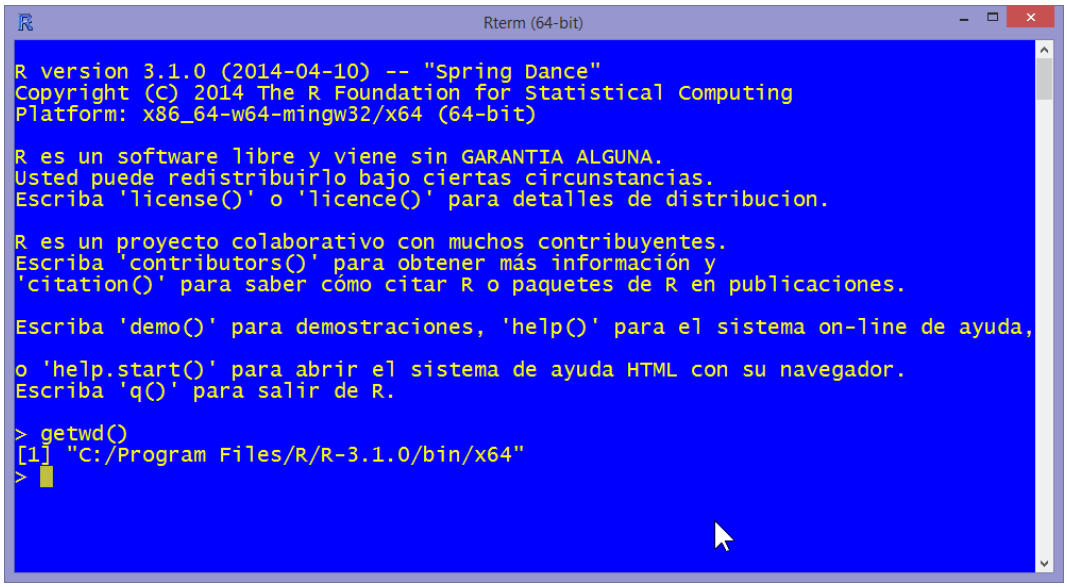


Figura 1.3: CONSOLA DE R EN WINDOWS

Puedes cerrar la consola y salir de R en cualquier momento usando el comando `quit()`. Antes de salir R nos preguntará si deseamos guardar nuestro espacio de trabajo[5], de forma que al iniciar de nuevo la consola podamos recuperar el estado en la que la dejamos.

[5]Nos ocuparemos más adelante del almacenamiento y recuperación de nuestro espacio de trabajo.

Ejecución de módulos R

Asumiendo que tenemos código R almacenado en un módulo, este puede crearse con cualquier editor de texto y, por norma, tendrá extensión `.R`, encontrándonos en la consola de R podemos ejecutarlo mediante la función `source()`, tal y como se aprecia en la Figura 1.4.

Sintaxis 1.1 `source('modulo.R'[,echo=T|F, print.eval=T|F])`

Lee el contenido de un módulo de codigo R y lo ejecuta. El parámetro `echo` determina si los comandos ejecutados serán enviados a la salida o no. El parámetro `print.eval` determina si el resultado de la ejecución se envía a la salida o no.

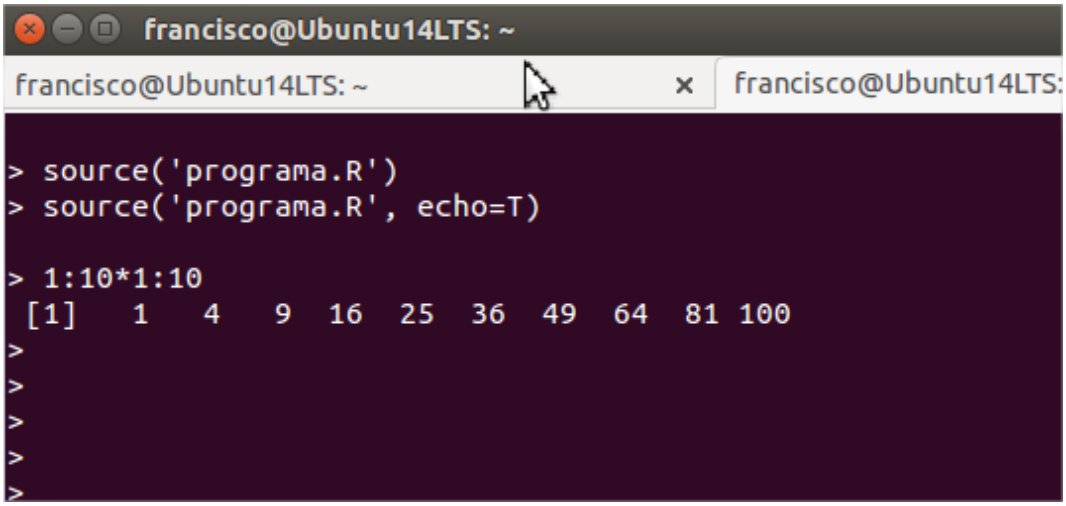


Figura 1.4: EJECUCIÓN DEL CONTENIDO DE UN MÓDULO R

Si únicamente nos interesa ejecutar el contenido del módulo y obtener el correspondiente resultado, sin interactuar con la consola de R, podemos invocar el ejecutable con las opciones CMD BATCH, facilitando dos parámetros:

- El nombre del archivo correspondiente al módulo R, incluyendo la ruta (si no está almacenado en la ruta actual) y la extensión.
- El nombre de un archivo en el que se escribirá el resultado obtenido.

La Figura 1.5 es un ejemplo de cómo utilizar R de esta forma. Además de los ya citados, R acepta muchos más parámetros que configuran el modo en que se procesará el archivo.

1.2.2 Una interfaz gráfica básica

A pesar que desde la consola de R podemos realizar cualquier tarea de forma interactiva, siempre que conozcamos los comandos adecuados y sus parámetros, algunas operaciones como la localización y almacenamiento de archivos resultan más comodas cuando se cuenta con una GUI (*Graphics User Interface*), aunque sea muy básica. Una alternativa, también relativamente sencilla, consiste en integrar R con editores como Emacs o Vim.

La versión de R para Windows incorpora una GUI específica (véase la Figura 1.6) de tipo MDI (*Multiple Document Interface*). Al iniciar esta aplicación (`Rgui.exe`) nos encontramos con la consola de R, pero como ventana hija de una ventana marco en la que encontramos múltiples opciones, entre ellas las necesarias para crear módulos de código R, instalar paquetes[6], acceder a la documentación integrada, etc.

[6]La funcionalidad inicial de R se extiende añadiendo complementos, denominados genéricamente *paquetes*, que contienen código R, bases de datos, documentación, ejemplos, etc. El repositorio de paquetes oficial de R se denomina CRAN (*The Comprehensive R Archive Network*).

Figura 1.5: EJECUCIÓN DEL CONTENIDO DE UN MÓDULO R Y ALMACENAMIENTO DEL RESULTADO

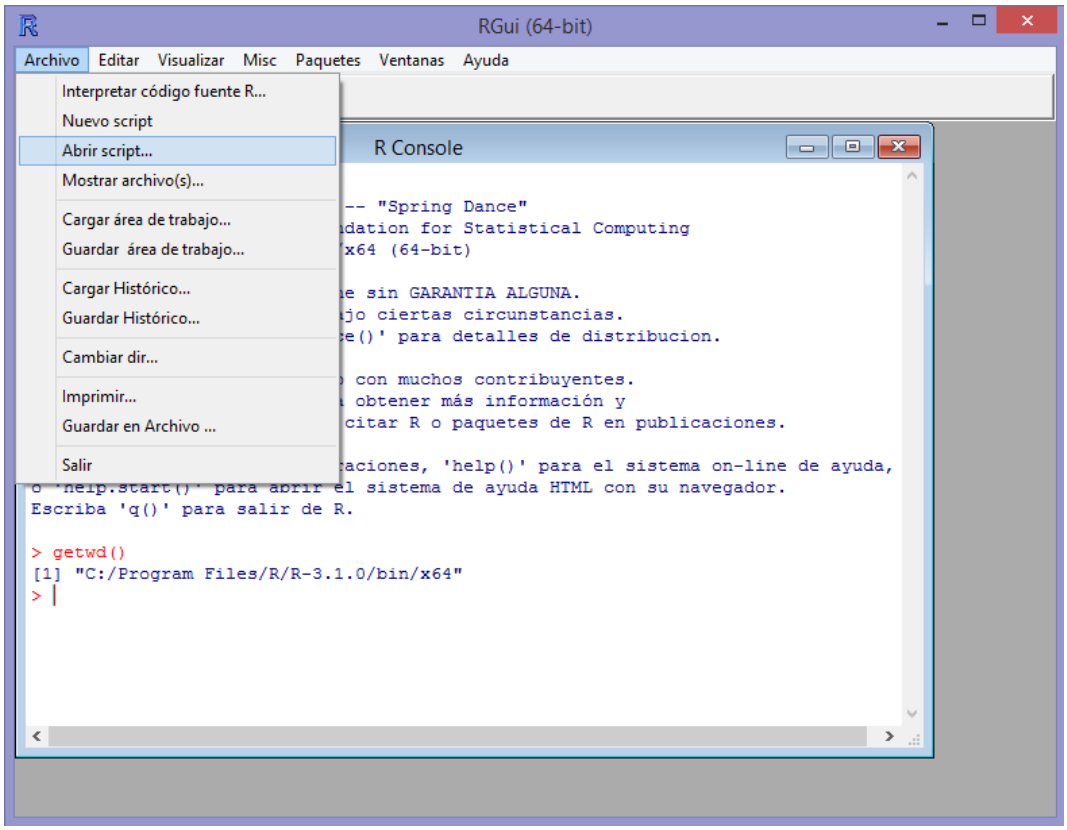


Figura 1.6: LA INTERFAZ DE USUARIO DE R EN WINDOWS

Aunque no incorpora todas las posibilidades integradas en la GUI para Windows, la versión de R para los demás sistemas también cuenta con una interfaz de usuario básica. Esta está desarrollada en Tcl/Tk y se abre añadiendo la opción `-gui=Tk` al iniciar R. Como puede

apreciarse en la Figura 1.7, esta interfaz es básicamente la consola de R con un menú de opciones muy básico, con apenas media docena de opciones y un mecanismo de acceso a la documentación integrada.

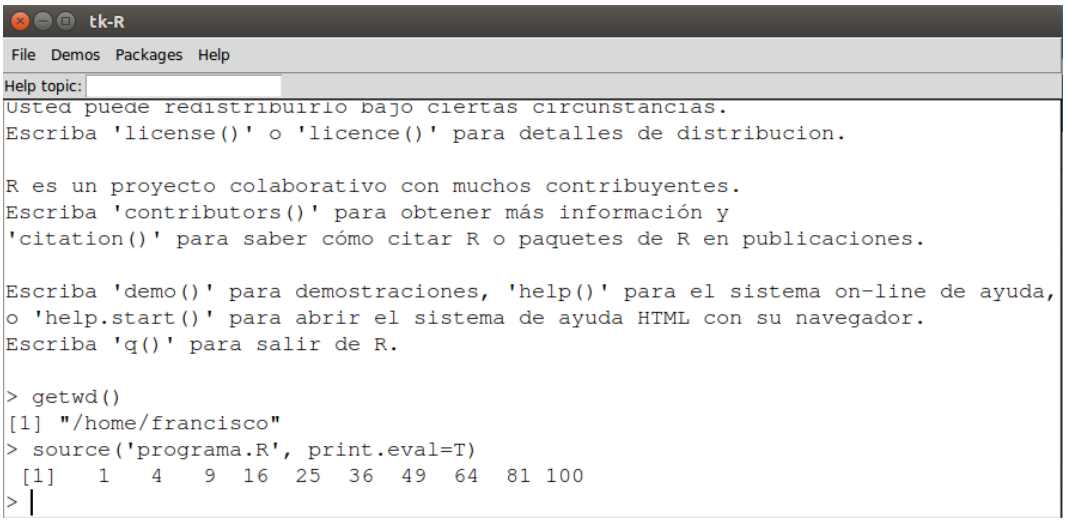


Figura 1.7: INTERFAZ DE USUARIO TK DE R EN LINUX

Al igual que en Windows, la instalación de R para OS X también incluye una interfaz de usuario específica: `R.app`. Su funcionalidad se encuentra a medio camino entre lo que nos ofrece la GUI de Linux y la de Windows. En la barra de herramientas (véase la Figura 1.8) se ofrecen opciones para crear módulos de código R, ejecutarlos, acceder a la documentación integrada, etc.

1.2.3 R como lenguaje de *script*

Además de usarse de manera interactiva, con la consola ya mencionada, o para escribir programas completos, que pueden ejecutarse según los procedimientos antes mencionados, R también puede utilizarse como un potente lenguaje de *script*. Esta es la finalidad de `RScript`, un intérprete de R que puede ser invocado facilitándole directamente el nombre de un archivo de texto conteniendo algunas sentencias R, sin más, a fin de ejecutarlas y obtener el resultado que produzcan. Por defecto ese resultado se obtendría en la salida estándar, es decir, la propia consola desde la que se ha invocado al guión. Un ejemplo podría ser el siguiente:

Ejercicio 1.1 Ejecución de un guión en R

```
Rscript programa.r
```

■

En sistemas operativos como Linux la ejecución del guión puede automatizarse, al igual que se hace con los guiones de Bash o Perl. Para ello es necesario incluir en la primera línea del archivo que contiene el codigo R la secuencia `#!/usr/bin/Rscript`[7] y dando permisos de ejecución del guión. En la Figura 1.9 puede verse un ejemplo de este uso. El guión en R, usando las funciones que conoceremos en los capítulos siguientes, podria tomar cualquier contenido que se facilite como entrada y procesarlo, extrayendo información estadística, generando gráficos, etc.

[7] En caso necesario cambiar la ruta según dónde se haya instala `Rscript` en nuestro sistema.

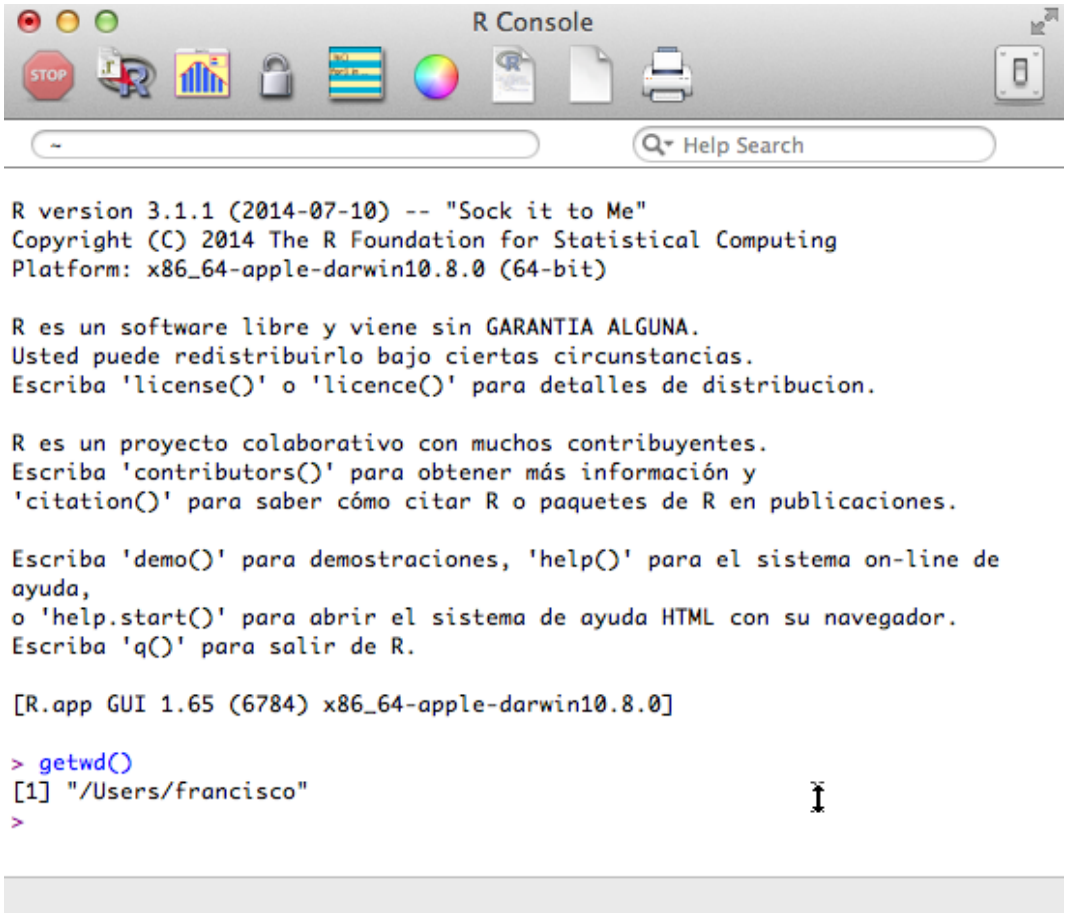


Figura 1.8: INTERFAZ DE USUARIO DE R EN OS X

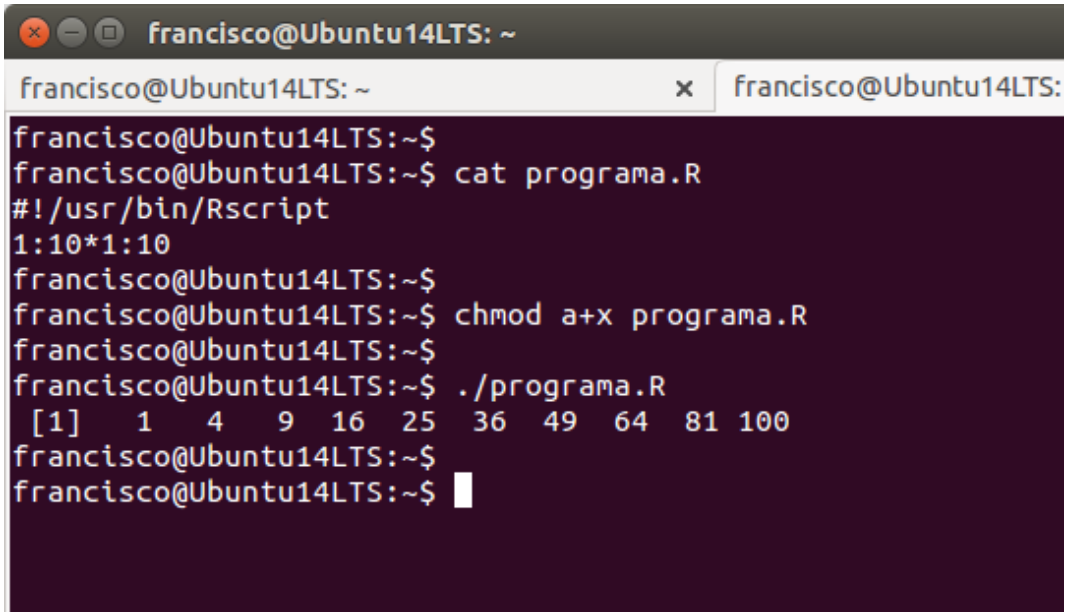


Figura 1.9: EJECUCIÓN DE UN MÓDULO R COMO SI FUESE UN *script* CUALQUIERA

1.3 RStudio

A pesar de que desde la consola de R y un editor de texto cualquiera es posible realizar cualquier tarea: escribir guiones y programa, crear nuevos paquetes, generar gráficos y documentación, etc., la mayor parte de esas tareas pueden simplificarse de manera considerable contando con un entorno de trabajo (IDE) adecuado.

RStudio es el IDE por excelencia para R. También se trata de un proyecto de código abierto, aunque puede adquirirse con una licencia comercial que incluye soporte, y está disponible para múltiples sistemas operativos.

Este no es un manual sobre RStudio, sino sobre R. Los procedimientos descritos en capítulos sucesivos pueden completarse desde la consola de R y, salvo contadas excepciones, se prescindirá de las opciones ofrecidas por la interfaz de RStudio en favor del uso de la línea de comandos.

1.3.1 Instalación de RStudio

En `http://www.rstudio.com/products/rstudio/download` encontraremos dos ediciones distintas de RStudio:

- **Desktop** Es la versión de RStudio adecuada para su ejecución local, en un equipo de escritorio o portátil, por parte de un único usuario. Cuenta con una GUI con el aspecto nativo del sistema operativo en que se instala.
- **Server** Esta versión de RStudio está diseñada para su instalación en un servidor, a fin de dar servicio a uno o más usuarios que accederían a la GUI de forma remota, desde su navegador web habitual.

De la versión para escritorio podemos tanto obtener versiones binarias de RStudio como el código fuente, en caso de que optemos por compilar nuestra propia versión de este software. Se ofrecen instaladores para Windows, OS X, Debian/Ubuntu y Fedora/Open Suse. No tenemos más que descargar la versión adecuada a nuestro sistema y ejecutar el paquete de instalación.

RStudio Server puede ser instalado directamente en Debian, Ubuntu, RedHat y CentOS. En la propia página de descarga se facilitan las instrucciones de instalación paso a paso. Para el resto de sistemas es necesario obtener el código fuente, configurarlo y compilarlo.

1.3.2 Introducción al IDE

Si hemos instalado la versión de escritorio de RStudio, para iniciarlo no tenemos más que usar el acceso directo que se habrá añadido o directamente ejecutar `RStudio` desde la consola. Se abrirá el IDE y podremos comenzar a trabajar con él. En caso de que tengamos RStudio Server instalado en un servidor, para acceder a él abriremos nuestro navegador por defecto e introduciremos el nombre o IP del servidor indicando que ha de usarse el puerto 8787. Por ejemplo: `http://localhost:8787`.

Al iniciar RStudio nos encontraremos habitualmente con tres paneles abiertos (véase la Figura 1.10):

- **Console** Ocupará la mitad izquierda de la ventana de RStudio. Es la consola de R que ya conocemos y, por tanto, la herramienta a usar para todo trabajo interactivo.
- **Environment/History** En la parte superior derecha tenemos un panel con dos páginas. En la página **Environment** irá apareciendo información sobre los objetos activos en la sesión actual: variables que se han definido, bases de datos cargadas, funciones definidas, etc. La página History es un historial de los comandos ejecutados, con opciones para recuperarlos en la consola o en un módulo de codigo R.

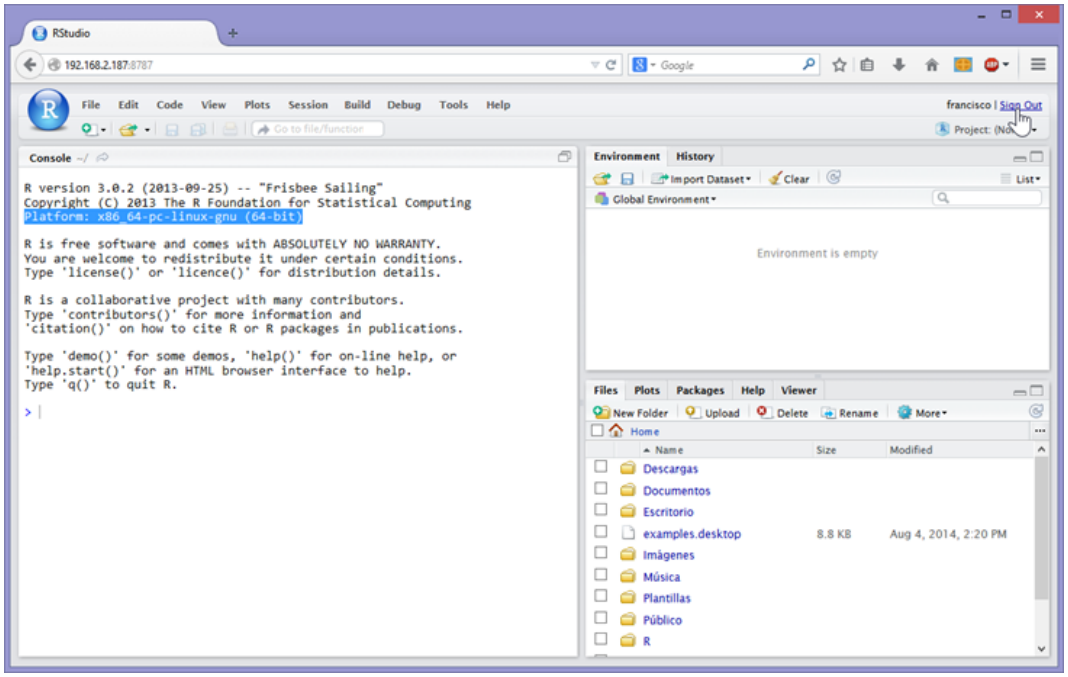


Figura 1.10: ACCESO A RSTUDIO SERVER DESDE UN NAVEGADOR EN WINDOWS

- **Files/Plots/Packages/Help/Viewer** Este panel, situado en la parte inferior derecha, ofrece páginas con opciones para acceder al sistema de archivos, visualizar gráficas, operar con paquetes R, acceder a la ayuda y obtener una vista previa de documentos.

Mediante la opción **File>New File>R Script** podemos crear tantos módulos con código R como necesitemos. Todos ellos apareceran en un nuevo panel en la mitad izquierda, quedando la consola en la parte inferior de dicha area (véase la Figura 1.11).

La edición de guiones R en RStudio cuenta con múltiples asistencias a la escritura de codigo: coloreado de código, autocompletado, ayuda sobre parámetros de funciones y miembros de objetos, etc.

Con las opciones ofrecidas en el menú de RStudio, y las barras de herramientas de los distintos paneles, podemos ejecutar módulos completos de código, instalar paquetes R, guardar las gráficas en distintos formatos, acceder a la documentación integrada, configurar el repositorio en el que residirá el código, crear nuevos paquetes, etc.

Mientras editamos código R en un módulo, podemos seleccionar una o más sentencias y pulsar **Control+Intro** para enviarlas a la consola y ejecutarlas. Esta opción nos resultará útil para ir probando porciones de código mientras escribimos un guión R.

1.4 Tareas habituales

En esta última sección del capítulo se explica cómo completar algunas tareas que necesitaremos llevar a cabo de manera habitual. Los procedimientos descritos tanto en este como en los capítulos siguientes son, salvo indicación en contrario, independientes de la herramienta que estemos utilizando. Se trata de comandos que introduciremos en la consola de R para su ejecución inmediata. Si utilizamos RStudio podemos usar el panel **Console** o,

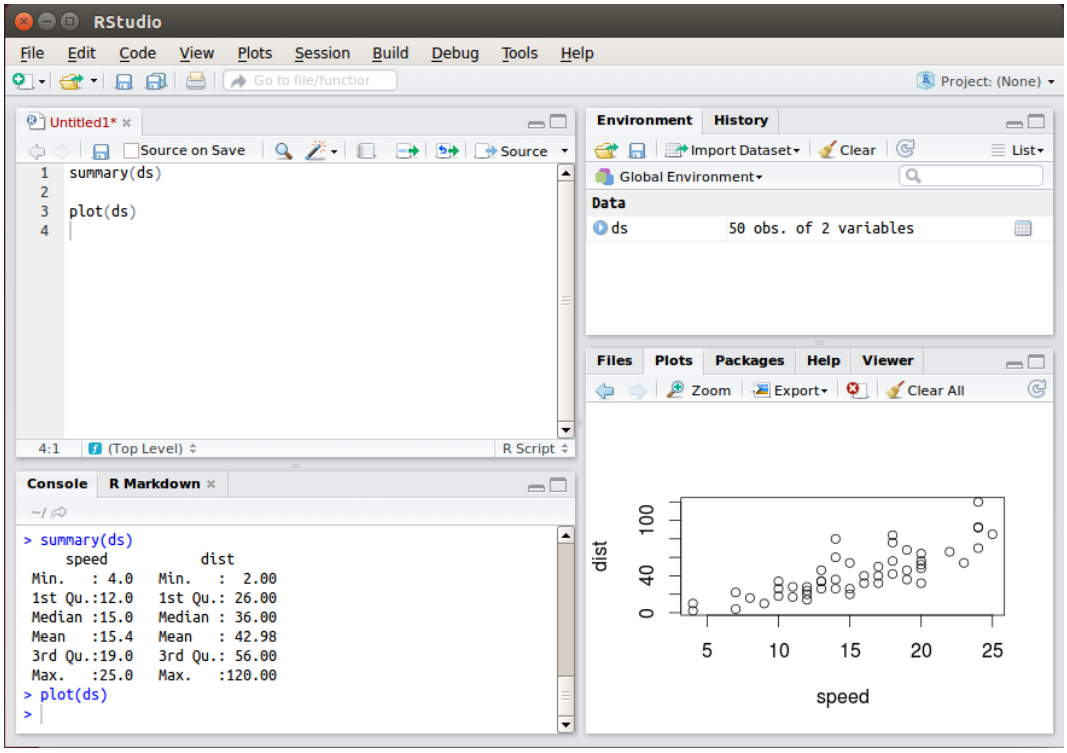


Figura 1.11: INTERFAZ DE USUARIO DE RSTUDIO EN LINUX

como alternativa, introducir las órdenes en un módulo y después ejecutarlo usando el botón **Source**[8].

1.4.1 Acceso a la ayuda

Al comenzar a trabajar con R necesitaremos información sobre cada instrucción, función y paquete. Toda la documentación se encuentra integrada, no tenemos más que usar la función `help()` para acceder a ella.

Sintaxis 1.2 `help(tema[, package = paquete])`

Facilita ayuda sobre comandos, operadores, funciones y paquetes de R. Si se incluye el parámetro `package` se buscará en el paquete indicado.

Dependiendo de la herramienta que estemos utilizando, la ayuda aparecerá directamente en la consola de R, se abrirá en nuestro navegador web por defecto o será mostrada en el panel correspondiente de RStudio. Independientemente de ello, el contenido en si sera exactamente el mismo.

Ejercicio 1.2 Acceso a la ayuda (No se muestra el resultado)

```
> help('source')
```

■

Podemos recurrir al comando `help('help')` para conocer todas las opciones de acceso a la ayuda.

[8] Este botón guardará el módulo actual y a continuación introducirá en la consola de R un comando `source('modulo.R')` para ejecutarlo.

Vignettes

Muchos paquetes R incluyen, además de la ayuda estándar accesible mediante la anterior función, documentación extendida en forma de artículos y manuales de uso, generalmente en formato PDF y HTML. Este material es accesible desde la consola de R mediante la función `vignette()`:

Sintaxis 1.3 `vignette([tema[, package = paquete]])`
Abre la documentación adicional sobre el tema indicado, generándola si fuese preciso. Si se incluye el parámetro `package` se buscará en el paquete indicado.
Al ejecutarse sin parámetros facilita una lista de todos los temas para los que hay disponibles *vignettes*.

Ejercicio 1.3 Abrir el PDF asociado al sistema de gráficos `grid`

```
> vignette('grid') # Se abrirá un PDF sobre 'grid Graphics'
```

■

Demos

El tercer recurso que nos ofrece R para aprender a usar sus posibilidades son las demostraciones, ejemplos prácticos en los que se muestran los resultados de utilizar ciertas funciones. Muchos paquetes incorporan elaboradas demostraciones de su funcionalidad, especialmente los correspondientes a gráficos.

Sintaxis 1.4 `demo([tema[, package = paquete]])`
Pone en marcha la demostración asociada al tema indicado. Si se incluye el parámetro `package` se buscará en el paquete indicado.
Al ejecutarse sin parámetros facilita una lista de todas las demos que hay disponibles.

Ejercicio 1.4 Ejecutar la demo asociada a la función `image`

```
> demo('image') # Se ejecutará la demo interactiva
```

■

1.4.2 Establecer la ruta de trabajo

Siempre que vayamos a trabajar con módulos de código R, generar gráficas, cargar datos de archivos externos, etc., hemos de tener en cuenta cuál es la ruta de trabajo actual y, si fuese preciso, cambiarla a la que sea adecuada. Esto es especialmente importante si pretendemos utilizar rutas relativas en un guión R.

La ruta actual se obtiene mediante la función `getwd()`. Esta devuelve como resultado una cadena de caracteres.

Sintaxis 1.5 `getwd()`
Devuelve la ruta de trabajo actual.

Para cambiar la ruta de trabajo usaremos la función setwd(). La ruta facilitada puede ser relativa o absoluta.

Sintaxis 1.6 `setwd(ruta)`
Cambia la ruta de trabajo según el parámetro ruta. Este será una cadena de caracteres con una ruta relativa o absoluta válida.

En el siguiente ejercicio se muestra cómo usar estas funciones para cambiar temporalmente la ruta de trabajo. `rutaPrevia` es una variable en la que guardamos la ruta actual y `<-` es el operador de asignación en R.

Ejercicio 1.5 Obtención y modificación de la ruta de trabajo actual

```
> getwd()

[1] "D:/FCharte/Estudios/CursoUNIA/book"

> rutaPrevia <- getwd()
> setwd('../data')
> getwd()

[1] "D:/FCharte/Estudios/CursoUNIA/data"

> # Trabajar con los datos y después restablecer la ruta previa
> setwd(rutaPrevia)
```

■

1.4.3 Guardar y recuperar el espacio de trabajo

Cuando se utiliza un módulo de codigo R, lo corriente es que este contenga todo lo necesario para cargar los datos que necesita, procesarlos, etc., sin depender de un estado previo. Durante una sesión de trabajo interactiva en R, por el contrario, iremos creando objetos que probablemente necesitemos en una sesión posterior. Si recrear dichos objetos requiere un cierto tiempo, porque sean obtenidos a partir de cálculos, almacenar su estado actual a fin de recuperarlo cuando se necesiten resultará mucho más eficiente.

Almacenar objetos R también nos permitirá eliminar dichos objetos de la memoria, por ejemplo antes de crear otros que precisen mayor espacio de memoria libre. En R no existe un mecanismo de liberación automática de objetos cuando estos no son necesarios, como en otros lenguajes de programación, ya que durante una sesión de trabajo interactiva es imposible saber si el usuario requerirá dichos objetos o no más adelante.

A fin de guardar, eliminar y recuperar objetos recurriremos a las tres siguientes funciones:

Sintaxis 1.7 `save(objetos, file = archivo)`

Guarda uno o más objetos en el archivo indicado, usando un formato de archivo binario y comprimido. El parámetro `file` establece la ruta y nombre del archivo en que se guardarán los objetos. Habitualmente la extensión para archivos que almacenan objetos R es `.RData`.

Sintaxis 1.8 `rm(objetos)`

Elimina de la memoria los objetos facilitados como parámetros, liberando la memoria que estos tuviesen asignada.

Sintaxis 1.9 `load(archivo)`

Recupera los objetos almacenados en un archivo de datos R y los incorpora al entorno actual. El parámetro facilitado ha de incluir la extensión del archivo y, si fuese preciso, también la ruta donde se encuentra.

En el siguiente ejercicio[9] se muestra cómo usar estos métodos durante una hipotética sesión de trabajo con R:

Ejercicio 1.6 Guardar objeto en un archivo, eliminar el objeto y recuperarlo

```
> rutaPrevia

[1] "D:/FCharte/Estudios/CursoUNIA/book"

> save(rutaPrevia, file = 'ruta.RData')
>
> rm(rutaPrevia)

> rutaPrevia

Error in eval(expr, envir, enclos) : objeto 'rutaPrevia' no
encontrado

> load('ruta.RData')
> rutaPrevia

[1] "D:/FCharte/Estudios/CursoUNIA/book"
```

■

Almacenar variables individuales tiene sentido en casos concretos, por ejemplo al trabajar con objetos complejos generados tras cálculos más o menos largos. De esta forma se puede recuperar el objeto fruto de ese procesamiento siempre que sea necesario, sin volver a realizar todos los cálculos. Habrá ocasiones, sin embargo, en los que necesitemos guardar todos los objetos existentes en la sesión de trabajo en curso, antes de cerrar la consola, a fin de poder continuar trabajando en una sesión posterior en el punto en el que lo habíamos dejado, sin partir de cero.

Aunque podríamos obtener una lista de los objetos existentes en el entorno y guardarlos con la anterior función `save()`, nos resultará más cómodo recurrir a la siguiente función:

Sintaxis 1.10 `save.image()`

Guarda todos los objetos existentes en el espacio de trabajo actual en un archivo llamado `.RData`. Dicho archivo se creará en la ruta actual (la devuelta por `getwd()`).

En realidad no tenemos que invocar a `save.image()` explícitamente justo antes de cerrar la consola o RStudio, ya que en ese momento la propia herramienta nos preguntará si deseamos guardar el estado de nuestro entorno de trabajo. Al responder afirmativamente se llamará a `save.image()`.

Al abrir RStudio en una sesión de trabajo posterior, lo primero que hará esta herramienta será cargar el estado de la sesión previa, recuperando el contenido del archivo `.RData` y recreando a partir de él todos los objetos que teníamos. De esta forma podremos seguir trabajando como si no hubiese existido interrupción alguna. También podemos cargar explícitamente dicho archivo mediante la función `load()` antes indicada.

[9] La mayoría de los ejercicios propuestos en cada capítulo de este libro dependen de los propuestos anteriormente. En este caso, por ejemplo, se asume que el objeto `rutaPrevia` ya existe, puesto que fue creado en el ejercicio previo.

Además de los objetos obtenidos a partir de la ejecución de sentencias R, también podemos guardar y recuperar la propia secuencia de órdenes que hemos usado. Trabajando en la consola de R, no tenemos más que usar la tecla de movimiento arriba para recuperar sentencias previas del historial. En caso de que estemos usando RStudio, el panel **History** nos ofrece más funciones de acceso al historial, tal y como se explica en [Pau13b]. El historial puede guardarse y recuperarse, usando para ello las dos siguientes funciones:

Sintaxis 1.11 `savehistory([file = archivo])`
Guarda el contenido del historial de trabajo en el archivo indicado. Si no se facilita un nombre de archivo, por defecto se usará `.Rhistory`. Dicho archivo se creará en la ruta actual (la devuelta por `getwd()`).

Sintaxis 1.12 `loadhistory([file = archivo])`
Recupera el contenido del historial de trabajo del archivo indicado. Si no se facilita un nombre de archivo, por defecto se usará `.Rhistory`. Dicho archivo se buscará en la ruta actual (la devuelta por `getwd()`) si no especifica otra explícitamente.

1.4.4 Cargar e instalar paquetes

El paquete base de R, siempre disponible desde que abrimos la consola, incorpora un gran abanico de funcionalidades con las que podemos cargar datos de fuentes externas, llevar a cabo análisis estadísticos y también obtener representaciones gráficas. No obstante, hay multitud de tareas para las que necesitaremos recurrir a paquetes externos, incorporando al entorno de trabajo las funciones y objetos definidos en ellos. Algunos de esos paquetes ya se encontrarán instalados en el sistema, pero otros será preciso descargarlos e instalarlos.

Cargar un paquete

Solamente pueden cargarse aquellos paquetes que estén instalados en el sistema. Tenemos dos funciones para llevar a cabo esta acción:

Sintaxis 1.13 `library(nombrePaquete)`
Carga el paquete indicado si está disponible o genera un error en caso contrario. `nombrePaquete` puede ser una cadena de caracteres o el nombre del paquete tal cual.

Sintaxis 1.14 `require(nombrePaquete)`
Carga el paquete indicado si está disponible o devuelve `FALSE` en caso contrario. `nombrePaquete` puede ser una cadena de caracteres o el nombre del paquete tal cual.

Habitualmente usaremos `library()` cuando estemos trabajando de manera interactiva, mientras que en guiones suele utilizarse `require()` comprobando el valor devuelto y evitando la interrupción abrupta del *script* por no encontrar un paquete. En el siguiente ejercicio puede apreciarse la diferencia entre ambas funciones:

Ejercicio 1.7 Cargar un paquete

```
> library(utils)  # El paquete está disponible, no hay problema
>
> library(openxlsx)  # El paquete no está disponible

Error in library(openxlsx) : there is no package called 'openxlsx'

> require(openxlsx)
```

Comprobar si un paquete está instalado

Mediante la función `installed.packages()` se obtiene información sobre todos los paquetes instalados en R:

Sintaxis 1.15 `installed.packages()`
Devuelve una matriz con información relativa a los paquetes instalados actualmente.

En lugar de examinar visualmente todos los elementos devueltos por la anterior función, podemos usar la función `is.element()` para verificar la presencia de un paquete concreto:

Sintaxis 1.16 `is.element(elemento, conjunto)`
Comprueba si elemento aparece en el conjunto o no, devolviendo `TRUE` o `FALSE` respectivamente.

Utilizando las dos anteriores funciones es fácil escribir una propia que nos facilite la comprobación de si un paquete está instalado o no. Es lo que se hace en el siguiente ejercicio, en el que definimos una nueva función a la que llamamos `is.installed()`. Esta precisa como único parámetro el nombre del paquete que nos interesa, devolviendo `TRUE` o `FALSE` según corresponda.

Ejercicio 1.8 Comprobar si un paquete está instalado

```
> is.installed <- function(paquete) is.element(
+   paquete, installed.packages())
>
> is.installed('XLConnect')

[1] TRUE
```

Instalar un paquete

Cualquier paquete disponible en el repositorio oficial (CRAN) puede ser instalado fácilmente desde el propio R. No hay más facilitar su nombre a la función `install.packages()`:

Sintaxis 1.17 `install.packages(paquetes, repos=repositorio)`
Busca, descarga e instala los paquetes indicados por el parámetro `paquetes`. Por defecto se buscará en CRAN, pero es posible facilitar la dirección de otro repositorio mediante el parámetro `repos`.

En el siguiente ejercicio se comprueba si un cierto paquete está instalado antes de cargarlo, procediendo a su instalación su fuese preciso:

Ejercicio 1.9 Cargar un paquete, verificando antes si está disponible e instalándolo en caso necesario

```
> if(!is.installed('sos'))  # Ayuda para buscar en paquetes
+   install.packages("sos")
>
```

```
> library("sos")
```

■

En este ejercicio se ha utilizado el condicional `if` para comprobar el valor devuelto por la función `is.installed()`. El simbolo `!` es el operador `NOT` en R.

Ejercicio 1.10 Cargar un paquete desde un archivo local descargado previamente

```
> install.packages('<pathtopackage>', repos = NULL, type = 'source')
```

■

Uno de los problemas a los que solemos enfrentarnos los usuarios de R es cómo saber qué paquete debemos instalar para satisfacer una cierta necesidad. Es una situación en la que nos será de utilidad el paquete sos instalado en el ejercicio previo. Dicho paquete facilita una función que se encargará de buscar en todos los paquetes disponibles en CRAN la cadena que facilitemos como parámetro, generando como resultado una página web (que se abrirá automáticamente en nuestro navegador por defecto) con información de cada paquete y enlaces a páginas con documentación adicional. De esta forma es fácil decidir qué paquete nos interesa. El siguiente ejercicio muestra cómo utilizar la citada función:

Ejercicio 1.11 Buscar información de paquetes relacionados con Excel

```
> findFn("excel")
```

■

2. Tipos de datos (I)

En R prácticamente todos los datos pueden ser tratados como objetos, incluidos los tipos de datos más simples como son los números o los caracteres. Entre los tipos de datos disponibles tenemos vectores, matrices, *factors*, `data frames` y listas.

El objetivo del presente capítulo es el de introducir todos esos tipos de datos y familiarizarnos con su uso a través de algunos ejercicios.

2.1 Tipos de datos simples

Los tipos de datos simples o fundamentales en R son los siguientes:

- `numeric`: Todos los tipos numéricos, tanto enteros como en coma flotante y los expresados en notación exponencial, son de esta clase. También pertenecen a ellas las constantes `Inf` y `NaN`. La primera representa un valor infinito y la segunda un valor que no es numérico.

  > Aunque todos los tipos numéricos comparten una misma clase, que es `numeric`, el tipo de dato (que determina la estructura interna del almacenamiento del valor) es `double`. Distinguiremos entre clase y tipo más adelante.

- `integer`: En R por defecto todos los tipos numéricos se tratan como `double`. El tipo `integer` se genera explícitamente mediante la función `as.integer()`. El objetivo es facilitar el envío y recepción de datos entre código R y código escrito en C.
- `complex`: Cualquier valor que cuente con una parte imaginaria, denotada por el sufijo **i**, será tratado como un número complejo.
- `character`: Los caracteres individuales y cadenas de caracteres tienen esta clase.Se delimitan mediante comillas simples o dobles.
- `logical`: Esta es la clase de los valores booleanos, representados en R por las constantes `TRUE` y `FALSE`.

2.1.1 Clase y tipo de un dato

Para los tipos de datos simples, en general la clase y el tipo coinciden salvo en el caso de datos numéricos no enteros, cuya clase es `numeric` siendo su tipo `double`. Podemos obtener

la clase y tipo de cualquier dato, ya sea constante o una variable, mediante las dos siguientes funciones:

Sintaxis 2.1 `class(objeto)`

Devuelve[a] un vector con los nombres de las clases a las que pertenece el objeto.

[a] También es posible asignar un valor, según la sintaxis `class(objeto) <- 'clase'`, lo cual permite cambiar la clase de un objeto

Sintaxis 2.2 `typeof(objeto)`

Devuelve una cadena indicando el tipo de dato interno usado por el objeto.

El siguiente ejercicio utiliza las dos funciones citadas para comprobar cuál es la clase y el tipo de varios datos:

Ejercicio 2.1 Comprobación de la clase y tipo de distintos datos simples

```
> class(45)

[1] "numeric"

> class(34.5)

[1] "numeric"

> class("R")

[1] "character"

> class(TRUE)

[1] "logical"

> class(Inf)

[1] "numeric"

> class(1+2i)

[1] "complex"

> class(NaN)

[1] "numeric"

> typeof(45)

[1] "double"

> typeof(34.5)

[1] "double"

> typeof("R")
```

```
[1] "character"

> typeof(TRUE)

[1] "logical"

> typeof(Inf)

[1] "double"

> typeof(1+2i)

[1] "complex"

> typeof(NaN)

[1] "double"
```

2.1.2 Almacenamiento de valores en variables

Las variables en R no se definen ni declaran previamente a su uso, creándose en el mismo momento en que se les asigna un valor. El operador de asignación habitual en R es `<-`, pero también puede utilizarse `=` y `->` tal y como se aprecia en el siguiente ejercicio. Introduciendo en cualquier expresión el nombre de una variable se obtendrá su contenido. Si la expresión se compone únicamente del nombre de la variable, ese contenido aparecerá por la consola.

