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.

Sintáxis 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)
```

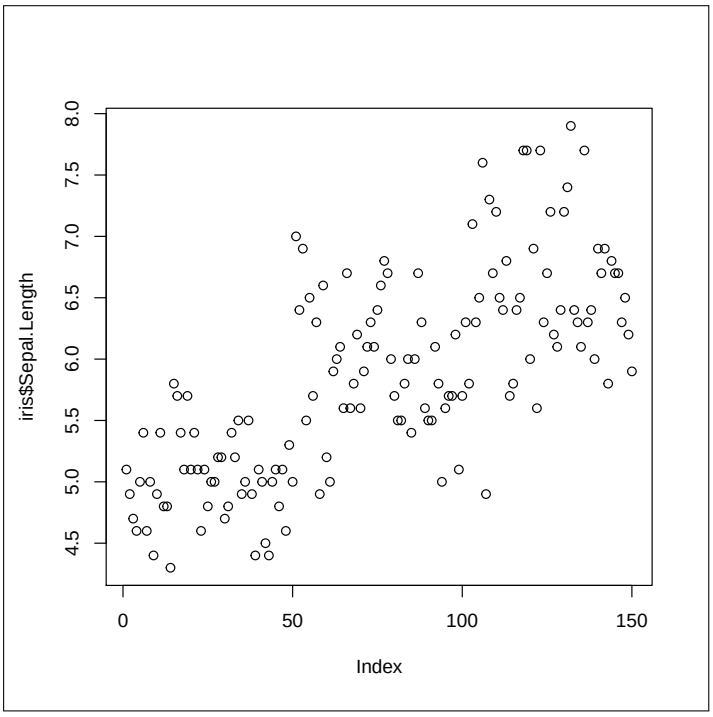

Figura 7.1: Nube de puntos mostrando una variable

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)
```

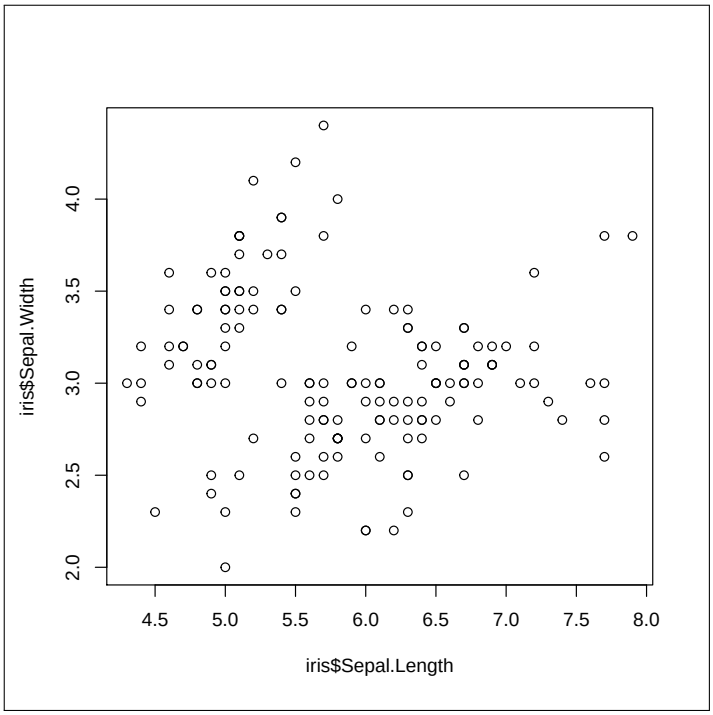

Figura 7.2: Nube de puntos mostrando dos variables

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)
```

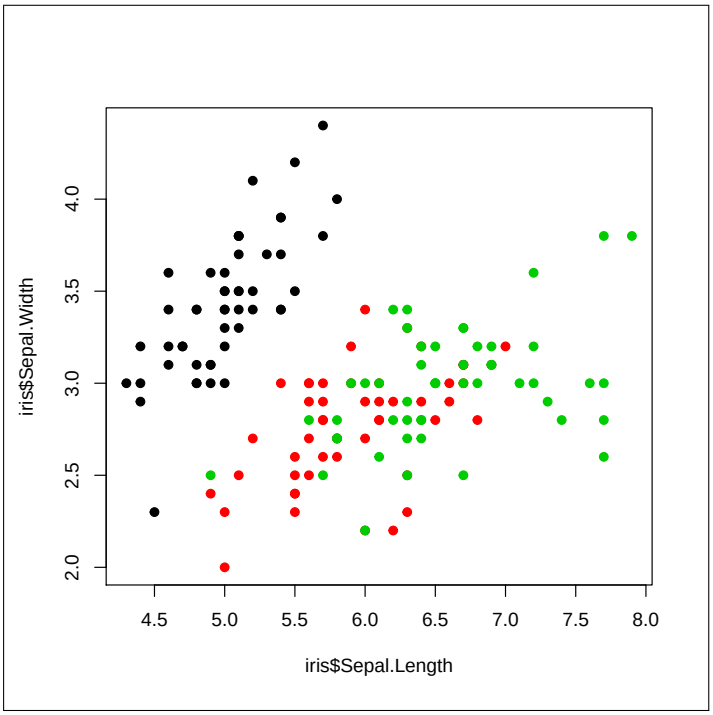

Figura 7.3: Nube de puntos con dos variables y la clase en color

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.

Sintáxis 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.

Sintáxis 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")
```