Ejercicio 2.2 Asignación de valores a una variable

```
> a <- 45
> a

[1] 45

> a = 3.1416
> a

[1] 3.1416

> "Hola" -> a
> a

[1] "Hola"
```

2.1.3 Comprobar el tipo de una variable antes de usarla

Aunque no es una necesidad habitual mientras se trabaja interactivamente con R, al escribir funciones y paquetes sí que es necesario comprobar el tipo de un dato antes de proceder a utilizarlo. De esta manera se evitan errores en caso de recibir un parámetro de tipo inadecuado. La comprobación la llevaremos a cabo con alguna de las funciones `is.tipo()`:

Sintaxis 2.3 `is.TIPO(objeto)`

Las funciones `is.numeric()`, `is.character()`, `is.integer()`, `is.infinite()` e `is.na()` comprueban si el objeto entregado como parámetro es del tipo correspondiente, devolviendo `TRUE` en caso afirmativo o `FALSE` en caso contrario.

2.1.4 Objetos en el espacio de trabajo

A medida que vayamos almacenando valores en variables, estas quedarán vivas en nuestro espacio de trabajo hasta en tanto no cerremos la consola. En RStudio el panel **Environment** facilita una lista de todos los objetos existentes. Desde la consola, podemos recurrir a la función `ls()` para obtener esa misma lista:

Sintaxis 2.4 `ls([entorno,pattern=])`

Facilita un vector de cadenas de caracteres conteniendo los nombres de los objetos existentes en el entorno. Esta función puede opcionalmente tomar varios parámetros, cuyo objeto es establecer el entorno cuyo contenido se quiere obtener y establecer filtros par solamente recuperar objetos cuyos nombres se ajustan a un cierto patrón.

Tal y como se explicó en el capítulo previo, es posible almacenar el contenido de los objetos en un archivo y recuperarlo con posterioridad. También podemos eliminar cualquier objeto existente en el entorno, usando para ello la función `rm()`:

Sintaxis 2.5 `rm(objetos)`

Elimina del entorno los objetos cuyos nombres se facilitan como parámetros, liberando la memoria ocupada por estos.

En ejercicios previos ya hemos creado algunas variables. El propuesto a continuación enumera esas variables y elimina una de ellas:

Ejercicio 2.3 Enumeración de objetos en el entorno y eliminación

```
> ls()

[1] "a"             "is.installed" "rutaPrevia"

> rm(a)
> ls()

[1] "is.installed" "rutaPrevia"
```

■

La función `is.installed()`, que definíamos en un ejercicio anterior, también es una variable, concretamente de tipo `function`. Por esa razón aparece en la lista facilitada por `ls()`. Como variable que es, puede ser eliminada mediante la función `rm()`.

2.2 Vectores

Los vectores en R contienen elementos todos del mismo tipo, accesibles mediante un índice y sin una estructura implícita: por defecto no hay dimensiones, nombres asignados a los elementos, etc. A partir de un vector es posible crear otros tipos de datos que sí tienen estructura, como las matrices o los *data frames*.

2.2.1 Creación de vectores

Existe un gran abanico de funciones para generar vectores con distintos contenidos. La más usada es la función `c()`:

Sintaxis 2.6 `c(par1, ..., parN)`
Crea un vector introduciendo en él todos los parámetros recibidos y lo devuelve como resultado.

En el siguiente ejercicio puede comprobarse cómo se utiliza esta función. Aunque en este caso el resultado se almacena en sendas variables, podría en su lugar mostrarse directamente por la consola o ser enviado a una función.

Ejercicio 2.4 Creación de vectores facilitando una enumeración de los elementos

```
> diasMes <- c(31,29,31,30,31,30,31,31,30,31,30,31)
> dias <- c('Lun','Mar','Mié','Jue','Vie','Sáb','Dom')
>
> diasMes

 [1] 31 29 31 30 31 30 31 31 30 31 30 31

> dias

[1] "Lun" "Mar" "Mié" "Jue" "Vie" "Sáb" "Dom"
```

■

Si los valores a introducir en el vector forman una secuencia de valores consecutivos, podemos generarla utilizando el operador `:` en lugar de enumerarlos todos individualmente. También podemos recurrir a las funciones `seq()` y `rep()` para producir secuencias y vectores en los que se repite un contenido.

Sintaxis 2.7 `seq(from=inicio, to=fin[, by=incremento] [,length.out=longitud])`
Crea un vector con valores partiendo de inicio y llegando como máximo a fin. Si se facilita el parámetro `by`, este será aplicado como incremento en cada paso de la secuencia. Con el parámetro `length.out` es posible crear vectores de una longitud concreta, ajustando automáticamente el incremento de cada paso.

Sintaxis 2.8 `rep(valor, veces)`
Crea un vector repetiendo el objeto entregado como primer parámetro tantas veces como indique el segundo.

El siguiente ejercicio muestra algunos ejemplos de uso de las anteriores funciones, generando varios vectores con distintos contenidos:

Ejercicio 2.5 Creación de vectores facilitando una enumeración de los elementos

```
> quincena <- 16:30
> quincena

 [1] 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
```

```
> semanas <- seq(1,365,7)
> semanas

 [1]   1   8  15  22  29  36  43  50  57  64  71  78  85  92
[15]  99 106 113 120 127 134 141 148 155 162 169 176 183 190
[29] 197 204 211 218 225 232 239 246 253 260 267 274 281 288
[43] 295 302 309 316 323 330 337 344 351 358 365

> rep(T,5)

[1] TRUE TRUE TRUE TRUE TRUE

> c(rep(T,5),rep(F,5))

 [1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE
[10] FALSE

> rep(c(T,F), 5)

 [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
[10] FALSE
```

2.2.2 Acceso a los elementos de un vector

Podemos saber cuántos elementos contiene un vector mediante la función `length()`. Esta información nos será útil a la hora de acceder a cualquiera de los elementos del vector, usando para ello la notación `vector[elemento]`.

Sintaxis 2.9 `length(vector)`
`length(vector) <- longitud`

Devuelve un entero indicando el número de elementos contenidos en el vector. También permite establecer el tamaño de un vector.

Al acceder al contenido de un vector se pueden obtener uno o más elementos, facilitando entre corchetes uno o más índices. Para usar más de un índice es necesario facilitar un vector entre los corchetes. También pueden utilizarse números negativos que, en este contexto, indican qué elementos no desean obtenerse. Algunos ejemplos:

Ejercicio 2.6 Acceso al contenido de un vector

```
> length(dias)

[1] 7

> length(semanas)

[1] 53

> dias[2]        # Solo el segundo elemento

[1] "Mar"
```

```
> dias[-2]        # Todos los elementos menos el segundo

[1] "Lun" "Mié" "Jue" "Vie" "Sáb" "Dom"

> dias[c(3,7)]        # Los elementos 3 y 7

[1] "Mié" "Dom"
```

■

Los valores simples, como la variable a que creábamos en un ejercicio previo, también son vectores, concretamente vectores de un solo elemento. A pesar de ello, podemos utilizar con estos datos la función `length()` y el operador `[]` para acceder a su contenido, incluso con valores constantes:

Ejercicio 2.7 Datos simples como vectores

```
> length(5)

[1] 1

> 5[1]

[1] 5
```

■

2.2.3 Generación de vectores aleatorios

En ocasiones es necesario crear vectores con valores que no son secuencias ni valores repetidos, sino a partir de datos aleatorios que siguen una cierta distribución. Es usual cuando se diseñan experimentos y no se dispone de datos reales. R cuenta con un conjunto de funciones capaces de generar valores aleatorios siguiendo una distribución concreta. Las dos más usuales son `rnorm()`, asociada a la distribución normal, y `runif()`, que corresponde a la distribución uniforme:

Sintaxis 2.10 `rnorm(longitud[,mean=media][,sd=desviación])`

Genera un vector de valores aleatorios con el número de elementos indicado por `longitud`. Por defecto se asume que la media de esos valores será 0 y la desviación 1, pero podemos usar los parámetros `mean` y `sd` para ajustar la distribución.

Sintaxis 2.11 `runif(longitud[,min=mínimo][,max=máximo])`

Genera un vector de valores aleatorios con el número de elementos indicado por `longitud`. Por defecto se asume que el valor mínimo será 0 y el máximo 1, pero podemos usar los parámetros `min` y `max` para ajustar la distribución.

Antes de utilizar estas funciones, podemos establecer la semilla del generador de valores aleatorios a fin de que el experimento sea reproducible. Para ello recurriremos a la función `set.seed()`:

Sintaxis 2.12 `set.seed(semilla)`

Inicializa el algoritmo de generación de valores aleatorios con la semilla facilitada como parámetro.

La diferencia entre los valores generados por las dos anteriores funciones puede apreciarse en la Figura 2.1. En ella aparecen dos histogramas representando un conjunto de 1000 valores producidos por `rnorm()` (arriba) y `runif()`.

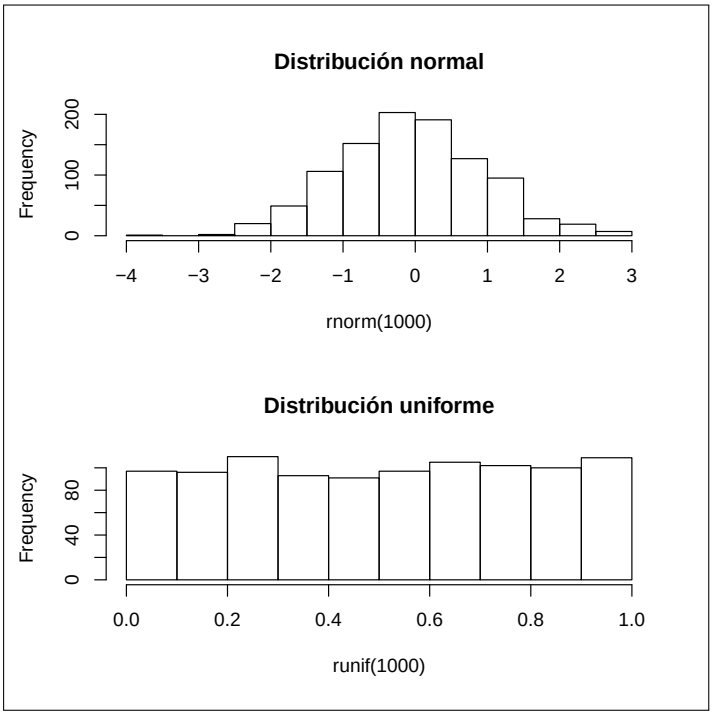


Figura 2.1: DISTRIBUCIÓN DE LOS VALORES GENERADOS POR `RNORM()` Y `RUNIF()`

En el siguiente ejercicio se muestra cómo establecer la semilla para el generador de números aleatorios y cómo obtener sendos vectores de valores generados por las dos funciones citadas:

Ejercicio 2.8 Generación de números aleatorios

```
> set.seed(4242)
> rnorm(100, mean=10, sd=3)

  [1] 16.601503  7.782490 11.597301 12.759985 11.251900
  [6]  7.049153  4.087217 12.464106  7.505729  9.416750
 [11]  5.900797  6.930702 11.462176 13.634359 11.915339
 [16]  8.320269 12.219524  4.553933 14.626155  9.829156
 [21] 10.867099  7.073053  9.469537  9.398100  9.629391
 [26] 12.712182  5.536700 16.427711  4.279690 12.181423
 [31]  4.775551  6.310720 14.329029 11.314088  5.968193
 [36] 13.372781  5.319390  5.886031 19.273538  7.377150
 [41]  9.222643  8.400000 16.307809  8.762114  6.517521
 [46]  7.350390 15.326032  9.000971 10.128321 10.220721
 [51] 12.423237  9.295735  7.661076 10.664515 15.588475
 [56] 15.646877  9.293694  9.181816 14.327371  9.470363
 [61]  6.989486 11.248542 11.961460 12.153842  6.076185
```

```
 [66]  9.772697 10.623449 10.217452 12.734511 15.803807
 [71] 13.701893  7.093037 16.272279 11.458514  8.253390
 [76] 10.349666 10.063023  9.581695 10.499814  9.124369
 [81]  8.715328 11.906492  6.666753 15.347326 12.292075
 [86]  6.786419 13.456128  5.749136 10.983657  2.637713
 [91]  9.446325  9.363261  9.336171 15.940844  9.236966
 [96]  6.066087 10.272637 12.942529  8.637694 12.164843

> loto <- as.integer(runif(6, min=1, max=49))
> loto

[1] 15  9 22 43 16 15
```

■

2.2.4 Operar sobre vectores

La mayor parte de funciones y operadores con que cuenta R están preparados para actuar sobre vectores de datos. Esto significa que no necesitamos codificar un bucle a fin de aplicar una operación a cada elemento de forma individual. Si bien esto resulta totalmente posible, en general el rendimiento es considerablemente peor.

Supongamos que tenemos dos vectores con un gran volumen de datos y que precisamos operar a fin de obtener un nuevo vector, en el cada elemento contenga por cada elemento de los originales el cuadrado del primero más el segundo. Es la operación que se efectúa en el siguiente ejercicio, primero recorriendo los elementos y efectuando la operación de manera individual y después operando sobre el vector completo como un todo:

Ejercicio 2.9 Operar sobre todos los elementos de un vector

```
> vect1 <- rnorm(100000)
> vect2 <- rnorm(100000)
>
> vect3 <- c()  # El vector de resultados está inicialmente vacío
> system.time(for(idx in 1:length(vect1))
+     vect3[idx] <- vect1[idx] * vect1[idx] + vect2[idx]
+ )

   user  system elapsed
  23.71    0.20   23.91

> system.time(vect4 <- vect1 * vect1 + vect2)

   user  system elapsed
      0       0       0

> stopifnot(vect3 == vect4) # Resultados deben ser idénticos
```

■

La sentencia `for` es similar al bucle del mismo nombre existente en multitud de lenguajes de programación. En este caso la variable `idx` tomará los valores en el intervalo que va de 1 al número de elementos del primer vector. A cada ciclo se hace el cálculo para un elemento

del vector. Mediante la función `system.time()` obtenemos el tiempo que ha tardado en ejecutarse la expresión facilitada como parámetro, en este caso todo el bucle.

En la segunda llamada a `system.time()` podemos comprobar cómo se opera sobre vectores completos. La sintaxis es sencilla, solamente hemos de tener en cuenta que los operadores, en este caso `*` y `+`, actúan sobre cada elemento. La sentencia final verifica que los resultados obtenidos con ambos métodos son idénticos, de no ser así la ejecución se detendría con un error.

En R el operador para comprobar la igualdad entre dos operandos es == no =, como en la mayoría de lenguaje derivados de C.

Sintaxis 2.13 `for(variable in secuencia) sentencia`

Recorre todos los valores en la secuencia. A cada ciclo la `variable` tomará un valor y se ejecutará la `sentencia`. Si necesitamos ejecutar varias sentencias podemos encerrarlas entre llaves.

Sintaxis 2.14 `system.time(expresión)`

Ejecuta la expresión y devuelve el tiempo que se ha empleado en ello.

Sintaxis 2.15 `stopifnot(expresion1, ..., expresionN)`

Verifica que todas las expresiones indicadas devuelven `TRUE`, en caso contrario se genera un error y la ejecución se detiene.

2.3 Matrices

Una matriz R no es más que un vector que cuenta con un atributo llamado `dim` indicando el número de filas y columnas de la matriz. Se trata, por tanto, de una estructura de datos en la que todos los elementos han de ser del mismo tipo. Podemos crear una matriz a partir de un vector, asi como indicando el número de filas y columnas dejando todos sus elementos vacíos. También cabe la posibilidad de agregar manualmente el citado atributo `dim` a un vector, convirtiéndolo en una matriz.

2.3.1 Creación de una matriz

La función encargada de crear una nueva matriz es `matrix()`:

Sintaxis 2.16 `matrix(vector[, nrow=numFilas, ncol=numCols, byrow=TRUE|FALSE], dimnames=nombres)`

Crea una matriz a partir del vector de datos entregado como primer parámetro. Si no es posible deducir a partir de la longitud del vector el número de filas y columnas, estos son parámetros que también habrá que establecer. El argumento `byrow` determina si los elementos del vector se irán asignando por filas o por columnas. El uso de los nombres para las dimensiones se detalla más adelante.

Al igual que ocurre con los vectores, la función `length()` devuelve el número de elementos de una matriz. Para conocer el número de filas y columnas podemos usar las función `nrow()`, `ncol()` y `dim()`.

Sintaxis 2.17 `nrow(matriz)`

Devuelve un entero indicando el número de filas que tiene la matriz.

Sintaxis 2.18 `ncol(matriz)`
Devuelve un entero indicando el número de columnas que tiene la matriz.

Sintaxis 2.19 `dim(matriz)`
Devuelve un vector con el número de filas y columnas que tiene la matriz. También es posible asignar un vector a fin de modificar la estructura de la matriz.

A continuación se muestran varias forma distintas de crear una matriz:

Ejercicio 2.10 Creación de matrices y obtención de su estructura

```
> mes <- matrix(1:35,ncol=7)  # Dos formas de generar exactamente
> mes <- matrix(1:35,nrow=5) # la misma matriz
> mes

     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]    1    6   11   16   21   26   31
[2,]    2    7   12   17   22   27   32
[3,]    3    8   13   18   23   28   33
[4,]    4    9   14   19   24   29   34
[5,]    5   10   15   20   25   30   35

> mes <- matrix(1:35,nrow=5,ncol=7,byrow=T)
> mes

     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]    1    2    3    4    5    6    7
[2,]    8    9   10   11   12   13   14
[3,]   15   16   17   18   19   20   21
[4,]   22   23   24   25   26   27   28
[5,]   29   30   31   32   33   34   35

> length(mes)

[1] 35

> nrow(mes)

[1] 5

> ncol(mes)

[1] 7

> dim(mes)

[1] 5 7
```

■

Podemos determinar si un cierto objeto es o no una matriz mediante la función `is.matrix()`. También tenemos a nuestra disposición una función para convertir objetos de otros tipos en matrices: `as.matrix()`.

Sintaxis 2.20 `is.matrix(objeto)`
Devuelve `TRUE` si el objeto es una matriz o `FALSE` en caso contrario.

Sintaxis 2.21 `as.matrix(objeto)`
Intenta convertir el objeto facilitado como parámetro en una matriz.

En el siguiente ejemplo se muestra cómo es posible generar una matriz a partir de un vector, sencillamente estableciendo sus dimensiones mediante la función `dim()`.

Ejercicio 2.11 Conversión de un vector en una matriz

```
> is.matrix(vect4)

[1] FALSE

> dim(vect4)

NULL

> dim(vect4) <- c(1000, 100)
> is.matrix(vect4)

[1] TRUE

> dim(vect4)

[1] 1000  100
```

■

2.3.2 Acceso a los elementos de una matriz

El acceso a los elementos de una matriz es similar al caso de los vectores. Se usa el mismo operador `[]`, pero en este caso se esperan dos parámetros separados por una coma: la fila y la columna. Cualquiera de ellos puede obviarse, recuperando filas y columnas completas. También es posible facilitar vectores de índices.

Ejercicio 2.12 Acceso a los elementos de una matriz

```
> mes[2,5]  # Quinto elemento de la segunda fila

[1] 12

> mes[2,]   # Segunda fila completa

[1]  8  9 10 11 12 13 14

> mes[,2]   # Segunda columna completa
```

```
[1]  2  9 16 23 30
```

■

2.3.3 Columnas y filas con nombre

Por defecto las filas y columnas de una matriz tienen asignado únicamente un índice, pero es posible establecer nombres en ambos dimensiones. Esa es la finalidad del parámetro `dimnames` de la función `matrix()`. También podemos usar las funciones `rownames()` y `colnames()` tanto para obtener como para establecer los nombres asociados a filas y columnas, respectivamente.

Sintaxis 2.22 `rownames(matriz)`
`rownames(matriz) <- nombres`

Devuelve o establece los nombres asociados a las filas de la matriz. Estos se facilitan como un vector de cadenas de caracteres.

Sintaxis 2.23 `colnames(matriz)`
`colnames(matriz) <- nombres`

Devuelve o establece los nombres asociados a las columnas de la matriz. Estos se facilitan como un vector de cadenas de caracteres.

Los nombres asignados a filas y columnas no solamente sirven como descripción de la estructura de la matriz, apareciendo cuando esta se muestra en la consola, sino que también pueden ser utilizados para acceder a los elementos de la matriz. Por ejemplo:

Ejercicio 2.13 Asignación de nombres a columnas y filas

```
> rownames(mes) <- c('Semana1','Semana2','Semana3',
+                    'Semana4','Semana5')
> colnames(mes) <- dias
> mes

        Lun Mar Mié Jue Vie Sáb Dom
Semana1   1   2   3   4   5   6   7
Semana2   8   9  10  11  12  13  14
Semana3  15  16  17  18  19  20  21
Semana4  22  23  24  25  26  27  28
Semana5  29  30  31  32  33  34  35

> attributes(mes)

$dim
[1] 5 7

$dimnames
$dimnames[[1]]
[1] "Semana1" "Semana2" "Semana3" "Semana4" "Semana5"

$dimnames[[2]]
[1] "Lun" "Mar" "Mié" "Jue" "Vie" "Sáb" "Dom"
```

```
> mes[,'Jue']

Semana1 Semana2 Semana3 Semana4 Semana5
      4      11      18      25      32

> mes['Semana4',]

Lun Mar Mié Jue Vie Sáb Dom
 22  23  24  25  26  27  28

> mes['Semana2', 'Vie']

[1] 12
```

Es posible editar interactivamente, en un sencillo editor, el contenido de una matriz utilizando la función `fix()`. Esta precisa el nombre de la matriz como único parámetro.

2.4 Factors

Al trabajar con bases de datos es habitual que tengamos que operar con datos de tipo categórico. Estos se caracterizan por tener asociada una descripción, una cadena de caracteres, y al mismo tiempo contar con un limitado número de valores posibles. Almacenar estos datos directamente como cadenas de caracteres implica un uso de memoria innecesario, ya que cada uno de las apariciones en la base de datos puede asociarse con un índice numérico sobre el conjunto total de valores posibles, obteniendo una representación mucho más compacta. Esta es la finalidad de los *factors* en R.

Desde una perspectiva conceptual un *factor* de R sería equivalente a un tipo enumerado de de C++/Java y otros lenguajes de programación.

Por tanto, un *factor* internamente se almacena como un número. Las etiquetas asociadas a cada valor se denomina niveles. Podemos crear un *factor* usando dos funciones distintas `factor()` y `ordered()`. El número de niveles de un factor se obtiene mediante la función `nlevels()`. El conjunto de niveles es devuelto por la función `level()`.

Sintaxis 2.24 `factor(vectorDatos[,levels=niveles])`
Genera un vector a partir del vector de valores facilitado como entrada. Opcionalmente puede establecerse el conjunto de niveles, entregándolo como un vector de cadenas de caracteres con el argumento `levels`.

Sintaxis 2.25 `ordered(vectorDatos)`
Actúa como la anterior función, pero estableciendo una relación de orden entre los valores del *factor*. Esto permitirá, por ejemplo, usar operadores de comparación entre los valores.

Sintaxis 2.26 `nlevels(factor)`
Devuelve un entero indicando el número de niveles con que cuenta el *factor*.

Sintaxis 2.27 `levels(factor)`
Devuelve el conjunto de niveles asociado al *factor*.

En el siguiente ejercicio se crea un vector de 1000 elementos, cada uno de los cuales contendrá un valor que representa un día de la semana. Como puede apreciarse, la ocupación de memoria cuando se utiliza un *factor* es muy importante.

Ejercicio 2.14 Definición y uso de *factors*

```
> mdias <- c(dias[as.integer(runif(1000,0,7)+1)])
> mdias[1:10]

 [1] "Jue" "Mar" "Mar" "Jue" "Jue" "Mar" "Mar" "Lun" "Jue"
[10] "Sáb"

> object.size(mdias)

8376 bytes

> fdias <- factor(mdias)
> fdias[1:10]

 [1] Jue Mar Mar Jue Jue Mar Mar Lun Jue Sáb
Levels: Dom Jue Lun Mar Mié Sáb Vie

> object.size(fdias)

4800 bytes

> nlevels(fdias)

[1] 7

> levels(fdias)

[1] "Dom" "Jue" "Lun" "Mar" "Mié" "Sáb" "Vie"

> levels(fdias)[1] <- 'Sun'
```

■

Los niveles de un *factor* no pueden compararse entre si a menos que al definirlos se establezca una relación de orden entre ellos. Con este objetivo usaríamos la función `ordered()` antes indicada. El siguiente ejercicio muestra un ejemplo de uso:

Ejercicio 2.15 Definición y uso de *factors* ordenados

```
> peso <- ordered(c('Ligero','Medio','Pesado'))
> tam <- peso[c(sample(peso,25,replace=T))]
> tam
```

```
 [1] Pesado Pesado Medio  Ligero Medio  Medio  Ligero Medio
 [9] Ligero Ligero Medio  Medio  Pesado Ligero Pesado Pesado
[17] Pesado Medio  Medio  Medio  Medio  Pesado Pesado Ligero
[25] Medio
Levels: Ligero < Medio < Pesado

> tam[2] < tam[1]

[1] FALSE
```

Otra ventana del uso de *factors* es que sus elementos pueden ser utilizados como valores numéricos, algo que no es posible hacer con una cadena de caracteres tal y como se aprecia en el siguiente ejemplo:

Ejercicio 2.16 Conversión de elementos de un *factor* a valores numéricos

```
> dias[3]

[1] "Mié"

> fdias[3]

[1] Mar
Levels: Sun Jue Lun Mar Mié Sáb Vie

> as.numeric(fdias[3])

[1] 4

> as.numeric(dias[3])

Warning: NAs introducidos por coerción

[1] NA
```

3. Tipos de datos (II)

Los tipos de datos descritos en el capítulo previo comparten una característica común: todos los datos que contienen han de ser del mismo tipo. En un vector o una matriz, por ejemplo, no es posible guardar un número, una cadena de caracteres y un valor lógico. Todos los valores serán convertidos a un tipo común, habitualmente `character`. Esto dificulta la realización de operaciones sobre los valores de otros tipos.

En este capítulo conoceremos dos estructuras de datos más avanzadas de R: los *data frame* y las listas.

3.1 *Data frames*

El *data frame* es seguramente el tipo de dato más utilizado en R, implementándose internamente en forma de lista que tiene una determinada estructura. Un *data frame* está compuesto de múltiples columnas y filas, como una matriz, pero cada columna puede ser un tipo distinto. Al igual que las matrices, las columnas y filas pueden tener nombres, lo cual simplifica el acceso a la información como se explicará a continuación.

3.1.1 Creación de un *data frame*

La función encargada de crear objetos de este tipo es `data.frame()`. El resultado obtenido es un objeto clase `data.frame`, en el que cada columna aparecerá como una variable y cada fila como una observación. Todas las columnas han de tener el mismo número de filas.

Sintaxis 3.1 `data.frame(vector1, ..., vectorN [, row.names= nombresFilas, stringsAsFactors=TRUE|FALSE])`

Genera un nuevo data frame a partir de los datos contenidos en los vectores entregados como parámetros. Todos ellos deben tener el mismo número de filas. Los nombres de las columnas se establecerán a partir de los nombres de los vectores. Opcionalmente pueden facilitarse nombres para las filas con el parámetro `row.names`. Habitualmente `data.frame()` convertirá las cadenas de caracteres en *factors*. Este comportamiento puede controlarse mediante el parámetro `stringsAsFactors`, asignándole el valor `FALSE` si deseamos preservar las cadenas como tales.

Las funciones `length()`, `ncol()` y `nrow()`, que usábamos en el capítulo previo con matrices, también se utilizan con *data frames*. En este caso, no obstante, la primera y la segunda son equivalentes, devolviendo el número de columnas. En el siguiente ejercicio se muestra cómo generar un *data frame* con tres columnas, llamadas `Dia`, `Estimado` y `Lectura`, conteniendo datos de un vector creado antes y dos creados dinámicamente mediante repetición y la función `rnorm()`:

Ejercicio 3.1 Creación de un *data frame*, visualización y obtención de su estructura

```
> df <- data.frame(Dia = fdias[1:20],
+                  Estimado = rep(c(T,F),10),
+                  Lectura = rnorm(20,5))
> head(df)

  Dia Estimado  Lectura
1 Dom     TRUE 3.845764
2 Mié    FALSE 5.986513
3 Jue     TRUE 2.547441
4 Jue    FALSE 5.714854
5 Dom     TRUE 6.426501
6 Lun    FALSE 5.417223

> c(length(df), ncol(df), nrow(df))

[1]  3  3 20
```

■

Podemos crear un *data frame* vacío, conteniendo únicamente el nombre de las columnas y sus respectivos tipos, facilitando a `data.frame()` exclusivamente esa información. Opcionalmente pueden indicarse un número de filas entre paréntesis, tomando estas un valor por defecto. Por ejemplo:

Ejercicio 3.2 Creación de *data frames* inicialmente vacios

```
> df2 <- data.frame(Dia = numeric(),
+                   Estimado = logical(),
+                   Lectura = numeric())
> df2

[1] Dia      Estimado Lectura
<0 rows> (or 0-length row.names)

> df3 <- data.frame(Dia = numeric(10),
+                   Estimado = logical(10),
+                   Lectura = numeric(10))
> df3

   Dia Estimado Lectura
1    0    FALSE       0
2    0    FALSE       0
3    0    FALSE       0
```

```
4    0    FALSE       0
5    0    FALSE       0
6    0    FALSE       0
7    0    FALSE       0
8    0    FALSE       0
9    0    FALSE       0
10   0    FALSE       0
```

■

También puede crearse un *data frame* a partir de una matriz y otros tipos de datos, mediante la función `as.data.frame()`. Para comprobar si un cierto objeto es o no un *data frame* podemos usar la función `is.data.frame()`.

Sintaxis 3.2 `as.data.frame(objeto [, stringsAsFactors=TRUE|FALSE])`

Facilita la conversión de objetos de otros tipos a tipo `data.frame`. Cada tipo de dato puede aportar su propia versión de esta función, con una implementación específica del proceso de conversión.

Sintaxis 3.3 `is.data.frame(objeto)`

Comprueba si el objeto facilitado como parámetro es o no de tipo `data.frame`, devolviendo `TRUE` o `FALSE` según corresponda.

3.1.2 Acceder al contenido de un *data frame*

A pesar de que en un `data.frame` cada columna puede contener datos de un tipo distinto, su estructura es similar a la de una matriz al ser una estructura de datos bidimensional, compuesta de filas y columnas. Por ello el método de acceso a su contenido, mediante el operador `[]`, sigue el mismo patrón:

Ejercicio 3.3 Acceso al contenido de un *data frame*

```
> df[5,3]  # Tercera columna de la quinta fila

[1] 6.426501

> df[5,]   # Quinta fila completa

  Dia Estimado  Lectura
5 Dom     TRUE 6.426501

> df[,3]   # Tercera columna completa

 [1] 3.845764 5.986513 2.547441 5.714854 6.426501 5.417223
 [7] 6.952092 3.963256 5.608792 4.580951 5.793826 4.644867
[13] 4.846451 6.433905 4.745405 6.325569 5.618622 4.464703
[19] 6.854527 4.002581

> df[c(-3,-6),] # Todo menos filas 3 y 6
```

```
   Dia Estimado  Lectura
1  Dom     TRUE 3.845764
2  Mié    FALSE 5.986513
4  Jue    FALSE 5.714854
5  Dom     TRUE 6.426501
7  Jue     TRUE 6.952092
8  Sáb    FALSE 3.963256
9  Mié     TRUE 5.608792
10 Mié    FALSE 4.580951
11 Jue     TRUE 5.793826
12 Mié    FALSE 4.644867
13 Mié     TRUE 4.846451
14 Dom    FALSE 6.433905
15 Vie     TRUE 4.745405
16 Jue    FALSE 6.325569
17 Jue     TRUE 5.618622
18 Sáb    FALSE 4.464703
19 Jue     TRUE 6.854527
20 Mar    FALSE 4.002581
```

■

ⓘ Al trabajar con *data frames* es habitual utilizar la terminología de SQL para referirse a las operaciones de acceso al contenido de la estructura de datos. Así, al filtrado de filas se le llama normalmente *selección*, mientras que el filtrado de columnas es conocido como *proyección*.

Las columnas de un *data frame* son directamente accesibles mediante la notación `dataFrame$columna`[1], tal y como se muestra en el siguiente ejemplo:

Ejercicio 3.4 Acceso al contenido de un *data frame*

```
> df$Lectura

 [1] 3.845764 5.986513 2.547441 5.714854 6.426501 5.417223
 [7] 6.952092 3.963256 5.608792 4.580951 5.793826 4.644867
[13] 4.846451 6.433905 4.745405 6.325569 5.618622 4.464703
[19] 6.854527 4.002581
```

■

Proyección y selección pueden combinarse a fin de poder ejecutar consultas más complejas sobre el contenido de un *data frame*. En el sigueinte ejercicio se proponen dos ejemplos de esta técnica:

Ejercicio 3.5 Ejemplos de proyección y selección de datos en un *data frame*

```
> df$Estimado==F
```

[1] Esta es la notación genérica que se usa en R para acceder a cualquier atributo de un objeto. Los *data frames* son objetos en los que cada columna es definida como un atributo.

```
 [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
[10]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
[19] FALSE  TRUE

> # Obtener el día y lectura de todas las filas en las que no se
> # haya estimado
> df[df$Estimado == F, c('Dia','Lectura')]

   Dia  Lectura
2  Mié 5.986513
4  Jue 5.714854
6  Lun 5.417223
8  Sáb 3.963256
10 Mié 4.580951
12 Mié 4.644867
14 Dom 6.433905
16 Jue 6.325569
18 Sáb 4.464703
20 Mar 4.002581

> # Filtrar también las filas cuya lectura sea <= que 3
> df[df$Estimado == F & df$Lectura > 3, c('Dia','Lectura')]

   Dia  Lectura
2  Mié 5.986513
4  Jue 5.714854
6  Lun 5.417223
8  Sáb 3.963256
10 Mié 4.580951
12 Mié 4.644867
14 Dom 6.433905
16 Jue 6.325569
18 Sáb 4.464703
20 Mar 4.002581
```

Además de para recuperar el contenido del *data frame*, las anteriores notaciones pueden también ser utilizadas para modificar cualquier dato. En el ejemplo siguiente se muestra cómo cambiar el mismo dato usando dos notaciones diferentes de las antes explicadas:

Ejercicio 3.6 Modificar el contenido de un *data frame*

```
> df[15,1] <- 'Vie'    # Acceso al mismo dato usando
> df$Dia[15] <- 'Vie'  # dos notaciones distintas
> df[12:17,]

   Dia Estimado  Lectura
12 Mié    FALSE 4.644867
13 Mié     TRUE 4.846451
14 Dom    FALSE 6.433905
```

```
15 Vie     TRUE 4.745405
16 Jue    FALSE 6.325569
17 Jue     TRUE 5.618622
```

3.1.3 Agregar filas y columnas a un *data frame*

Añadir e insertar nuevas filas y columnas en un *data frame* con contenido son operaciones que pueden efectuarse de diversas formas. La longitud de cualquier vector o matriz puede extenderse directamente con el operador `[]`, usando como índice el valor siguiente a la actual longitud. Esto también es válido para los *data frame*, pero hemos de tener en cuenta cómo afectará la operación a los tipos de las columnas.

Adición de nuevas filas

En el siguiente ejemplo se muestra cómo agregar una nueva fila con esta técnica básica. Antes y después de hacerlo se usa la función `str()` para obtener información básica de la estructura del *data frame*. Hemos de prestar atención a los tipos de cada variable:

Ejercicio 3.7 Agregar nuevas filas a un *data frame*

```
> str(df)

'data.frame':        20 obs. of  3 variables:
 $ Dia     : Factor w/ 7 levels "Dom","Jue","Lun",..: 1 5 2 2 1 3 2 6 5 5 ...
 $ Estimado: logi  TRUE FALSE TRUE FALSE TRUE FALSE ...
 $ Lectura : num  3.85 5.99 2.55 5.71 6.43 ...

> # Cuidado, se pierden los tipos de las columnas y todas pasan a ser character
> df[nrow(df)+1,] <- c('Vie', FALSE, 5)

> str(df)

'data.frame':        21 obs. of  3 variables:
 $ Dia     : Factor w/ 7 levels "Dom","Jue","Lun",..: 1 5 2 2 1 3 2 6 5 5 ...
 $ Estimado: chr  "TRUE" "FALSE" "TRUE" "FALSE" ...
 $ Lectura : chr  "3.84576411" "5.98651328" "2.54744080" "5.71485449" ...
```

La función `c()` que utilizamos para facilitar los datos de la nueva fila crea un vector temporal. Como ya sabemos, los vectores son estructuras en las que todos los elementos han de ser del mismo tipo, razon por la que `FALSE` y `5` se convierten al tipo `character`, al ser este el único compatible para los tres elementos. Al agregar la nueva fila el *data frame* establece como tipos de las columnas los correspondientes a los nuevos datos.

Para crear la nueva fila debemos usar la función `data.frame()`, de forma que lo que haríamos seria concatenar un nuevo *data frame* al final del ya existente. En lugar de usar la notación `df[nrow(df)+1,]`, que es completamente válida, podemos recurrir a la función `rbind()`.

Sintaxis 3.4 `rbind(objeto1, ..., objetoN)`

Concatena los objetos facilitados como argumentos por filas. Los objetos pueden ser vectores, matrices o `data.frames`. El tipo del resultado dependerá de los tipos de los objetos.

En el siguiente ejercicio se ofrecen dos ejemplos en los que se añade una nueva fila al final de las ya existentes:

Ejercicio 3.8 Agregar nuevas filas a un *data frame*

```
> df[nrow(df)+1,] <- data.frame('Vie', F, 5)
> str(df)

'data.frame':        21 obs. of  3 variables:
 $ Dia     : Factor w/ 7 levels "Dom","Jue","Lun",..: 1 5 2 2 1 3 2 6 5 5 ...
 $ Estimado: logi  TRUE FALSE TRUE FALSE TRUE FALSE ...
 $ Lectura : num  3.85 5.99 2.55 5.71 6.43 ...

> tail(df)

   Dia Estimado  Lectura
16 Jue    FALSE 6.325569
17 Jue     TRUE 5.618622
18 Sáb    FALSE 4.464703
19 Jue     TRUE 6.854527
20 Mar    FALSE 4.002581
21 Vie    FALSE 5.000000

> df <- rbind(df, data.frame(
+   Dia = fdias[1],
+   Estimado = T,
+   Lectura = 3.1415926))
> str(df)

'data.frame':        22 obs. of  3 variables:
 $ Dia     : Factor w/ 7 levels "Dom","Jue","Lun",..: 1 5 2 2 1 3 2 6 5 5 ...
 $ Estimado: logi  TRUE FALSE TRUE FALSE TRUE FALSE ...
 $ Lectura : num  3.85 5.99 2.55 5.71 6.43 ...

> tail(df)

   Dia Estimado  Lectura
17 Jue     TRUE 5.618622
18 Sáb    FALSE 4.464703
19 Jue     TRUE 6.854527
20 Mar    FALSE 4.002581
21 Vie    FALSE 5.000000
22 Dom     TRUE 3.141593
```

■

Inserción de filas

En caso de que las nuevas filas no hayan de añadirse al final de las ya existentes en el *data frame*, sino insertarse en una posición concreta, habremos de partir el *data frame* original en dos partes y colocar entre ellas la nueva fila. Para ello recurriremos nuevamente a la función `rbind()`, tal y como se aprecia en el siguiente ejercicio:

Ejercicio 3.9 Insertar filas en un *data frame*

```
> nuevaFila <- data.frame(Dia = fdias[1],
+                         Estimado = F,
+                         Lectura = 4242)
>
> df <- rbind(df[1:9,], nuevaFila, df[10:nrow(df),])
> df[8:14,]
```

```
    Dia Estimado     Lectura
8   Sáb    FALSE    3.963256
9   Mié     TRUE    5.608792
10  Dom    FALSE 4242.000000
101 Mié    FALSE    4.580951
11  Jue     TRUE    5.793826
12  Mié    FALSE    4.644867
13  Mié     TRUE    4.846451
```

Observa en el ejercicio previo cómo el número asociado a la antigua fila se ha cambiado, para evitar repeticiones. El nuevo identificador no es consecutivo, es decir, no se renumeran las filas del *data frame*.

Adición de nuevas columnas

Para agregar una nueva columna a un *data frame* hemos de facilitar un vector con la información. Dicho vector debería tener tantos elementos como filas haya actualmente en el *data frame*. La nueva columna puede añadirse usando directamente la notación `objeto$columna` o bien usando la función `cbind()`.

Sintaxis 3.5 `cbind(objeto1, ..., objetoN)`

Concatena los objetos facilitados como argumentos por columnas. Los objetos pueden ser vectores, matrices o `data.frames`. El tipo del resultado dependerá de los tipos de los objetos.

El siguiente ejercicio muestra cómo agregar dos nuevas columnas a nuestro anterior *data frame*. La primera se llamará `Lectura` y sería de tipo numérico, mientras que segunda tendrá el nombre `Fecha` y contendra una fecha.

Ejercicio 3.10 Agregar nuevas columnas a un *data frame*

```
> df$Ajustado <- df$Lectura + rnorm(nrow(df), 2)
> df <- cbind(df, Fecha = date())
> head(df)

  Dia Estimado  Lectura  Ajustado                    Fecha
1 Dom     TRUE 3.845764  6.623803 Sat Dec 27 13:18:51 2014
2 Mié    FALSE 5.986513  7.016683 Sat Dec 27 13:18:51 2014
3 Jue     TRUE 2.547441  4.659956 Sat Dec 27 13:18:51 2014
4 Jue    FALSE 5.714854  7.738610 Sat Dec 27 13:18:51 2014
5 Dom     TRUE 6.426501 10.375874 Sat Dec 27 13:18:51 2014
6 Lun    FALSE 5.417223  6.175288 Sat Dec 27 13:18:51 2014
```


Inserción de columnas

Al igual que ocurría con las filas, para insertar una columna en una posición concreta es necesario dividir el actual *data frame*, uniendo las partes para formar el nuevo mediante la función `cbind()`. El siguiente ejercicio demuestra cómo reconstruir el *data frame* de forma

que las dos primeras columnas sean las antiguas primera y tercera. Estan irían seguidas de una nueva columna, tras la cual aparecería la antigua segunda. El resultado solamente se muestra por la consola, sin llegar a almacenarse en la variable:

Ejercicio 3.11 Insertar nuevas columnas en un *data frame*

```
> head(cbind(df[,c(1,3)],
+      Ajustado = df$Lectura + rnorm(nrow(df),2), df$Estimado))

  Dia  Lectura Ajustado df$Estimado
1 Dom 3.845764 7.378324        TRUE
2 Mié 5.986513 9.469816       FALSE
3 Jue 2.547441 4.657718        TRUE
4 Jue 5.714854 7.508201       FALSE
5 Dom 6.426501 9.779609        TRUE
6 Lun 5.417223 8.106187       FALSE
```

■

3.1.4 Nombres de filas y columnas

Al igual que las matrices, las filas y columnas de un *data frame* pueden tener asignados nombres. Estos se obtienen y modifican con las funciones `colnames()` y `rownames()` que conocimos en el capítulo previo, a las que hay que sumar la función `names()` que, en este contexto, sería equivalente a `colnames()`.

Ejercicio 3.12 Nombres de columnas y filas en un *data frame*

```
> names(df)

[1] "Dia"      "Estimado" "Lectura"  "Ajustado" "Fecha"

> colnames(df)

[1] "Dia"      "Estimado" "Lectura"  "Ajustado" "Fecha"

> rownames(df)

 [1] "1"   "2"   "3"   "4"   "5"   "6"   "7"   "8"   "9"
[10] "10"  "101" "11"  "12"  "13"  "14"  "15"  "16"  "17"
[19] "18"  "19"  "20"  "21"  "22"
```

■

3.1.5 *Data frames* y la escalabilidad

Como se apuntó anteriormente, el *data frame* es posiblemente el tipo de dato más usado en R. Esto, sin embargo, no significa que sea el más apropiado en todos los casos. Dependiendo de la cantidad de información a manejar, y de las operaciones a efectuar sobre ella, probablemente nos encontremos con problemas de escalabilidad. Leer en un *data frame* cientos de miles de filas, agregándolas una a una, puede requerir un tiempo considerable, ya

que cada adición implica crear un nuevo *data frame* como resultado y descargar el anterior, dando más trabajo al recolector de basura[2].

Si se conoce de antemano el número de filas que tendrá el *data frame*, siempre puede reservarse toda la memoria que será necearia al principio, asignando después los valores a fila tal y como se demuestra en el siguiente ejercicio:

Ejercicio 3.13 Creación de un *data frame* reservando memoria al inicio

```
> n <- 15
> df <- data.frame(Lectura = numeric(n),
+                  Fecha = character(n),
+                  stringsAsFactors = FALSE)
> for(idx in 1:n) {
+   df$Lectura[idx] <- rnorm(1,10)
+   df$Fecha[idx] <- as.character(Sys.Date())
+ }
> head(df)

    Lectura      Fecha
1  9.860742 2014-12-27
2  9.570793 2014-12-27
3 10.829137 2014-12-27
4  9.619860 2014-12-27
5 11.259605 2014-12-27
6  8.705983 2014-12-27
```

■

Una alternativa, válida no solo para cargar datos más rápido sino en general para obtener un mejor rendimiento mientras trabajamos con grandes volúmenes de datos, es el tipo `data.table` ofrecido en el paquete del mismo nombre. Puedes instalar dicho paquete y usar la función `vignette()` para acceder a una introducción a su uso. En *Handling big data in R* [Pau13a] puedes encontrar algunos ejemplos y comparaciones de rendimiento interesantes.

3.2 Listas

Es el tipo de dato más polifacético con que cuenta R. Una lista puede contener elementos de cualquier tipo, sin una estructura predefinida. Esto nos permite almacenar cualquier información e interpretarla como nos convenga en cada caso.

3.2.1 Creación de una lista

La función encargada de crear una nueva lista es `list()`. Podemos saber cuántos elementos contiene una lista con la función `length()`, que ya hemos usado en casos previos.

Sintaxis 3.6 `list(objeto1, ..., objetoN)`

Crea una nueva lista introduciendo como elementos los objetos entregados como parámetros.

[2] R cuenta con un GC (*Garbage collector*) o recolector de basura, encargado de liberar la memoria de los objetos que ya no son necesarios, por ejemplo tras haberlos destruido con `rm()`. También podemos invocarlo explícitamente mediante la función `gc()`

Los objetos alojados en una lista pueden ser de cualquier tipo, incluyendo otras listas. Esto también significa que podemos incluir *data frames* como elementos de una lista. No hay más límite para la cantidad y complejidad de la información almacenada que la memoria disponible en el sistema.

Ejercicio 3.14 Creación de nuevas listas

```
> lst1 <- list(3.1415927, 'Hola', TRUE, fdias[4])
> lst2 <- list(fdias[1:10], mes, df)
>
> length(lst1)

[1] 4

> lst1

[[1]]
[1] 3.141593

[[2]]
[1] "Hola"

[[3]]
[1] TRUE

[[4]]
[1] Jue
Levels: Dom Jue Lun Mar Mié Sáb Vie

> length(lst2)

[1] 3

> lst2

[[1]]
 [1] Dom Mié Jue Jue Dom Lun Jue Sáb Mié Mié
Levels: Dom Jue Lun Mar Mié Sáb Vie

[[2]]
        Lun Mar Mié Jue Vie Sáb Dom
Semana1   1   2   3   4   5   6   7
Semana2   8   9  10  11  12  13  14
Semana3  15  16  17  18  19  20  21
Semana4  22  23  24  25  26  27  28
Semana5  29  30  31  32  33  34  35

[[3]]
     Lectura      Fecha
1   9.860742 2014-12-27
2   9.570793 2014-12-27
```

```
3  10.829137 2014-12-27
4   9.619860 2014-12-27
5  11.259605 2014-12-27
6   8.705983 2014-12-27
7  11.711551 2014-12-27
8   7.472963 2014-12-27
9  10.626633 2014-12-27
10  9.567412 2014-12-27
11  9.242191 2014-12-27
12 11.192863 2014-12-27
13 10.303550 2014-12-27
14  8.053791 2014-12-27
15 10.192173 2014-12-27
```

3.2.2 Acceso a los elementos de una lista

Al trabajar con listas, el habitual operador `[]` que hemos usado con matrices, vectores y *data frames* no devuelve el contenido de un elemento, sino una lista con el elemento o elementos desgignados por los índices. Para acceder al contenido propiamente dicho tenemso que utilizar el operador `[[]]`. El siguiente ejercicio muestra la diferencia entre ambos:

Ejercicio 3.15 Acceso a los elementos de una lista

```
> lst1[2]   # Una lista con el segundo elemento

[[1]]
[1] "Hola"

> lst1[[2]] # El contenido del segundo elemento

[1] "Hola"

> lst1[c(2,3)] # Una sublista

[[1]]
[1] "Hola"

[[2]]
[1] TRUE

> lst2[[3]][1] # Un elemento del dato contenido en un elemento

     Lectura
1   9.860742
2   9.570793
3  10.829137
4   9.619860
5  11.259605
6   8.705983
```

```
7  11.711551
8   7.472963
9  10.626633
10  9.567412
11  9.242191
12 11.192863
13 10.303550
14  8.053791
15 10.192173
```

Mediante la función `unlist()` podemos convertir una lista en un vector, facilitando así el acceso a su contenido. Esto tiene sentido especialmente en listas cuyos elementos son datos simples: números, cadenas, valores lógicos, etc. Si existen elementos complejos el resultado puede resultar difícil de tratar.

Sintaxis 3.7 `unlist(lista[, recursive=TRUE|FALSE])`

Genera un vector a partir del contenido de una lista. Si esta contuviese elementos complejos, tales como otras listas, el parámetro `recursive` determinará si también ha de aplicarse el proceso de simplificación a ellos.

El siguiente ejercicio muestra la diferencia entre aplicar recursivamente el proceso de conversión a cada elemento o no hacerlo:

Ejercicio 3.16 Conversión de listas a vectores

```
> unlist(lst2)
```

```
   "1"    "5"    "2"
   "2"    "1"    "3"
   "2"    "6"    "5"
   "5"    "1"    "8"
  "15"   "22"   "29"
   "2"    "9"   "16"
  "23"   "30"    "3"
  "10"   "17"   "24"
  "31"    "4"   "11"
  "18"   "25"   "32"
   "5"   "12"   "19"
```

```
             "26"              "33"               "6"

             "13"              "20"              "27"

             "34"               "7"              "14"

             "21"              "28"              "35"
          Lectura1          Lectura2          Lectura3
"9.86074237245708" "9.57079288767732" "10.8291371095133"
          Lectura4          Lectura5          Lectura6
"9.61985988287166"  "11.259605075469" "8.70598333620988"
          Lectura7          Lectura8          Lectura9
"11.7115514267809" "7.47296302377809" "10.6266327189315"
         Lectura10         Lectura11         Lectura12
 "9.5674122142443" "9.24219085599938" "11.1928627509596"
         Lectura13         Lectura14         Lectura15
"10.3035496828115" "8.05379071914909" "10.1921726943101"
            Fecha1            Fecha2            Fecha3
      "2014-12-27"      "2014-12-27"      "2014-12-27"
            Fecha4            Fecha5            Fecha6
      "2014-12-27"      "2014-12-27"      "2014-12-27"
            Fecha7            Fecha8            Fecha9
      "2014-12-27"      "2014-12-27"      "2014-12-27"
           Fecha10           Fecha11           Fecha12
      "2014-12-27"      "2014-12-27"      "2014-12-27"
           Fecha13           Fecha14           Fecha15
      "2014-12-27"      "2014-12-27"      "2014-12-27"

> unlist(lst2, recursive = FALSE)

[[1]]
[1] 1

[[2]]
[1] 5

[[3]]
[1] 2

[[4]]
[1] 2

[[5]]
[1] 1

[[6]]
[1] 3

[[7]]
```

```
[1] 2

[[8]]
[1] 6

[[9]]
[1] 5

[[10]]
[1] 5

[[11]]
[1] 1

[[12]]
[1] 8

[[13]]
[1] 15

[[14]]
[1] 22

[[15]]
[1] 29

[[16]]
[1] 2

[[17]]
[1] 9

[[18]]
[1] 16

[[19]]
[1] 23

[[20]]
[1] 30

[[21]]
[1] 3

[[22]]
[1] 10

[[23]]
[1] 17
```

```
[[24]]
[1] 24

[[25]]
[1] 31

[[26]]
[1] 4

[[27]]
[1] 11

[[28]]
[1] 18

[[29]]
[1] 25

[[30]]
[1] 32

[[31]]
[1] 5

[[32]]
[1] 12

[[33]]
[1] 19

[[34]]
[1] 26

[[35]]
[1] 33

[[36]]
[1] 6

[[37]]
[1] 13

[[38]]
[1] 20

[[39]]
[1] 27

[[40]]
```

```
[1] 34

[[41]]
[1] 7

[[42]]
[1] 14

[[43]]
[1] 21

[[44]]
[1] 28

[[45]]
[1] 35

$Lectura
 [1]  9.860742  9.570793 10.829137  9.619860 11.259605
 [6]  8.705983 11.711551  7.472963 10.626633  9.567412
[11]  9.242191 11.192863 10.303550  8.053791 10.192173

$Fecha
 [1] "2014-12-27" "2014-12-27" "2014-12-27" "2014-12-27"
 [5] "2014-12-27" "2014-12-27" "2014-12-27" "2014-12-27"
 [9] "2014-12-27" "2014-12-27" "2014-12-27" "2014-12-27"
[13] "2014-12-27" "2014-12-27" "2014-12-27"
```

3.2.3 Asignación de nombres a los elementos

Los elementos de una lista pueden tener asignados nombres. Estos se obtienen y establecen mediante la función `names()` que ya conocemos. El uso de nombres facilita el acceso a los elementos, especialmente cuando se usa la notación `lista$nombre` ya que permite prescindir del operador `[[]]` tal y como se aprecia en el siguiente ejercicio.

Ejercicio 3.17 Uso de nombres con listas

```
> names(lst1) <- c('PI','Mensaje','Activado','Inicio')
> lst1[[1]]

[1] 3.141593

> lst1[['PI']]

[1] 3.141593

> lst1$PI
```

```
[1] 3.141593
```

4. Carga de datos

En los capítulos previos hemos usado datos introducidos manualmente en los guiones R o, a lo sumo, generados aleatoriamente mediante funciones como `rnorm()` y `runif()`. En la práctica, la información a la que habremos de aplicar los algoritmos casi siempre se encontrará almacenada en algún tipo de archivo, siendo necesario importarla desde R. También es posible que esa información esté alojada en la web, en el portapapeles o algún otro tipo de recurso.

Este capítulo enumera las funciones que necesitaremos utilizar para importar datos desde fuentes externas, explicando los procedimientos a seguir en cada caso.

4.1 Datos en formato CSV

El formato CSV (*Comma Separated Values*) es uno de los más comunes a la hora de intercambiar datos entre aplicaciones. Un archivo en este formato es un archivo de texto, en el que cada fila es una muestra u ocurrencia y cada columna una variable. Los datos en cada fila se separan entre sí mediante comas, de ahí la denominación del formato.

Un archivo CSV puede incluir o no un encabezado, una primera línea especificando el nombre de cada una de las columnas de datos. El archivo mostrado en la Figura 4.1 cuenta con dicha cabecera. En caso de que el separador para la parte decimal de los números sea la coma, en lugar del punto, los datos de cada fila se separan entre sí mediante puntos y comas. Los datos no numéricos, especialmente cadenas de caracteres, pueden ir delimitados por comillas o no.

En suma, existe una cierta variabilidad en el formato CSV si bien su estructura fundamental es siempre la misma. Por ello R cuenta con varias funciones distintas para operar con este tipo de archivos.

4.1.1 Lectura de archivos CSV

En la mayoría de las ocasiones necesitaremos importar datos almacenados en formato CSV, obteniendo como resultado un *data frame* R. Con este fin podemos recurrir a funciones como `read.table()`, `read.csv()` y `read.csv2()`, siendo estas dos últimas versiones especializadas de la primera, en las que se asigna un valor por defecto concreto a algunos de los parámetros.

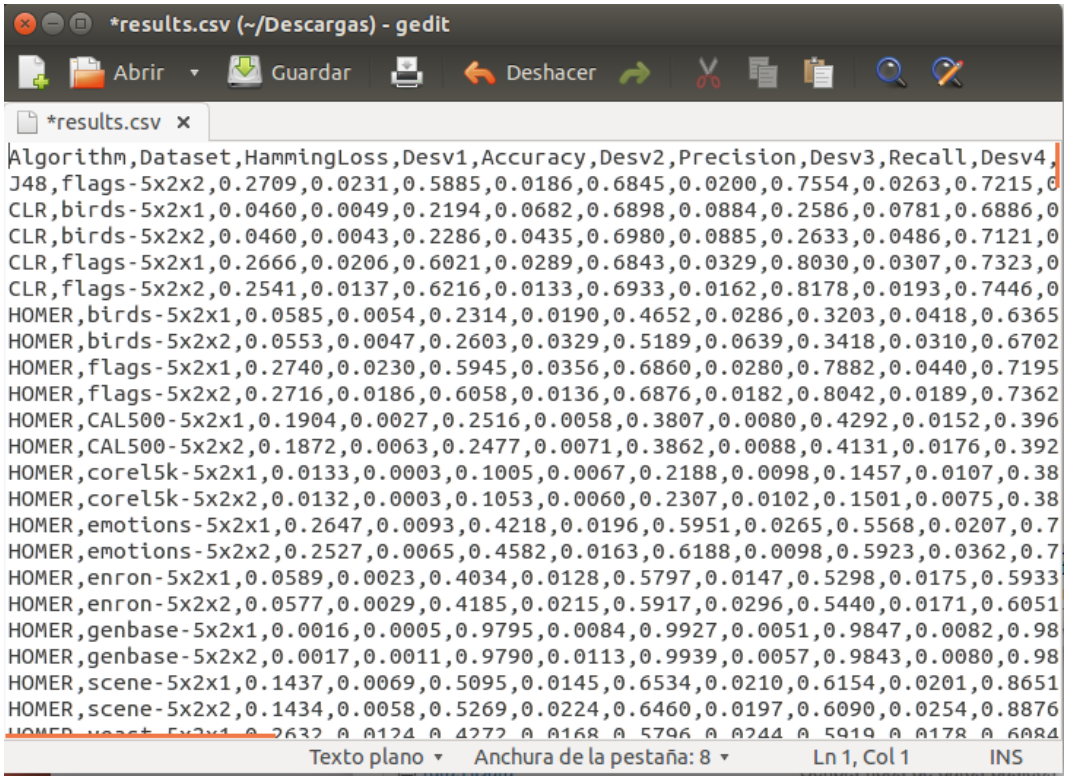


Figura 4.1: VISTA PARCIAL DEL CONTENIDO DE UNA ARCHIVO CSV

Sintaxis 4.1
```
read.table(file = archivo[, header = TRUE|FALSE,
sep = separadorDatos, dec = separadorDecimal,
quote = delimitadorCadenas,
stringsAsFactors = TRUE|FALSE])
```

Función genérica para la lectura de datos en formato CSV. El único argumento obligatorio es `file`, con el que se facilita el nombre del archivo a leer. Este debe incluir la extensión y, si fuese preciso, también la ruta. El resto de parámetros tienen la siguiente finalidad:

- `header`: De tipo `logical`. Indica a la función si el archivo contiene una cabecera o no.
- `sep`: Determina cuál es el separador de datos en cada fila.
- `dec`: Para datos numéricos, establece cuál es el separador entre parte entera y decimal.
- `quote`: Para datos alfanuméricos, establece el carácter que se usa como delimitador.
- `stringsAsFactors`: Tiene la misma finalidad que en la función `data.frame()` que conocimos en el capítulo previo.

Sintaxis 4.2 `read.csv(archivo)`

Es una implementación especializada de `read.table()` en la que se asume que los parámetros `header`, `sep` y `dec` toman los valores `TRUE`, `","` y `"."`, respectivamente. Acepta los mismos parámetros que `read.table()`.

Sintaxis 4.3 `read.csv2(archivo)`

Es una implementación especializada de `read.table()` en la que se asume que los parámetros `header`, `sep` y `dec` toman los valores `TRUE`, `";"` y `","`, respectivamente. Acepta los mismos parámetros que `read.table()`.

Sintaxis 4.4 `read.delim(archivo)`

Es una implementación especializada de `read.table()` en la que se asume que los parámetros `header`, `sep` y `dec` toman los valores `TRUE`, `"\t"` y `"."`, respectivamente. Acepta los mismos parámetros que `read.table()`.

Los archivos de datos utilizados en los ejercicios de este capítulo han de ser descargados previamente desde la página web asociada al curso o bien desde `https://github.com/fcharte/ExploraVisualizaconR`. Se asume que que en la ruta actual, que podemos modificar con la función `setwd()` como ya sabemos, se habrá creado un subdirectorio `data` y que en él estarán almacenados los archivos de datos.

En el siguiente ejercicio se carga el contenido de un archivo CSV, con datos sobre el rendimiento de diversos algoritmos, y se muestra información sobre su estructura y parte de los datos leídos:

Ejercicio 4.1 Lectura del contenido de un archivo CSV

```
> results <- read.csv('data/results.csv')
> class(results)

[1] "data.frame"

> str(results)

'data.frame':        188 obs. of  34 variables:
$ Algorithm : Factor w/ 6 levels "BR-J48","CLR",..: 1 1 1 1
   1 ...
$ Dataset : Factor w/ 32 levels "bibtex-5x2x1",..: 5 6 9 10
   11 ...
$ HammingLoss : num 0.163 0.163 ...
$ Desv1 : num 0.0015 0.002 0.0001 0.0001 0.0111 ...
$ Accuracy : num 0.213 0.214 ...
$ Desv2 : num 0.0095 0.0063 0.0025 0.0015 0.0246 ...
$ Precision : num 0.439 0.44 ...
$ Desv3 : num 0.0151 0.006 0.014 0.0155 0.0269 ...
$ Recall : num 0.297 0.296 ...
$ Desv4 : num 0.0171 0.013 0.0027 0.0017 0.0354 ...
$ FMeasure : num 0.346 0.346 ...
$ Desv5 : num 0.0126 0.009 0.0109 0.0047 0.007 ...
$ SubsetAccuracy : num 0 0 ...
$ Desv6 : num 0 0 0.0017 0.001 0.0278 ...
$ MacroFMeasure : num 0.295 0.292 ...
$ Desv7 : num 0.007 0.0093 0.0105 0.0144 0.02 ...
$ MacroPrecision : num 0.251 0.249 ...
$ Desv8 : num 0.0161 0.0213 0.0468 0.0403 0.0147 ...
$ MacroRecall : num 0.119 0.12 ...
```

```
$ Desv9 : num 0.0069 0.0078 0.0013 0.0011 0.0248 ...
$ MicroFMeasure : num 0.348 0.349 ...
$ Desv10 : num 0.0122 0.0091 0.0038 0.0022 0.023 ...
$ MicroPrecision : num 0.433 0.435 ...
$ Desv11 : num 0.0168 0.0078 0.0183 0.0197 0.0184 ...
$ MicroRecall : num 0.292 0.292 ...
$ Desv12 : num 0.0154 0.0132 0.0026 0.0016 0.0298 ...
$ OneError : num 0.709 0.731 ...
$ Desv13 : num 0.0339 0.0257 0.0096 0.013 0.0642 ...
$ Coverage : num 169 169 ...
$ Desv14 : num 1.13 0.765 ...
$ RankingLoss : num 0.314 0.318 ...
$ Desv15 : num 0.0134 0.0105 0.0017 0.0034 0.0374 ...
$ AveragePrecision: num 0.347 0.344 ...
$ Desv16 : num 0.013 0.0068 0.0067 0.0059 0.0341 ...

> head(results[,c('Algorithm','Dataset','Accuracy')])

  Algorithm        Dataset Accuracy
1    BR-J48   CAL500-5x2x1   0.2134
2    BR-J48   CAL500-5x2x2   0.2136
3    BR-J48  corel5k-5x2x1   0.0569
4    BR-J48  corel5k-5x2x2   0.0604
5    BR-J48 emotions-5x2x1   0.4343
6    BR-J48 emotions-5x2x2   0.4527
```

En este caso podemos comprobar que el *data frame* obtenido cuenta con 34 variables (columnas) y 188 observaciones (filas). Una vez leídos los datos, operaríamos sobre ellos como lo haríamos con cualquier otro *data frame*, según se explicó en el capítulo previo.

4.1.2 Exportación de datos a CSV

Si necesitamos guardar un *data frame* podemos usar, como con cualquier otro objeto, la función `save()` que conocimos en un capítulo previo. Esta almacena el contenido del objeto en formato binario. Dicho formato es más compacto y rápido de procesar, pero no resulta útil en caso de que se precise leer la información desde otras aplicaciones.

Las funciones `read.table()`, `read.csv()` y `read.csv2()` antes descritas cuentan con las complementarias `write.table()`, `write.csv()` y `write.csv2()`.

Sintaxis 4.5

```
write.table(objeto, file = archivo[,
          sep = separadorDatos, dec = separadorDecimal,
          quote = delimitadorCadenas, col.names = TRUE|FALSE,
          append = TRUE|FALSE])
```

Función genérica para la escritura de datos en formato CSV. El único argumento obligatorio, aparte del objeto a almacenar, es `file`, con el que se facilita el nombre del archivo en el que se escribirá. El contenido de este se perderá a penos que se dé el valor `TRUE` al parámetro `append`. El parámetro `col.names` determina si se agregará o no una fila como cabecera, con los nombres de las columnas. El resto de parámetros son equivalentes a los de `read.table()`.

Sintaxis 4.6 `write.csv(objeto, file = archivo)`
Es una implementación especializada de `write.table()` en la que se asume que los parámetros `sep` y `dec` toman los valores `","` y `"."`, respectivamente. Acepta los mismos parámetros que `write.table()`.

Sintaxis 4.7 `write.csv2(archivo)`
Es una implementación especializada de `write.table()` en la que se asume que los parámetros `sep` y `dec` toman los valores `";"` y `","`, respectivamente. Acepta los mismos parámetros que `write.table()`.

4.2 Importar datos desde Excel

Microsoft Excel es una de las aplicaciones más utilizadas para analizar datos y elaborar representaciones gráficas. Por ello es bastante habitual encontrarse con la necesidad de importar a R información alojada en una hoja de cálculo Excel. Para esta tarea podemos recurrir a múltiples paquetes disponibles en CRAN. En esta sección se explica cómo usar dos de ellos.

4.2.1 XLConnect

Este paquete cuenta con varias funciones capaces de leer el contenido de un archivo Excel y generar un *data frame* a partir de todo o parte del contenido de una de sus hojas. Una de esas funciones es `readWorksheetFromFile()`:

Sintaxis 4.8 `readWorksheetFromFile(archivo, sheet = hojaALeer[, header = TRUE|FALSE, region = rango])`
Abre el archivo Excel indicado por el primer parámetro, recupera el contenido de la hoja indicada por `sheet` y genera un *data frame* con esa información. Opcionalmente puede indicarse si los datos cuentan con una cabecera o no, mediante el parametro `header`. Asimismo, puede facilitarse un rango de celdillas a leer mediante el parámetro `region`. Este contendrá una cadena con un rango del tipo `"B12:D18"`.

Las funciones del paquete `XLConnect` permiten operar con archivos Excel en el formato heredado `.xls` y también con el nuevo formato `.xlsx`.

En el siguiente ejercicio se utiliza esta función para obtener información sobre subastas en eBay almacenadas en una hoja de cálculo Excel. A continuación se usan las funciones `str()` y `tail()` para tener una idea sobre la estructura de los datos, como hemos hecho en ejemplos previos.

Ejercicio 4.2 Lectura del contenido de una hoja Excel

```
> if(!is.installed('XLConnect'))
+   install.packages('XLConnect')
> library('XLConnect')
>
> ebay <- readWorksheetFromFile('data/eBayAuctions.xls',sheet=1)
> class(ebay)

[1] "data.frame"
```

```
> str(ebay)

'data.frame':        1972 obs. of  8 variables:
$ Category : chr "Music/Movie/Game" "Music/Movie/Game" ...
$ currency : chr "US" "US" ...
$ sellerRating: num 3249 3249 ...
$ Duration : num 5 5 5 5 5 ...
$ endDay : chr "Mon" "Mon" ...
$ ClosePrice : num 0.01 0.01 0.01 0.01 0.01 ...
$ OpenPrice : num 0.01 0.01 0.01 0.01 0.01 ...
$ Competitive.: num 0 0 0 0 0 ...

> tail(ebay)

      Category currency sellerRating Duration endDay
1967 Automotive       US          142        7    Sat
1968 Automotive       US         2992        5    Sun
1969 Automotive       US           21        5    Sat
1970 Automotive       US         1400        5    Mon
1971 Automotive       US           57        7    Fri
1972 Automotive       US          145        7    Sat
     ClosePrice OpenPrice Competitive.
1967     521.55    200.00            1
1968     359.95    359.95            0
1969     610.00    300.00            1
1970     549.00    549.00            0
1971     820.00    650.00            1
1972     999.00    999.00            0
```

Como puede apreciarse, tenemos datos sobre 1972 transacciones y por cada una de ellas tenemos 8 variable distintas: la categoría asociada al objeto puesto en venta, la moneda de pago, la calificación del vendedor, el tiempo que el objeto ha estado en venta, el día de finalización de la subasta, el precio de salida y precio final y un indicador de competitividad.

4.2.2 xlsx

Este paquete también es capaz de trabajar con archivos Excel en los dos formatos habituales, `.xls` y `.xlsx`, ofreciendo funciones que permiten tanto escribir como leer datos de sus hojas. Las dos funciones fundamentales son `write.xlsx()` y `read.xlsx()`. También están disponibles las funciones `write.xlsx2()` y `read.xlsx2()`, funcionalmente equivalentes a las anteriores pero que ofrecen un mejor rendimiento cuando se trabaja con hojas de cálculo muy grandes.

Sintaxis 4.9
```
write.xlsx(objeto, archivo[ , sheetName = nombreHoja,
append = TRUE|FALSE, col.names = TRUE|FALSE,
row.names = TRUE|FALSE])
```

Crea un archivo Excel con el nombre indicado por el parámetro `archivo`, a menos que se dé el valor `TRUE` al parámetro `append` en cuyo caso se abre un archivo existente para añadir datos. El objeto a escribir ha de ser un *data frame*. Opcionalmente puede establecerse un nombre para la hoja en la que se introducirá la información, mediante el

parámetro `sheetName`. La adición de un línea de cabecera y de los nombres de cada fila están regidos por los parámetros `col.names` y `row.names`, respectivamente.

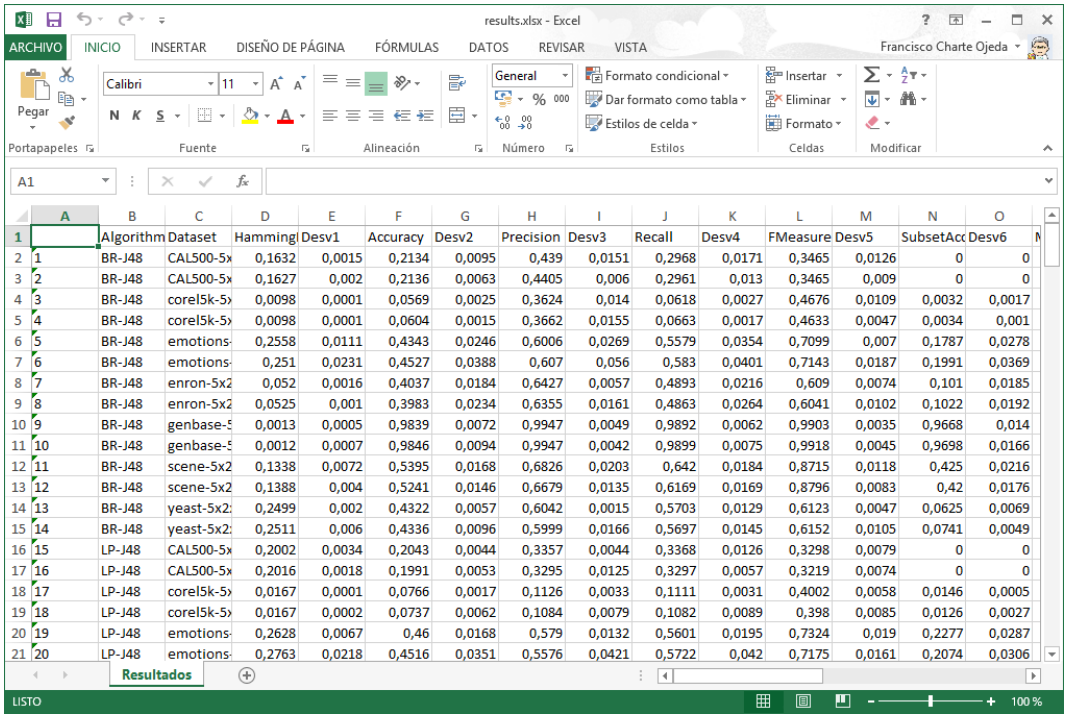
Sintaxis 4.10 `read.xlsx(archivo, sheetIndex = hojaALeer[, sheetName = hojaALeer, header = TRUE|FALSE])`

Abre el archivo Excel indicado por el primer parámetro, recupera el contenido de la hoja indicada por `sheetIndex` (índice numérico de la hoja en el libro Excel) o `sheetName` (nombre de la hoja) y genera un *data frame* con esa información. Opcionalmente puede indicarse si los datos cuentan con una cabecera o no, mediante el parametro `header`.

En el ejercicio siguiente se utiliza la función `write.xlsx()` para guardar en formato Excel los datos que habíamos obtenido antes de un archivo CSV. La Figura 4.2 muestra el archivo generado por R abierto en Excel.

Ejercicio 4.3 Creación de un archivo Excel con datos desde R

```
> if(!is.installed('xlsx'))
+   install.packages('xlsx')
> library('xlsx')
>
> write.xlsx(results, file = 'data/results.xlsx',
+            sheetName = 'Resultados')
```



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1		Algorithm	Dataset	Hamming	Desv1	Accuracy	Desv2	Precision	Desv3	Recall	Desv4	FMeasure	Desv5	SubsetAcc	Desv6
2	1	BR-J48	CAL500-5x	0,1632	0,0015	0,2134	0,0095	0,439	0,0151	0,2968	0,0171	0,3465	0,0126	0	0
3	2	BR-J48	CAL500-5x	0,1627	0,002	0,2136	0,0063	0,4405	0,006	0,2961	0,013	0,3465	0,009	0	0
4	3	BR-J48	corel5k-5x	0,0098	0,0001	0,0569	0,0025	0,3624	0,014	0,0618	0,0027	0,4676	0,0109	0,0032	0,0017
5	4	BR-J48	corel5k-5x	0,0098	0,0001	0,0604	0,0015	0,3662	0,0155	0,0663	0,0017	0,4633	0,0047	0,0034	0,001
6	5	BR-J48	emotions-	0,2558	0,0111	0,4343	0,0246	0,6006	0,0269	0,5579	0,0354	0,7099	0,007	0,1787	0,0278
7	6	BR-J48	emotions-	0,251	0,0231	0,4527	0,0388	0,607	0,056	0,583	0,0401	0,7143	0,0187	0,1991	0,0369
8	7	BR-J48	enron-5x2	0,052	0,0016	0,4037	0,0184	0,6427	0,0057	0,4893	0,0216	0,609	0,0074	0,101	0,0185
9	8	BR-J48	enron-5x2	0,0525	0,001	0,3983	0,0234	0,6355	0,0161	0,4863	0,0264	0,6041	0,0102	0,1022	0,0192
10	9	BR-J48	genbase-5	0,0013	0,0005	0,9839	0,0072	0,9947	0,0049	0,9892	0,0062	0,9903	0,0035	0,9668	0,014
11	10	BR-J48	genbase-5	0,0012	0,0007	0,9846	0,0094	0,9947	0,0042	0,9899	0,0075	0,9918	0,0045	0,9698	0,0166
12	11	BR-J48	scene-5x2	0,1338	0,0072	0,5395	0,0168	0,6826	0,0203	0,642	0,0184	0,8715	0,0118	0,425	0,0216
13	12	BR-J48	scene-5x2	0,1388	0,004	0,5241	0,0146	0,6679	0,0135	0,6169	0,0169	0,8796	0,0083	0,42	0,0176
14	13	BR-J48	yeast-5x2	0,2499	0,002	0,4322	0,0057	0,6042	0,0015	0,5703	0,0129	0,6123	0,0047	0,0625	0,0069
15	14	BR-J48	yeast-5x2	0,2511	0,006	0,4336	0,0096	0,5999	0,0166	0,5697	0,0145	0,6152	0,0105	0,0741	0,0049
16	15	LP-J48	CAL500-5x	0,2002	0,0034	0,2043	0,0044	0,3357	0,0044	0,3368	0,0126	0,3298	0,0079	0	0
17	16	LP-J48	CAL500-5x	0,2016	0,0018	0,1991	0,0053	0,3295	0,0125	0,3297	0,0057	0,3219	0,0074	0	0
18	17	LP-J48	corel5k-5x	0,0167	0,0001	0,0766	0,0017	0,1126	0,0033	0,1111	0,0031	0,4002	0,0058	0,0146	0,0005
19	18	LP-J48	corel5k-5x	0,0167	0,0002	0,0737	0,0062	0,1084	0,0079	0,1082	0,0089	0,398	0,0085	0,0126	0,0027
20	19	LP-J48	emotions-	0,2628	0,0067	0,46	0,0168	0,579	0,0132	0,5601	0,0195	0,7324	0,019	0,2277	0,0287
21	20	LP-J48	emotions-	0,2763	0,0218	0,4516	0,0351	0,5576	0,0421	0,5722	0,042	0,7175	0,0161	0,2074	0,0306

Figura 4.2: VISTA PARCIAL DEL ARCHIVO EXCEL CREADO DESDE R

La lectura de una hoja Excel con `read.xslx()` tiene prácticamente la misma sintaxis que la usada antes con `readWorksheetFromFile()` y, obviamente, el resultado es el mismo, como se aprecia en el siguiente ejercicio:

Ejercicio 4.4 Lectura del contenido de una hoja Excel

```
> ebay <- read.xlsx('data/eBayAuctions.xls', sheetIndex=1)
> class(ebay)

[1] "data.frame"

> str(ebay)

'data.frame':        1972 obs. of  8 variables:
$ Category : Factor w/ 18 levels "Antique/Art/Craft",..: 14
   14 14 14 14 ...
$ currency : Factor w/ 3 levels "EUR","GBP","US": 3 3 3 3 3
   ...
$ sellerRating: num 3249 3249 ...
$ Duration : num 5 5 5 5 5 ...
$ endDay : Factor w/ 7 levels "Fri","Mon","Sat",..: 2 2 2 2
   2 ...
$ ClosePrice : num 0.01 0.01 0.01 0.01 0.01 ...
$ OpenPrice : num 0.01 0.01 0.01 0.01 0.01 ...
$ Competitive.: num 0 0 0 0 0 ...

> tail(ebay)

       Category currency sellerRating Duration endDay
1967 Automotive       US          142        7    Sat
1968 Automotive       US         2992        5    Sun
1969 Automotive       US           21        5    Sat
1970 Automotive       US         1400        5    Mon
1971 Automotive       US           57        7    Fri
1972 Automotive       US          145        7    Sat
     ClosePrice OpenPrice Competitive.
1967     521.55    200.00            1
1968     359.95    359.95            0
1969     610.00    300.00            1
1970     549.00    549.00            0
1971     820.00    650.00            1
1972     999.00    999.00            0
```

■

En `http://fcharte.com/Default.asp?busqueda=1&q=Excel+y+R` podemos encontrar un trío de artículos dedicados al intercambio de datos entre R y Excel que pueden sernos útiles si trabajamos habitualmente con estas dos herramientas.

4.3 Importar datos en formato ARFF

El formato ARFF (*Attribute-Relation File Format*) fue creado para el software de minería de datos WEKA[1], siendo actualmente utilizado por muchos otros paquetes de software. Es muy probable que al trabajar con R necesitemos obtener información alojada en archivos `.arff`, que es la extensión habitual para datos con dicho formato.

Un archivo `.arff` es básicamente un archivo en formato CSV con una cabecera compuesta de múltiples líneas, definiendo cada una de ellas el nombre y tipo de los atributos (las columnas) de cada fila. Las secciones de cabecera y de datos están señaladas mediante etiquetas.

Al igual que ocurría con las hojas Excel, existen varios paquetes que nos ofrecen funciones capaces de leer el contenido de archivos en formato ARFF. En los siguientes apartados se describen dos de esos paquetes.

4.3.1 foreign

Este paquete ofrece múltiples funciones del tipo `read.XXX()`, representando `XXX` un formato de archivo de datos como puede ser `spss`, `dbf`, `octave` y `arff`. En total hay una decena de funciones de lectura, entre ellas `read.arff()`, asi como algunas funciones de escritura, incluyendo `write.arff()`.

Sintaxis 4.11 `read.arff(archivo)`
Lee el contenido del archivo ARFF indicado y genera un *data frame* que devuelve como resultado.

Sintaxis 4.12 `write.arff(objeto, file = archivo)`
Escribe el objeto entregado como parámetro, normalmente será un *data frame*, en el archivo ARFF indicado por el parámetro `file`.

4.3.2 RWeka

Este paquete actúa como una interfaz completa entre R y la funcionalidad ofrecida por el software WEKA, incluyendo el acceso a los algoritmos de obtención de reglas, agrupamiento, clasificación, etc. No es necesario tener instalado WEKA, al instalar el paquete `RWeka` también se instalarán las bibliotecas Java que forman el núcleo de dicho software.

Al igual que foreign, el paquete RWeka también aporta las funciones `read.arff()` y `write.arff()`. El siguiente ejercicio muestra cómo utilizar la primera de ellas para leer el dataset Covertype[2].

Ejercicio 4.5 Carga de un dataset en formato ARFF con el paquete RWeka

```
> if(!is.installed('RWeka'))
+   install.packages('RWeka')
> library('RWeka')
>
> covertype <- read.arff('data/covertype.arff')
> class(covertype)
```

[1] `http://www.cs.waikato.ac.nz/ml/weka/`

[2] Este dataset contiene una docena de variables cartográficas a partir de las cuales se trata de predecir el tipo cubierta forestal del terreno. Más información sobre este dataset en `https://archive.ics.uci.edu/ml/datasets/Covertype`.

```
[1] "data.frame"

> str(covertype)

'data.frame':        581012 obs. of  13 variables:
$ elevation : num 2596 2590 ...
$ aspect : num 51 56 139 155 45 ...
$ slope : num 3 2 9 18 2 ...
$ horz_dist_hydro: num 258 212 268 242 153 ...
$ vert_dist_hydro: num 0 -6 65 118 -1 ...
$ horiz_dist_road: num 510 390 3180 3090 391 ...
$ hillshade_9am : num 221 220 234 238 220 ...
$ hillshade_noon : num 232 235 238 238 234 ...
$ hillshade_3pm : num 148 151 135 122 150 ...
$ horiz_dist_fire: num 6279 6225 ...
$ wilderness_area: Factor w/ 4 levels "1","2","3","4": 1 1
   1 1 1 ...
$ soil_type : Factor w/ 40 levels "1","2","3","4",..: 29 29
   12 30 29 ...
$ class : Factor w/ 7 levels "1","2","3","4",..: 5 5 2 2 5
   ...

> head(covertype)

  elevation aspect slope horz_dist_hydro vert_dist_hydro
1      2596     51     3             258               0
2      2590     56     2             212              -6
3      2804    139     9             268              65
4      2785    155    18             242             118
5      2595     45     2             153              -1
6      2579    132     6             300             -15
  horiz_dist_road hillshade_9am hillshade_noon
1             510           221            232
2             390           220            235
3            3180           234            238
4            3090           238            238
5             391           220            234
6              67           230            237
  hillshade_3pm horiz_dist_fire wilderness_area soil_type
1           148            6279               1        29
2           151            6225               1        29
3           135            6121               1        12
4           122            6211               1        30
5           150            6172               1        29
6           140            6031               1        29
  class
1     5
2     5
3     2
4     2
```

```
5      5
6      2
```

■

En la información facilitada por la función `str()` puede apreciarse que el dataset cuenta con 581 012 observaciones, por lo que su carga puede precisar un cierto tiempo dependiendo de la potencia del equipo donde se esté trabajando.

4.4 Importar datos de otras fuentes

Aunque en la mayoría de los casos las funciones `read.csv()`, `read.xlsx()` y `read.arff()` serán suficientes para importar los datos con los que tengamos que trabajar, habrá ocasiones en que la información no se encuentre en un archivo que podamos leer. En esta sección se describen tres de esos casos.

4.4.1 Compartir datos mediante el portapapeles

Si los datos que necesitamos están en una aplicación que no nos permite exportar a CSV o algún otro formato más o menos estándar, por regla general siempre podremos recurrir al uso del portapapeles. En R el portapapeles se trata como si fuese un archivo cualquiera, lo único característico es su nombre: `clipboard`. También podemos usar este recurso en sentido inverso, copiando datos desde R al portapapeles para que otra aplicación pueda recuperarlos.

Funciones que ya conocemos, como `read.delim()` y `write.table()`, pueden utilizarse para importar y exportar datos al portapapeles. Si tenemos una hoja de cálculo Excel abierta, tras seleccionar un rango de celdillas y copiarlo al portapapeles la función `read.delim()` sería la adecuada para obtener su contenido desde R, ya que por defecto Excel usa tabuladores para separar los datos. De igual manera, podríamos copiar en el portapapeles un *data frame* R mediante la función `write.table()`, especificando que el separador debe ser el tabulador.

El siguiente ejemplo muestra cómo usar el portapales, en este caso exportando a él parte de un `data frame` que después es importado desde el propio R en otra variable:

Ejercicio 4.6 Copiar información a y desde el portapapeles

```
> write.table(results[1:100,], 'clipboard', sep='\t')
> partial.results <- read.delim('clipboard')
```

■

4.4.2 Obtener datos a partir de una URL

Si los datos con los necesitamos trabajar están alojados en un sitio web, como puede ser un repositorio de **GitHub**, no es necesario que descarguemos el archivo a una ubicación local para a continuación leerlo. Gracias a la función `getURL()` del paquete `RCurl` podemos leer directamente desde la URL. Esto nos permitirá trabajar siempre con la última versión disponible de los datos, sin necesidad de comprobar manualmente si ha habido cambios desde la última vez que los descargamos desde el repositorio.

Sintaxis 4.13 `getURL(URL[, write = funcionLectura)`

Descarga la información especificada por el parámetro `URL`. Opcionalmente puede facilitarse una función a la que se irá invocando repetidamente a medida que lleguen

bloques de datos con la respuesta. Se aceptan muchos otros parámetros cuya finalidad es controlar la solicitud HTTP y la gestión de la respuesta.

El valor devuelto por la función `getURL()` representa la información descargada. Podemos usarlo como parámetro para crear un objeto textConnection, del que podemos leer como si de un archivo cualquiera se tratase. Esto es lo que se hace en el siguiente ejercicio a fin de obtener el contenido de un archivo CSV desde un repositorio GitHub:

Ejercicio 4.7 Lectura de un archivo alojado en un repositorio GitHub

```
> if(!is.installed('RCurl'))
+   install.packages('RCurl')
> library('RCurl')

> url <- getURL(
+   'https://raw.githubusercontent.com/fcharte/ExploraVisualizaconR/
+   master/data/results.csv', ssl.verifypeer = FALSE)

> results2 <- read.csv(textConnection(url))

> str(results2)

'data.frame':        188 obs. of  34 variables:
 $ Algorithm       : Factor w/ 6 levels "BR-J48","CLR",..: 1 1 1 1 1 ...
 $ Dataset         : Factor w/ 32 levels "bibtex-5x2x1",..: 5 6 9 10 11 ...
 $ HammingLoss     : num  0.163 0.163 ...
 $ Desv1           : num  0.0015 0.002 0.0001 0.0001 0.0111 ...
 $ Accuracy        : num  0.213 0.214 ...
 $ Desv2           : num  0.0095 0.0063 0.0025 0.0015 0.0246 ...
 $ Precision       : num  0.439 0.44 ...
 $ Desv3           : num  0.0151 0.006 0.014 0.0155 0.0269 ...
 $ Recall          : num  0.297 0.296 ...
 $ Desv4           : num  0.0171 0.013 0.0027 0.0017 0.0354 ...
 $ FMeasure        : num  0.346 0.346 ...
 $ Desv5           : num  0.0126 0.009 0.0109 0.0047 0.007 ...
 $ SubsetAccuracy  : num  0 0 ...
 $ Desv6           : num  0 0 0.0017 0.001 0.0278 ...
 $ MacroFMeasure   : num  0.295 0.292 ...
 $ Desv7           : num  0.007 0.0093 0.0105 0.0144 0.02 ...
 $ MacroPrecision  : num  0.251 0.249 ...
 $ Desv8           : num  0.0161 0.0213 0.0468 0.0403 0.0147 ...
 $ MacroRecall     : num  0.119 0.12 ...
 $ Desv9           : num  0.0069 0.0078 0.0013 0.0011 0.0248 ...
 $ MicroFMeasure   : num  0.348 0.349 ...
 $ Desv10          : num  0.0122 0.0091 0.0038 0.0022 0.023 ...
 $ MicroPrecision  : num  0.433 0.435 ...
 $ Desv11          : num  0.0168 0.0078 0.0183 0.0197 0.0184 ...
 $ MicroRecall     : num  0.292 0.292 ...
 $ Desv12          : num  0.0154 0.0132 0.0026 0.0016 0.0298 ...
 $ OneError        : num  0.709 0.731 ...
 $ Desv13          : num  0.0339 0.0257 0.0096 0.013 0.0642 ...
 $ Coverage        : num  169 169 ...
 $ Desv14          : num  1.13 0.765 ...
 $ RankingLoss     : num  0.314 0.318 ...
 $ Desv15          : num  0.0134 0.0105 0.0017 0.0034 0.0374 ...
```

```
$ AveragePrecision: num  0.347 0.344 ...
$ Desv16          : num  0.013 0.0068 0.0067 0.0059 0.0341 ...
```

■

4.4.3 Datasets integrados

Muchos paquetes R incorporan datasets propios preparados para su uso. La instalación base de R aporta un paquete, llamado `datasets`, con docenas de bases de datos. Podemos usar la función `data()` para obtener una lista completa de todos los datasets disponibles, así como para cargar cualquiera de ellos.

Sintaxis 4.14 `data([dataset, package = nombrePaquete)`

Ejecutada sin parámetros, esta función abre un documento enumerando todos los datasets disponibles en los paquetes actualmente cargados en la sesión de trabajo. Si se facilita uno o más nombres de datasets, estos son cargados en memoria y quedan preparados para su uso. Opcionalmente puede especificarse el paquete en que están alojados los datasets.

Una vez se ha cargado un dataset, podemos usarlo como haríamos con cualquier *data frame*. En el siguiente ejercicio se utiliza el conocido dataset `iris`:

Ejercicio 4.8 Lectura de un archivo alojado en un repositorio GitHub

```
> library(datasets)
> data(iris)

> str(iris)

'data.frame':        150 obs. of  5 variables:
$ Sepal.Length: num 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
$ Sepal.Width : num 3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1
   ...
$ Petal.Length: num 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5
   ...
$ Petal.Width : num 0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1
   ...
$ Species : Factor w/ 3 levels "setosa","versicolor",..: 1
   1 1 1 1 1 1 1 1 1 ...

> head(iris)

  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

■

5. Tratamiento de datos ausentes

La existencia de datos ausentes, también conocidos como *missing values* y representados habitualmente como `NA` en R, es una casuística habitual en muchas bases de datos. La mayoría de las veces se deben a problemas durante la recopilación de datos, por ejemplo la incapacidad para obtener una cierta medida o respuesta, o fallos en la transcripción.

> Al leer datos de una fuente externa, por ejemplo un archivo CSV, los datos ausentes pueden aparecer como comillas vacías, estar representadas por un cierto valor clave o, sencillamente, estar ausentes. Funciones como `read.table()` permiten indicar qué casos han de ser interpretados como valores ausentes y, en consecuencia, aparecer como `NA` en el *data frame*.

Tratar con datasets en los que existen datos ausentes puede generar diversos problemas. Por dicha razón en este capítulo aprenderemos a tratar con ellos en R, a fin de evitar los incovenientes que suelen producir.

5.1 Problemática

La presencia de datos ausentes dificulta la mayoría de operaciones matemáticas y de análisis. ¿Cuál es el resultado de sumar `NA` a cualquier número? ¿Es `NA` menor o mayor que un cierto valor? ¿Cómo se calcula la media de una lista de valores en los que aparece `NA`? Estos son algunos casos sin respuesta y, en consecuencia, el resultado de todos ellos es también `NA`.

El siguiente ejercicio genera un conjunto de valores que, hipotéticamente, se han obtenido de una encuesta. Cinco de los encuestados no han respondido, por lo que el valor asociado es `NA`. En los pasos siguientes se efectúan algunas operaciones cuyo resultado puede apreciarse en la consola:

Ejercicio 5.1 Algunas operaciones con datos ausentes

```
> # Número de horas trabajadas semanalmente en una encuesta
> valores <- as.integer(runif(50,1,10))
> indices <- as.integer(runif(5,1,50))  # Sin respuesta 5 casos
> valores[indices] <- NA
```

```
> valores

 [1]  1  4  6  4  1  5  4  3  6 NA  7 NA  3  2  5  8  9  4
[19]  3  1  9  4  1  6  1  6  5  4  9  8  5  3  4  7  6  7
[37]  4  7 NA  5 NA  8  9  7  3  1  5  6  4  7

> valores > 5 # Los valores NA no pueden ser comparados

 [1] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE
[10]    NA  TRUE    NA FALSE FALSE FALSE  TRUE  TRUE FALSE
[19] FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE  TRUE FALSE
[28] FALSE  TRUE  TRUE FALSE FALSE FALSE  TRUE  TRUE  TRUE
[37] FALSE  TRUE    NA FALSE    NA  TRUE  TRUE  TRUE FALSE
[46] FALSE FALSE  TRUE FALSE  TRUE

> valores + 10 # Ni se puede operar con ellos

 [1] 11 14 16 14 11 15 14 13 16 NA 17 NA 13 12 15 18 19 14
[19] 13 11 19 14 11 16 11 16 15 14 19 18 15 13 14 17 16 17
[37] 14 17 NA 15 NA 18 19 17 13 11 15 16 14 17

> mean(valores)

[1] NA
```

5.2 Detectar existencia de valores ausentes

Antes de operar con un conjunto de datos, por tanto, deberíamos verificar si existen valores ausentes y, en caso afirmativo, planificar cómo se abordará su tratamiento. Con este fin podemos usar funciones como `is.na()` y `na.fail()`, entre otras.

Sintaxis 5.1 `is.na(objeto)`

Devuelve `TRUE` si el objeto es un valor ausente o `FALSE` en caso contrario. Si el objeto es compuesto, como un vector, una matriz o *data frame*, la comprobación se efectúa elemento a elemento.

Sintaxis 5.2 `na.fail(objeto)`

En caso de que el objeto facilitado como argumento contenga algún valor ausente, esta función genera un error y detiene la ejecución del guión o programa.

En caso de que solamente queramos saber si un objeto contiene valores ausentes o no, sin obtener un vector lógico para cada elemento, podemos combinar la salida de `is.na()` mediante la función `any()`, tal y como se muestra en el siguiente ejercicio:

Ejercicio 5.2 Detectar la presencia de valores nulos antes de operar

```
> is.na(valores)

 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[10]  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
[19] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[28] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[37] FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE
[46] FALSE FALSE FALSE FALSE FALSE

> any(is.na(valores))

[1] TRUE

> na.fail(valores)

Error in na.fail.default(valores) : missing values in object
```

5.3 Eliminar datos ausentes

Dependiendo de cómo vayamos a operar sobre los datos, es posible que antes de trabajar con ellos prefiramos eliminar los datos ausentes para evitar problemas como los antes expuestos. Con este fin recurriremos a la función `na.omit()`:

Sintaxis 5.3 `na.omit(objeto)`
Eliminará del objeto entregado como argumento cualquier dato ausente que exista, devolviendo un objeto del mismo tipo sin dichos valores. Los indices que ocupaban los datos ausentes se facilitan en un atributo asociado al objeto y llamado `na.action`.

Otra posibilidad consiste en utilizar la función `complete.cases()`. Esta resulta especialmente útil al trabajar con *data frames*, ya que verifica que ninguna de las columnas de cada fila contenga valores ausentes. El valor devuelto es un vector de lógicos, con `TRUE` en las filas completas (sin valores ausentes) y `FALSE` en las demás. Dicho vector puede ser utiliza para seleccionar las filas que interesen.

Sintaxis 5.4 `complete.cases(objeto)`
Devuelve un vector de valores lógicos indicando cuáles de las filas del objeto entregado como parámetro están completas, no conteniendo ningún valor ausente.

El dataset integrado `airquality` contiene 42 filas con valores ausentes de un total de 153 observaciones. En el siguiente ejercicio se muestra cómo obtener únicamente las filas sin valores nulos, ya sea utilizando `na.omit()` o `complete.cases()`:

Ejercicio 5.3 Eliminación de valores ausentes

```
> str(airquality)

'data.frame':        153 obs. of  6 variables:
 $ Ozone  : int  41 36 12 18 NA 28 23 19 8 NA ...
 $ Solar.R: int  190 118 149 313 NA NA 299 99 19 194 ...
 $ Wind   : num  7.4 8 12.6 11.5 14.3 14.9 8.6 13.8 20.1 8.6 ...
 $ Temp   : int  67 72 74 62 56 66 65 59 61 69 ...
 $ Month  : int  5 5 5 5 5 5 5 5 5 5 ...
 $ Day    : int  1 2 3 4 5 6 7 8 9 10 ...

> nrow(airquality)
```

```
[1] 153

> nrow(na.omit(airquality))

[1] 111

> nrow(airquality[complete.cases(airquality),])

[1] 111
```

5.4 Operar en presencia de datos ausentes

Algunas funciones R están preparadas para trabajar en presencia de datos ausentes, aceptando un parámetro que determina cómo han de ser tratados. Un par de ejemplos de este caso son las funciones `mean()` y `lm()`, usadas para obtener el valor promedio (media aritmética) y ajustar un modelo lineal. La primera acepta el parámetro `na.rm`, de tipo lógico, con el que se indica si los valores ausentes deben ser ignorados durante el cálculo o no. La segunda tiene un parámetro llamado `na.action` que, entre otros, acepta el valor `omit`, con exactamente el mismo resultado.

En ocasiones, en lugar de eliminar filas completas de datos de un *data frame* lo que se hace es sustituir los valores ausentes por el valor promedio de la columna en la que aparece, o bien con el valor más frecuente o bien algún valor especial. En el siguiente ejercicio se pone en práctica la primera técnica:

Ejercicio 5.4 Operar en presencia de valores ausentes

```
> promedio <- mean(valores, na.rm = TRUE)
> promedio

[1] 4.934783

> valores[is.na(valores)] <- promedio
> mean(valores)

[1] 4.934783

> lm(Solar.R ~ Temp, airquality, na.action=na.omit)


Call:
lm(formula = Solar.R ~ Temp, data = airquality, na.action = na.omit)

Coefficients:
(Intercept)         Temp
    -24.431        2.693
```

6. Análisis exploratorio

Salvo que lo hayamos creado nosotros mismos o estemos familiarizados con él por alguna otra razón, tras cargar un dataset en R generalmente lo primero que nos interesará será obtener una idea general sobre su contenido. Con este fin se aplican operaciones de análisis exploratorio, recuperando la estructura del dataset (columnas que lo forman y su tipo, número de observaciones, etc.), echando un vistazo a su contenido, aplicando funciones de estadística descriptiva para tener una idea general sobre cada variable, etc.

También es habitual que necesitemos agrupar los datos según distintos criterios, asi como particionarlos en conjuntos disjuntos a fin de usar una parte de ellos para construir modelos y otra parte para comprobar su comportamiento.

Este capítulo enumera muchas de las funciones de R que necesitaremos usar durante la exploración del contenido de un dataset, asumiendo que ya lo hemos cargado en un *data frame* mediante las técnicas descritas en el cuarto capítulo.

6.1 Información general

Asumiendo que comenzamos a trabajar con un nuevo dataset, lo primero que nos interesará sera saber qué atributos contiene, cuántas observaciones hay, etc. En capítulos previos se definieron funciones como `class()` y `typeof()`, con las que podemos conocer la clase de un objeto y su tipo. La función str() aporta más información, incluyendo el número de variables y observaciones y algunos detalles sobre cada una de las variables (columnas).

Sintaxis 6.1
```
str(objeto[, max.level = nivelExpl,
              vec.len = numElementos, ...])
```

Muestra la estructura de un objeto R cualquiera, incluyendo objetos compuestos como *data frames* y listas. El formato de salida puede ajustar con multitud de parámetros opcionales, incluyendo `max.level` para indicar hasta que nivel se explorarán estructuras anidadas (por ejemplo listas que contienen otras listas), `vec.len` a fin de limitar el número de elementos de muestra que se visualizarán de cada vector, y algunos parámetros de formato como `indent.str` y `strict.width`.

En el siguiente ejercicio puede verse la información devuelta por cada una de las tres funciones citadas al aplicarse al mismo objeto: el dataset integrado `iris`.

Ejercicio 6.1 Obtención de información general de los datos

```
> class(iris)  # Clase del objeto

[1] "data.frame"

> typeof(iris) # Tipo del objeto

[1] "list"

> # Información sobre su estructura

> str(iris)

'data.frame':        150 obs. of  5 variables:
$ Sepal.Length: num 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
$ Sepal.Width : num 3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1
   ...
$ Petal.Length: num 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5
   ...
$ Petal.Width : num 0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1
   ...
$ Species : Factor w/ 3 levels "setosa","versicolor",..: 1
   1 1 1 1 1 1 1 1 ...
```

■

6.1.1 Exploración del contenido

Aunque la función `str()` facilita una muestra del contenido de cada variable, en general dicha información nos resultará insuficiente. Podemos recurrir a funciones como `head()` y `tail()` para obtener los primeros y últimos elementos, respectivamente, de un objeto R. Asimismo, la función `summary()` ofrece un resumen global del contenido de cada variable: su valor mínimo, máximo y medio, mediana, cuartiles y, en el caso de las variables qualitativas, el número de veces que aparece cada valor posible.

Sintaxis 6.2 `head(objeto[, n = numElementos])`

Facilita los primeros elementos de un objeto R, habitualmente los primeros elementos de un vector o bien las primeras filas de un *data frame* o una matriz. Si no se facilita el parámetro `n`, por defecto este toma el valor 6. Puede usarse otro valor entero positivo para modificar el número de elementos devuelto. Si este parámetro es un entero negativo, se devolverán todos los elementos menos los `n` primeros, invirtiendo el resultado.

Sintaxis 6.3 `tail(objeto[, n = numElementos])`

Facilita los últimos elementos de un objeto R, habitualmente los últimos elementos de un vector o bien las últimas filas de un *data frame* o una matriz. El parámetro `n` funciona como en la función `head()`.

Sintaxis 6.4 `summary(objeto)`

Genera un resumen del contenido del objeto entregado como parámetro. Para variables numéricas se aportan estadísticos básicos, como la media, mediana y cuartiles. Para variables qualitativas se entrega un conteo de apariciones para cada posible valor.

La función `summary()` puede utilizarse también con otros tipos de objetos, como los devueltos por las funciones de ajuste de modelos, a fin de obtener un resumen del modelo.

Aparte de funciones como `head()` y `tail()`, que en el caso de un dataset devolverían unas pocas filas del inicio o final, también podemos recurrir a las operaciones de selección y proyección que conocimos en un capítulo previo. Obteniendo únicamente las columnas y filas que nos interesen.

En el siguiente ejercicio se utilizan las cuatro técnicas mencionadas sobre el dataset `iris`:

Ejercicio 6.2 Exploración del contenido de un *data frame*

```
> summary(iris) # Resumen de contenido

  Sepal.Length    Sepal.Width     Petal.Length
 Min.   :4.300   Min.   :2.000   Min.   :1.000
 1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600
 Median :5.800   Median :3.000   Median :4.350
 Mean   :5.843   Mean   :3.057   Mean   :3.758
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100
 Max.   :7.900   Max.   :4.400   Max.   :6.900
  Petal.Width          Species
 Min.   :0.100   setosa    :50
 1st Qu.:0.300   versicolor:50
 Median :1.300   virginica :50
 Mean   :1.199
 3rd Qu.:1.800
 Max.   :2.500

> head(iris)  # Primeras filas

  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa

> tail(iris)  # Últimas filas

    Sepal.Length Sepal.Width Petal.Length Petal.Width
145          6.7         3.3          5.7         2.5
146          6.7         3.0          5.2         2.3
```

```
147          6.3         2.5          5.0         1.9
148          6.5         3.0          5.2         2.0
149          6.2         3.4          5.4         2.3
150          5.9         3.0          5.1         1.8
      Species
145 virginica
146 virginica
147 virginica
148 virginica
149 virginica
150 virginica

> # Selección de filas y columnas
> iris$Sepal.Length[which(iris$Species == 'versicolor')]

 [1] 7.0 6.4 6.9 5.5 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1
[15] 5.6 6.7 5.6 5.8 6.2 5.6 5.9 6.1 6.3 6.1 6.4 6.6 6.8 6.7
[29] 6.0 5.7 5.5 5.5 5.8 6.0 5.4 6.0 6.7 6.3 5.6 5.5 5.5 6.1
[43] 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7
```

6.2 Estadística descriptiva

R cuenta con multitud de funciones de tipo estadístico, entre ellas las que permiten obtener información descriptiva sobre la distribución de valores en un vector. Estas funciones pueden también aplicarse a objetos más complejos, como comprobaremos después. Asimismo, existen funciones que permiten obtener un resumen descriptivo en un solo paso.

6.2.1 Funciones básicas

La sintaxis de las funciones de estadística descriptiva más comunes es la indicada a continuación.

Sintaxis 6.5 `min(vector[, na.rm = TRUE|FALSE])`

Devuelve el valor mínimo existente en el vector facilitado como parámetro. El resultado será `NA` si el vector contiene algún valor ausente, a menos que se entregue el parámetro `na.rm` con el valor `TRUE`.

El tratamiento de valores ausentes es idéntico en todas las funciones de este grupo, por lo que lo explicado para `min()` se cumple para el resto de ellas.

Sintaxis 6.6 `max(vector[, na.rm = TRUE|FALSE])`

Devuelve el valor máximo existente en el vector facilitado como parámetro.

Sintaxis 6.7 `range(vector[, finite = TRUE|FALSE, na.rm = TRUE|FALSE])`

Devuelve un vector de dos elementos con el valor mínimo y máximo de los existentes en el vector facilitado como parámetro. El parámetro `finite` determina si se obviarán los valores no finitos, valor `TRUE`, o no.

Sintaxis 6.8 `mean(vector[, trim = numValores, na.rm = TRUE|FALSE])`

Devuelve el valor promedio existente en el vector facilitado como parámetro. Si no se facilita el parámetro `trim` este tomará por defecto el valor 0 y el resultado será la media aritmética. Cualquier otro valor asignado a dicho parámetro, en el rango `(0, 0.5]`, provocará que antes de cálcular el promedio se recorten de manera simétrica la porción de muestras indicada de cada extremo. Esto permite obviar la presencia de valores extremos (*outliers*), muy pequeños o muy grandes, que podrían sesgar la media.

Sintaxis 6.9 `var(vector[, na.rm = TRUE|FALSE])`

Devuelve la varianza total calculada a partir de los valores existentes en el vector facilitado como parámetro.

Sintaxis 6.10 `sd(vector[, na.rm = TRUE|FALSE])`

Devuelve la desviación estándar calculada a partir de los valores existentes en el vector facilitado como parámetro.

Sintaxis 6.11 `median(vector[, na.rm = TRUE|FALSE])`

Devuelve la mediana de los valores existentes en el vector facilitado como parámetro.

Sintaxis 6.12 `quantile(vector[, probs = c(cortes), na.rm = TRUE|FALSE])`

Facilita un vector con tantos elementos como tenga el parámetro `probs`, correspondiente cada uno de ellos a un cuantil. Por defecto probs contiene los valores `c(0, 0.25, 0.5, 0.75, 1)`, por lo que se obtendrán los cuartiles.

Partiendo del vector `valores` que generábamos en uno de los ejercicios de un capítulo previo, el ejercicio siguiente muestra el resultado de aplicar sobre él las funciones de estadística descriptiva. A fin de obtener un resultado más compacto, se crea una lista con el valor devuelto por cada operación y, finalmente, se usa la función `unlist()` para generar un vector con la información a mostrar:

Ejercicio 6.3 Funciones básicas de estadística descriptiva

```
> unlist(list(
+   media = mean(valores), desviacion = sd(valores),
+   varianza = var(valores),  minimo = min(valores),
+   maximo = max(valores),  mediana = median(valores),
+   rango = range(valores), quartiles = quantile(valores)))

         media     desviacion       varianza         minimo
      4.934783       2.280370       5.200089       1.000000
        maximo        mediana         rango1         rango2
      9.000000       4.967391       1.000000       9.000000
  quartiles.0%  quartiles.25%  quartiles.50%  quartiles.75%
```

```
      1.000000       4.000000       4.967391       6.750000
quartiles.100%
      9.000000
```

6.2.2 Aplicación a estructuras complejas

Las anteriores funciones pueden aplicarse sobre estructuras más complejas que los vectores, como matrices y *data frames*, pero en la mayoría de los casos no nos interesará obtener la media o desviación de todo su contenido, sino de cada una de las variables (columnas) por separado.

Un método simple y directo consiste en seleccionar exactamente la información a la que queremos aplicar la función, como puede verse en el siguiente ejemplo:

Ejercicio 6.4 Selección de una variable de un objeto complejo

```
> mean(iris$Sepal.Length) # Seleccionamos una variable del dataset

[1] 5.843333
```

Si el objeto cuenta con multitud de variables, seleccionar manualmente cada una de ellas para poder aplicar una función resultará tedioso. Es una tarea que puede automatizarse gracias a las funciones `lapply()` y `sapply()`.

Sintaxis 6.13 `lapply(objeto, función)`

Aplica la función entregada segundo parámetro a cada uno de los elementos del objeto. Los resultados se devuelven en forma de lista.

Sintaxis 6.14 `sapply(objeto, función)`

Aplica la función entregada segundo parámetro a cada uno de los elementos del objeto. Los resultados se devuelven en forma de vector.

En el siguiente ejercicio puede verse cómo utilizar `lapply()` para obtener la media de cada una de las cuatro columnas numéricas existentes en el dataset `iris`:

Ejercicio 6.5 Aplicar una función a múltiples variables de un objeto

```
> lapply(iris[,1:4], mean)

$Sepal.Length
[1] 5.843333

$Sepal.Width
[1] 3.057333

$Petal.Length
[1] 3.758
```

```
$Petal.Width
[1] 1.199333
```

La función entregada como segundo parámetro puede estar predefinida, como es el caso de `mean()` o cualquier otra de las antes enumeradas, pero también es posible definirla en ese mismo punto. Para ello se usará la sintaxis `function(par) operación-a-efectuar`, siendo `par` cada uno de los elementos generados por `lapply()/sapply()` a partir del objeto entregado como primer argumento.

En el ejercicio siguiente puede verse cómo se obtienen los distintos valores de la columna `Species` del dataset `iris` y, para cada uno de ellos, se obtiene la media de la longitud de sépalo.

Ejercicio 6.6 Definir una función a medida para `sapply()/lapply()`

```
> sapply(unique(iris$Species), function(especie) mean(
+          iris$Sepal.Length[iris$Species == especie]))

[1] 5.006 5.936 6.588
```

Además de seleccionar columnas concretas, a fin de obtener un resumen descriptivo de su contenido, también podemos filtrar filas. En este contexto nos serán de utilidad funciones como `which()` y `subset()`, ya que simplifican la selección de datos en un dataset.

Sintaxis 6.15 `which(vector)`

Partiendo de un vector con valores `TRUE` y `FALSE`, presumiblemente obtenido a partir de una operación relacional, devuelve un vector con los índices correspondientes a los valores `TRUE`. De esta forma se simplica la selección de las filas que cumplen una cierta condición.

Sintaxis 6.16 `subset(objeto, subset = exprLogica, select = columnas)`

Esta función toma del objeto entregado como primer parámetro las filas en las que se cumple la expresión facilitada por `subset` y, a continuación, extrae las columnas indicadas por `select` y las devuelve como resultado. La clase del resultado sera habitualmente data.frame, conteniendo el subconjunto de filas y columnas, por lo que podemos aplicar sobre él los mecanismos de selección y proyección que ya conocemos.

El siguiente ejercicio muestra cómo obtener el mismo subconjunto de datos utilizando las dos funciones anteriores:

Ejercicio 6.7 Aplicar una función a una selección de datos de un objeto

```
> # Media de longitud de sépalo de la especie versicolor
> mean(iris$Sepal.Length[which(iris$Species == 'versicolor')])

[1] 5.936

> mean(subset(iris, Species == 'versicolor',
+             select = Sepal.Length)$Sepal.Length)
```

```
[1] 5.936
```

■

6.2.3 La función `describe()`

Aunque usando todas las funciones antes descritas, y algunas de las que conocimos en capítulos previos, podemos explorar el contenido de cualquier dataset y obtener una visión general sobre su estructura, esta es una tarea que puede simplificarse enormemente gracias a funciones como `describe()`. Esta se encuentra en el paquete `Hmisc`.

Sintaxis 6.17 `describe(objeto[, descript = título, digits = numDigitosDec)`

Facilita información descriptiva del objeto entregado como primer argumento. Opcionalmente puede facilitarse un título, asi como establecer el número de dígitos a mostrar tras el punto decimal.

Como se aprecia en el resultado generado por el siguiente ejercicio, la información generada por `describe()` incluye para cada variable el número de valores ausentes, el número de valores únicos, el porcentaje para cada valor, etc.

Ejercicio 6.8 Uso de la función `describe()` del paquete `Hmisc`

```
> # Instalar el paquete Hmisc si es preciso
> if(!is.installed('Hmisc'))
+   install.packages('Hmisc')
> library('Hmisc')
>
> describe(ebay)

ebay

 8  Variables      1972  Observations
---------------------------------------------------------------
Category
      n missing  unique
   1972       0      18

Antique/Art/Craft (177, 9%)
Automotive (178, 9%), Books (54, 3%)
Business/Industrial (18, 1%)
Clothing/Accessories (119, 6%)
Coins/Stamps (37, 2%)
Collectibles (239, 12%)
Computer (36, 2%), Electronics (55, 3%)
EverythingElse (17, 1%)
Health/Beauty (64, 3%)
Home/Garden (102, 5%)
Jewelry (82, 4%)
Music/Movie/Game (403, 20%)
```

```
Photography (13, 1%)
Pottery/Glass (20, 1%)
SportingGoods (124, 6%)
Toys/Hobbies (234, 12%)
---------------------------------------------------------------
currency
      n missing  unique
   1972       0       3

EUR (533, 27%), GBP (147, 7%)
US (1292, 66%)
---------------------------------------------------------------
sellerRating
      n missing  unique    Info    Mean     .05     .10
   1972       0     461       1    3560    50.0   112.1
    .25     .50     .75     .90     .95
  595.0  1853.0  3380.0  5702.8 22501.0

lowest :     0     1     4     5     6
highest: 25433 27132 30594 34343 37727
---------------------------------------------------------------
Duration
      n missing  unique    Info    Mean
   1972       0       5    0.86   6.486

           1   3   5   7  10
Frequency 23 213 466 967 303
%          1  11  24  49  15
---------------------------------------------------------------
endDay
      n missing  unique
   1972       0       7

          Fri Mon Sat Sun Thu Tue Wed
Frequency 287 548 351 338 202 171  75
%          15  28  18  17  10   9   4
---------------------------------------------------------------
ClosePrice
      n missing  unique    Info    Mean     .05     .10
   1972       0     852       1   36.45   1.230   2.241
    .25     .50     .75     .90     .95
  4.907   9.995  28.000  80.999 153.278

lowest :   0.01   0.06   0.10   0.11   0.17
highest: 820.00 860.00 863.28 971.00 999.00
---------------------------------------------------------------
OpenPrice
      n missing  unique    Info    Mean     .05     .10
   1972       0     569       1   12.93    0.01    0.99
    .25     .50     .75     .90     .95
```

```
   1.23    4.50    9.99   24.95   49.99

lowest :   0.01000   0.01785   0.10000   0.25000   0.50000
highest: 300.00000 359.95000 549.00000 650.00000 999.00000
------------------------------------------------------------
Competitive.
      n missing  unique    Info     Sum    Mean
   1972       0       2    0.75    1066  0.5406
------------------------------------------------------------
```

6.3 Agrupamiento de datos

Al trabajar con datasets es muy frecuente que se necesite agrupar su contenido según los valores de ciertos atributos. Sobre esos grupos pueden aplicarse funciones de resumen, generando una tabla de contigencia de datos, o bien extraer subconjuntos del dataset para operar independientemente sobre ellos.

6.3.1 Tablas de contigencia

Una tabla de contingencia permite, a partir de una tabulación cruzada, obtener un conteo de casos respecto a los valores de dos variables cualesquiera. En R este tipo de tablas se generan mediante la función `table()`:

Sintaxis 6.18 `table(objeto1, ..., objetoN)`

Genera una o más tablas de contigencia usando los objetos entregados como parámetros. Para dos objetos, normalmente dos variables de un *data frame*, se obtiene una tabla.

En el siguiente ejercicio se usa esta función explorar el dataset `ebay` y saber el número de vendedores por reputación y moneda. Aplicando la función `tail()` obtenemos únicamente los mayores valores de la primera variable, es decir, los resultados que corresponden a los mejores vendedores, a fin de saber en qué moneda operan. El segundo caso muestra para cada longitud de sépalo en `iris` el número de ocurrencias para cada especie.

Ejercicio 6.9 Generación de tablas de contingencia de datos

```
> # Conteo de vendedores según reputación y moneda
> tail(table(ebay$sellerRating, ebay$currency))


       EUR GBP US
  22501   0   0 27
  25433   0   0 35
  27132   0   0 46
  30594   0   0  1
  34343   0   0  1
  37727   0   0  4
```

```
> # Conteo para cada lóngitud de sépalo por especie
> table(iris$Sepal.Length, iris$Species)


     setosa versicolor virginica
  4.3      1          0         0
  4.4      3          0         0
  4.5      1          0         0
  4.6      4          0         0
  4.7      2          0         0
  4.8      5          0         0
  4.9      4          1         1
  5        8          2         0
  5.1      8          1         0
  5.2      3          1         0
  5.3      1          0         0
  5.4      5          1         0
  5.5      2          5         0
  5.6      0          5         1
  5.7      2          5         1
  5.8      1          3         3
  5.9      0          2         1
  6        0          4         2
  6.1      0          4         2
  6.2      0          2         2
  6.3      0          3         6
  6.4      0          2         5
  6.5      0          1         4
  6.6      0          2         0
  6.7      0          3         5
  6.8      0          1         2
  6.9      0          1         3
  7        0          1         0
  7.1      0          0         1
  7.2      0          0         3
  7.3      0          0         1
  7.4      0          0         1
  7.6      0          0         1
  7.7      0          0         4
  7.9      0          0         1
```

6.3.2 Discretización de valores

Las tablas de contingencias se usan normalmente sobre variables discretas, no numéricas, ya que estas últimas tienden a producir tablas muy grandes que dificultan el análisis. Es lo que ocurre en el ejemplo anterior, ya que hay muchos valores distintos en la variable `iris$Sepal.Length`.

En estos casos podemos discretizar la variable continua y obtener una serie de rangos que, a la postre, pueden ser tratados como variables discretas. La función a usar en este caso es `cut()`, obteniendo una transformación de la variable original que después sería usada con la función `table()`.

Sintaxis 6.19 `cut(vector, breaks = cortes)`

Discretiza los valores contenidos en el vector entregado como primer parámetro según lo indicado por el argumento `breaks`. Este puede ser un número entero, en cuyo caso se harán tantas divisiones como indique, o bien un vector de valores que actuarían como puntos de corte.

El siguiente ejercicio muestra cómo discretizar la variable `iris$Sepal.Length` a fin de obtener una tabla de contigencia más compacta, de la cual es fácil inferir cómo cambia la longitud de sépalo según la especia de la flor:

Ejercicio 6.10 Tabla de contigencia sobre valores discretizados

```
> # Discretizar la longitud de sépalo
> cortes <- seq(from=4, to=8, by=0.5)
> seplen <- cut(iris$Sepal.Length, breaks = cortes)
>
> # Usamos la variable discretizada con table()
> table(seplen, iris$Species)


seplen    setosa versicolor virginica
  (4,4.5]      5          0         0
  (4.5,5]     23          3         1
  (5,5.5]     19          8         0
  (5.5,6]      3         19         8
  (6,6.5]      0         12        19
  (6.5,7]      0          8        10
  (7,7.5]      0          0         6
  (7.5,8]      0          0         6
```

■

6.3.3 Agrupamiento y selección

Es posible dividir un dataset en varios subdatasets o grupos atendiendo a los valores de una cierta variable, usando para ello la función `split()`. También es posible seleccionar grupos de datos mediante la función `subset()` antes descrita, asi como recurrir a la función `sample()` para obtener una selección aleatoria de parte de los datos.

Sintaxis 6.20 `split(objeto, variableFactor)`

Separa un objeto en varios subobjetos conteniendo aquellos casos en los que la `variableFactor` toma cada uno de los posibles niveles. Si usamos `split()` con un *data frame*, el resultado obtenido es una lista en la que cada elemento sería también *data frame*.

Una lista con partes de un objeto puede entregarse como parámetro a la función `unsplit()` para invertir la operación, uniendo las partes a fin de obtener el objeto original.

Sintaxis 6.21 `sample(vector, numElementos[, replace = TRUE|FALSE])`

El objetivo de esta función es obtener una muestra aleatoria compuesta por `numElementos` tomados del vector entregado como primer parámetro. Por defecto el parametro `replace` toma el valor `FALSE`, por lo que la selección de elementos se hace sin reemplazamiento, es decir, no se puede tomar más de una vez cada elemento en el vector original.

El siguiente ejercicio utiliza la función `split()` para dividir el dataset iris en varios datasets, conteniendo cada uno las muestras pertenecientes a una especie de flor. También muestra cómo usar un elemento del resultado.

Ejercicio 6.11 División de un dataset en grupos

```
> # Separar en grupos según un factor
> bySpecies <- split(iris, iris$Species)

> str(bySpecies)

List of 3
$ setosa :'data.frame': 50 obs. of 5 variables:
..$ Sepal.Length: num [1:50] 5.1 4.9 4.7 4.6 5 ...
..$ Sepal.Width : num [1:50] 3.5 3 3.2 3.1 3.6 ...
..$ Petal.Length: num [1:50] 1.4 1.4 1.3 1.5 1.4 ...
..$ Petal.Width : num [1:50] 0.2 0.2 0.2 0.2 0.2 ...
..$ Species : Factor w/ 3 levels "setosa","versicolor",..:
   1 1 1 1 1 ...
$ versicolor:'data.frame': 50 obs. of 5 variables:
..$ Sepal.Length: num [1:50] 7 6.4 6.9 5.5 6.5 ...
..$ Sepal.Width : num [1:50] 3.2 3.2 3.1 2.3 2.8 ...
..$ Petal.Length: num [1:50] 4.7 4.5 4.9 4 4.6 ...
..$ Petal.Width : num [1:50] 1.4 1.5 1.5 1.3 1.5 ...
..$ Species : Factor w/ 3 levels "setosa","versicolor",..:
   2 2 2 2 2 ...
$ virginica :'data.frame': 50 obs. of 5 variables:
..$ Sepal.Length: num [1:50] 6.3 5.8 7.1 6.3 6.5 ...
..$ Sepal.Width : num [1:50] 3.3 2.7 3 2.9 3 ...
..$ Petal.Length: num [1:50] 6 5.1 5.9 5.6 5.8 ...
..$ Petal.Width : num [1:50] 2.5 1.9 2.1 1.8 2.2 ...
..$ Species : Factor w/ 3 levels "setosa","versicolor",..:
   3 3 3 3 3 ...

> # Media de longitud de sépalo de la especie 'setosa'
> mean(bySpecies$setosa$Sepal.Length)

[1] 5.006
```

Los ejemplos del siguiente ejercicio muestran cómo utilizar las funciones `subset()` y `sample()` para obtener parte de las muestras del dataset `covertype`. En el primer caso se obtienen la elevación, pendiente y clase de cubierta forestal de aquellos casos en los que la pendiente es superior a 45 y el tipo de suelo es 1. En el segundo se toma una muestra aleatoria con el 10 % de las filas del dataset.

Ejercicio 6.12 Agrupamiento y selección de datos

```
> # Media de longitud de sépalo de la especie 'setosa'
> mean(bySpecies$setosa$Sepal.Length)

[1] 5.006

> str(covertype)

'data.frame':        581012 obs. of  13 variables:
$ elevation : num 2596 2590 ...
$ aspect : num 51 56 139 155 45 ...
$ slope : num 3 2 9 18 2 ...
$ horz_dist_hydro: num 258 212 268 242 153 ...
$ vert_dist_hydro: num 0 -6 65 118 -1 ...
$ horiz_dist_road: num 510 390 3180 3090 391 ...
$ hillshade_9am : num 221 220 234 238 220 ...
$ hillshade_noon : num 232 235 238 238 234 ...
$ hillshade_3pm : num 148 151 135 122 150 ...
$ horiz_dist_fire: num 6279 6225 ...
$ wilderness_area: Factor w/ 4 levels "1","2","3","4": 1 1
   1 1 1 ...
$ soil_type : Factor w/ 40 levels "1","2","3","4",..: 29 29
   12 30 29 ...
$ class : Factor w/ 7 levels "1","2","3","4",..: 5 5 2 2 5
   ...

> # Selección de filas y columnas
> subset(covertype, slope > 45 & soil_type == '1',
+        select = c(elevation, slope, class))

       elevation slope class
2697        2001    46     3
12243       2025    46     3
247707      1991    47     3
248257      1985    46     3
253725      2237    47     3
254322      2265    48     3
254926      2293    48     3

> # Selección aleatoria
> subcovertype <- covertype[sample(1:nrow(covertype),
+                                  nrow(covertype)*.1),]
```

```
> str(covertype)

'data.frame':        581012 obs. of  13 variables:
$ elevation : num 2596 2590 ...
$ aspect : num 51 56 139 155 45 ...
$ slope : num 3 2 9 18 2 ...
$ horz_dist_hydro: num 258 212 268 242 153 ...
$ vert_dist_hydro: num 0 -6 65 118 -1 ...
$ horiz_dist_road: num 510 390 3180 3090 391 ...
$ hillshade_9am : num 221 220 234 238 220 ...
$ hillshade_noon : num 232 235 238 238 234 ...
$ hillshade_3pm : num 148 151 135 122 150 ...
$ horiz_dist_fire: num 6279 6225 ...
$ wilderness_area: Factor w/ 4 levels "1","2","3","4": 1 1
   1 1 1 ...
$ soil_type : Factor w/ 40 levels "1","2","3","4",..: 29 29
   12 30 29 ...
$ class : Factor w/ 7 levels "1","2","3","4",..: 5 5 2 2 5
   ...
```

6.4 Ordenación de datos

Cuando se lee un dataset, usando para ello cualquiera de las funciones descritas en un capítulo previo, el orden en que aparecen las observaciones en el *data frame* es el orden en que se han leído del archivo CSV, ARFF o la hoja de cálculo Excel. Algunas funciones ordenan internamente los datos sobre los que van a operar, pero sin afectar al orden original en que aparecen en el objeto en que están almacenados.

Mediante la función `sort()` podemos ordenar vectores cuyos elementos son numéricos, cadenas de caracteres, *factors* y lógicos. El resultado es un nuevo vector con los elementos ordenados. Una alternativa a la anterior es la función `order()`, cuya finalidad es devolver los posiciones que deberían ocupar los elementos para estar ordenados.

Sintaxis 6.22 `sort(vector[, decreasing = TRUE|FALSE])`

Ordena los elementos del vector generando como resultado un nuevo vector. El orden es ascedente a menos que se entregué el valor `TRUE` para el parámetro `decreasing`.

Sintaxis 6.23 `order(vector1, ..., vectorN[, decreasing = TRUE|FALSE])`

Genera un vector con las posiciones que deberían tener los elementos de vector1 para estar ordenados. En caso de empaquete, cuando varios elementos del primer vector tienen el mismo valor, se usarán los elementos del segundo vector para determinar el orden. Este proceso se repite tantas veces como sea necesaria. El orden por defecto es ascendente, pudiendo cambiarse dando el valor `TRUE` al parámetro `decreasing`.

En el siguiente ejemplo puede verse claramente la diferencia entre usar `sort()` y `order()` sobre un vector de valores. En el primer caso el nuevo vector contiene los elementos del original, pero ordenados de menor a mayor. En el segundo lo que se obtiene son los índices en que habría que tomar los elementos del vector original para obtener uno ordenado:

Ejercicio 6.13 Ordenación de los datos

```
> valores # Vector original

 [1] 1.000000 4.000000 6.000000 4.000000 1.000000 5.000000
 [7] 4.000000 3.000000 6.000000 4.934783 7.000000 4.934783
[13] 3.000000 2.000000 5.000000 8.000000 9.000000 4.000000
[19] 3.000000 1.000000 9.000000 4.000000 1.000000 6.000000
[25] 1.000000 6.000000 5.000000 4.000000 9.000000 8.000000
[31] 5.000000 3.000000 4.000000 7.000000 6.000000 7.000000
[37] 4.000000 7.000000 4.934783 5.000000 4.934783 8.000000
[43] 9.000000 7.000000 3.000000 1.000000 5.000000 6.000000
[49] 4.000000 7.000000

> sort(valores) # Vector ordenado

 [1] 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
 [7] 2.000000 3.000000 3.000000 3.000000 3.000000 3.000000
[13] 4.000000 4.000000 4.000000 4.000000 4.000000 4.000000
[19] 4.000000 4.000000 4.000000 4.934783 4.934783 4.934783
[25] 4.934783 5.000000 5.000000 5.000000 5.000000 5.000000
[31] 5.000000 6.000000 6.000000 6.000000 6.000000 6.000000
[37] 6.000000 7.000000 7.000000 7.000000 7.000000 7.000000
[43] 7.000000 8.000000 8.000000 8.000000 9.000000 9.000000
[49] 9.000000 9.000000

> order(valores) # Orden en que se toman los elementos

 [1]  1  5 20 23 25 46 14  8 13 19 32 45  2  4  7 18 22 28
[19] 33 37 49 10 12 39 41  6 15 27 31 40 47  3  9 24 26 35
[37] 48 11 34 36 38 44 50 16 30 42 17 21 29 43
```

■

La función `order()` es especialmente útil a la hora de ordenar estructuras de datos complejas, como los emphdata frame, ya que los índices devueltos como resultado pueden utilizarse para tomar las filas en el orden adecuado. Dado que `order()` puede tomar varios vectores como parámetro, es posible también ordenar por varias columnas, como se hace en el segundo ejemplo del siguiente ejercicio:

Ejercicio 6.14 Ordenación de los datos

```
> # Ordenar un data frame por una cierta columna
> sortedIris <- iris[order(iris$Petal.Length), ]
> head(sortedIris)

   Sepal.Length Sepal.Width Petal.Length Petal.Width
23          4.6         3.6          1.0         0.2
14          4.3         3.0          1.1         0.1
15          5.8         4.0          1.2         0.2
```

```
36          5.0         3.2          1.2         0.2
3           4.7         3.2          1.3         0.2
17          5.4         3.9          1.3         0.4
   Species
23  setosa
14  setosa
15  setosa
36  setosa
3   setosa
17  setosa

> # Ordenar de mayor a menor por una columna y
> # de menor a mayor por otra
> sortedIris <- iris[with(iris,
+                         order(-Sepal.Length, Petal.Length)), ]
> head(sortedIris)

    Sepal.Length Sepal.Width Petal.Length Petal.Width
132          7.9         3.8          6.4         2.0
136          7.7         3.0          6.1         2.3
118          7.7         3.8          6.7         2.2
123          7.7         2.8          6.7         2.0
119          7.7         2.6          6.9         2.3
106          7.6         3.0          6.6         2.1
      Species
132 virginica
136 virginica
118 virginica
123 virginica
119 virginica
106 virginica
```

La función `with()` usada en este ejemplo permite simplificar expresiones de acceso a elementos de una estructura de datos compleja. En lugar de escribir `order(-iris$Sepal.Length, iris$Petal.Length)`, incluyendo el nombre del *data frame* en cada referencia, con `with(objeto, expresión)` las referencias a elementos del objeto en la expresión no necesitan el prefijo `objeto$`.

6.4.1 Generación de rankings

En ocasiones más que ordenar los datos en sí nos interesará generar rankings a partir de ellos. Esto es útil, por ejemplo, para saber la posición que ocupa cada algoritmo según los resultados que ha generado sobre un conjunto de datasets. Obteniendo el ranking medio para cada algoritmo puede obtenerse una idea general sobre su comportamiento y competitividad respecto a los demás. La función `rank()` es la encargada de generar un ranking a partir de un vector de datos de entrada.

Sintaxis 6.24 `rank(vector[, ties.method = resoluciónEmpates])`

Produce un ranking a partir de los datos contenidos en el vector de entrada. Por defecto se usa el metodo `'average'` para la resolución de empates, lo que significa que a igualdad de valor se comparte la posición en el ranking. Mediante el parámetro `ties.method` es posible elegir entre los métodos disponibles: `'min'`, `'max'`, `'average'`, `'first'` y `'random'`.

En el siguiente ejercicio se generan dos rankings a partir del mismo conjunto de datos. En el primer caso se usa el método de resolución de empaquetes por defecto, mientras que en el segundo se ha optado por, en caso de empaquete, colocar primero en el ranking el valor que aparece primero en el vector:

Ejercicio 6.15 Generación de rankings a partir de los datos

```
> valores   # Valores a tratar

 [1] 1.000000 4.000000 6.000000 4.000000 1.000000 5.000000
 [7] 4.000000 3.000000 6.000000 4.934783 7.000000 4.934783
[13] 3.000000 2.000000 5.000000 8.000000 9.000000 4.000000
[19] 3.000000 1.000000 9.000000 4.000000 1.000000 6.000000
[25] 1.000000 6.000000 5.000000 4.000000 9.000000 8.000000
[31] 5.000000 3.000000 4.000000 7.000000 6.000000 7.000000
[37] 4.000000 7.000000 4.934783 5.000000 4.934783 8.000000
[43] 9.000000 7.000000 3.000000 1.000000 5.000000 6.000000
[49] 4.000000 7.000000

> rank(valores)  # Ranking con método "average"

 [1]  3.5 17.0 34.5 17.0  3.5 28.5 17.0 10.0 34.5 23.5 40.5
[12] 23.5 10.0  7.0 28.5 45.0 48.5 17.0 10.0  3.5 48.5 17.0
[23]  3.5 34.5  3.5 34.5 28.5 17.0 48.5 45.0 28.5 10.0 17.0
[34] 40.5 34.5 40.5 17.0 40.5 23.5 28.5 23.5 45.0 48.5 40.5
[45] 10.0  3.5 28.5 34.5 17.0 40.5

> rank(valores, ties.method='first') # Ranking con método "first"

 [1]  1 13 32 14  2 26 15  8 33 22 38 23  9  7 27 44 47 16
[19] 10  3 48 17  4 34  5 35 28 18 49 45 29 11 19 39 36 40
[37] 20 41 24 30 25 46 50 42 12  6 31 37 21 43
```

6.5 Particionamiento de los datos

El último tema que abordamos en este capítulo es el particionamiento de los datos contenidos en un *data frame* o estructura de datos similar. Esta es una tarea necesaria siempre que va a construirse un modelo predictivo, siendo habitual dividir el dataset original en dos (entrenamiento y test) o tres particiones (entrenamiento, validación y test).

En realidad, puesto que sabemos cómo seleccionar parte de las filas de un *data frame*, no tendremos problema alguno en usar la función `nrow()` para obtener el número de filas y, a

partir de ahí, dividir el conjunto de observaciones en las partes que nos interese. Es lo que se hace en el siguiente ejercicio, en el que se obtiene una partición de entrenamiento y otra de test:

Ejercicio 6.16 Particionamiento de datos en conjuntos disjuntos

```
> # Primeras n filas para training restantes para test
> nTraining <- as.integer(nrow(iris) * .75)
> training <- iris[1:nTraining, ]
> test <- iris[(nTraining + 1):nrow(iris), ]
>
> nrow(training)

[1] 112

> nrow(test)

[1] 38

> # Verificar que el particionamiento es correcto
> stopifnot(nrow(training) + nrow(test) == nrow(iris))
```

■

Un método alternativo al anterior es el particionamiento aleatorio del conjunto de filas. En este caso usaríamos la función `sample()` que se definió anteriormente en este mismo capítulo, obteniendo un conjunto de índices aleatorio tal y como se muestra en el siguiente ejercicio:

Ejercicio 6.17 Particionamiento de datos en conjuntos disjuntos

```
> # Toma de un conjunto aleatorio para training y test
> set.seed(4242)
> indices <- sample(1:nrow(iris), nTraining)
> # Obtenemos una lista con los dos subconjuntos
> particion <- list(training = iris[indices, ],
+                   test = iris[-indices, ])
>
> lapply(particion, nrow) # Filas en cada subconjunto

$training
[1] 112

$test
[1] 38

> particion$test    # Contenido del conjunto de test

    Sepal.Length Sepal.Width Petal.Length Petal.Width
1            5.1         3.5          1.4         0.2
13           4.8         3.0          1.4         0.1
15           5.8         4.0          1.2         0.2
```

```
16           5.7         4.4          1.5         0.4
19           5.7         3.8          1.7         0.3
27           5.0         3.4          1.6         0.4
30           4.7         3.2          1.6         0.2
31           4.8         3.1          1.6         0.2
32           5.4         3.4          1.5         0.4
33           5.2         4.1          1.5         0.1
36           5.0         3.2          1.2         0.2
37           5.5         3.5          1.3         0.2
47           5.1         3.8          1.6         0.2
48           4.6         3.2          1.4         0.2
49           5.3         3.7          1.5         0.2
55           6.5         2.8          4.6         1.5
61           5.0         2.0          3.5         1.0
67           5.6         3.0          4.5         1.5
73           6.3         2.5          4.9         1.5
75           6.4         2.9          4.3         1.3
76           6.6         3.0          4.4         1.4
79           6.0         2.9          4.5         1.5
81           5.5         2.4          3.8         1.1
85           5.4         3.0          4.5         1.5
88           6.3         2.3          4.4         1.3
104          6.3         2.9          5.6         1.8
118          7.7         3.8          6.7         2.2
120          6.0         2.2          5.0         1.5
123          7.7         2.8          6.7         2.0
133          6.4         2.8          5.6         2.2
139          6.0         3.0          4.8         1.8
140          6.9         3.1          5.4         2.1
142          6.9         3.1          5.1         2.3
143          5.8         2.7          5.1         1.9
144          6.8         3.2          5.9         2.3
146          6.7         3.0          5.2         2.3
147          6.3         2.5          5.0         1.9
149          6.2         3.4          5.4         2.3
       Species
1       setosa
13      setosa
15      setosa
16      setosa
19      setosa
27      setosa
30      setosa
31      setosa
32      setosa
33      setosa
36      setosa
37      setosa
47      setosa
48      setosa
```

```
49      setosa
55  versicolor
61  versicolor
67  versicolor
73  versicolor
75  versicolor
76  versicolor
79  versicolor
81  versicolor
85  versicolor
88  versicolor
104  virginica
118  virginica
120  virginica
123  virginica
133  virginica
139  virginica
140  virginica
142  virginica
143  virginica
144  virginica
146  virginica
147  virginica
149  virginica
```

7. Gráficos con R (I)

Uno de los mecanismos de exploración de datos más usuales y útiles consiste en generar representaciones gráficas de las variables que componen el dataset. Es frecuente que a partir de la observación de dichas representaciones pueda obtenerse información más fácilmente interpretable que la que nos ofrecen los métodos de exploración descritos en el capítulo previo.

R cuentan en su paquete base con múltiples funciones para la producción de gráficas, pudiendo generar representaciones en forma de nubes de puntos, líneas, barras, gráficos circulares, etc. También tenemos funciones para elaborar histogramas y curvas de densidad. Esos gráficos, además de ser útiles como vía de exploración de los datos, pueden ser almacenados para su posterior reutilización en cualquier tipo de documento.

En este capítulo conoceremos algunas de las posibilidades gráficas de R, alojadas en su mayor parte en el paquete `graphics`[1], aprendiendo a usarlas con casos prácticos en los se utilizarán los datasets que hemos conocido en capítulos anteriores.

7.1 Gráficos básicos

Muchos de los gráficos básicos que es posible generar con R son resultado de la función `plot()`. Esta acepta un importante número de parámetros con los que es posible configurar el gráfico resultante, establecer títulos y otros parámetros gráficos como colores, tipos de marcadores, grosor de líneas, etc.

Sintaxis 7.1
```
plot(valoresX[, valoresY type = tipoGráfico,
              main = tituloPrincipal, sub = subtítulo,
              xlab = tituloEjeX, ylab = tituloEjeY])
```
Genera una representación gráfica de los valores facilitados acorde a la configuración especificada por los parámetros adicionales. El parámetro `type` toma por defecto el valor `"p"`, dibujando un punto por cada dato. Otros posibles valores son `"l"`, para dibujar líneas, y `"h"`, para obtener un histograma.

[1] Este paquete forma parte de la instalación base de R, por lo que no necesitamos instalarlo ni cargarlo. Puedes obtener una lista de todas las funciones de este paquete con el comando `library(help = "graphics")`

En esta sección aprenderemos a usar la función `plot()` para obtener distintos tipos de gráficas, usando para ellos los datasets que obteníamos en un capítulo previo de distintas fuentes.

7.1.1 Gráficas de puntos

Este tipo de representación, conocida habitualmente como *nube de puntos*, dibuja un punto por cada observación existente en el conjunto de datos. La posición de cada punto en el plano dependerá de los valores que tomen para el dato correspondiente las variables representadas, una para el eje X y otra para el eje Y.

La mejor forma de aprender a usar `plot()` es a través de ejemplos. A continuación se ofrecen varios que producen distintas configuraciones de nubes de puntos.

Nube de puntos de una variable

Si facilitamos a `plot()` solamente un vector de datos, los valores extremos de dichos datos se usarán para establecer el rango del eje Y. El rango del eje X dependerá del número de observaciones existentes en el vector. Los datos se representarán de izquierda a derecha en el eje X según el orden que ocupan en el vector, siendo su altura (posición en el eje Y) proporcional al valor del dato en sí.

En el siguiente ejemplo se representa la longitud de sépalo para las muestras de `iris`. En el eje X aparecen las 150 observaciones, mientras que en el eje Y se indica la longitud de sépalo para cada observación.

Ejercicio 7.1 Gráficas de nubes de puntos (I)

```
> plot(iris$Sepal.Length)
```

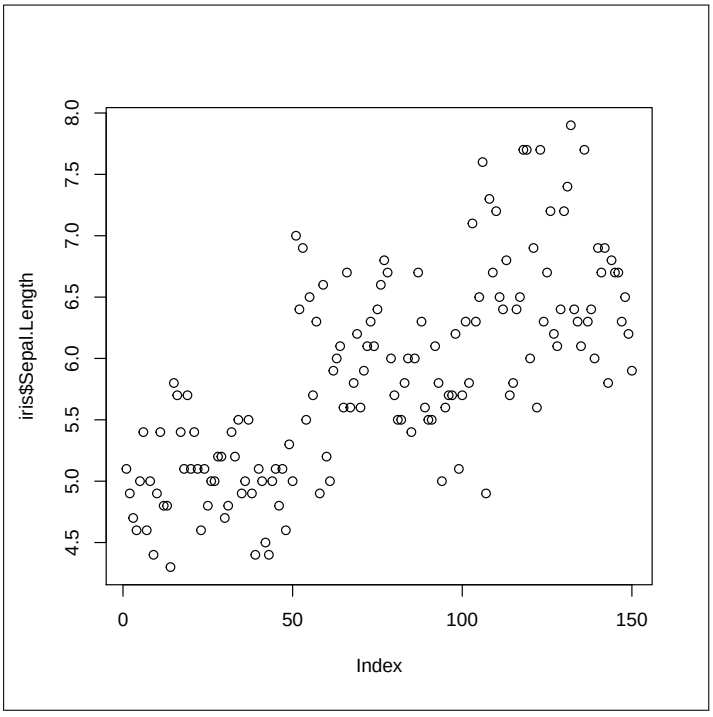


Figura 7.1: NUBE DE PUNTOS MOSTRANDO UNA VARIABLE

Lo que puede deducirse a partir de esta gráfica es que las observaciones están en el dataset original más o menos ordenadas según la longitud del sépalo. Las primeras muestran valores inferiores a las últimas, con una evolución ascendente y cierto nivel de dispersión.

Nube de puntos de dos variables

Por regla general, las nubes de puntos resultan más útiles cuando se representan dos variables, una frente a otra en los ejes X e Y, a fin de determinar si existe o no algún tipo de correlación entre ellas. Lo único que tenemos que hacer es facilitar a `plot()` dos vectores de valores, uno para el eje X y otro para el eje Y.

El ejercicio siguiente usa esta técnica para observar la relación entre la longitud y la anchura de sépalo:

Ejercicio 7.2 Gráficas de nubes de puntos (II)

```
> plot(iris$Sepal.Length, iris$Sepal.Width)
```

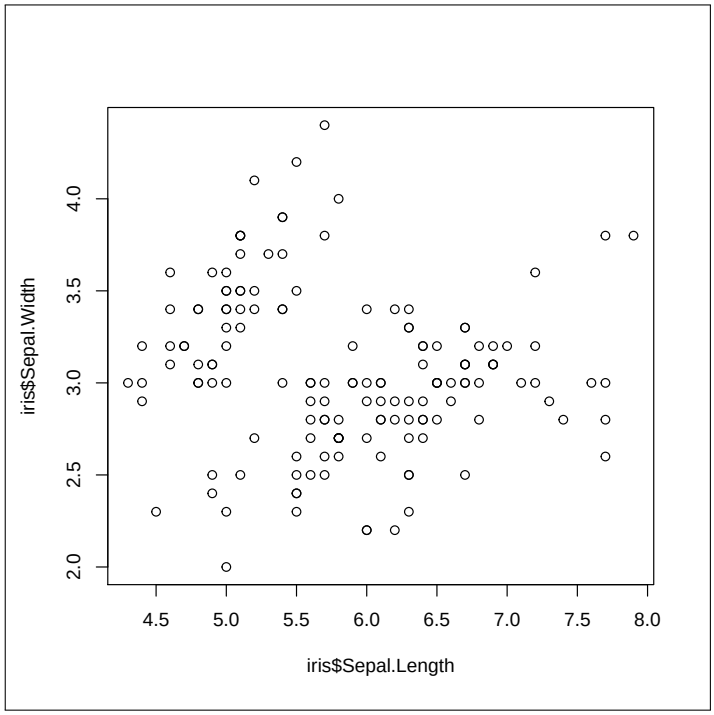


Figura 7.2: NUBE DE PUNTOS MOSTRANDO DOS VARIABLES

Como se aprecia en la gráfica, a simple vista no es fácil determinar la existencia de una relación entre estas dos variables.

Nubes de puntos de tres variables

Aunque en estas gráficas solamente contamos con dos ejes, y por tanto pueden representarse los valores correspondientes a dos variables, es posible aprovechar los atributos de los puntos: color, tipo de símbolo y tamaño, para mostrar variables adicionales.

En el siguiente ejercicio se usa el color de los puntos para representar la especie de cada observación, añadiendo el parámetro `col` de la función `plot()`. En este caso sí que puede

apreciarse un cierto agrupamiento de las muestras de una especie, mientras que las otras no son fácilmente separables atendiendo a las variables representadas.

Ejercicio 7.3 Gráficas de nubes de puntos (III)

```
> plot(iris$Sepal.Length, iris$Sepal.Width,
+      col = iris$Species, pch = 19)
```

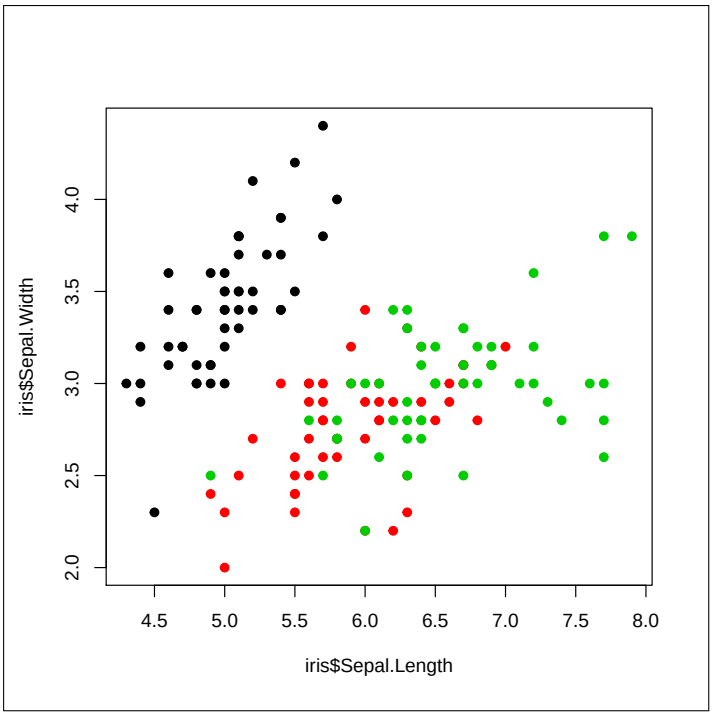


Figura 7.3: NUBE DE PUNTOS CON DOS VARIABLES Y LA CLASE EN COLOR

El parámetro `pch` permite elegir entre distintos tipos de símbolos pare representar los puntos. Es posible asignar tanto un carácter como un valor entero para elegir entre los símbolos predefinidos. Consulta la documentación de la función `points()` de R, en ella se facilita una lista de los posibles valores y los símbolos correspondientes.

Configuración de títulos y leyendas

La función `plot()` acepta cuatro parámetros que sirven para mostrar un título general, un subtítulo, un título para el eje de abscisas y otro para el eje de ordenadas. Para agregar títulos a un gráfico también podemos usar la función `title()`, que acepta los mismos cuatro parámetros: `main`, `sub`, `xlab` e `ylab`, así como otros que determinan los atributos de los titulos: fuente de letra, tamaño, color, etc.

Sintaxis 7.2 `title(main = tituloPrincipal, sub = tituloSecundario, ...)`

Añade títulos a un gráfico generado previamente, por ejemplo mediante la función `plot()`.

Además de títulos, es habitual incluir en las gráficas una leyenda con la clave de color que corresponde a cada punto, línea o barra. Estas leyendas se agregan mediante la función `legend()` que, entre otros atributos, permite establecer su posición en la gráfica, la separación de esta con un borde alrededor de las leyendas, etc.

Sintaxis 7.3
```
legend(posición, legend = títulos[, col = color,
       bty = 'o'|'n', pch = símbolo, ncol = numColumnas])
```

Añade una leyenda a un gráfico generado previamente, por ejemplo mediante la función `plot()`.

El siguiente ejercicio genera una gráfica de nube de puntos con toda la información necesaria, incluyendo la leyenda que indica a qué especie pertenece cada color y los titulos:

Ejercicio 7.4 Gráficas de nubes de puntos (IV)

```
> plot(iris$Petal.Length, iris$Petal.Width,
+      col = iris$Species,  pch = 19,
+      xlab = 'Longitud del pétalo', ylab = 'Ancho del pétalo')
>
> title(main = 'IRIS',
+       sub = 'Exploración de los pétalos según especie',
+       col.main = 'blue', col.sub = 'blue')
>
> legend("bottomright", legend = levels(iris$Species),
+        col =  unique(iris$Species), ncol = 3, pch = 19, bty = "n")
```

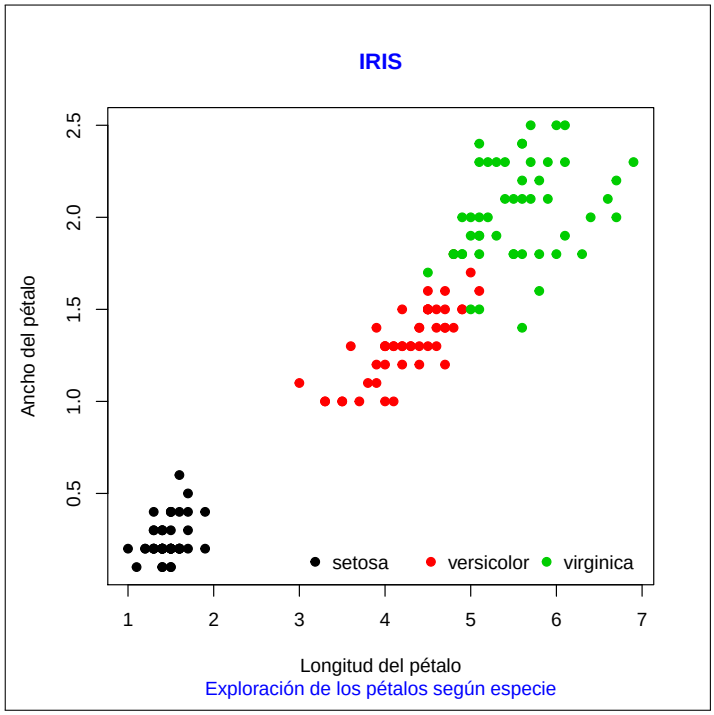


Figura 7.4: NUBE DE PUNTOS CON DOS VARIABLES Y TÍTULOS

El parámetro bty determina si se dibujará un borde alrededor de la leyenda o no. Por defecto se dibuja, pero al asignarle el valor 'n' desactivamos esta característica. Mediante el parámetro ncol establecemos el número de columnas en que se distribuirán las leyendas. Por defecto se una sola columna, por lo que aparecerían una debajo de la otra en tantas filas como leyendas haya.

Lo más importante de la gráfica anterior es el uso de los distintos valores de `iris$Species` para determinar tanto el color de los símbolos que acompañan a las leyendas como el texto de estas.

7.1.2 Gráficas de cajas

Este tipo de diagrama, conocido como gráfica de *cajas y bigotes* o *box-and-whisker plot*, permite apreciar de un vistazo cómo se distribuyen los valores de una variable, si están más o menos concentrados o dispersos respecto a los cuartiles centrales, y si existen valores anómalos (*outliers*).

En R podemos generar este tipo de gráficas con la función `plot()`, facilitando como parámetro un objeto de tipo `formula` en lugar de un vector. El operador nos permite generar un objeto de dicho tipo a partir de una o dos variables. Es lo que se hace en el siguiente ejercicio:

Ejercicio 7.5 Gráfica de cajas generada con `plot()`

```
> plot(iris$Petal.Length ~ iris$Species)
```

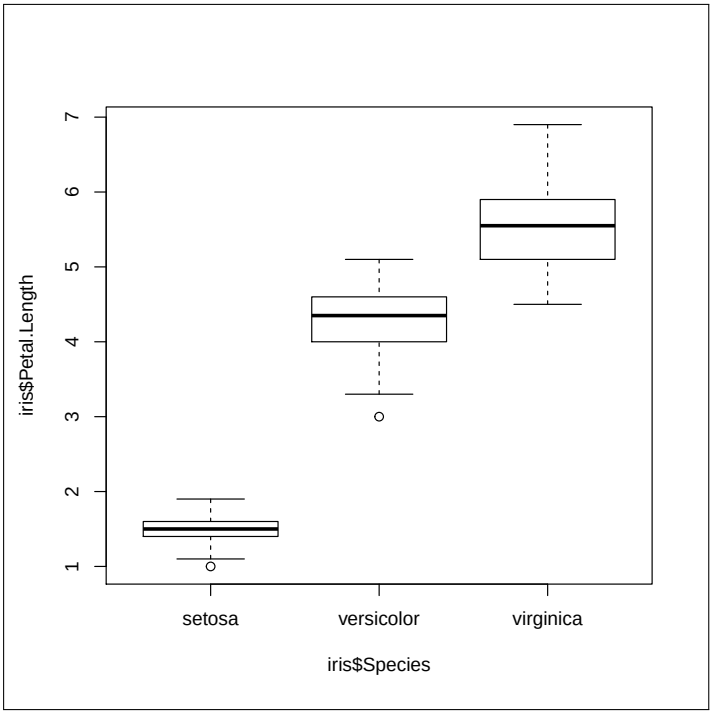


Figura 7.5: Gráfica de cajas de la longitud de pétalo por especie

Se aprecia claramente cómo los valores correspondientes a la longitud de pétalo están mucho más agrupados en la especie setosa que en las otras dos. Los círculos que aparecen

debajo del bigote inferior de las os primeras especies denotan la presencia de valores anormalmente pequeños en dicha variable.

Si precisamos mayor control sobre la configuración de este tipo de gráfica, en lugar de `plot()` podemos usar la función `boxplot()`.

Sintaxis 7.4
```
boxplot(formula[, data = dataFrame,
        range = rangoBigotes,
        outline = TRUE|FALSE,
        horizontal = TRUE|FALSE
        notch = TRUE|FALSE,
        width = anchoRelativo])
```

Genera una gráfica de cajas y bigotes a partir de la fórmula entregada como primer argumento. Si las variables usadas en la fórmula son parte de una *data frame*, el parámetro `data` permite indicarlo y prescindir del prefijo `objeto$` en la fórmula. Los demás parámetros tienen la siguiente finalidad:

- `range`: Un número entero que actúa como multiplicador del rango intercuartil, representado por la caja, para determinar el rango hasta el que se extenderán los bigotes. Si este parámetro toma el valor 0 los bigotes se extienden hasta los extremos.
- `outline`: Determina si se dibujarán o no los valores anómalos, aquellos no cubiertos por el rango de los bigotes.
- `horizontal`: Dando el valor `TRUE` a este parámetro la gráfica se rotará 90 grados, dibujándose las cajas en horizontal.
- `notch`: Si se le da el valor `TRUE`, las cajas se dibujarán con una muesca respecto al valor central a fin de facilitar la comparación con las demás cajas.
- `width`: Un vector con tantos elementos como cajas van a dibujarse, estableciendo el ancho relativo de unas respecto a otras.

En el siguiente ejemplo se genera una gráfica a partir de la misma fórmula, pero haciendo la primera caja más estrecha, ajustando el rango de los bigotes y añadiendo la muesca en las cajas. También se han añadido títulos, usando para ello la función `title()` antes mencionada. El resultado que obtenemos, a pesar de mostrar la misma información, es apreciablemente distinto al que producía el ejemplo previo con la función `plot()`.

Ejercicio 7.6 Gráfica de cajas generada con `boxplot()`

```
> boxplot(Petal.Length ~ Species, data = iris, notch = T,
+         range = 1.25, width = c(1.0, 2.0, 2.0))
> title(main = 'IRIS', ylab = 'Longitud pétalo',
+       sub = 'Análisis de pétalo por familia')
```

■

7.1.3 Gráficas de líneas

Uno de los tipos de gráfica más utilizados es la de líneas, especialmente cuando se quieren comparar visualmente varias variables a lo largo del tiempo o algún otro parámetro. Para generar un gráfico de este tipo con la función `plot()` habremos de dar el valor 'o'

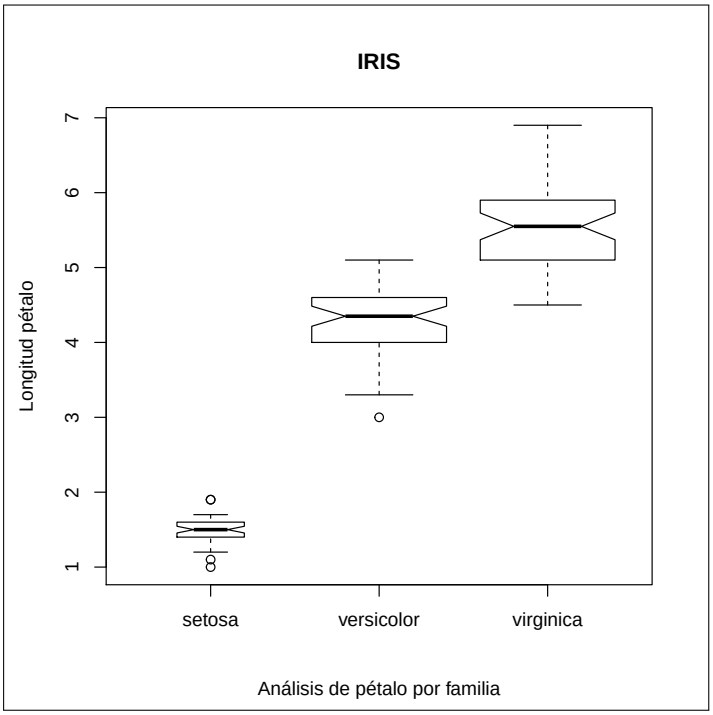


Figura 7.6: GRÁFICA DE CAJAS CON TÍTULOS EN EJES Y TÍTULO PRINCIPAL

al parámetro `type`. Además podemos ajustar algunos atributos de las líneas mediante los siguientes parámetros adicionales:

- `lty`: Tipo de línea a dibujar. Los valores 1 a 6 corresponden a distintos tipos de trazos: continuo, guionado, punteado, etc.
- `lwd`: Grosor de la línea. Por defecto es 1.
- `col`: Establece el color que tendrá la línea.
- `pch`: Símbolo a mostrar en cada punto de corte respecto al eje X.

En caso de que queramos dibujar más de una línea en la misma gráfica, la primera se generaría con la función `plot()` y la segunda se añadiría mediante la función `lines()`. La primera se encarga de inicializar los parámetros del gráfico, por ejemplo los límites de los ejes X e Y, así como de dibujar los ejes en sí y mostrar sus respectivos títulos. La segunda se limitaría a añadir líneas adicionales, generada cada una a partir de variables conteniendo el mismo número de observaciones que la original.

Sintaxis 7.5 `lines(vector, ...)`

Añade líneas a un gráfico previamente generado. Acepta la mayor parte de los parámetros gráficos de la función `plot()`.

Supongamos que queremos ver cómo cambia el precio de cierre de las subastas en eBay dependiendo del día de la semana y comparando esta evolución según que la moneda usada sea el dólar o el euro.

Comenzaremos preparando los datos a representar, extrayendo del dataset `ebay` la información que nos interesa tal y como se muestra a continuación. En la parte final del ejercicio se muestran en la consola los datos que servirán para generar la gráfica:

Ejercicio 7.7 Preparación de los datos a representar

```
> # Separar por moneda
> ebayPerCurr <- split(ebay, ebay$currency)
>
> # En cada moneda, separar por dias
> endPricePerDay <- lapply(ebayPerCurr,
+       function(curr) split(curr$ClosePrice, curr$endDay))
>
> # Precio medio de cierre para moneda
> meanPricesUS <- sapply(endPricePerDay$US, mean)
> meanPricesEUR <- sapply(endPricePerDay$EUR, mean)
> meanPricesUS[is.na(meanPricesUS)] <- mean(meanPricesUS, na.rm=T)
>
> # Obtener el rango a representar
> rango <- range(meanPricesUS, meanPricesEUR)
>
> meanPricesEUR

     Fri      Mon      Sat      Sun      Thu      Tue
20.47578 43.07168 45.63229 40.85346 25.75291 23.29060
     Wed
31.47493

> meanPricesUS

     Fri      Mon      Sat      Sun      Thu      Tue
48.21471 36.83621 39.28396 42.27805 39.71355 31.95483
     Wed
39.71355

> rango

[1] 20.47578 48.21471
```

■

En `meanPriceUS` tenemos la media de los precios de cierre por día para las transacciones en dólares, y en `meanPricesEUR` la misma información para las transacciones en euros. Además, en la variable `rango` hemos obtenido el rango total de precios de cierre, información que necesitaremos a fin de ajustar el eje Y adecuadamente.

Usamos los anteriores resultados para dibujar la gráfica. Utilizamos el parámetro `ylim` de la función `plot()` para indicarle cuál será el rango de valores a representar. Si no lo hiciésemos asi, el rango del eje Y se ajustaría usando solo los datos entregados a `plot()`, por lo que la segunda línea podría tener puntos fuera de dicho rango. También damos el valor `FALSE` a los parámetros axes y ann, indicando a la función que no debe dibujar los ejes ni tampoco mostrar los títulos asociados a estos. Toda esa información se agrega después de dibujar la segunda línea, usando para ello las funciones `axis()` y `title()`.

Ejercicio 7.8 Gráfica de líneas mostrando dos conjuntos de datos

```
> # Inicializa gráfico con la primera línea y sin ejes
> plot(meanPricesUS, type = "o", axes = F, ann = F,
+      col = "blue", ylim = rango)
> # Añade la segunda línea
> lines(meanPricesEUR, type = "o", col = "red")
>
> # Colocamos los ejes
> axis(1, at = 1:length(meanPricesUS), lab = names(meanPricesUS))
> axis(2, at = 3*0:rango[2], las = 1)
>
> # Y finalmente los títulos y leyendas
> title(main = 'Precio de cierre según día',
+       xlab = 'Día', ylab = 'Precio final')
> legend("bottomright", c("$","E"),
+        col = c("blue","red"), lty = c(1,1))
```

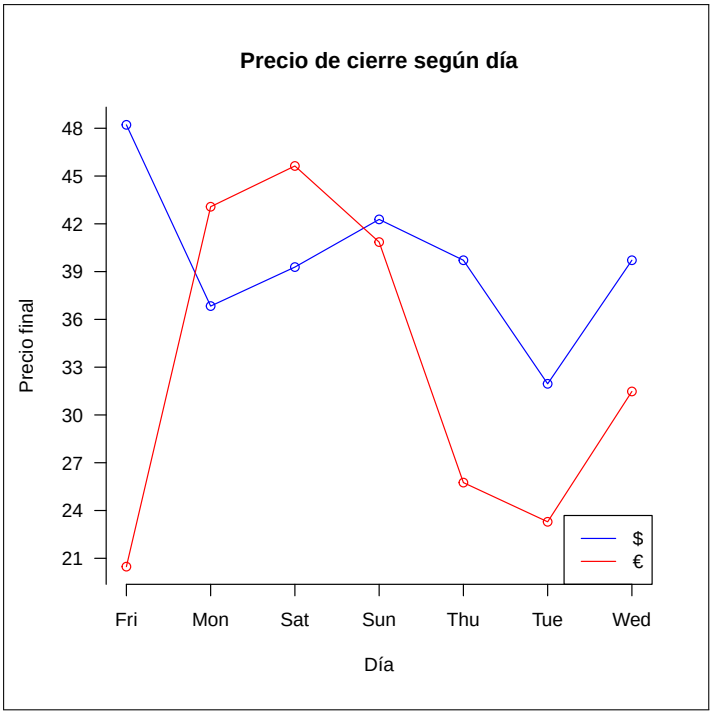


Figura 7.7: Comparación de precios de cierre por día y moneda

Puedes dar el valor `TRUE` a los parámetros `axes` y `ann` de la función `plot()` y eliminar las dos llamadas a la función `axis()` para apreciar la diferencia entre los ejes dibujados por defecto y los que se obtienen en el ejemplo previo.

7.1.4 Gráficas de barras

Otro de los tipos de representación más habitual es el que usa barras para representar los valores. La dimensión de la barra es proporcional al valor a representar, pudiendo utilizarse colores y tramas de relleno para diferenciarlas. En R este tipo de gráfica se genera con la función `barplot()`.

Sintaxis 7.6
```
barplot(objeto[, width = anchos, space = separación,
        horiz = TRUE|FALSE, beside = TRUE|FALSE,
        names.arg = títulosGrupos, legend.text = títulos])
```

El objeto a representar puede ser un vector de valores, en cuyo caso habrá un único grupo de barras, o bien una matriz con varias columnas, caso este en que los valores de cada columna se representarán como un grupo de barras. En este último caso el parámetro `names.arg` permite establecer un título para cada grupo, y el parámetro `beside` determina si las barras se dibujarán apiladas o yuxtapuestas. Asimismo, el parámetro `legend.text` permitirá configurar la leyenda asociada al gráfico. Los parámetros `width` y `space` serán vectores indicand el ancho de cada barra y la separación entre estas.

Además de los anteriores, `barplot()` acepta muchos de los parámetros gráficos que ya conocemos.

En el siguiente ejemplo se muestra cómo crear una gráfica de barras simple, a partir de un único vector de datos. Este es generado aplicando la función length() a cada una de las columnas de un *data frame*, obteniendo asi el número de transacciones por día.

Ejercicio 7.9 Gráfica de barras simple

```
> barplot(sapply(endPricePerDay$EUR, length), col = rainbow(7))
> title(main='Número de operaciones por día')
```

■

Si tenemos varios grupos de datos a representar, por ejemplo los resultados de clasificación de varios algoritmos obtenidos con dos medidas distintas, podemos preparar una matriz y a continuación entregarla como parámetro a `barplot()` para obtener varios grupos de barras, dando el valor `TRUE` al parámetro `beside`. Es lo que se hace en el siguiente ejemplo, en el que se usa el contenido del archivo CSV que obteníamos en un capítulo previo. A fin de agregar los datos de todos los datasets, calculando promedios por algoritmo, se ha utilizado la función `aggregate()` de R.

Sintaxis 7.7
```
aggregate(objeto|formula[, data = data.frame,
          by = valoresAgregado, FUN = funciónAgregado])
```

Aplica a un conjunto de datos una función de agregación, generando como resultado un *data frame* con los resultados. El primer parámetro puede ser una fórmula o una variable, normalmente una columna de un *data frame*. Si no se usa una fórmula, es preciso utilizar el parámetro `by` para indicar cuál será el criterio de agregación. El argumento `data` especifica el *data frame* al que pertenecen las variables implicadas en la fórmula. La función de agregación a aplicarse viene indicada por el parametro `FUN`.

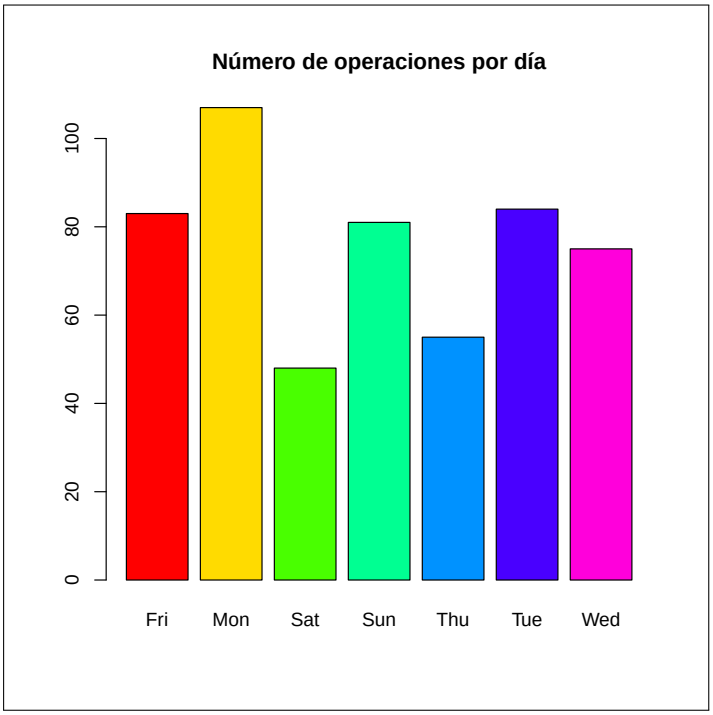


Figura 7.8: GRÁFICA DE BARRAS SIMPLE

Ejercicio 7.10 Gráfica de barras a partir de datos agregados

```
> accuracy <- aggregate(Accuracy ~ Algorithm, results, mean)
> precision <- aggregate(Precision ~ Algorithm, results, mean)
>
> valMedios <- matrix(c(precision$Precision, accuracy$Accuracy),
+                     nrow=6, ncol=2)
> rownames(valMedios) <- accuracy$Algorithm
> barplot(valMedios, beside = T, horiz = T, col = cm.colors(6),
+         legend.text = T, names.arg = c('Accuracy', 'Precision'))
```

■

7.1.5 Gráficas de sectores (circular)

Las gráficas de sectores, también conocidas como gráficas *de tarta*, se usan exclusivamente para representar la parte de un todo que corresponde a distintas componentes. Un caso típico sería representar el tiempo que una persona emplea emplea a cada tarea durante las 24 horas del día. En este caso la función que nos interesa es `pie()`.

Sintaxis 7.8
```
pie(vector[, labels = títulos, radius = radioCirc,
clockwise = TRUE|FALSE, col = colores])
```

Genera una gráfica de sectores con tantas divisiones como elementos existan en el vector facilitado como primer argumento. Los grados de cada arco serán proporcionales a los valores contenidos en dicho vector. El resto de los parámetros tienen la siguiente finalidad:

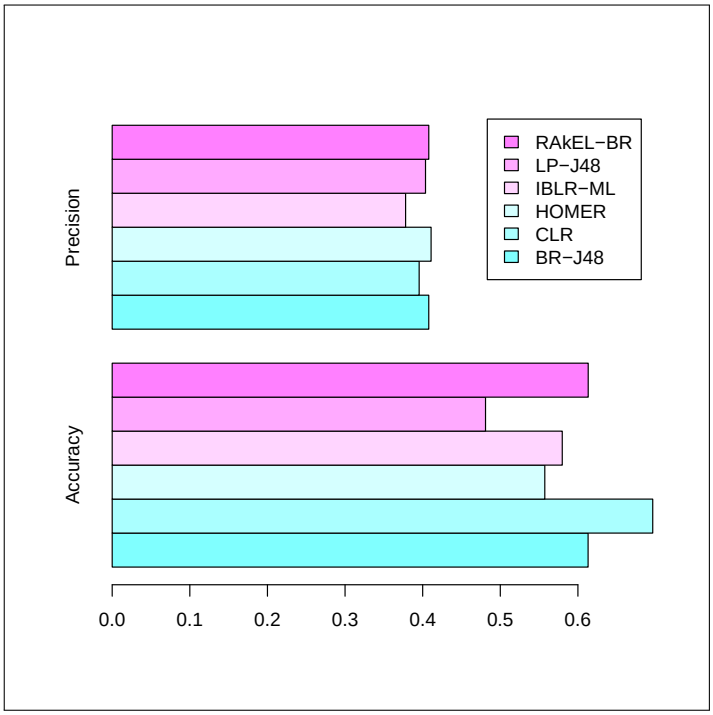


Figura 7.9: GRÁFICA DE BARRAS A PARTIR DE DATOS AGREGADOS

- `labels`: Títulos a mostrar junto a cada uno de los sectores. El orden de los títulos será el mismo que el de los valores del vector a representar.
- `cols`: Colores a utilizar para cada sector.
- `radius`: Radio del gráfico. Por defecto es 0.8, pudiendo llegar hasta 1.0. Si los títulos entregados con `labels` son extensos, este parámetro permitirá ajustar el tamaño de la gráfica para poder mostrarlos.
- `clockwise`: Determina si los valores se irán dibujando siguiendo el sentido de las agujas del reloj o el sentido inverso.

En el siguiente ejemplo se usa de nuevo la función `aggregate()`, en este caso para contar cuántas operaciones hay registradas para cada categoría en el dataset `ebay`. En la fórmula puede utilizarse cualquier variable junto con `Category`, no tiene necesariamente que ser `ClosePrice`, ya que lo que va a hacerse es contar el número de casos con `length()`, en lugar de aplicar cualquier otro cálculo. De los datos agregados tomamos las 8 primeras filas y las representamos en un gráfico de sectores, mostrando junto a cada sector el número de operaciones y agregando también una leyenda para identificar las categorías.

Ejercicio 7.11 Gráfica de sectores mostrando proporción de productos por categoría

```
> # Obtener número de operaciones por categoría
> opPorCategoria <- aggregate(
+   ClosePrice ~ Category,
+   ebay, length)[1:8,] # Tomar solo las primeras 8 filas
>
> colores <- topo.colors(length(opPorCategoria$Category))
>
> pie(opPorCategoria$ClosePrice,
```

```
+     labels = opPorCategoria$ClosePrice,
+     col = colores, main='Productos por categoría')
>
> legend("bottom", "Categoría", opPorCategoria$Category,
+        cex = 0.6, fill = colores, ncol = 4)
```

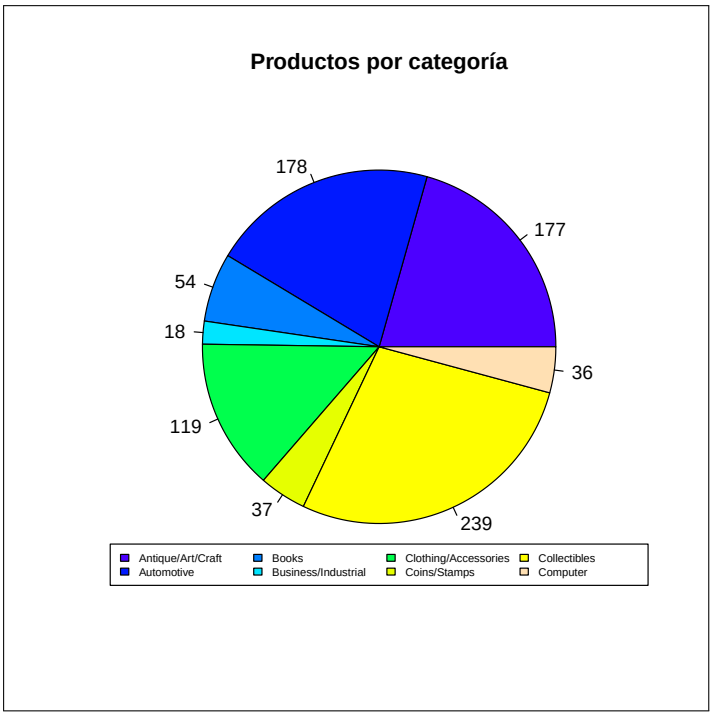


Figura 7.10: PROPORCIÓN DE PRODUCTOS POR CATEGORÍA EN GRÁFICA DE SECTORES

El parámetro `cex` usado en el ejemplo previo con la función `legend()` tiene la finalidad de reducir el tamaño de letra usado en las leyendas, de forma que sea posible visualizarlas en el espacio disponible.

7.2 Histogramas

Cuando es necesario analizar la distribución de una variable con un gran conjunto de valores, una de las herramientas habituales es el histograma. Se trata de un gráfico de barras con una configuración específica: el rango de los valores a representar se divide en intervalos, el ancho de las barras es proporcional a la amplitud de cada intervalo y su altura lo es a la frecuencia del rango de valores representados (el número de casos en que la variable toma algún valor en dicho intervalo).

Habitualmente la amplitud de los intervalos es idéntica, por lo que las barras tendrian la misma anchura, caso en el que prestaríamos atención especialmente a la altura de cada barra.

7.2.1 Histograma básico

Teniendo un vector con los valores a representar, podemos generar un histograma entregándolo como parámetro a la función `hist()`. Esta se encargará de definir los intervalos,

hacer el conteo de valores existentes para cada uno y elaborar la gráfica, como se aprecia en el ejemplo siguiente. En él se quiere estudiar la distribución de la elevación del terreno para el dataset covertype, obteniéndose el resultado que puede verse en la Figura 7.11. En ella puede comprobarse que la mayor parte de los casos estudiados tienen una elevación en torno a los 3000 metros.

Ejercicio 7.12 Histograma básico

```
> hist(covertype$elevation,
+      main = 'Elevación del terreno', xlab = 'Metros')
```

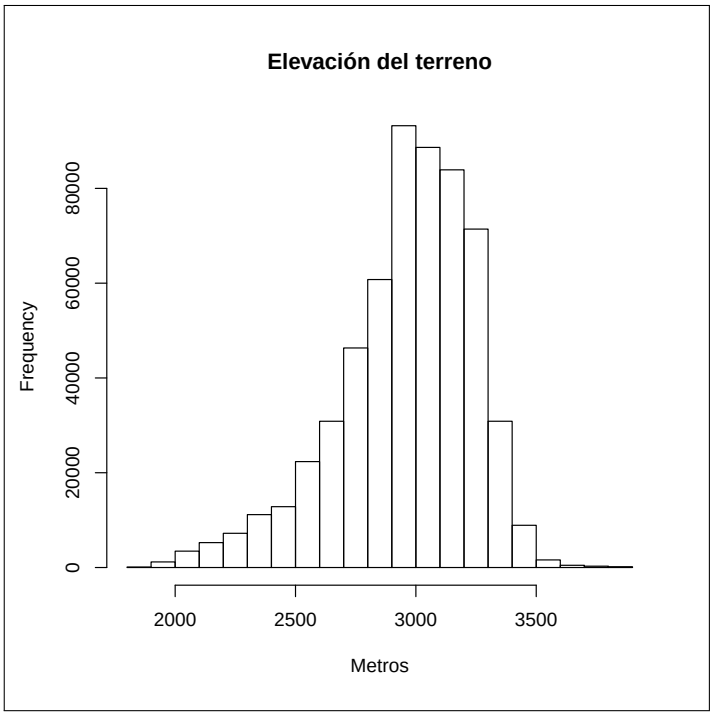


Figura 7.11: HISTOGRAMA BÁSICO

La gráfica anterior es una representación de un conjunto de datos preparado por la función `hist()`. Podemos almacenar dicha información en una variable y comprobar cuál es su estructura, como se hace a continuación:

Ejercicio 7.13 Información generada por la función `hist()`

```
> # El parámetro plot = FALSE desactiva la visualización
> histograma <- hist(covertype$elevation, plot = F)
>
> str(histograma, strict.width = 'wrap')

List of 6
$ breaks : num [1:22] 1800 1900 2000 2100 2200 2300 2400
   2500 2600 2700 ...
$ counts : int [1:21] 93 1180 3456 5255 7222 11161 12837
```

```
   22350 30864 46331 ...
$ density : num [1:21] 1.60e-06 2.03e-05 5.95e-05 9.04e-05
   1.24e-04 ...
$ mids : num [1:21] 1850 1950 2050 2150 2250 2350 2450 2550
   2650 2750 ...
$ xname : chr "covertype$elevation"
$ equidist: logi TRUE
- attr(*, "class")= chr "histogram"
```

Como puede apreciarse, en este caso hay datos sobre 21 intervalos. Los atributos `breaks` y `mids` indican contienen los valores que corresponden a la división de cada intervalo y su punto medio, mientras que los atributos `counts` y `density` almacenan la frecuencia de valores en cada intervalo y su densidad.

Aunque podriamos manipular directamente el contenido de esos atributos, mediante los parámetros aceptados por `hist()` es posible personalizar los datos generados para elaborar la gráfica, por ejemplo modificando el número de divisiones.

Sintaxis 7.9
```
hist(vector[, breaks = divisiones, labels = etiquetas,
freq = TRUE|FALSE, right = TRUE|FALSE
plot = TRUE|FALSE])
```

Toma los valores existentes en el vector de entrada, define los intervalos de acuerdo a la configuración del parámetro `break` y lleva a cabo el conteo de valores para cada intervalo, calculando también su densidad. Los parámetros de configuración son los siguientes:

- `breaks`: Este parámetro puede ser un número entero indicando el número de intervalos que se desea obtener en el histograma, así como un vector de valores especificando los puntos de división de dichos intervalos. También puede ser una cadena, especificando el algoritmo que se utilizará para calcular los intervalos, asi como el nombre de una función que se usaría para realizar dicho cálculo.
- `labels`: Puede tomar un valor lógico, que en caso de ser `TRUE` mostraría sobre cada barra del histograma su frecuencia, o bien un vector con tantas etiquetas como intervalos. Esas etiquetas se mostrarían sobre las barras.
- `freq`: Por defecto el eje Y muestra la frecuencia, el conteo de número de casos. Dándole el valor `FALSE` se mostraría la densidad en lugar de la frecuencia.
- `right`: Por defecto toma el valor `TRUE`, de forma que los intervalos sean abiertos por la izquierda y cerrados por la derecha.
- `plot`: Controla la visualización de la gráfica. Dándole el valor FALSE solamente se devolverá la estructura de datos generada por la función `hist()`, sin mostrar el histograma.

7.2.2 Personalización de divisiones y colores

Además de los parámetros específicos, la función `hist()` también acepta muchos de los parámetros gráficos que hemos ido conociendo en apartados previos de este capítulo. Podemos, por ejemplo, establecer el color de las barras mediante el atributo `col`. Hasta ahora siempre hemos asignado a dicho parámetro un color o un vector de colores. También es posible utilizar una función para determinar el color, algo que en el caso de los histogramas

resulta interesante ya que podemos usar la información sobre los intervalos para establecer un color u otro.

En el siguiente ejercicio se usa el parámetro breaks para efectuar 100 divisiones. El color de estas vendrá determinado por los valores representados, para aquellos inferiores a 2500 se usará el verde, para los que estan entre 2500 y 3000 el azul y para los superiores a 3000 el rojo. La función `ifelse()` actúa como el habitual condicional if, tomando como primer argumento un condicional cuyo resultado determinará si se devuelve el segundo o tercer argumento. Como se aprecia en la Figura 7.12, el resultado es mucho más atractivo, visualmente hablando, que en el caso anterior.

Ejercicio 7.14 Personalización del histograma

```
> plot(histograma, col = ifelse(histograma$breaks < 2500, 'green',
+                ifelse(histograma$breaks > 3000, "red", "blue")),
+      main='Elevación del terreno', xlab='Metros')
```

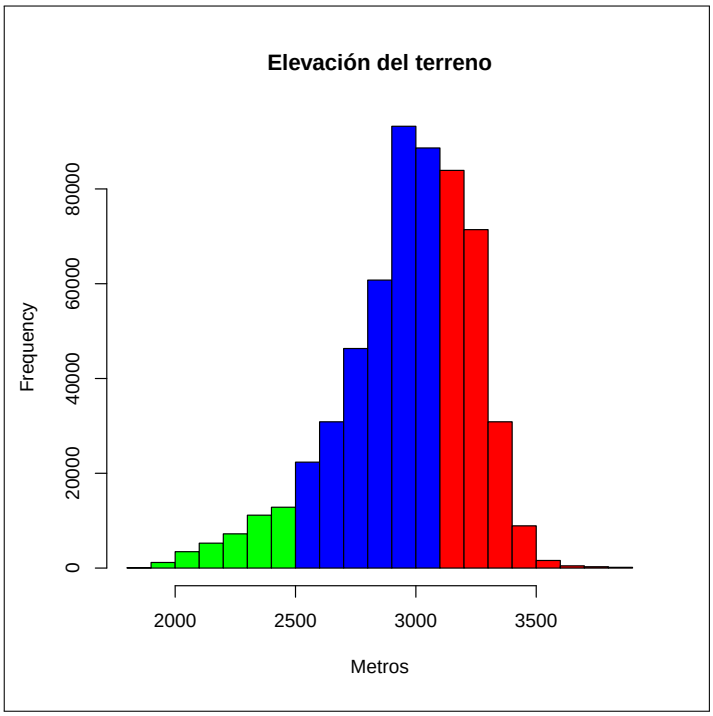


Figura 7.12: PERSONALIZACIÓN DEL HISTOGRAMA

7.2.3 Curva de densidad

Al trabajar con variables cuantitativas continuas, como es el caso de la elevación del terreno de las muestras en `covertype`, el número de divisiones que es posible realizar en el histograma es, en teoría, infinito. Cuantas más divisiones se haga más estrechas serán las barras, llegando a convertirse en líneas que solamente tienen altura, no anchura, y cuyos extremos son puntos que dibujan una curva. Esta seria la curva de densidad de la variable.

Podemos estimar la curva de densidad de una variable a partir de un vector que contiene sus valores, usando para ello la función `density()`. El resultado puede dibujarse mediante

la función `plot()`, como una especial nube de puntos, tal y como se hace en el siguiente ejercicio.

Sintaxis 7.10 `density(vector[, adjust = multiplicador, bw = factorSuavizado])`

Estima la curva de densidad a partir de los valores contenidos en el vector entregado como primer argumento. El parámetro `adjust` es un multiplicador que se aplica sobre el factor de suavizado o *bandwidth*. Este también puede personalizarse mediante el parámetro `bw`.

Ejercicio 7.15 Curva de densidad sin histograma

```
> plot(density(covertype$elevation, adjust = 5),
+      col = 'black', lwd = 3)
```

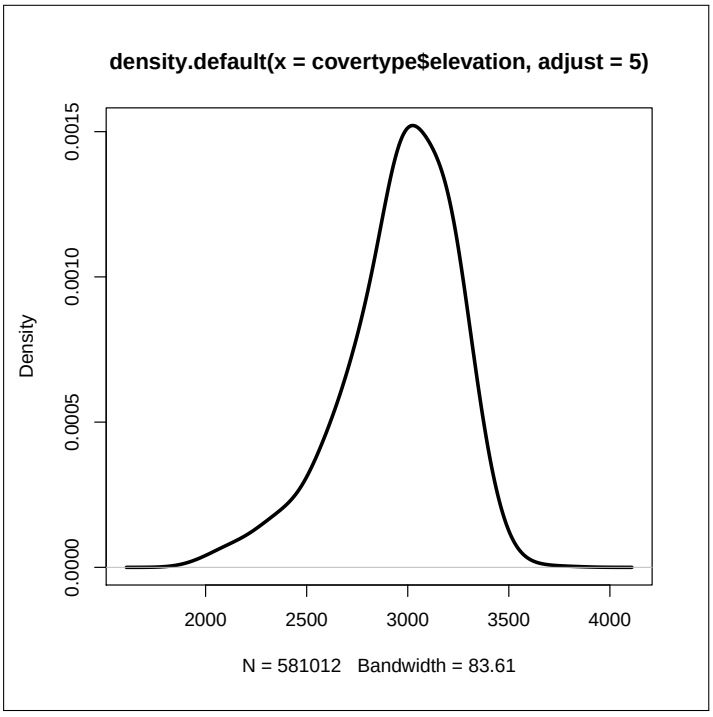


Figura 7.13: CURVA DE DENSIDAD SIN HISTOGRAMA

Si lo deseamos, podemos superponer la curva de densidad al histograma, usando la función `lines()` en lugar de `plot()` para dibujar la curva, tal y como se muestra en el siguiente ejemplo (véase la Figura 7.14):

Ejercicio 7.16 Histograma y curva de densidad

```
> hist(covertype$elevation,
+      prob = T, col = "grey",
+      main = 'Elevación del terreno', xlab = 'Metros')
>
```

```
> lines(density(covertype$elevation, adjust = 5),
+       col = 'black', lwd = 3)
```

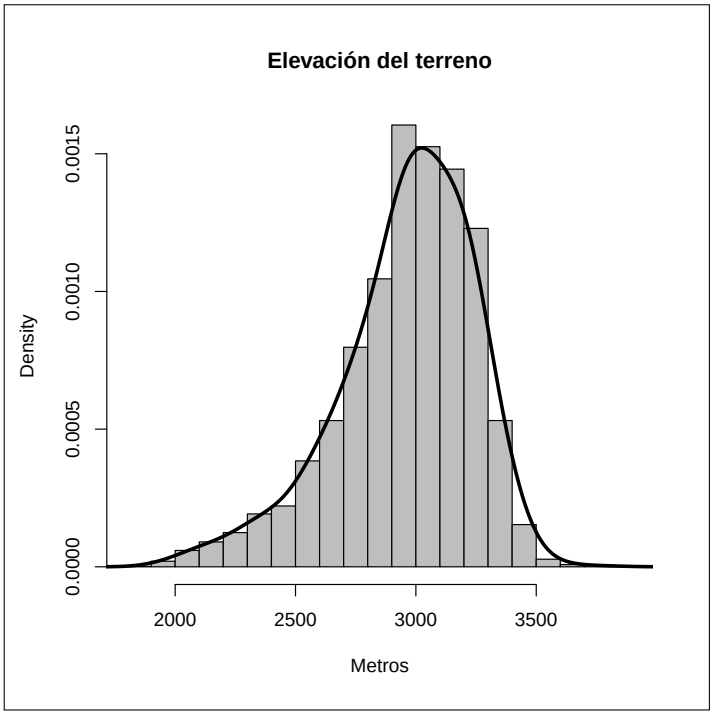


Figura 7.14: HISTOGRAMA Y CURVA DE DENSIDAD

7.2.4 Histogramas de objetos complejos

En los ejemplos previos siempre hemos usado la función `hist()` para procesar un vector de valores, concretamente una variable de un *data frame*. Si se entrega como parámetro de entrada un objeto complejo, como puede ser un *data frame*, el resultado será la obtención de una gráfica múltiple conteniendo un histograma para cada variable.

La Figura 7.15 es el resultado producido por el siguiente ejemplo, en el que se entrega a `hist()` el dataset `iris` completo, sin más parámetros ni ajustes.

Ejercicio 7.17 Histograma de objetos complejos

```
> hist(iris)
```

7.3 Cómo agrupar varios gráficos

En el último ejemplo de la sección previa (véase la Figura 7.15) hemos comprobado que es posible obtener un único resultado con múltiples gráficas. Esta es una técnica que puede resultarnos útiles en diversos contextos, ya sea simplemente para explorar un dataset o bien para preparar un informe o documento similar.

La finalidad de esta sección es describir las distintas vías que nos ofrece R para agrupar varios gráficos en un único resultado.

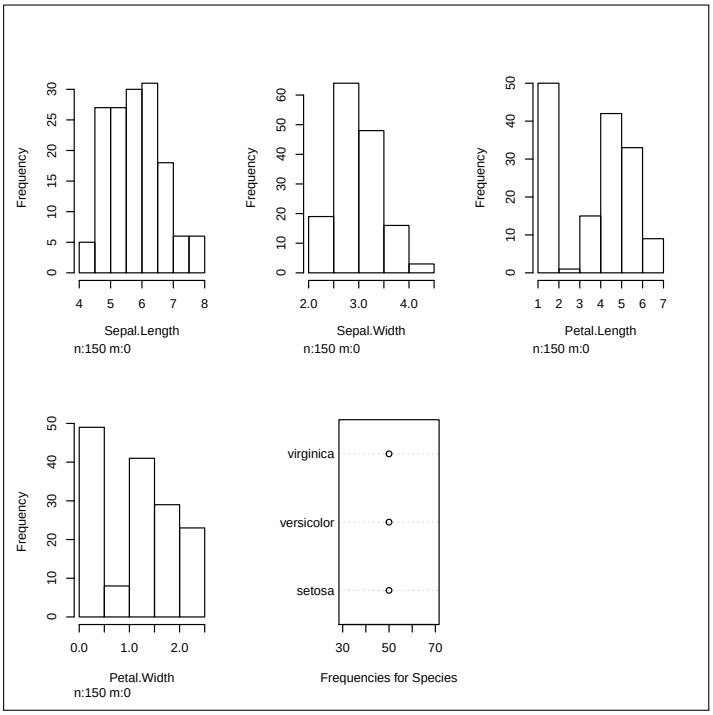


Figura 7.15: HISTOGRAMA DE OBJETOS COMPLEJOS

7.3.1 Gráficas cruzadas por atributos

La función `plot()` no solamente acepta uno o dos vectores de valores a representar, también es posible entregar como primer parámetro una estructura compleja, como puede ser un *data frame*, conteniendo varias variables. En estos casos lo que se obtiene es una gráfica compuesta, con cada una de las combinaciones por pares de las variables facilitadas.

En el siguiente ejercicio se entregan a `plot()` las cuatro variables numéricas del dataset `iris`. El resultado, como se aprecia en la Figura 7.16, es una combinación de 12 gráficas que nos permite analizar la relación entre cualquier par de variables.

Ejercicio 7.18 Combinaciones de los atributos en `iris`

```
> plot(iris[ ,1:4], col = iris$Species)
```

■

Análisis de correlación entre variables

Para casos como el anterior, en el que nos interesa analizar la correlación entre pares de variables de un dataset, podemos obtener una representación más compacta mediante la función `plotcorr()` del paquete `ellipse`. Tomando los mismos parámetros que `plot()`, esta función dibuja por cada pareja de variables una elipse cuya forma denota el tipo de correlación que existe entre ellas.

El siguiente ejercicio, previa instalación del paquete si es necearia, usa `plotcorr()` para obtener el resultado mostrado en la Figura 7.17.

Ejercicio 7.19 Gráfica de correlación entre los atributos de iris

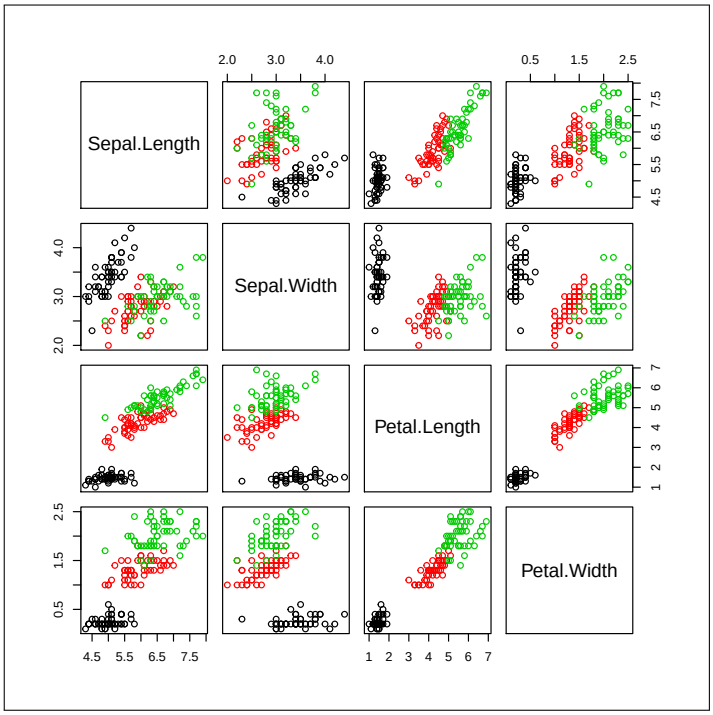


Figura 7.16: COMBINACIONES DE LOS ATRIBUTOS EN IRIS

```
> if(!is.installed('ellipse'))
+   install.packages('ellipse')
> library(ellipse)
>
> plotcorr(cor(iris[ ,1:4]), col = heat.colors(10))
```

7.3.2 Composiciones de múltiples gráficas

Si el formato producido automáticamente por `plot()` cuando se le entregan varias variables no es lo que nos interesa, siempre podemos generar gráficas individuales, ya sea con `plot()` o cualquier otra de las funciones explicadas en secciones previas, distribuyéndolas en una matriz de N filas por M columnas. Para ello configuraremos un parámetro gráfico global de dos posibles:

- `mfrow`: Toma como parámetro un vector con dos valores enteros, indicando el número de filas y de columnas en que se distribuirán los gráficos. Estos serán generados a continuación, con funciones como `plot()`, `barplot()`, `pie()` o `hist()`, y se irán distribuyendo por filas, de izquierda a derecha primero y de arriba a abajo después.
- `mfcol`: Funciona exactamente igual que el parámetro anterior, con la diferencia de que las gráficas se distribuirán por columnas, primero de arriba a abajo y después de izquierda a derecha.

El cambio de parámetros gráficos globales se lleva a cabo mediante la función `par()`. Esta establece la nueva configuración y devuelve un objeto con la anterior, lo cual nos permitirá restablecerla al final, para que gráficos que se generen posteriormente aparezcan individualmente que es lo usual.

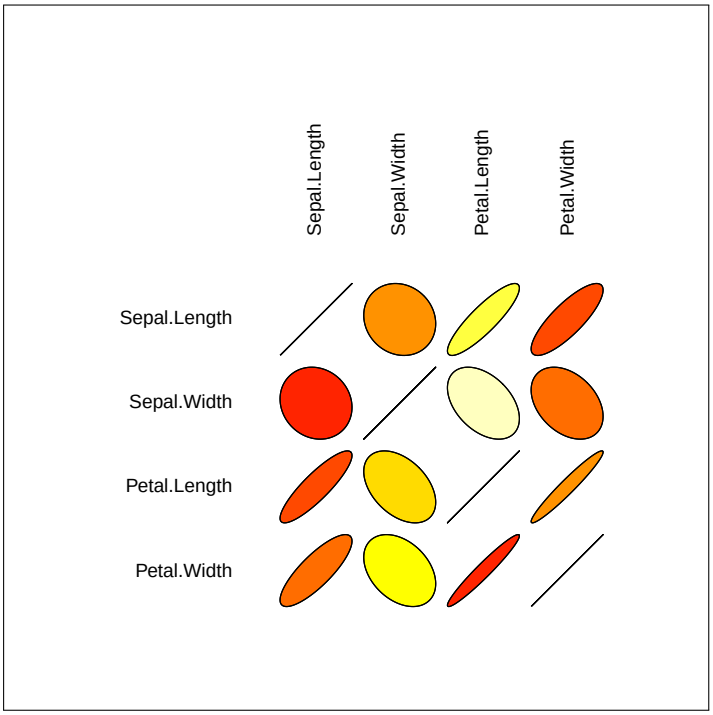


Figura 7.17: GRÁFICA DE CORRELACIÓN ENTRE LOS ATRIBUTOS DE IRIS

Sintaxis 7.11 `par(parámetroGraf = valor, ...)`

Establece nuevos valores para distintos parámetros gráficos. Consultar la documentación de esta función en R para acceder a la lista completa de parametros existentes.

En el siguiente ejercicio se usa esta técnica para obtener cuatro gráficas de nubes de puntos en una sola imagen (véase la Figura 7.18):

Ejercicio 7.20 Composición de nXm gráficas usando `par()`

```
> prev <- par(mfrow = c(2, 2))
> plot(iris$Sepal.Length, iris$Sepal.Width,
+      col = iris$Species)
> plot(iris$Petal.Length, iris$Petal.Width,
+      col = iris$Species)
> plot(iris$Sepal.Length, iris$Petal.Width,
+      col = iris$Species)
> plot(iris$Petal.Length, iris$Sepal.Width,
+      col = iris$Species)

> par(prev)
```

■

Composiciones más flexibles

Con los parámetros `mfrow/mfcol` podemos obtener en una imagen tantas gráficas como necesitemos, pero su distribución siempre ha de ser el de una matriz y todas las gráficas

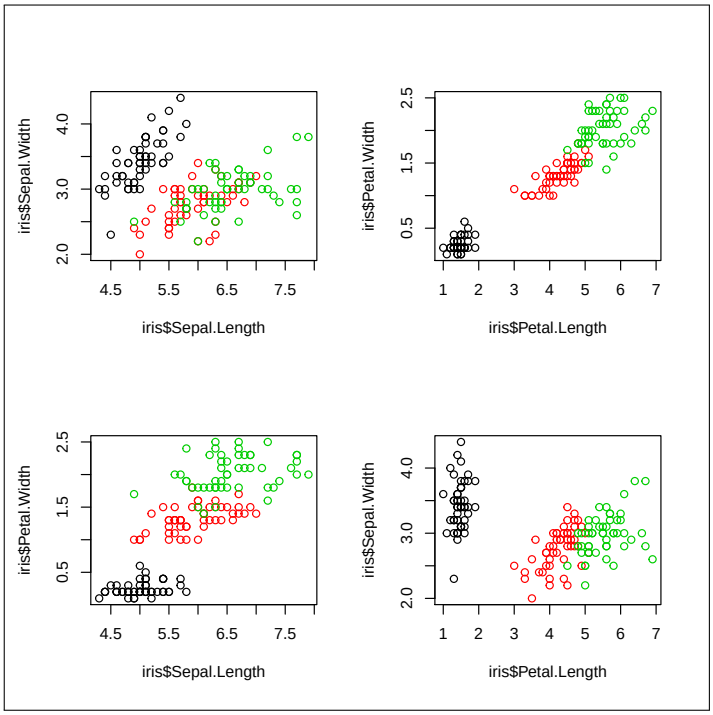


Figura 7.18: COMPOSICIÓN DE NXM GRÁFICAS USANDO `par()`

tendrán el mismo tamaño. Podemos conseguir composiciones más flexibles mediante la función `layout()`.

Sintaxis 7.12 `layout(matriz[, widths = anchos, heights = altos])`

Configura la distribución de las gráficas que se generen a continuación de acuerdo al contenido de la matriz entregada como primer argumento, esa matriz seria una representación numérica de la división espacial que se hará de la superficie de dibujo. Cada elemento de dicha matriz contendrá un valor entero, indicando el número de gráfica a dibujar en cada celda.

Opcionalmente pueden usarse los parámetros `widths` y `heights` para establecer un ancho y alto para cada columna y fila de la matriz, respectivamente. Por defecto se usa el mismo ancho para todas las columnas y el mismo alto para todas las filas.

Supongamos que queremos resumir en una imagen algunos datos del datatset `ebay`, como puede ser un histograma de la duración de las subastas, un gráfico de barras con el número de operaciones por dia y una gráfica de líneas con el precio de cierre por dia. El histograma sería más ancho y ocuparía la parte superior de la imagen. Tendríamos, por tanto, un matriz de 2×2 en la que las dos columnas de la primera fila las ocuparía el histograma, mientras que la fila inferior contendria las otras dos gráficas. Es la configuración mostrada en la Figura 7.19, generada por el siguiente ejercicio:

Ejercicio 7.21 Composición libre de gráficas usando `layout()`

```
> # Establecemos la distribución de las gráficas
> layout(matrix(c(1,1,2,3), 2, 2, byrow = T))
>
> # Un histograma
```

```
> hist(ebay$Duration, main = 'Duración subasta', xlab = 'Días')
>
> # Una gráfica de barras
> barplot(sapply(endPricePerDay$EUR, length),
+         col = rainbow(7),horiz=T,las=1)
> title(main='Operaciones por día')
>
> # Y una gráfica de líneas
> plot(meanPricesEUR, type = "o", axes = F, ann = F)
> title(main = 'Precio cierre por día',
+       ylab = 'Euros', xlab = 'Día')
> axis(1, at = 1:length(meanPricesEUR),
+      lab = names(meanPricesEUR), las = 2)
> axis(2, at = 3*0:rango[2], las = 1)

> # Restablecemos la configuración previa
> par(prev)
```

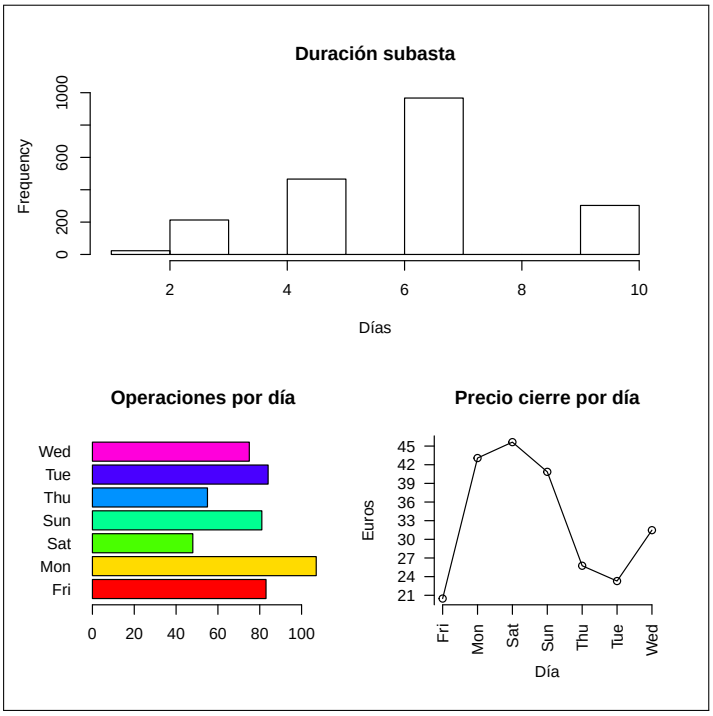


Figura 7.19: COMPOSICIÓN LIBRE DE GRÁFICAS USANDO `LAYOUT()`

7.4 Cómo guardar los graficos

El fin de muchos de los gráficos que generemos con R será formar parte de algún tipo de documento. Por ello será preciso almacenarlos en el formato apropiado, para lo cual podemos seguir fundamentalmente dos caminos posibles:

- Si estamos trabajando con RStudio, en el panel **Plots** encontraremos un menú **Export** con opciones de exportación a distintos formatos (véase la Figura 7.20). No tenemos

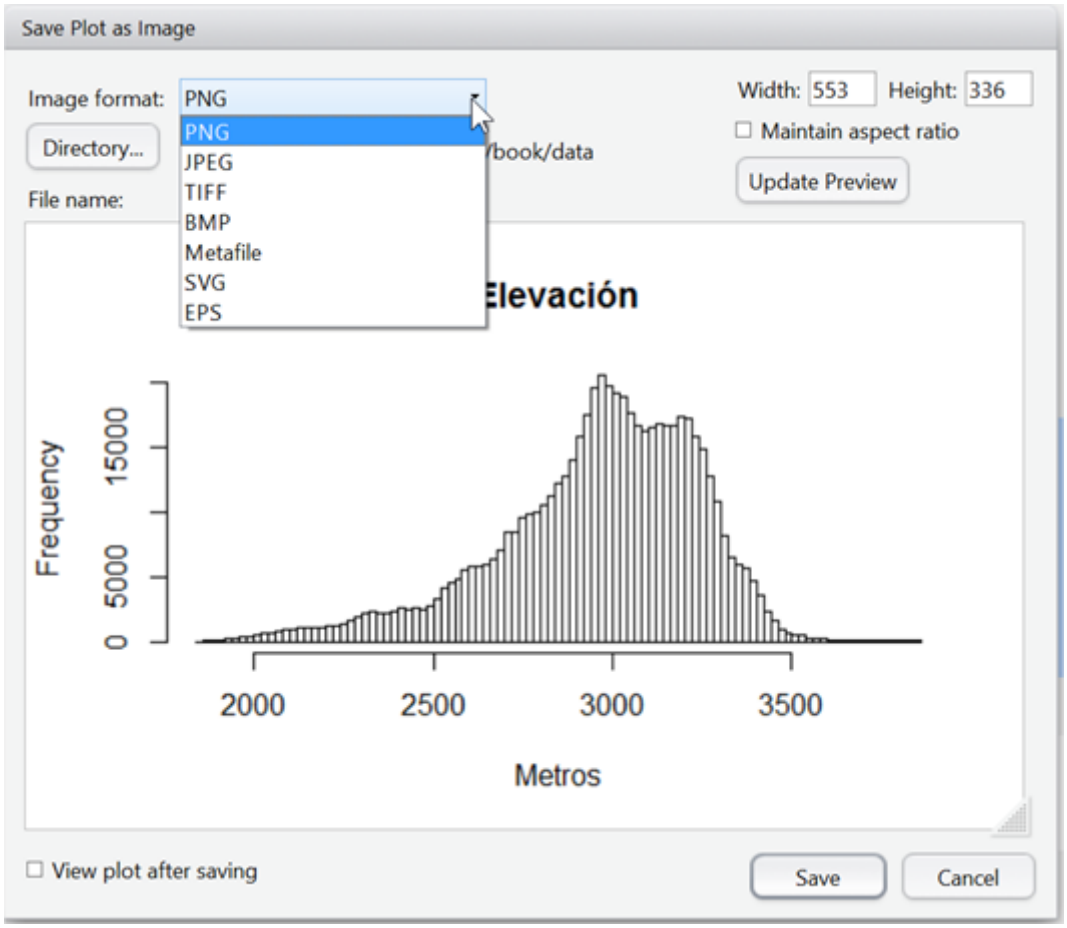


Figura 7.20: RSTUDIO NOS OFRECE MÚLTIPLES OPCIONES DE EXPORTACIÓN DE GRÁFICOS

más que elegir la que nos interese, configurar las dimensiones del gráfico, indicar el nombre del archivo y guardarlo. Este método es cómodo para guardar una gráfica de forma puntual, pero no resulta el más indicado si generamos muchas gráficas y las actualizados frecuentemente.

- Antes de generar la gráfica podemos usar funciones como `pdf()`, `png()`, `jpeg()`, `tiff()`, etc., a fin de abrir un dispositivo de salida en el formato correspondiente y dirigirlo a un archivo. A continuación generaríamos el gráfico y, finalmente, cerraríamos el dispositivo con la función `dev.off()`. Esta técnica permite generar tantos archivos gráficos como se precisen, actualizarlos automáticamente sin más que volver a ejecutar el codigo R y generar documentos PDF conteniendo múltiples gráficas.

Sintaxis 7.13

```
png(nombreArch[, width = ancho, height = alto])
jpeg(nombreArch[, width = ancho, height = alto])
bmp(nombreArch[, width = ancho, height = alto])
tiff(nombreArch[, width = ancho, height = alto])
win.metafile(nombreArch[, width = ancho, height = alto])
postscript(nombreArch[, width = ancho, height = alto])
```

Estas funciones crean un archivo en el formato adecuado, guardando en él la salida generada por funciones como `plot()`, `hist()`, `boxplot()`, `pie()`, etc. Los parámetros width y height establecen el ancho y alto de la imagen. En formatos de bapas de bits la

unidad de medida es el píxel. Para los PDF se expresa en pulgadas. Algunas de estas funciones también aceptan parámetros específicos, por ejemplo para establecer el nivel de calidad, el uso de compresión, etc.

El siguiente ejercicio generará un archivo PDF llamado `AnalisisSubastas.pdf` conteniendo tres gráficas. Tras ejecutar el código no tenemos más que abrirlo en nuestro visor de PDF para comprobar el resultado.

Ejercicio 7.22 Guardar gráficas en un archivo PDF

```
> # Abrimos el archivo PDF y establecemos las dimensiones
> pdf('AnalisisSubastas.pdf', width = 8.3, height = 11.7)
>
> # Introducimos un histograma
> hist(ebay$Duration, main='Duración subasta',xlab = 'Días')
>
> # Una gráfica de barras
> barplot(sapply(endPricePerDay$EUR, length),
+         col = rainbow(7), horiz = T, las = 1)
> title(main='Operaciones por día')
>
> # Y una gráfica de líneas
> plot(meanPricesEUR, type = "o", axes = F, ann = F)
> title(main = 'Precio cierre por día',
+       ylab = 'Euros', xlab = 'Día')
> axis(1, at = 1:length(meanPricesEUR),
+      lab = names(meanPricesEUR), las = 2)
> axis(2, at = 3*0:rango[2], las = 1)
>
> # Cerramos el archivo
> dev.off()
```

■

7.4.1 Animaciones

En ocasiones una gráfica aislada puede resultar insuficiente para transmitir una cierta idea, como puede ser la evolución de una cierta magnitud a lo largo del tiempo, el cambio que introduce en una función la modificación de ciertos parámetros, etc. Son situaciones en las que básicamente existen dos alternativas: generar varias gráficas individuales, a fin de comparar unas con otras, o combinar múltiples gráficas para producir una animación, incluso ofreciendo al usuario la posibilidad de controlarla en cierta medida. Esta segunda opción suele ser mucho más efectiva.

Desde R podemos generar animaciones a partir de cualquier secuencia de gráficas. Las imágenes individuales puedes prepararse usando cualquiera de las funciones que hemos conocido en este capítulo. Para ello, no obstante, necesitaremos instalar un paquete R que depende de la instalación de un software externo.

Instalación del paquete `animation` y sus dependencias

La instalación del paquete `animation` se realiza como la de cualquier otro paquete R. La función `install.packages()` se encargará de descargar el paquete si no se encuentra

en el sistema, asi como de configurarlo e instalarlo de forma que quede preparado para su uso. Por tanto, esta parte del procedimiento será idéntica a la de otros casos:

Ejercicio 7.23 Instalación y carga del paquete animation

```
> if(!is.installed('animation'))
+   install.packages('animation')
> library('animation')
```

■

Las funciones aportadas por este paquete dependen para su funcionamiento de un software externo que es necesario instalar por separado. Concretamente se trata de la utilidad `convert` que forma parte del software **ImageMagick**. Esta aplicación es software libre y está disponible para Windows, Linux, OS X, etc. en `http://www.imagemagick.org`.

Tras la instalación, deberíamos asegurarnos de que el directorio donde se haya instalado la utilidad `convert` se encuentra en la ruta de búsqueda del sistema, de forma que sea fácilmente accesible desde R.

Generación de una animación

El paquete `animation` aporta cuatro funciones que permiten generar una animación en otros tantos formatos. En todos los casos el primer parámetro será un bloque de código, normalmente un bucle, en el que se preparará cada una de las gráficas a incluir en la animación. Con el resto de parámetros es posible configurar el tamaño de la animación, la velocidad con que se cambiará de una imagen a la siguiente, el archivo donde sera almacenada, etc.

Sintaxis 7.14
```
saveGIF(genGráficas[, opcionesConfiguración])
saveVideo(genGráficas[, opcionesConfiguración])
saveHTML(genGráficas[, opcionesConfiguración])
saveSWF(genGráficas[, opcionesConfiguración])
saveLatex(genGráficas[, opcionesConfiguración])
```

Generan a partir de las imágenes producidas por el código facilitado como primer argumento una animación en el formato adecuado: como imagen GIF, como vídeo (MP4, AVI, etc.), como página HTML con controles de animación, como archivo SWF de Flash o como documento LaTeX con controles de animación. Algunos de los parámetros de uso más frecuente son:

- `ani.width` y `ani.height`: Establecen el tamaño que tendrá la animación. Es útil para fijar el tamaño que tendrá la imagen resultante.
- `interval`: Tiempo en segundos de intervalo entre cada paso de la animación.
- `nmax`: Número máximo de pasos en la animación.
- `loop`: Determina si la animación se repetirá de forma continua o no.
- `movie.name/latex.filename/htmlfile`: Nombre del archivo en el que se guardará la animación.

Algunas de las funciones pueden tomar parámetros adicionales. `saveHTML()`, por ejemplo, acepta los parámetros *title* y `description` para configurar el título y descripción de la página HTML.

En el siguiente ejercicio se muestra cómo usar la función `saveGIF()` para obtener un GIF animado de la función seno. El resultado mostrado en la Figura 7.21 se ha obtenido con

el mismo código, pero cambiando la función `saveGIF()` por `saveLatex()` para incluir el resultado en este documento.

Ejercicio 7.24 Generación de una animación en un archivo GIF

```
> saveGIF({
+   for(lim in seq(-3.14,3.14,by=0.1)) {
+     curve(sin, from=lim, to=lim + 9.42)
+   }
+ }, movie.name = "animacion.gif",
+ interval = 0.1, ani.width = 640, ani.height = 640)
```

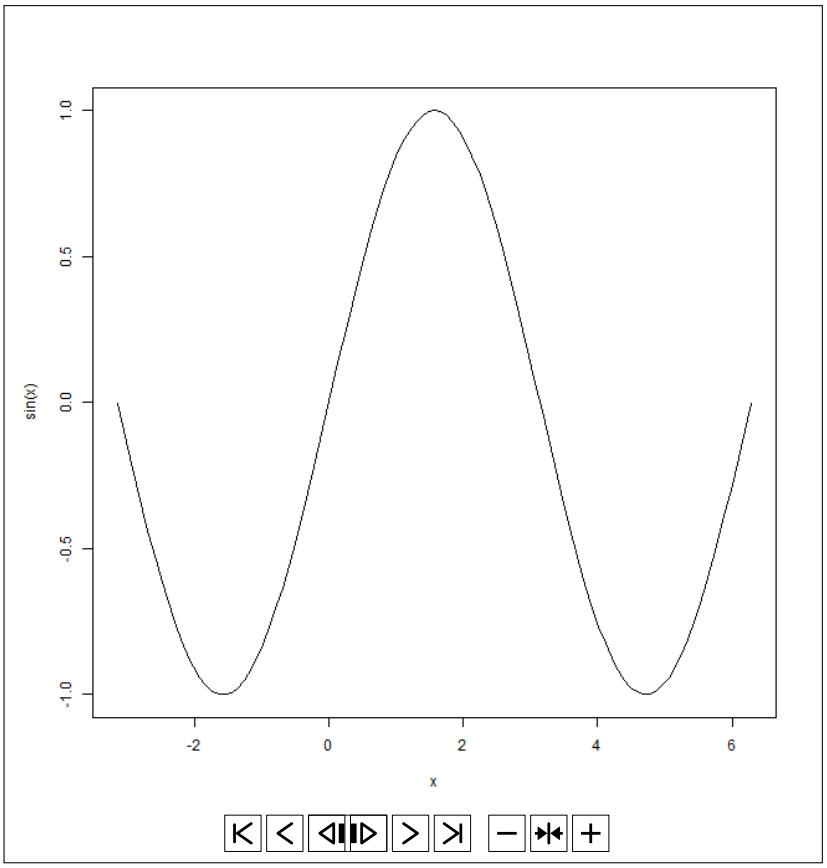


Figura 7.21: Animación de una función generada con saveLatex()

8. Gráficos con R (II)

8.1 Introducción a ggplot2

`ggplot2` es un paquete gráfico basado en el libro *The Grammar of Graphics* [Wil05]. Nos ofrece una función llamada `qplot()` que podemos utilizar para producir gráficos relativamente simples, como los mostrados en el capítulo previo, aunque con una apariencia algo más cuidada. También podemos generar un gráfico mediante la función `ggplot()`, obteniendo un objeto en el que se combinarán varios tipos de elementos: asociaciones entre las variables presentes en los datos y las entidades gráficas a dibujar, configuración de las entidades gráficas (*geoms*), operaciones estadísticas a aplicar sobre los datos, etc.

> En esta sección únicamente se ofrecerán algunos ejemplos sencillos de uso de ggplot2. En `http://docs.ggplot2.org/current/` encontraremos la documentación completa de dicho paquete, en la que se detallan las decenas de *geoms* existentes y todos los parámetros de configuración.

8.1.1 Nubes de puntos

Los *scatter plots*, o gráficas de nubes de puntos, están entre los tipos de representación gráfica más comunes y también son los más simples de generar. Con `ggplot2` únicamente tenemos que facilitar a la función `qplot()` dos atributos, cuyos valores formarán las escalas X e Y. Opcionalmente pueden usarse atributos adicionales para configurar el color de los puntos, su tamaño, símbolo, etc.

Sintaxis 8.1
```
qplot(valoresX[, valoresY, data = data.frame,
      colour = atributo, size = atributo,
      xlim = límitesEjeX, ylim = límitesEjeY,
      xlab = etiquetasEjeX, ylab = etiquetasEjeY,
      main = títuloPrincipal, geom = tipoGráfica])
```

Genera una representación gráfica de los valores facilitados acorde a la configuración especificada por los parámetros adicionales. El parámetro `geom` toma por defecto el valor `."uto"`, dibujando un histograma si solamente se facilita `valoresX` o un punto por cada dato si también se entrega `valoresY`. Si `valoresX` y `valoresY` son atributos de un

`data.frame`, en lugar de vectores de valores, mediante el parámetro `data` facilitaremos la referencia al `data.frame`.

Si únicamente se facilita una variable a representar el resultado obtenido por defecto sera un histograma con la distribución de valores de dicha variable, por lo que parámetros como `colour` y `size` no se utilizaran.

En el siguiente ejercicio se representa la longitud frente al ancho de sépalo del dataset `iris`, diferenciando cada especie de flor en un color distinto y ajustando el tamaño de cada punto según el ancho de pétalo. De esta forma estamos representando cuatro de los atributos de dicho dataset en una misma gráfica.

Ejercicio 8.1 Gráfica de nube de puntos

```
> if(!is.installed('ggplot2'))
+   install.packages('ggplot2')
> library(ggplot2)
>
> # Nube de puntos
> qplot(Sepal.Length, Sepal.Width, data = iris,
+       colour = Species, size = Petal.Width)
```

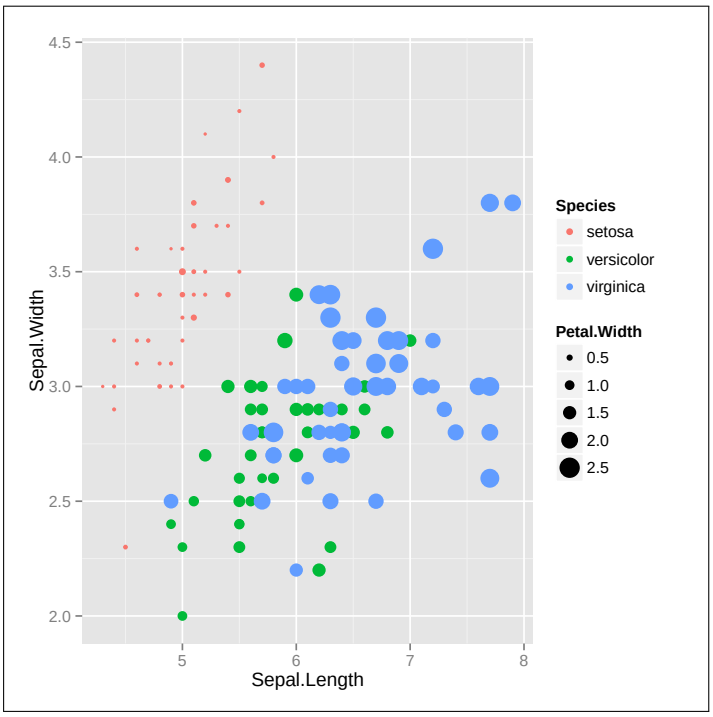


Figura 8.1: NUBE DE PUNTOS MOSTRANDO CINCO VARIABLES

8.1.2 Gráficas de líneas

La función `qplot()` devuelve un objeto con todos los parámetros necesarios para generar el gráfico, objeto al que es posible agregar otros elementos mediante el operador `+`. Los elementos más comunes que podemos añadir a un gráfico `ggplot2` son los *geoms*, nombre que reciben las distintas entidades gráficas con la que se representan los datos: puntos, líneas, barras, etc.

En `http://docs.ggplot2.org/current/` encontraremos una lista completa de las entidades existentes. Todas ellas tienen nombres del tipo `geom_ENTIDAD()`, pudiendo tomar o no parámetros según los casos. Una de ellas es `geom_line()`, gracias a la cual podemos convertir nuestra anterior gráfica de nube de puntos en una gráfica de líneas, tal y como se muestra en el siguiente ejercicio:

Ejercicio 8.2 Gráfica de líneas

```
> qplot(Petal.Length, Sepal.Length, data=iris, color=Species) +
+   geom_line()
```

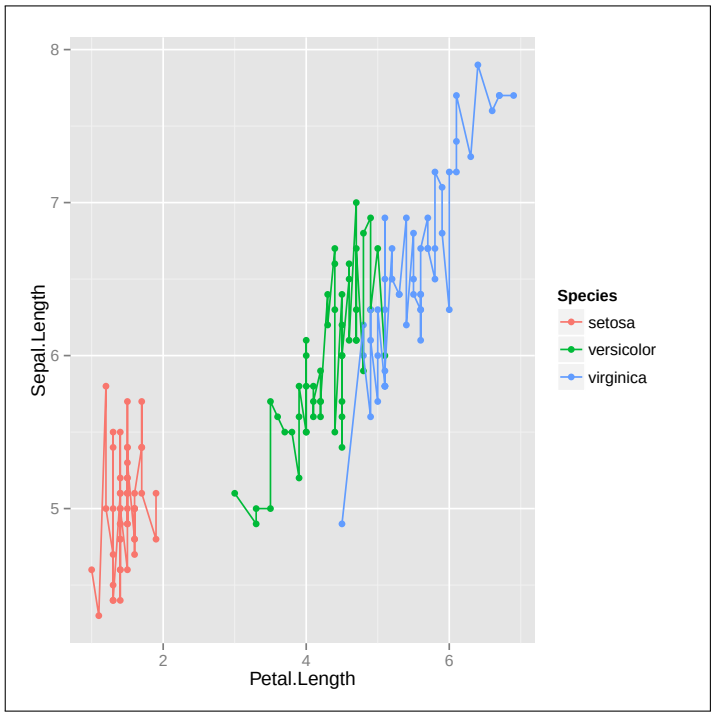


Figura 8.2: GRÁFICA DE LÍNEAS

Es posible facilitar a `geom_line()` un objeto `aes()` con información de configuración estética, incluyendo parámetros como linetype, size o colour para establecer el tipo de línea (continua, punteada, a trazos, etc.), su grosor y color. Por ejemplo:

Ejercicio 8.3 Gráfica de líneas con distintos tipos de trazo

```
> qplot(Petal.Length, Sepal.Length, data=iris, color=Species) +
+   geom_line(aes(linetype = Species))
```

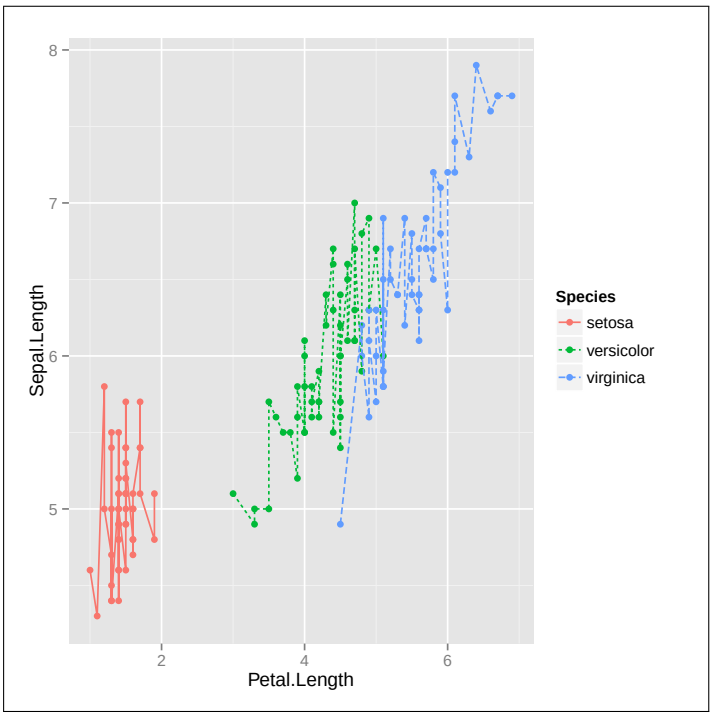


Figura 8.3: GRÁFICA DE LÍNEAS CON DISTINTOS TIPOS DE TRAZO

Cuando no se especifica de manera explícita el tipo de entidad a usar en la grafica, como hacíamos en el ejemplo previo de nube de puntos, se asume `geom_point()`.

8.1.3 Añadir curva de regresión

Es posible usar varios *geoms* en una misma gráfica, simplemente añadiéndolos con el operador +. Esta es una técnica útil para, por ejemplo, agregar curvas de regresión sobre una nube de puntos, a fin de identificar tendencias para cada categoría. Con este fin agregaremos un `geom_smooth()`. Esta función agrega por defecto una curva de regresión y también dibuja una franja con el error estándar, asumiendo un nivel del 95 % para el intervalo de confianza.

Sintaxis 8.2
```
geom_smooth([se = TRUE|FALSE,
             level = nivelIntervalo,
             method = método])
```

Con el parámetro `se` se puede desactivar la visualización de intervalo de confianza que, por defecto, siempre se dibuja. El nivel de dicho de intervalo, que es del 95 % por defecto, puede cambiarse con el parámetro `level`. Finalmente, el parámetro `method` nos permite escoger entre diferentes métodos para obtener la curva. Para obtener más información sobre los parámetros de configuración de `geom_smooth()` podemos recurrir a `http://docs.ggplot2.org/0.9.3.1/stat_smooth.html`.

Además de los anteriores, también es posible facilitar como parámetro un objeto `aes` para configurar el aspecto de las curvas, ajustando su color, grosort, etc.

El siguiente ejemplo muestra nuevamente el dataset `iris` con una curva de regresión para cada categoría:

Ejercicio 8.4 Nube de puntos con regresión entre ancho y alto de pétalo por cada especie

```
> qplot(Petal.Length, Petal.Width, data = iris, color = Species) +
+   geom_point() + geom_smooth()
```

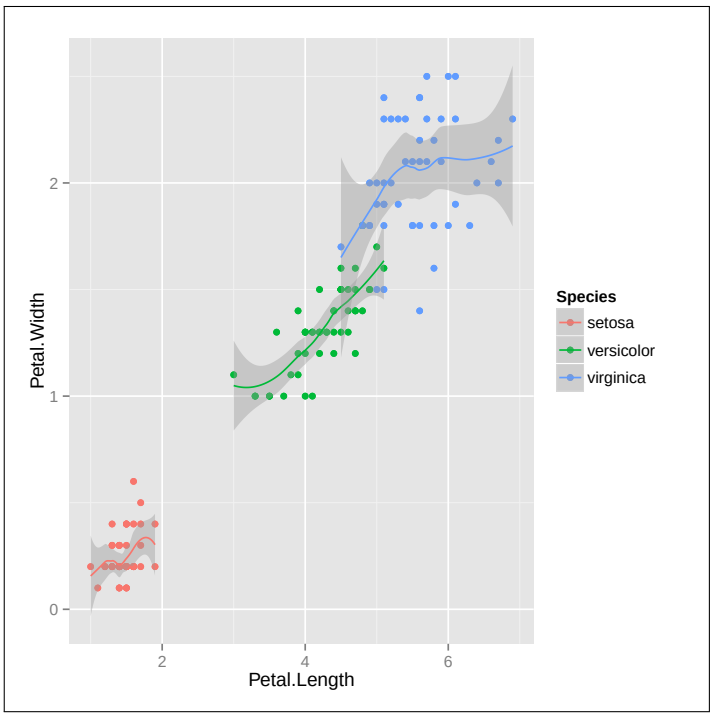


Figura 8.4: NUBE DE PUNTOS CON REGRESIÓN ENTRE ANCHO Y ALTO DE PÉTALO POR CADA ESPECIE

En ocasiones, dependiendo de la complejidad de los datos representados, el fondo gris de la gráfica puede dificultar la correcta apreciación de la franja que representa el nivel de confianza. Podemos eliminar el fondo cambiando el estilo de la gráfica, simplemente agregando `theme_bw()` a la configuración. Es lo que se hace en el siguiente ejemplo, en el que se utilizan los datos de cubierta forestal:

Ejercicio 8.5 Nube de puntos con datos de regresión y tema BW para eliminar el fondo

```
> qplot(elevation, slope, data =
+         covertype[sample(1:nrow(covertype), 500),],
+       geom = c("point", "smooth"), color = wilderness_area) +
+   theme_bw()
```

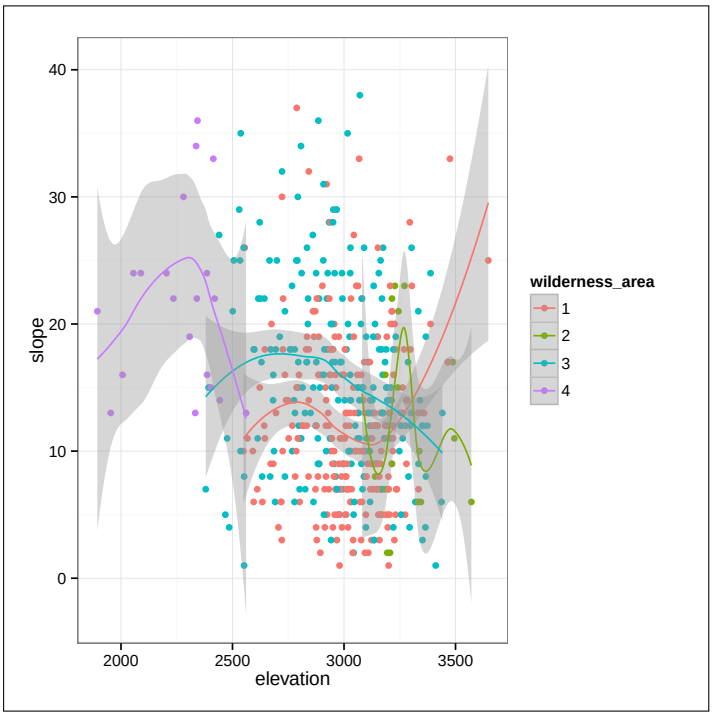


Figura 8.5: ELIMINAMOS EL FONDO PARA APRECIAR MEJOR LA GRÁFICA

8.1.4 Curva de densidad

Cuando se dispone de un gran número de muestras de datos, como ocurre en el dataset `covertype`, la representación en forma de nube de puntos, líneas o cualquier otra representación individual puede resultar insuficiente para obtener una visión general de la distribución de valores de cada variable. Es una situación en la que una curva de densidad suele aportar más información.

En este caso vamos a utilizar la función `ggplot()`, en lugar de `qplot()`, facilitando como parámetros el nombre del dataset y un objeto `aes`. Este último indicara qué variable se quiere representar y otros atributos visuales. Para indicar que deseamos generar una curva de densidad agregaremos la función `geom_density()`, tal y como se muestra en el siguiente ejemplo:

Ejercicio 8.6 Curva de densidad general

```
> ggplot(covertype, aes(x = elevation)) + geom_density()
```

En esta gráfica puede apreciarse que la elevación del terreno va desde poco menos de 2 000 metros a algo más de 3 500, encontrándose la mayor parte de las muestras en torno a los 3 000 metros.

La anterior gráfica nos permite apreciar la distribución general de la variable elegida. Podemos obtener múltiples curvas de densidad, en lugar de una general, diferenciando por tipo de área forestal. Para ello estableceremos en el objeto `aes` que cierto atributo de la

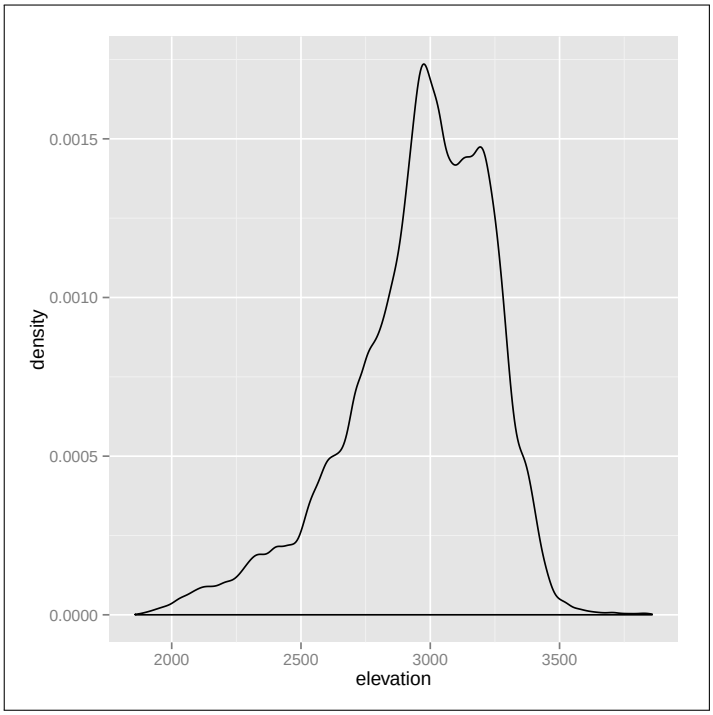


Figura 8.6: CURVA DE DENSIDAD

curva, como puede ser el color (`color`), tipo de trazo (`linetype`) o relleno (`fill`), viene determinado por el atributo que indidca la clase, que en este caso es `wilderness_area`.

El siguiente ejemplo usa la técnica que acaba de describirse, obteniendo cuatro curvas de densidad con su área rellena de distinto color. Esto nos permite saber, por ejemplo, que para el área forestal de tipo 4 la elevación es inferior que para la de tipo 2.

Ejercicio 8.7 Curva de densidad por clase

```
> ggplot(covertype,
+        aes(x = elevation, fill = wilderness_area)) +
+   geom_density()
```

■

Por defecto la estimación de densidad para cada variable se efectúa mediante una función *kernel* de tipo gausiano. Podemos elegir otro tipo de función de estimación mediante el parámetro `kernel`. Las distintas funciones disponibles son las mismas ofrecidas por la función `density()` (véase `http://www.inside-r.org/r-doc/stats/density`)

El resto de parámetros aceptados por `geom_density()` podemos encontrarlos en `http://docs.ggplot2.org/current/stat_density.html`.

8.1.5 Composición de múltiples gráficas

Indistintamente de cuál sea la entidad gráfica utilizada para representar los datos, puntos, líneas, barras, etc., siempre podemos realizar una composición de graficas en las que la

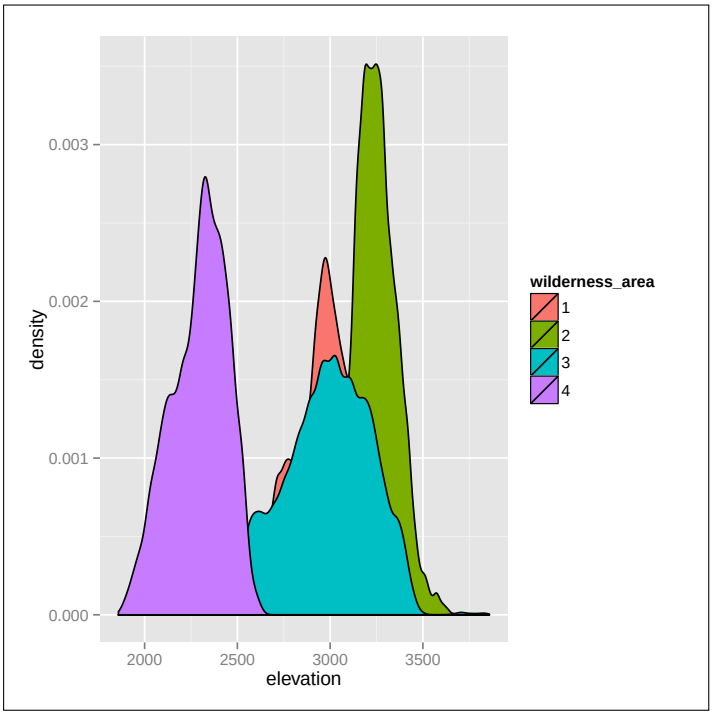


Figura 8.7: CURVA DE DENSIDAD POR CLASE DE ÁREA

información se agrupe en función de los valores de una o dos variables. Estas normalmente serán siempre de tipo `factor` y con un número de valores distintos no muy grande.

Vamos a partir de una grafica representando en forma de nube de puntos el precio de cierre de subastas en función de la valoración que tiene el vendedor, seleccionando tres dias de concretos de la semana como días de cierre y solamente transacciones en moneda europea. La configuración y aspecto inicial de esta gráfica sería la mostrada a continuación:

Ejercicio 8.8 Precio de cierre de subasta frente a valoración de los vendedores

```
> grp <- ggplot(ebay[ebay$endDay %in% c('Wed','Thu','Fri') &
+                      ebay$currency != 'US',],
+        aes(x = ClosePrice, y = sellerRating)) +
+   geom_point(aes(color = Duration))
>
> grp
```

■

ⓘ Guardar en una variable el valor devuelto por la función `ggplot()` nos permite tanto dibujar la gráfica, simplemente imprimiendo dicha variable, como agregar posteriormente otros elementos gráficos mediante el operador +.

Los datos globales mostrados en esta gráfica, almacenados en la variable `grp`, podemos separarlos en varios grupos y generar múltiples gráficas usando la función `facet_wrap()`. Esta toma como parámetro una fórmula del tipo `variable1 ~ variable2`, pudiendo utilizarse una sola variable o dos. Las gráficas se dispondrán una tras otra, primero de izquierda a derecha y después de arriba a abajo según el espacio disponible, mostrando en la parte superior el valor para cada una de las variables.

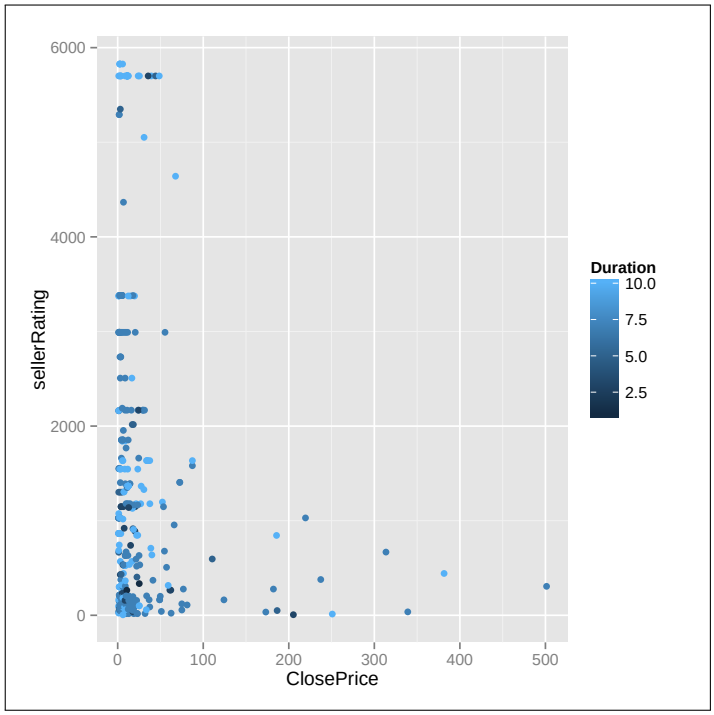


Figura 8.8: PRECIO DE CIERRE DE SUBASTA FRENTE A VALORACIÓN DE LOS VENDEDORES

En el siguiente ejemplo, partiendo de los datos de la gráfica previa, se agrupan los datos por moneda y día de finalización de la subasta:

Ejercicio 8.9 Facets: Representar 5 variables en una gráfica

```
> grp + facet_wrap(currency ~ endDay) # Una gráfica tras otra
```

Si conocemos exactamente el número de graficas que van a generarse, y queremos disponerlas en un número concreto de filas y columnas, podemos usar los parámetros `nrow` y `ncol` de `facet_wrap()`, respectivamente. Solamente es preciso establecer uno de los dos. El siguiente ejercicio muestra los mismos datos del ejemplo precio, pero a cuatro columnas en lugar de dos filas por dos columnas.

Ejercicio 8.10 Facets: Configuración del número de columnas

```
> grp + facet_wrap(currency ~ endDay, ncol = 4) # Cuatro columnas
```

Una alternativa a `facet_wrap()` es la función `facet_grid()` que, partiendo de la misma fórmula de entrada, generará una cuadrícula de gráficas disponiendo los valores de variable1 como filas y los de variable2 como columnas. Esto provoca que se incluyan en la composición incluso gráficas para las que no existen datos, tal y como se aprecia en el siguiente ejemplo:

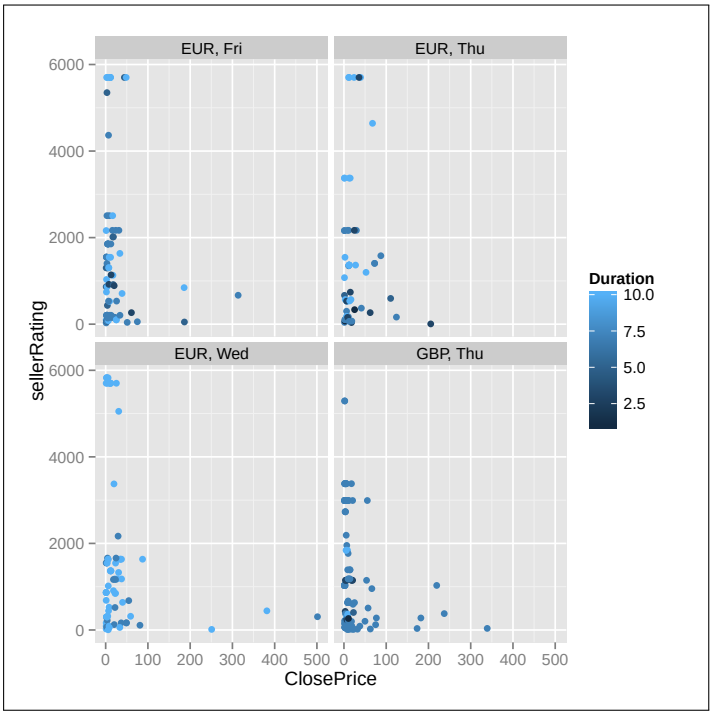


Figura 8.9: FACETS: REPRESENTAR 5 VARIABLES EN UNA GRÁFICA

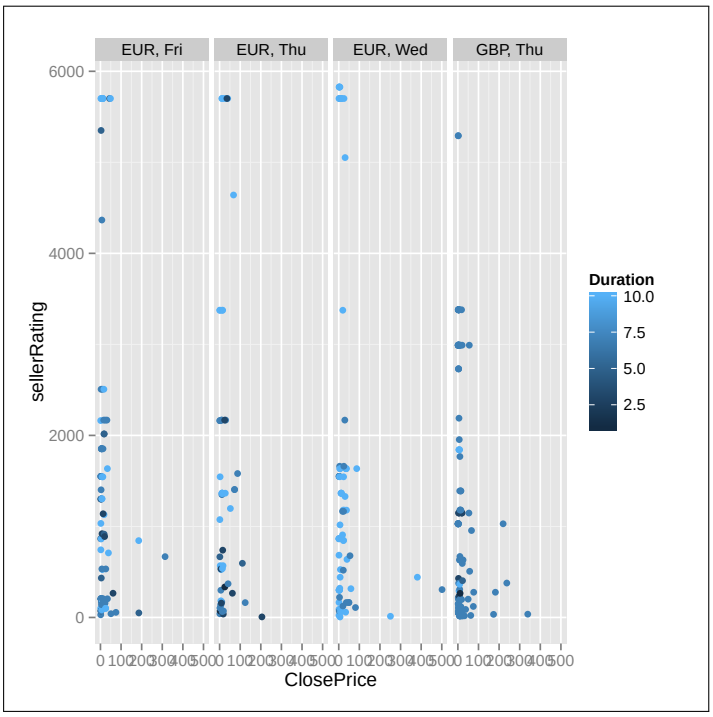


Figura 8.10: FACETS: CONFIGURACIÓN DEL NÚMERO DE COLUMNAS

Ejercicio 8.11 Facets: Representar 5 variables en una gráfica en formato cuadrícula

```
> grp + facet_grid(currency ~ endDay) # Formato cuadrícula
```

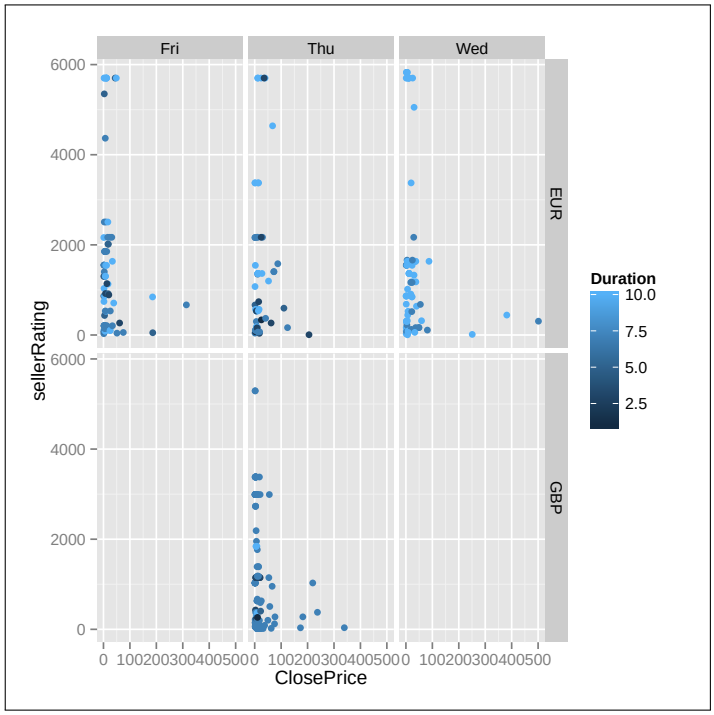


Figura 8.11: FACETS: REPRESENTAR 5 VARIABLES EN UNA GRÁFICA EN FORMATO CUADRÍCULA

Escalas y espaciado

Por defecto todas las gráficas de una composición compartirán la misma escala para el eje X y también para el eje Y. En ocasiones esto puede resultar inadecuado si la distribución de los datos entre gráficas no es relativamente homogénea, lo cual dificultará su interpretación.

Tanto `face_wrap()` como `face_grid()` aceptan un parámetro, llamado `scales`, que puede tomar cuatro valores posibles: `"fixed"`, `"free_x"`, `"free_y"` y `"free"`. El primero es el que se usa cuando no se especifica lo contrario, generando un escala fija para cada eje común a todas las gráficas. Con los dos valores siguientes obtendríamos escalas independientes para el eje X o para el eje Y, mientras que el último ambas escalas serían independientes.

El siguiente ejercicio vuelve a usar los mismos daos de los ejemplos previos, pero en este caso liberando ambos ejes para cada gráfica. La diferencia es apreciable a simple vista:

Ejercicio 8.12 Facets: Escalas independientes

```
> grp + facet_wrap(currency ~ endDay, scales = "free")
```

A pesar de que las escalas sean independientes, el espacio utilizado para dibujar cada gráfica de la composición será por defecto idéntico al del resto, es decir, el espacio disponible se reparte proporcionalmente entre el número de filas y columnas que existan. Utilizando `facet_grid()` es posible modificar esta configuración por defecto, de tal forma que el ancho y/o alto de cada gráfica esté en concordancia con la escala asociada. Para ello ha de utilizarse el parámetro `space`, que acepta los mismos cuatro valores que `scales`.

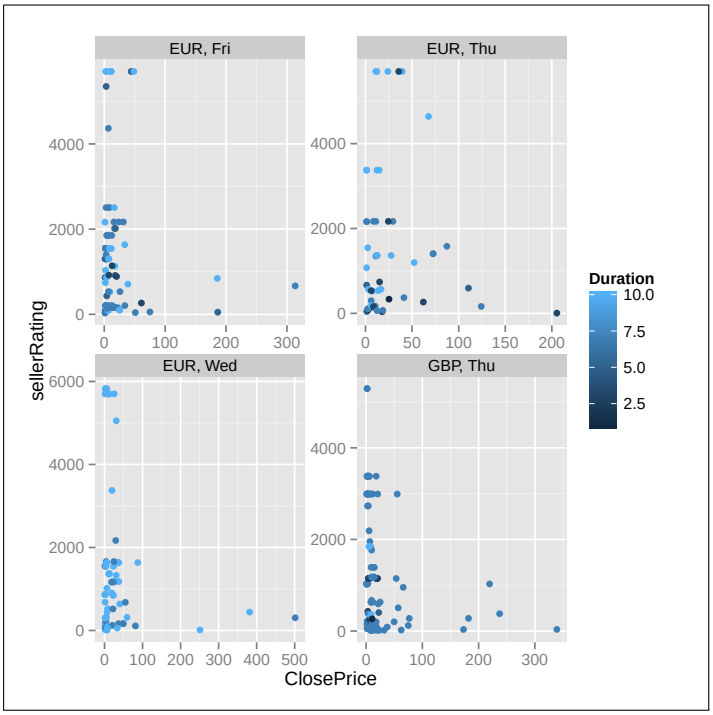


Figura 8.12: ESCALAS INDEPENDIENTES

Gráficas múltiples con otras entidades gráficas

Como se indica anteriormente, en lugar de dos variables podemos agrupar los datos usando solo una de ellas, usando para ello la sintaxis `~variable`. Asimismo las gráficas pueden usar otro tipo de entidad. En el siguiente ejemplo usamos `geom_bar()` para obtener una gráfica de barras en cada grupo:

Ejercicio 8.13 Facets: Precio de cierre por moneda y día de finalización

```
> qplot(currency, ClosePrice, data=ebay[ebay$endDay != 'Wed',],
+       fill = currency) + geom_bar(stat = 'identity') +
+   facet_wrap(~endDay)
```

■

Podemos encontrar información sobre todo los parámetros aceptados por `facet_wrap()` y `facet_grid()` en `http://docs.ggplot2.org/0.9.3.1/facet_wrap.html` y `http://docs.ggplot2.org/0.9.3.1/facet_grid.html`, respectivamente.

8.2 Otras posibilidades gráficas

Además del paquete base (`graphics`) y de `ggplot2`, que posiblemente es el más conocido, existen multitud de paquetes R para generación de diferentes tipos de gráficas. En los apartados de esta sección se mencionan algunos de ellos, así como otras funciones del paquete base que no están tan enfocadas a la representación de conjuntos de datos que, hasta ahora, es lo que se ha perseguido en todos los ejercicios propuestos.

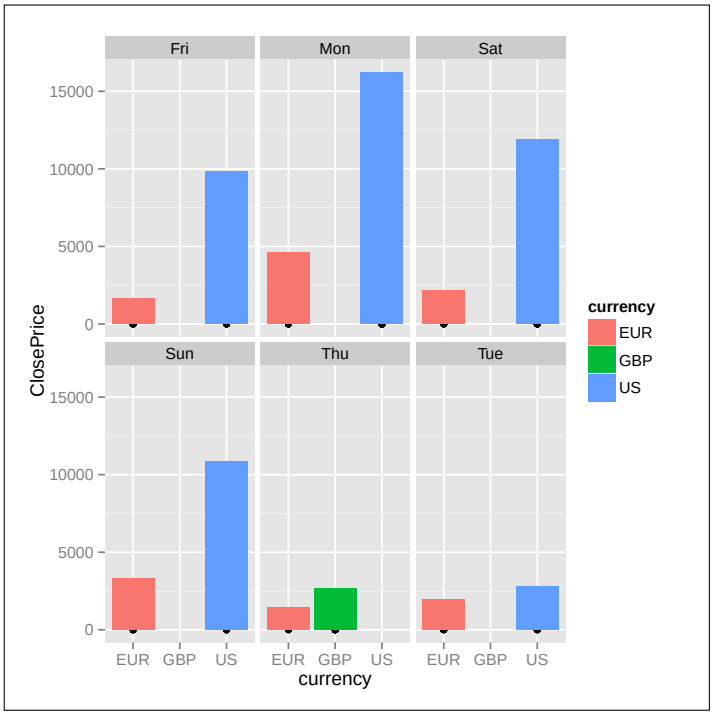


Figura 8.13: FACETS: PRECIO DE CIERRE POR MONEDA Y DÍA DE FINALIZACIÓN

8.2.1 Dibujo de funciones y polinomios

En el paquete base de R hay dos funciones que nos permiten dibujar líneas a partir de un conjunto de puntos, de una función predefinida o de una expresión como podria ser un polinomio.

Conectando puntos con líneas

Partiendo de las coordenadas X,Y de una serie de puntos, que pueden estar dibujados o no según interese, es posible conectar unos puntos con otros mediante líneas rectas usando la función `lines()`. Esta forma parte del paquete base de R, por lo que podemos utilizarla sin necesidad de importar paquete alguno.

Sintaxis 8.3 `lines(x, ...)`

Esta función agrega a una gráfica previamente generada, por ejemplo con la función `plot()`, segmentos de línea conectando los puntos del vector entregado como parámetro. Además del citado vector, esta función acepta la mayoría de los parámetros gráficos de `plot()`, incluyendo `lwd` (grosor de las líneas) y `lty` (tipo de trazo).

En el siguiente ejemplo se muestra cómo generar una serie de puntos en un intervalo para las `X`, de `-10` a `10` y calculando el valor de las `y` mediante la función `tan`. A continuación se usa `plot()` para preparar la grafica con esos puntos, aunque en realidad no llegan a dibujarse ya que se usa el parametro `type = "n"`. Finalmente se dibujan los segmentos de línea conectando los puntos, entregando a `lines()` los dos vectores de coordenadas en forma de matriz:

Ejercicio 8.14 Líneas a partir de función existente (tan)

```
> x <- seq(-10, 10, .25)
> y <- tan(x)
> plot(x, y, type = "n")
> lines(matrix(c(x,y), ncol = 2))
```

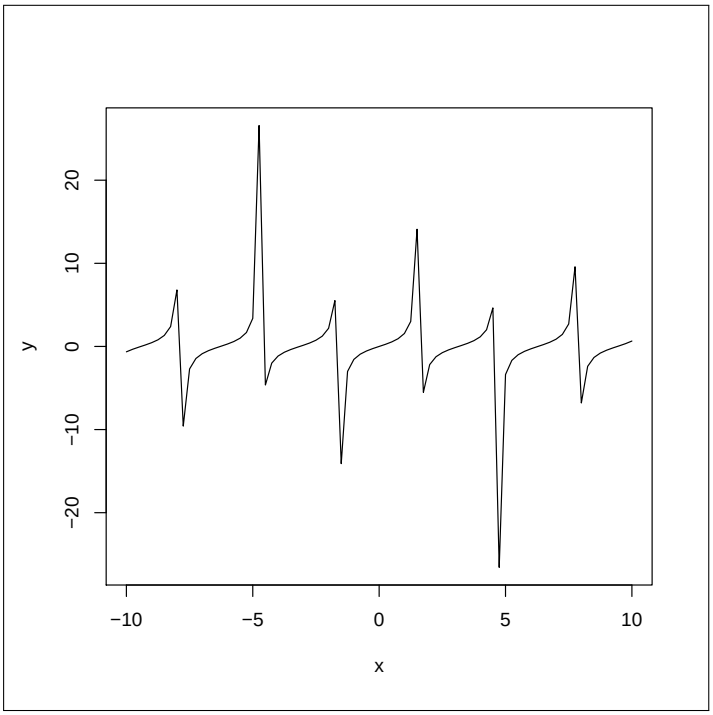


Figura 8.14: DIBUJO DE LA FUNCIÓN TANGENTE

También podemos obtener las coordenadas de los puntos a partir de cualquier función paramétrica, en lugar de usar una secuencia lineal para uno de los ejes. En el siguiente ejercicio se ha definido una función paramétrica que devuelve para cada valor t de entrada un vector (x,y). A continuación usamos dicha función para obtener la matriz de puntos que entregamos tanto a `plot()` como a `lines()`:

Ejercicio 8.15 Líneas a partir de función paramétrica

```
> heart <- function(t) {
+   x <- 16 * sin(t)^3
+   y <- 13 * cos(t) - 5 * cos(2*t) - 2 * cos(3*t) - cos(4*t)
+   c(x,y)
+ }
> mpoints <- matrix(sapply(seq(0, 2 * pi, 0.1), heart),
+                   ncol = 2, byrow = TRUE)
> plot(mpoints)
> lines(mpoints)
```

La función `sapply()` fue descrita en la sección 6.2.2.

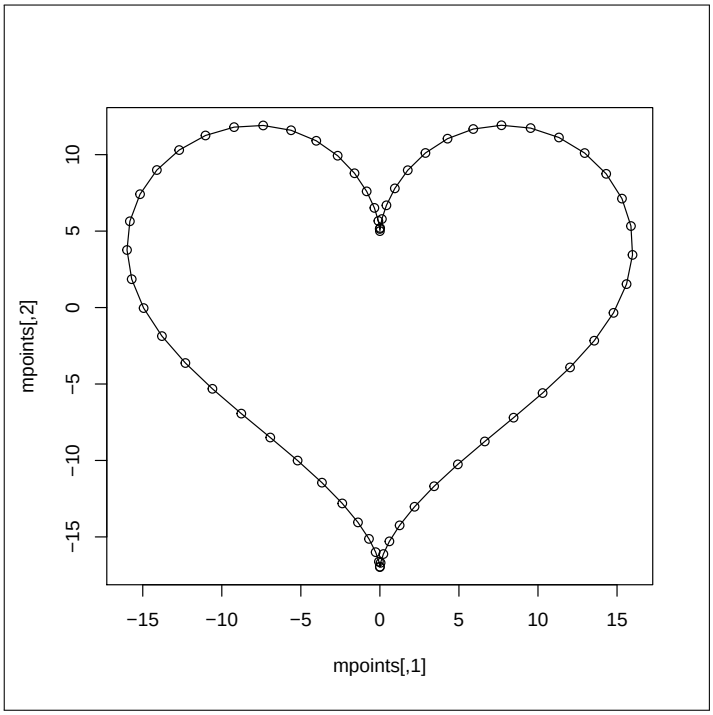


Figura 8.15: DIBUJO DE UNA FUNCIÓN PARAMÉTRICA

Dibujo de curvas de una función en un intervalo

Cuando lo que se quiere dibujar es el resultado de aplicar una cierta función a un rango de valores de entrada, el valor para x seria la entrada y la función nos daria la salida para y, en lugar de lines() podemos usar la función `curve()`. Esta nos ahorra el paso de generar nosotros mismos los puntos, ya que toma como parámetro la función y el intervalo en que se aplicará, y además genera la gráfica por sí misma, por lo que no necesitamos llamar previamente a `plot()`.

Sintaxis 8.4
```
curve(función|expresión[, from = inicio,
              to = fin, n = numPuntos, add = Añadir,
              xname = "nombre"])
```

Genera una gráfica dibujando una curva a partir de la `función` o `expresión` entregada como primer parámetro. Si no se especifica un intervalo de valores, mediante los argumentos `from` y `to`, el intervalo será $[0, 1]$. El número de puntos en que se evaluará la función determinará el grado de refinamiento de la curva obtenida. Dando el valor `TRUE` al parámetro `add` no se generará una nueva gráfica, sino que la curva se agregará a una gráfica existente. El parámetro `xname` da nombre a los valores generados para el eje X. Además también se aceptan otros argumentos generales para gráficas, como `xlab`, `ylab`, `type`, etc.

En el siguiente ejercicio se demuestra cómo utilizar funciones trascendentales, como `sin` y `cos`, para generar una gráfica en la que se muestran las curvas que estas generan en un cierto intervalo:

Ejercicio 8.16 Dibujo de funciones seno y coseno

```
> curve(sin, from = -4, to = 4, col = 'blue',
+       lty = 'dotted', lwd = 2, ylab = 'sin(x) vs cos(x)')
>
> curve(cos, from = -4, to = 4, col = 'cyan',
+       lty = 'dashed', lwd = 2, add = T)
>
> legend("topleft", c("sin(x)", "cos(x)"),
+        col = c('blue', 'cyan'), lty = c('dotted','dashed'),
+        lwd = 2, ncol = 2)
```

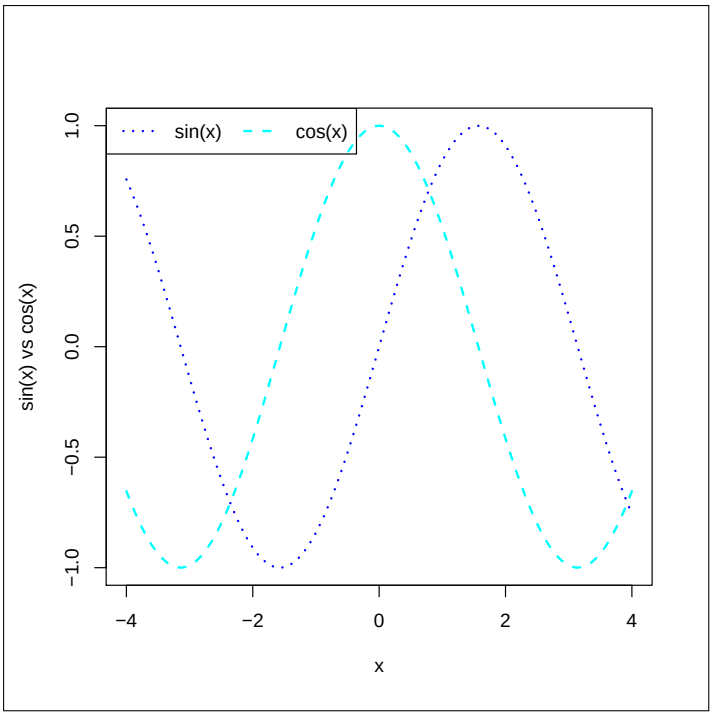


Figura 8.16: DIBUJO DE FUNCIONES SENO Y COSENO

La función `legend()` fue descrita en la sección 7.3.

También existe la posibilidad de entregar como primer argumento una expresión que será interpretada como un polinomio en función de X, tal y como se muestra en el ejemplo siguiente:

Ejercicio 8.17 Dibujo de un polinomio

```
> curve(x^2 - x, lty = 3, lwd = 2,
+       from = -10, to = 10)
>
> curve(x^3 - x^2 + 1, lty = 2, lwd = 2,
+       from = -10, to = 10, add = TRUE)
```

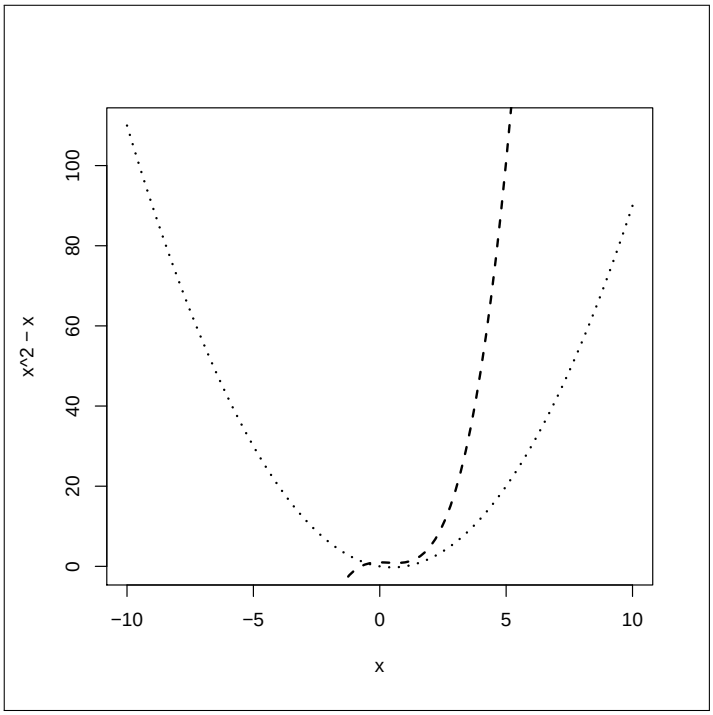


Figura 8.17: DIBUJO DE UN POLINOMIO

8.2.2 circlize

Los gráficos de nubes de puntos, líneas y barras son los más usuales, pero no los únicos que nos permiten representar información almacenada en una matriz o un `data.frame`. Un tipo alternativo es el gráfico tipo *circos*, muy usado en genética pero que puede tener otras aplicaciones, por ejemplo para mostrar interacciones de todo tipo. Un ejemplo seria la visualización del volumen de inmigracion-emigración entre zonas geográficas, o la aparición conjunta de las mismas etiquetas en noticias o documentos.

En adelante nos referiremos a este tipo de gráfica como GCI (*Gráfica circular de interacción*).

Como es habitual con R, en CRAN podemos encontrar múltiples paquetes que permiten elaborar este tipo de gráficas. Uno de ellos es el paquete `circlize`, cuya documentación podemos encontrar en `http://cran.r-project.org/web/packages/circlize/vignettes/circlize.pdf`. Lo que se ofrece a continuación es una introducción breve a parte de las posibilidades de este paquete.

Datos de entrada

Para generar una GCI con el paquete `circlize` tendremos que utilizar la función `chordDiagram()`, pero previamente sera necesario preparar los datos que hay que entregarle como entrada, en forma de matriz, siendo esta el únicio parámetro imprescindible.

La matriz de datos de entrada indicará, por cada cruce de fila/columna, la magnitud de la interacción a representar. Un ejemplo podría ser el mostrado en la tabla siguiente, en la que supuestamente se tienen datos de migración de aves entre países del sur y del norte de

Europa. Cada una de las filas y de las columnas estaría representado en el GCI por un sector. Las interacciones entre los sectores se dibujan como un enlace entre ellos, con un ancho proporcional a la magnitud.

	Spain	France	Italy
Finland	35.00	17.00	42.00
Poland	12.00	13.00	17.00
Denmark	67.00	28.00	32.00

Cuadro 8.1: Migración de aves

Sintaxis 8.5
```
chordDiagram(datos[, grid.col = colorSectores,
              col = colorEnlaces, transparency = nivelTrans
              directional = TRUE|FALSE,
              annotationTrack = tipoAnotaciones
              preAllocateTracks = pistasAdicionales])
```

Genera una gráfica circular de interacción a partir de los datos de entrada facilitados como primer argumento. Este deberá ser una matriz numérica, indicando la magnitud de la interacción entre cada fila y columna. El objetivo de los demás parámetros es el indicado a continuación:

- `grid.col`: Colores para los sectores del GCI. Por ser un color único o bien un vector de colores con tantos elementos como columnas y filas distintas haya en los datos.
- `col`: Vector de colores para los enlaces entre los sectores.
- `transparency`: Valor entre 0 y 1 estableciendo el nivel de transparencia para los enlaces. Es habitual usar colores semi-transparentes para que unos enlaces no oculten completamente a otros.
- `directional`: Indica si los enlaces deben indicar direccionalidad o no.
- `annotationTrack`: Configuración de las anotaciones a mostrar en la GCI. Habitualmente se muestra el nombre que correpsponde a cada sector.
- `preAllocateTracks`: En caso de que vaya a mostrarse información adicional en la GCI, este parámetro determina cuántas pistas adicionales de datos existirán y cuál será su configuración.

Los nombres de las columnas y filas de la matriz actuarán como identificadores de los sectores de la GCI. Es posible que un mismo nombre aparezca como columna y como fila, en cuyo caso estaria representado por un único sector.

El siguiente ejercicio muestra cómo componer la matriz con la información de la tabla de datos anterior, facilitándola a la función `chordDiagram()` como único parámetro. La GCI obtenida nos permite apreciar fácilmente las interacciones entre los países, tal y como se pretendía:

Ejercicio 8.18 GCI básica

```
> if (!is.installed("circlize")) install.packages("circlize")
> library(circlize)
```

```
> migracion <- data.frame(Spain = c(35, 12, 67), France = c(17,
+     13, 28), Italy = c(42, 17, 32))
> rownames(migracion) <- c("Finland", "Poland", "Denmark")
> chordDiagram(as.matrix(migracion))

> circos.clear()  # Liberación de datos asociados a la gráfica
```

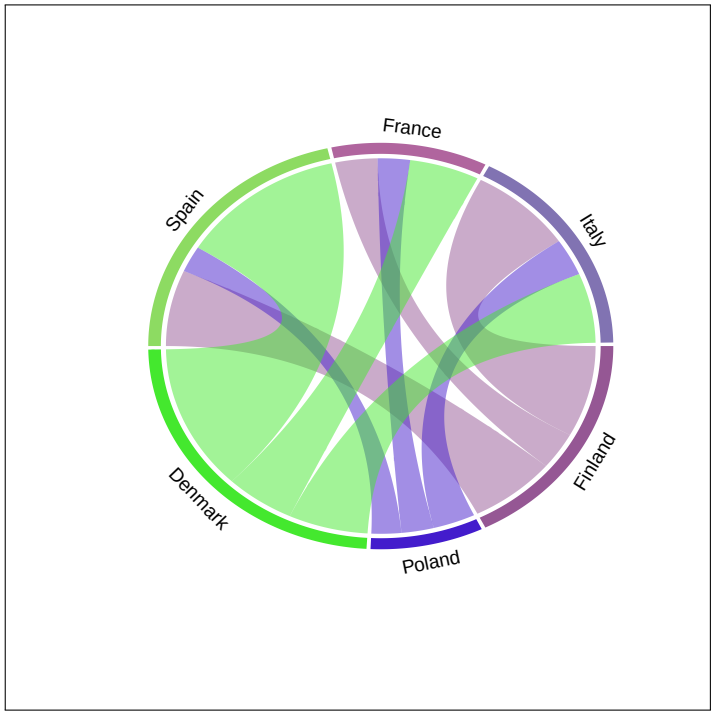


Figura 8.18: GCI BÁSICA

8.2.3 Pistas de información adicionales

La función `chordDiagram()` puede dibujar exclusivamente la GCI, los arcos que representan a cada fila/columna con los enlaces de interacción, o bien agregar una o más pistas adicionales con datos. Por defecto, cuando no se indica nada explícitamente, hay dos pistas con información: una muestra los nombres de cada sector y otra dibuja una rejilla, apenas perceptible, en el borde de cada sector. Este comportamiento viene determinado por el parámetro `annotationTrack` que, por defecto, contiene el vector de valores `'name'` y `'grid'`.

Usando el parámetro `preAllocateTracks` es posible reservar espacio para dibujar pistas adicionales con información, reduciendo el área dedicada a la GCI. Dicho parámetro puede tomar como parámetro un número entero, indicando el número de pistas, o bien una lista con parámetros de configuración para cada una de ellas, por ejemplo indicando su altura.

Tras la llamada a `chordDiagram()`, podemos dibujar y escribir en las pistas adicionales usando funciones como `circos.axis()` y `circos.trackPlotRegion()`, entre otras. Estas se caracterizan por tomar un parámetro llamado `track.index`, mediante el cual se indica la pista en la que se quiere trabajar.

El siguiente ejercicio muestra cómo utilizar las funciones citadas y algunas más del paquete `circlize`, consiguiendo una GCI mucho más informativa.

Ejercicio 8.19 GCI con pistas adicionales de datos

```
> if(!is.installed('circlize'))
+   install.packages('circlize')
> library(circlize)
> library(mldr)  # Paquete necesario para usar el dataset emotions
>
> # Preparar en tbl el conteo de muestras en que aparecen
> # conjuntamente cada pareja de etiquetas
> labels <- emotions$dataset[ , emotions$labels$index]
> nlabels <- ncol(labels)
> tbl <- sapply(1:nlabels, function(ind1)
+   sapply(1:nlabels, function(ind2)
+     if(ind2 < ind1) sum(labels[,ind1]*labels[,ind2]) else 0
+   ))
> colnames(tbl) <- colnames(labels)
> row.names(tbl) <- colnames(tbl)
>
> par(mar=c(0,0,0,0))
> # Separación entre cada arco y el siguiente
> circos.par(gap.degree = 3)
>
> # Diagrama principal
> chordDiagram(tbl, annotationTrack = "grid", transparency = 0.5,
+              preAllocateTracks = list(track.height = 0.1))
>
> # Un eje sobre cada arco
> for(si in get.all.sector.index()) {
+   circos.axis(h = "top", labels.cex = 0.3,
+               sector.index = si, track.index = 2)
+ }
>
> # Por encima de cada arco una línea relativa (con porcentajes)
> # y el nombre de cada etiqueta
> circos.trackPlotRegion(track.index = 1,
+                        panel.fun = function(x, y) {
+   xlim = get.cell.meta.data("xlim")
+   ylim = get.cell.meta.data("ylim")
+   sector.name = get.cell.meta.data("sector.index")
+
+   circos.lines(xlim, c(mean(ylim), mean(ylim)), lty = 3)
+   for(p in seq(0, 1, by = 0.25)) {
+     circos.text(p*(xlim[2] - xlim[1]) + xlim[1], mean(ylim),
+                 p, cex = 0.4, adj = c(0.5, -0.2),
+                 niceFacing = TRUE)
+   }
+   circos.text(mean(xlim), 1.1, sector.name,
```

```
+                cex = 0.5, niceFacing = TRUE)
+ }, bg.border = NA)

> circos.clear()  # Liberación de datos asociados a la gráfica
```

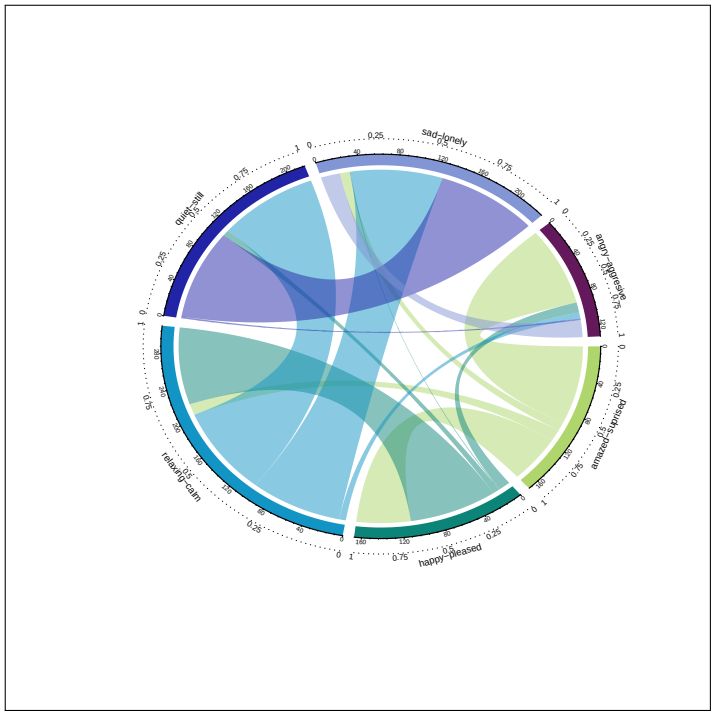


Figura 8.19: GCI CON PISTAS ADICIONALES DE DATOS

8.2.4 radarchart

Las gráficas de tipo *radar*, también conocidas como gráficas *spider*, permiten representar series de datos con múltiples variables en ejes que pueden ser homogéneos o no. En R hay múltiples paquetes que permiten crear este tipo de gráficas, entre ellos el paquete `fmsb` que es el que usaremos aquí.

El mencionado paquete nos ofrece la función `radarchart()` que, a partir de la información contenida en un `data.frame`, puede generar este tipo de gráficas con un mínimo de tres variables (un triángulo) y una serie de datos (una línea).

Sintaxis 8.6
```
radarchart(datos[, maxmin = TRUE|FALSE,
              axistype = etqEjes, seg = numSegmentos
              vlabels = titVariables, plty = tipoTrazo
              plwd = grosorLínea])
```

Genera una gráfica de tipo *radar* a partir de los datos de entrada facilitados como primer argumento. Este será un `data.frame` con las variables en columnas y las series en filas. La finalidad del resto de parámetros es la indicada a continuación:

- `maxmin`: Este parámetro determina si las dos primeras filas del `data.frame` contendrán o no los valores máximo y mínimo para cada una de las variables. Por

defecto toma el valor `TRUE`, por lo que se espera que dichas filas existan. Dándole el valor `FALSE` se consigue que el máximo y mínimo para cada variable se calcule a partir de los propios datos. El máximo y mínimo de cada variable determinarán la escala correspondiente a su eje.

- `axistype`: El valor asignado a este parámetro, un entero entre 0 y 5, determinará las etiquetas que se mostrarán a lo largo del eje principal y en el extremo de los ejes de cada una de las variables.
- `seg`: Establece el número de segmentos en que se dividirán los ejes. Por defecto se usan cuatro segmentos.
- `vlabels`: Vector con el nombre de las variables, a mostrar en el extremo de cada eje. Por defecto se usaran los nombres de las columnas del `data.frame`.
- `plty` y `plwd`: Mediante estos parámetros podemos configurar el tipo de trazo de las líneas y su grosor. Se aceptan parámetros adicionales para fijar el color y otros atributos.

Imaginemos que tenemos resultados de una experimentación correspondientes a cinco algoritmos distintos: C4.5 (un tipo de árbol de clasificación), CAR (*Classification Association Rules*), ANN (*Artificial Neural Network*), kNN (*k Nearest Neighborsh*) y SVM (*Support Vector Machine*). Dichos algoritmos serían en este caso las variables. Por cada algoritmo han sido obtenidas evaluaciones con tres medidas diferentes: *Precision*, *Recall* y *Accuracy*, que serían las series.

Aunque podríamos recuperar la anterior información de un archivo CSV o una hoja de cálculo, por simplicidad en el siguiente ejercicio los valores para cada caso se generan aleatoriamente, tomados de una distribución uniforme entre `0` y `1`, y el `data.frame` resultando se entrega como primer parámetro a `radarchart`. Dado que el `data.frame` no se ha incluido información sobre los valores mínimo y máximo para cada variable, que serán calculados de los valores contenidos en cada columna, es necesario asignar el valor `FALSE` al parámetro `maxmin`.

Ejercicio 8.20 Gráfica básica de tipo *radar*

```
> if(!is.installed('fmsb'))
+   install.packages('fmsb')
> library('fmsb')
>
> # Preparamos los datos de entrada
> set.seed(4242)
> dat <- data.frame(
+   C4.5 = runif(3, 0, 1),
+   SVM  = runif(3, 0, 1),
+   kNN  = runif(3, 0, 1),
+   ANN  = runif(3, 0, 1),
+   CAR  = runif(3, 0, 1)
+   )
>
> # Cada eje tendrá una escala propia
> radarchart(dat, maxmin = FALSE)
```

■

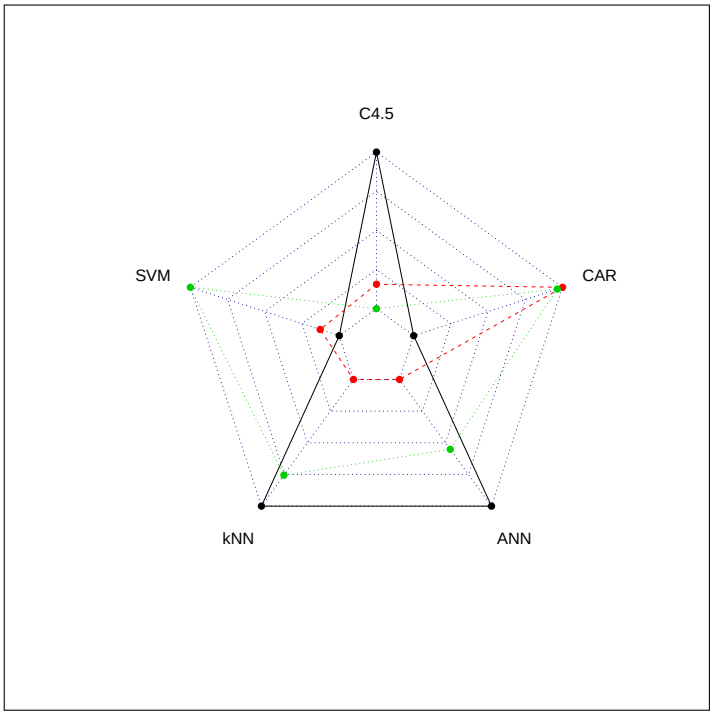


Figura 8.20: GRÁFICA BÁSICA DE TIPO *radar*

En la gráfica anterior cada uno de los cinco ejes tiene una escala distintas, cuyo mínimo y máximo dependerán de los valores aleatorios que se hayan generado. Lo habitual es que incluyamos como dos primeras filas el máximo y mínimo para cada eje, por ello el parámetro `maxmin` toma por defecto el valor `TRUE`.

Partiendo del ejemplo previo, en el siguiente ejercicio se agregan las dos filas y se dibuja de nuevo la gráfica, manteniendo exactamente los mismos valores que ya se tenían. Además, en este caso se muestras las etiquetas de valor en la escala central, configuración establecida con el parámetro `axistype`:

Ejercicio 8.21 Gráfica tipo *radar* con escalas homogéneas

```
> # Fijamos el máximo y mínimo para cada variable
> maxmin <- data.frame(
+   C4.5 = c(1, 0),
+   SVM  = c(1, 0),
+   kNN  = c(1, 0),
+   ANN  = c(1, 0),
+   CAR  = c(1, 0)
+ )
>
> dat <- rbind(maxmin, dat)
>
> # Escalas homogéneas para todas las variables
> radarchart(dat, axistype = 4)
```

■

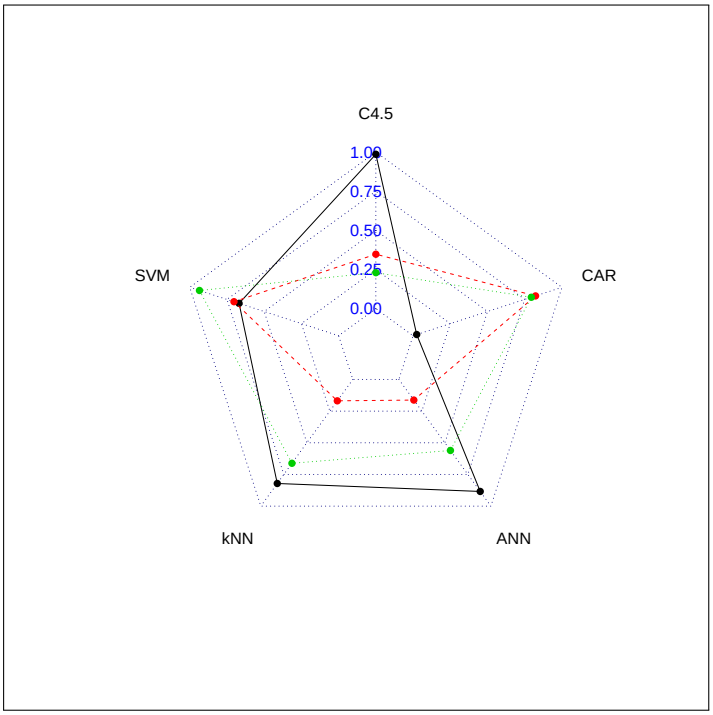


Figura 8.21: GRÁFICA TIPO *radar* CON ESCALAS HOMOGÉNEAS

Podemos hacer la gráfica algo más descriptiva agregando unas leyendas que indiquen a qué corresponde cada línea. También es posible ajustar la configuración de las líneas, establciendo su grosor, tipo de trazo y color. Es lo que se hace en el último ejercicio propuesto para la función `radarchart`.

Ejercicio 8.22 Gráfica tras agregar leyendas y configurar tipos de líneas

```
> radarchart(dat, axistype = 2,
+            plty = 1:3, plwd = 2,
+            pcol = c('black', 'green', 'red'),
+            vlabels = names(dat)
+            )
> legend("bottomleft", c("precision", "accuracy", "recall"),
+        col = c('black', 'green', 'red'),
+        lty = 1:3, lwd = 2, ncol = 3)
```

■

8.2.5 Gráficas 3D

Al igual que en otros apartados, en el de generación de gráficas 3D las posibilidades con R son muy extensas dado el número de paquetes disponibles para dicho fin. En esta sección se introducen únicamente dos de ellos, mediante los cuales es relativamente sencilo preparar graficas de puntos y líneas utilizando tres ejes en lugar de los dos habituales que hemos venido utilizando en los ejercicios previos.

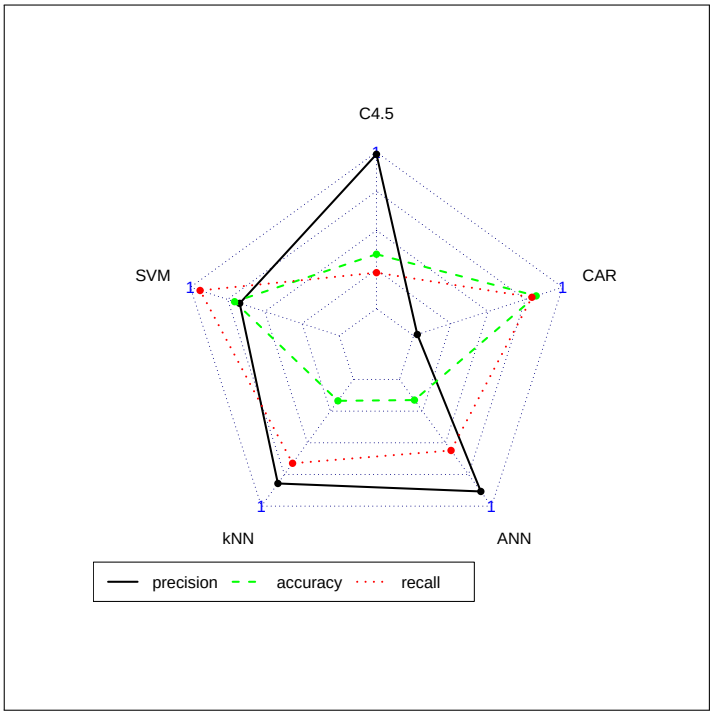


Figura 8.22: GRÁFICA TRAS AGREGAR LEYENDAS Y CONFIGURAR TIPOS DE LÍNEAS

scatterplot3d

El paquete `scatterplot3d` nos ofrece la función `scatterplot3d()` que, a partir de las coordenadas X, Y y Z de una lista de puntos, produce una gráfica con tres ejes y cada punto en la posición que le corresponde.

Sintaxis 8.7
```
scatterplot3d(x[, y, z, color, pch = símbolo,
              type = tipoGráfica, angle = ánguloGiro,
              highlight.3d = TRUE|FALSE, box = TRUE|FALSE
              grid = TRUE|FALSE, col.axis = colorEjes
              col.grid = colorRejilla])
```

Genera una gráfica a partir de coordenadas 3D de un conjunto de puntos, proyectándolos en el plano 2D según la configuración indicada. El único parámetro imprescindible es `x`, que puede ser un `data.frame` con las coordenadas de cada punto o bien un vector con los datos del eje x. En este último caso serán precisos también los parámetros `y` y `z`, dos vectores con los datos para los otros dos ejes. Además también pueden entregarse, entre otros, los parámetros siguientes:

- `color`: Vector especificando el color de los puntos. Este parámetro no está es necesario si se se usa `highlight.3d = TRUE`.
- `pch`: Indica el tipo de símbolo que se usará para dibujar los puntos.
- `type`: Tipo de gráfica a dibujar. Los tipos posibles son `'p'`, se dibuja un punto por cada trío de valores, `'l'`, se dibujan líneas conectando los puntos, y `'h'`, se dibuja el punto y una línea conectándolo con la base del plano x/y.
- `angle`: Ángulo entre los ejes X e Y. Permite cambiar la perspectiva de la gráfica.
- `highlight.3d`: Si se le da el valor `TRUE` se usarán distintos colores para dibujar los puntos según su posición en el eje Y.

- `box`: Por defecto es `TRUE`, dibujando una caja alrededor de la gráfica.
- `grid`: Por defecto es `TRUE`, dibujando una rejilla en la base de la gráfica.
- `col.axis`: y
- `col.grid`: Establecen los colores con los que se dibujarán los ejes y la rejilla, respectivamente.

Además de los anteriores, `scatterplot3d()` acepta muchos otros parámetros comunes a todas las funciones gráficas, como `xlab`, `ylab` y `zlab` para establecer los títulos de los ejes, `main` y `sub` para configurar el título principal y secundario de la gráfica, `xlim`, `ylim` y `zlim` para fijar los límites de cada eje, etc.

Si el primer parámetro entregado a la función `scatterplot3d()` es un `data.frame`, conteniendo en cada filas los valores (X, Y, Z) para cada punto, el resto de los parámetros es opcional.

En el siguiente ejercicio se muestra cómo preparar un `data.frame` para representar tres de las cuatro variables del dataset iris en una gráfica 3D. Esa estructura de datos es entregada como primer parámetro a `scatterplot3d()`, usándose además el parámetro `type`, para obtener una gráfica en que los puntos están conectados mediante líneas con la base del eje X,Y; el parámetro `color`, para distinguir cada punto según la especie de flor a la que pertenece; el parámetro `pch`, para seleccionar el símbolo con que se representará cada punto, y el parámetro `angle`, mediante el que podemos cambiar la perspectiva desde la que se ve la gráfica.

Ejercicio 8.23 Gráfica tridimensional con `scatterplot3d()` y datos de iris

```
> if(!is.installed('scatterplot3d'))
+   install.packages('scatterplot3d')
> library('scatterplot3d')
>
> # Representación de iris, ancho y alto de pétalo,
> # ancho de sépalo y especie,
> # en una grafica de  puntos 3D
> datos <- data.frame(AnchoSepalo = iris$Sepal.Width,
+                     LargoPetalo = iris$Petal.Length,
+                     AnchoPetalo = iris$Petal.Width)
>
> scatterplot3d(datos, type = "h",
+               color = as.numeric(iris$Species),
+               pch = 20, angle = 15)
```

La información para las coordenadas de los puntos puede proceder de un conjunto de datos real, como acaba de demostrarse, pero también es posible generarlas artificalmente, por ejemplo a partir de una secuencia de valores y algunos cálculos trigonométricos. Estos nos permite dibujar funciones 3D y representar cualquier concepto matemático que precise un espacio tridimensional.

El siguiente ejercicio usa `scatterplot3d()` para dibujar un lazo 3D, generado utilizando las funciones matemáticas `sin()` y `cos()` de R. En este caso se da el valor `TRUE` al

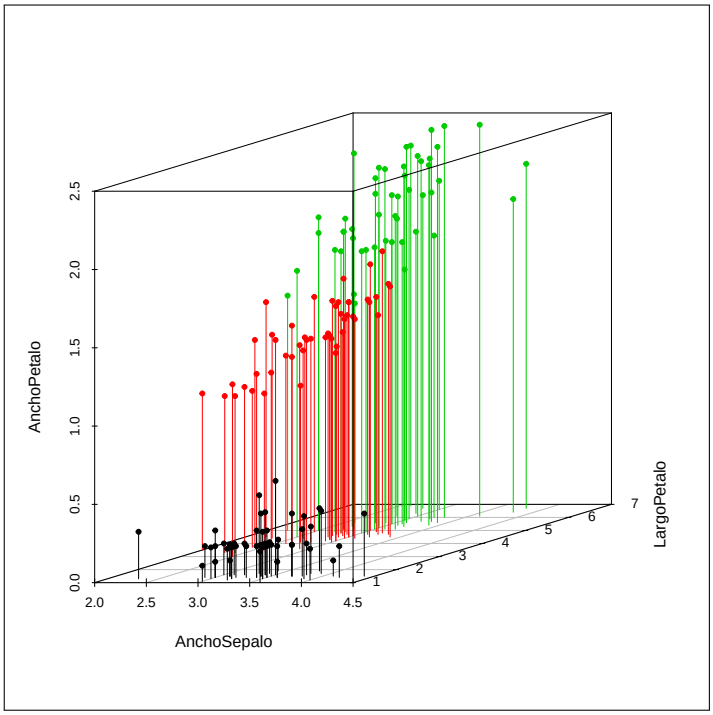


Figura 8.23: GRÁFICA TRIDIMENSIONAL CON `scatterplot3d()` Y DATOS DE IRIS

parámetro `highlight.3d`, de forma que el color de los puntos está en concordancia con su profundidad, lo cual permite apreciar mejor la perspectiva tridimensional:

Ejercicio 8.24 Gráfica tridimensional con `scatterplot3d()` y funciones trigonométricas

```
> x <- seq(-10, 10, 0.01)
> y <- sin(x) # Cálculo de las posiciones
> z <- cos(x)*sin(x) # en Y y Z
>
> scatterplot3d(x, y, z,
+               highlight.3d = TRUE,
+               col.axis = "blue",
+               col.grid = "lightblue",
+               pch = 20, angle = -30)
```

■

lattice

Otro de los paquetes que cuentan con funciones para la elaboración de gráficas en 3D es `lattice` que, entre otras funciones, nos permite representar superficies mediante la función `persp()`. Los datos de entrada para esta función son tres vectores, como en el caso de `scatterplot3d()`, con valores para X, Y y Z. Es posible, no obstante, facilitar únicamente una matriz con la elevación (el valor para Z), dejando que los valores de X e Y se generen automáticamente según las dimensiones de dicha matriz.

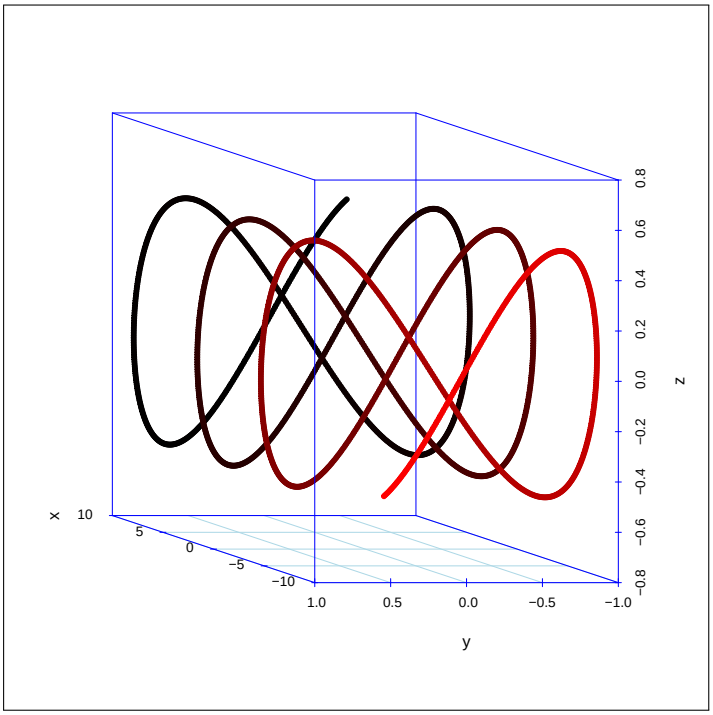


Figura 8.24: GRÁFICA TRIDIMENSIONAL CON SCATTERPLOT3D() Y FUNCIONES TRIGONOMÉTRICAS

Sintaxis 8.8 `persp(x, y, z[, xlim = límites, ylim = límites, zlim = límites, theta = ánguloAz, phi = ánguloLat, box = TRUE|FALSE axes = TRUE|FALSE, col = color])`

Produce una gráfica tridimensional representan en perspectiva una superficie respecto al plano formado por los ejes X e Y. Los límites para cada eje se obtienen mediante la función `range()` a partir de los vectores de entrada en caso de que no se entreguen explícitamente con los parámetros `xlim`, `ylim` y `zlim`. Mediante los parámetros `theta` y `phi` es posible cambiar la rotación de la gráfica.

En el siguiente ejercicio se muestra cómo utilizar la función persp() para representar una superficie definida como una matriz de 25 x 25 medidas. El valor base es 574 y a este se añade después una variación que produce una inclinación sucesiva, dando como resultado la gráfica que puede verse más adelante:

Ejercicio 8.25 Gráfica tridimensional con `lattice`

```
> if(!is.installed('lattice'))
+   install.packages('lattice')
> library('lattice')
>
> z <- matrix(rnorm(625) + 574, nrow = 25)
> z <- z + seq(50, 1, length = 25)
> persp(z, phi = 30, theta = 30,
+       shade = 0.5, col = heat.colors(6),
```

```
+        zlim = c(550,650), ticktype = "detailed",
+        xlab = "X", ylab = "Y", zlab='Z',
+        main = "Elevación del terreno")
```

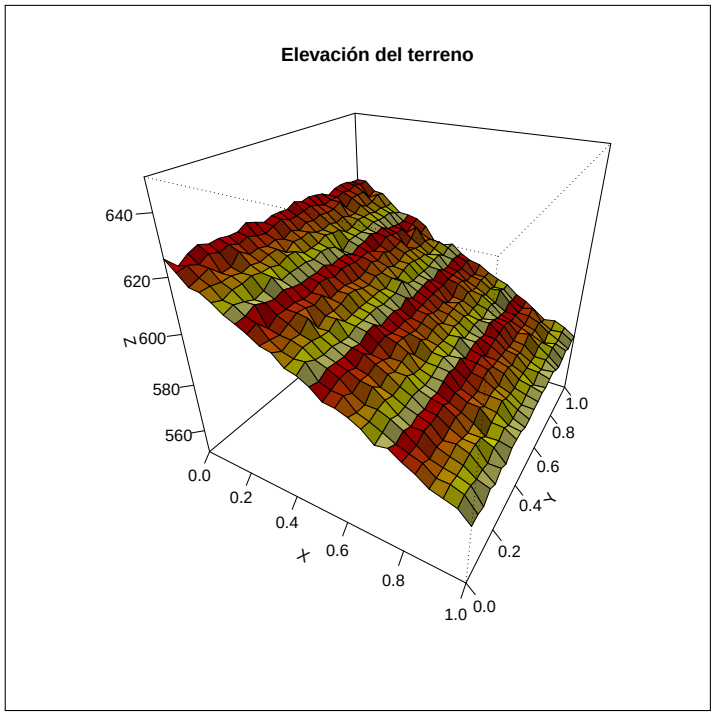


Figura 8.25: GRÁFICA TRIDIMENSIONAL CON LATTICE

Además de la función `persp()`, el paquete lattice ofrece otras para representación de nubes de puntos en 3D, función `cloud()`, o mallas de alambre, función `wireframe()`. En la documentación del paquete podemos encontrar más información sobre todas estas funciones. En el siguiente ejemplo se muestra cómo utilizar la función `cloud()` para repersentar en 3D el conjunto de datos `iris`, tal y como hacíamos antes con `scatterplot3d()`.

Ejercicio 8.26 Nube de puntos tridimensional con `lattice`

```
> # Representación de nube de puntos 3D con lattice
> cloud(iris$Petal.Width ~
+         iris$Petal.Length *
+         iris$Sepal.Width,
+       col = rainbow(3, alpha = 0.5),
+       pch = 19, cex = 1.5,
+       groups = iris$Species,
+       auto.key = list(space="top", columns = 3, points = FALSE,
+                       title="Specie", cex.title = 1.5,
+                       col = rainbow(3)))
```

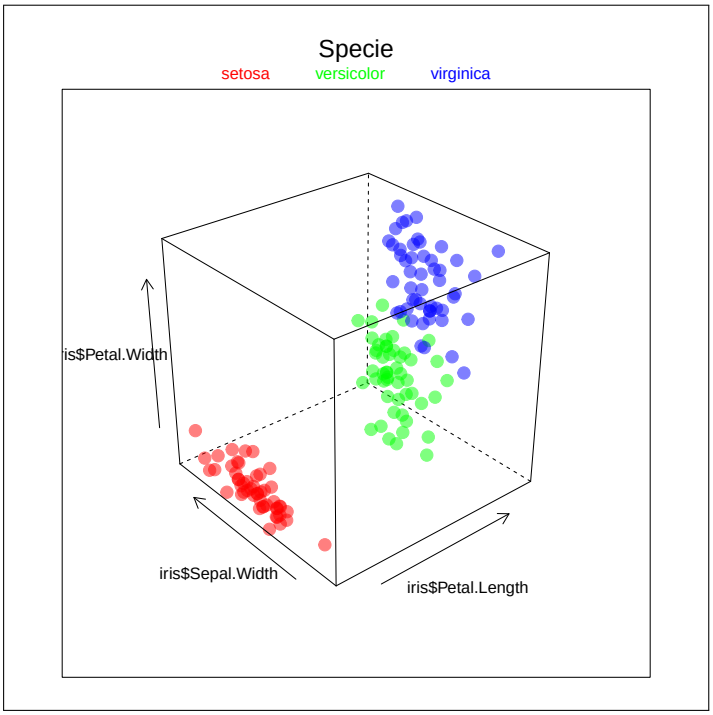


Figura 8.26: NUBE DE PUNTOS TRIDIMENSIONAL CON LATTICE

8.3 Gráficos de tortuga

Para concluir este capítulo, y más a modo de curiosidad que por su utilidad en la exploración o visualización de datos, apuntar la existencia de paquetes R para la generación de muchos otros tipos de gráficos. Sirva como ejemplo el paquete `TurtleGraphics`, mediante el que podemos usar los conocidos como *gráficos de tortuga* introducidos hace años por el lenguaje Logo `http://en.wikipedia.org/wiki/Logo_(programming_language)`.

En los gráficos de tortuga se cuenta con una especie de avatar, la **tortuga**, que cuenta con atributos como posición en el plano, orientación, color del lápiz y situación de este (bajado o subido), etc.

Con órdenes simples, del tipo *avanza n puntos*, *gira a la derecha g grados*, etc., es posible ir dibujando sobre el plano. Uno de los puntos fuertes de Logo era la recursividad, por lo que resultaba ideal para dibujar cualquier tipo de gráfico con una componente recursiva.

Un ejemplo muy conocido de esta técnica es el mostrado a continuación, con el que se genera uno de los conocidos fractales de Sierpiński:

Ejercicio 8.27 Gráficos de tortuga: Triángulo de Sierpinski

```
> if(!is.installed('TurtleGraphics'))
+   install.packages('TurtleGraphics')
> library('TurtleGraphics')
>
> drawTriangle <- function(points) {
+   turtle_setpos(points[1,1],points[1,2])
```

```
+   turtle_goto(points[2,1],points[2,2])
+   turtle_goto(points[3,1],points[3,2])
+   turtle_goto(points[1,1],points[1,2])
+ }
>
> getMid<- function(p1,p2) c((p1[1]+p2[1])/2, c(p1[2]+p2[2])/2)
>
> sierpinski <- function(points, degree){
+   drawTriangle(points)
+   if (degree  > 0) {
+     p1 <- matrix(c(points[1,], getMid(points[1,], points[2,]),
+                    getMid(points[1,], points[3,])),
+                  nrow=3, byrow=TRUE)
+
+     sierpinski(p1, degree-1)
+
+     p2 <- matrix(c(points[2,], getMid(points[1,], points[2,]),
+                    getMid(points[2,], points[3,])),
+                  nrow=3, byrow=TRUE)
+
+     sierpinski(p2, degree-1)
+
+     p3 <- matrix(c(points[3,], getMid(points[3,], points[2,]),
+                    getMid(points[1,], points[3,])),
+                  nrow=3, byrow=TRUE)
+
+     sierpinski(p3, degree-1)
+   }
+   invisible(NULL)
+ }
>
> turtle_init(520, 500, "clip")

> p <- matrix(c(10, 10, 510, 10, 250, 448), nrow=3, byrow=TRUE)
> turtle_col("red")
> turtle_do(sierpinski(p, 6))

> turtle_setpos(250, 448)
```

Podemos explorar por nosotros mismos las posibilidades de los gráficos de tortuga a partir de únicamente un par de funciones del paquete `TurtleGraphics`: `turtle_forward()` y `turtle_right()`. La primera hace avanzar a la tortuga el número de puntos indicado, dejando el correspondiente rastro que será una línea recta. La segunda cambia la orientación de la tortuga, girándola hacia a la derecha el número de grados que se especifique. Repitiendo estas dos operaciones, mediante un simple bucle, es posible dibujar cualquier polígono, cerrado o no.

El siguiente ejercicio usa dos bucles anidados a fin de dibujar un conjunto de hexágonos, cada uno de ellos girado 8 grados respecto al anterior, con 20 unidades de lado. El resultado,

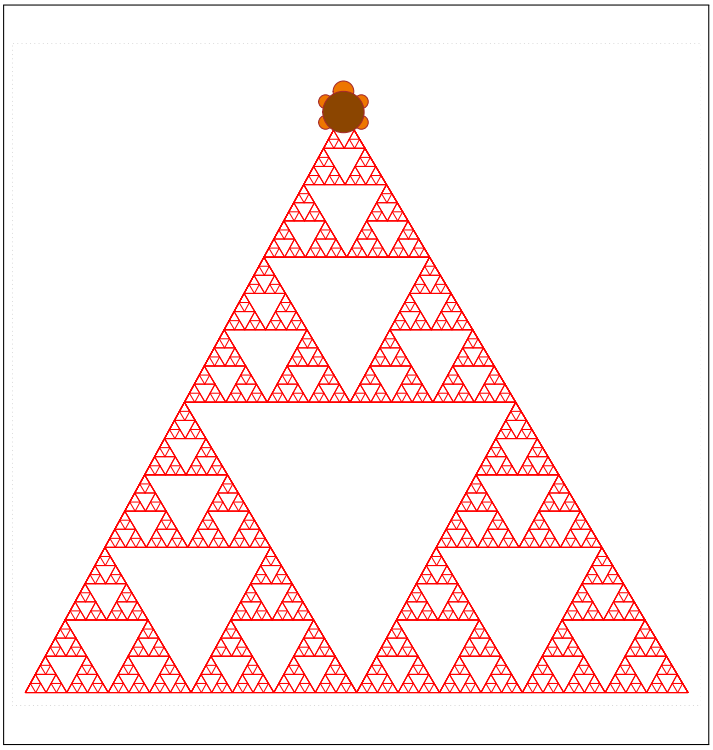


Figura 8.27: GRÁFICOS DE TORTUGA: TRIÁNGULO DE SIERPINSKI

a pesar de tratarse de un ejemplo muy sencillo, es realmente visto como se aprecia en la última figura de este capítulo.

Ejercicio 8.28 Gráficos de tortuga: Espiral de hexágonos

```
> turtle_init()

> turtle_do({
+ for(j in 1:45) {
+   for(i in 1:6) {
+     turtle_forward(20)
+     turtle_right(360/6)
+   }
+  turtle_right(360/45)
+ }
+ })
```

■

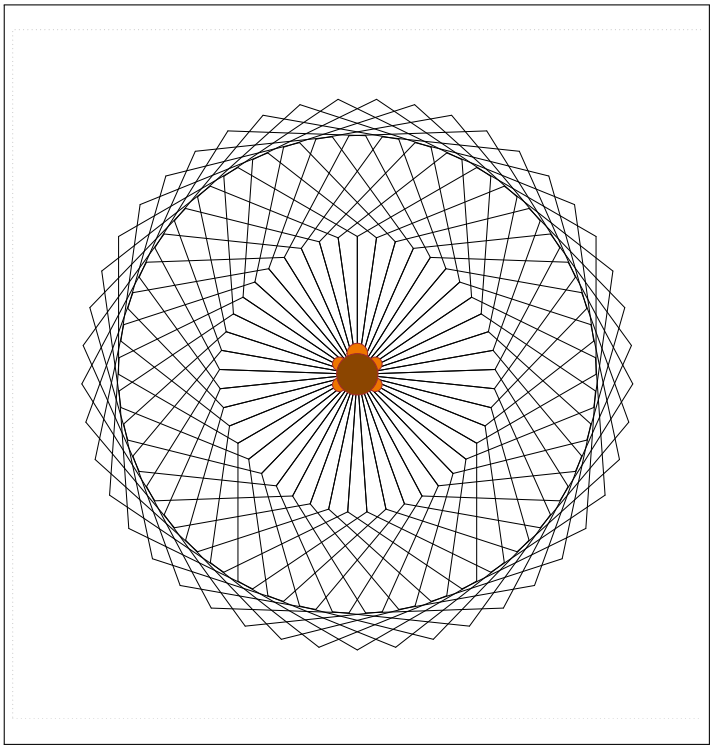


Figura 8.28: GRÁFICOS DE TORTUGA: ESPIRAL DE HEXÁGONOS

9. Introducción al análisis reproducible

En palabras de Donald E. Knuth, autor de los volúmenes *The Art of Computer Programming* y creador de TeX, al programar un algoritmo no deberíamos hacerlo pensando únicamente en instruir a la máquina sobre lo que debe hacer, sino que deberiamos preocuparnos más por explicar a otras personas qué es lo que pretendemos que el ordenador haga:

Definition 9.0.1 *Literate programming (Donald Knuth, 1984)*
Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.

La meta de la *literate programming* no se consigue incluyendo muchos comentarios en el código de un programa, sino trabajando como lo haria un escritor que explica a los usuarios lo que se está haciendo, introduciendo fragmentos de código que hacen lo que se está explicando. El objetivo final es conseguir que nuestro algoritmo sea **reproducible**, de forma que cualquiera pueda volver a implementarlo y obtener exactamente los mismos resultados que nosotros afirmamos obtener.

La reproducibilidad es un factor cada vez más importante en muchos ámbitos y, por supuesto, especialmente en el campo de la investigación. Conseguirlo, sin embargo, no es una tarea fácil y, en ocasiones, ni siquiera los autores originales de un trabajo son capaces de reproducirlo llegando a las mismas conclusiones.

9.1 Componentes de un proyecto de investigación

Mientras se trabaja en un proyecto de investigación o análisis de datos se utilizan multitud de componentes: datos de entrada, procedimientos para su tratamiento, software que permiten poner en práctica dichos procedimientos, etc. Veamos brevemente algunos de los componentes más comunes, enumerándolos en el orden en que se irían utilizando durante el tiempo de vida del proyecto.

9.1.1 Datos a analizar

Los datos que se quieren analizar, o que forman parte de la investigación a llevar a cabo, pueden proceder de muy distintos orígenes: archivos CSV (*Comma Separated Values*), hojas

de calculo (por ejemplo archivos Excel), bases de datos relacionales, conjuntos de datos en formatos como ARFF, etc. Dependiendo de los casos necesitaremos utilizar un programa u otro para acceder a esos datos, un softwate que ha de estar disponible en cualquier momento futuro en que se precise reproducir el procedimiento de análisis.

Inicialmente los datos deberán ser integrados, consiguiendo una representación común que podamos usar en los pasos posteriores del proyecto. Posiblemente también sea preciso aplicar tareas de transformación y limpieza de datos, por ejemplo unificando unidades de medida, convirtiendo todas las fechas a un formato común, eliminando o tratando valores ausentes y anomalías, etc. Para todas estas tareas será preciso recurrir a algoritmos de procesamiento de datos que pueden existir o han de ser definidos a medida de las necesidades del proyecto.

9.1.2 Algoritmos de tratamiento y aprendizaje

Una vez que los datos, obtenidos de orígenes heterogéneos, han sido integrados y usan una representación normalizada llega el momento de aplicar sobre ellos distintos algoritmos, por ejemplo para descubrir la estructura subyacente a los datos, mediante técnicas de agrupamiento (*clustering*), o bien para obtener modelos de regresión o de clasificación de patrones, que permitan predecir valores o clases de muestras en el futuro.

La implementación de los algoritmos se lleva a cabo en una plétora de lenguajes, desde los de propósito general y fácilmente accesibles, como es el caso de Java o C/C++, hasta los más especializados y cuyo uso conlleva un coste considerable, como ocurre ccon MatLab y productos similares.

Tanto el código de la implementación de los algoritmos como las herramientas utilizadas para compilar o ejecutar deberían estar disponibles siempre, para cualquiera que quisiera reproducir el procedimiento aplicado a los datos. El procedimiento en sí mismo, lógicamente, tendria que estar cuidadosamente documentado.

9.1.3 Análisis de resultados

Los algoritmos desarrollados en la fase anterior generarán unos resultados que es preciso analizar, a fin de determinar la bondad y el rendimiento de los modelos generados. Para ello se procede a agregar información, efectuar comparaciones y aplicar tests estadísticos.

En esta fase se recurre a herramientas específicas para el análisis de datos, tales como SPSS, Stata o R, entre otras. Al igual que en el caso previo, el software utilizado durante la experimentación original, asi como el procedimiento seguido para tratar los resultados y la naturaleza de los estadísticos empleados, ha de estar disponible también en cualquier momento posterior a fin de poder reproducirlo y verificar que las conclusiones del análisis son correctas.

9.1.4 Representación

A partir del análisis de los resultados, o como parte del propio proceso análisis, es habitual que se elaboren y compongan distintos tipos de gráficas, tablas y representaciones de los datos o incluso animaciones.

Nuevamente, en esta fase se usan las herramientas más adecuadas para la tarea a llevar a cabo, que en este caso suelen ser Excel, Matlab, GnuPlot y R. En cualquier caso la representación o gráfica debe ser reproducible a partir de los resultados obtenidos en el paso previo, lo cual implica conservar el código usado para ello, por ejemplo en el caso de R, o bien documentar cuidadosamente el procedimiento que se ha seguido para elaborarlos, en el caso de programas como Excel que se utilizan de forma interactiva.

9.1.5 Redacción

La fase final, en la que se plasma todo el trabajo desarrollado en los pasos previos, es la redacción del PFC/TFM, tesis o artículo, segun los casos. Es una tarea en la que ha de recopilarse la descripción del tratamiento aplicado a los datos, detalles sobre los algoritmos empleados, explicación de como se ha conducido la experimentación y el análisis e integración de las tablas, gráficas y cualquier otro elemento de representación. El objetivo ha de ser que cualquiera que lea este trabajo pueda seguir las indicaciones dadas para reproducir todo el proceso y, obviamente, obtener exactamente los mismos resultados.

9.1.6 Problemática

A raíz de los pasos que acaban de describirse es fácil detectar una cierta problemática, formada por múltiples obstáculos para conseguir que el trabajo sea finalmente reproducible. Podriamos resumir esta problemática en los siguientes puntos clave:

- Se usan demasiadas herramientas distintas durante el proceso, incluyendo múltiples lenguajes para la implementación de los algoritmos, el análisis, la representación y finalmente la redacción.
- En general el proceso tiende a ser complejo y muy específico. Cada investigador define un procedimiento a su medida.
- Se echa en falta una mayor integración y automatización de los procesados, lo cual nos lleva a plantearnos múltiples cuestiones a la hora de la redacción final:
  - ¿Se ha usado la última versión de los datos tras aplicar el un nuevo preprocesamiento?
  - ¿Corresponde la tabla o gráfica a los resultados más recientes?
  - ¿Se han incorporado las nuevas representaciones en el documento final o son una versión anterior?
- Globalmente hay muchas dificultades para reproducir los resultados, ya que la probabilidad de introducir errores de transcripción en cualquiera de los pasos va acumulándose.

Cabe preguntarse, por tanto, si es posible obtener el mismo documento final actualizado tras introducir un cambio en los datos de origen ¿Cómo estar seguro de que se han seguido todos los pasos sin ningún fallo? La respuesta, o al menos una de ellas, la encontramos en la automatización de la mayor parte del proceso.

9.1.7 Posible solución

Una posible solución a la anterior problemática, fundamentada en lo que ha venido a denominar *literate programming*, consistiría en utilizar el propio archivo desde el que se generará el documento final como contenedor universal, alojando en él la información y la lógica necesarios para realizar todo el trabajo.

Se precisaría un software capaz de automatizar todo el proceso, garantizando así la reproducibilidad del documento. Especialmente importante es la elaboración mediante código, no de forma interactiva por parte de una persona, de las tablas de datos, resultados y gráficas.

Este sistema debería contemplar la posibilidad de generar múltiples salidas, según la fase del proceso en que nos encontrásemos. De esta forma podríamos, partiendo del mismo contenedor, ejecutar los algoritmos de tratamiento de datos, aplicar el análisis apropiado a los resultados obtenidos y producir el documento final en los formatos en que interese.

El mayor beneficio de un sistema de este tipo sería la ausencia de errores de transcripción, garantizando la reproducibilidad por parte de cualquier otra persona.

Bibliografía

Libros

[SLP14] Victoria Stodden, Friedrich Leisch y Roger D. Peng, editores. *Implementing Reproducible Research*. The R Series. Chapman y Hall/CRC, abr. de 2014. ISBN: 1466561599. URL: `https://osf.io/s9tya/wiki/home/`.

[Wil05] Leland Wilkinson. *The Grammar of Graphics (Statistics and Computing)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005. ISBN: 0387245448 (véase página 125).

Artículos

[Lei13] Philip Leifeld. "texreg: Conversion of Statistical Model Output in R to LaTeX and HTML Tables". En: *Journal of Statistical Software* 55.8 (2013), páginas 1-24. URL: `http://www.jstatsoft.org/v55/i08/`.

[Sch+12] Eric Schulte y col. "A Multi-Language Computing Environment for Literate Programming and Reproducible Research". En: *Journal of Statistical Software* 46.3 (25 de ene. de 2012), páginas 1-24. ISSN: 1548-7660. URL: `http://www.jstatsoft.org/v46/i03`.

Páginas web

[Pau13a] Linus Pauling. *Handling big data in R*. Sep. de 2013. URL: `http://davetang.org/muse/2013/09/03/handling-big-data-in-r/` (véase página 48).

[Pau13b] Josh Paulson. *Using Command History*. Oct. de 2013. URL: `https://support.rstudio.com/hc/en-us/articles/200526217-Command-History` (véase página 20).

Índice alfabético