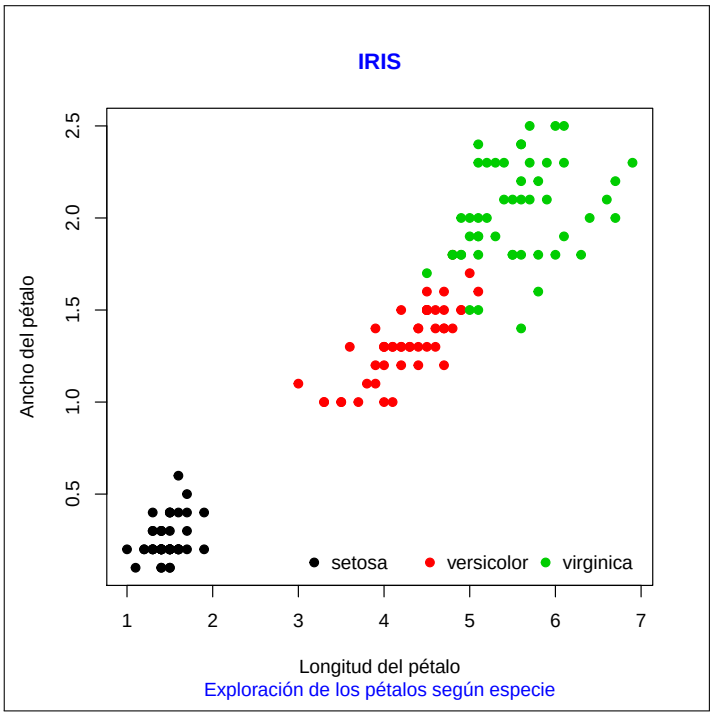

Figura 7.4: Nube de puntos con dos variables y títulos

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)
```

■

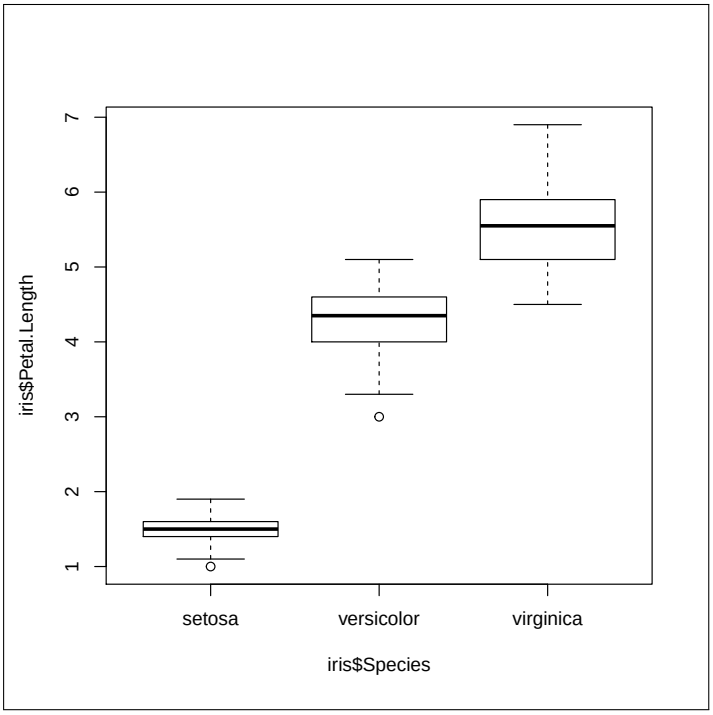

Figura 7.5: Gráfica de cajas de la longitud de pétalo por especie

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()`.

Sintáxis 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

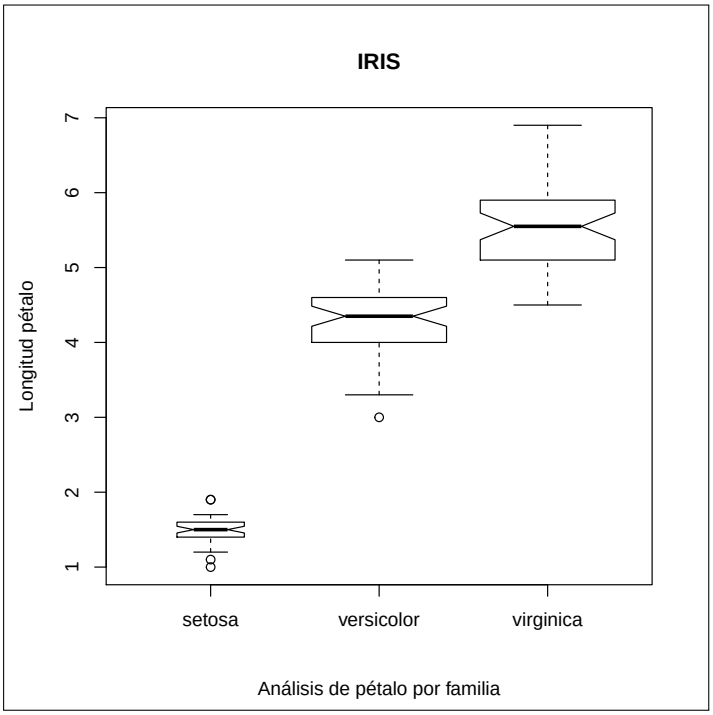

Figura 7.6: Gráfica de cajas con títulos en ejes y título principal

dar el valor 'o' 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.

Sintáxis 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      Wed
20.47578 43.07168 45.63229 40.85346 25.75291 23.29060 31.47493

> meanPricesUS

     Fri      Mon      Sat      Sun      Thu      Tue      Wed
48.21471 36.83621 39.28396 42.27805 39.71355 31.95483 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("$","€"),
+        col = c("blue","red"), lty = c(1,1))
```

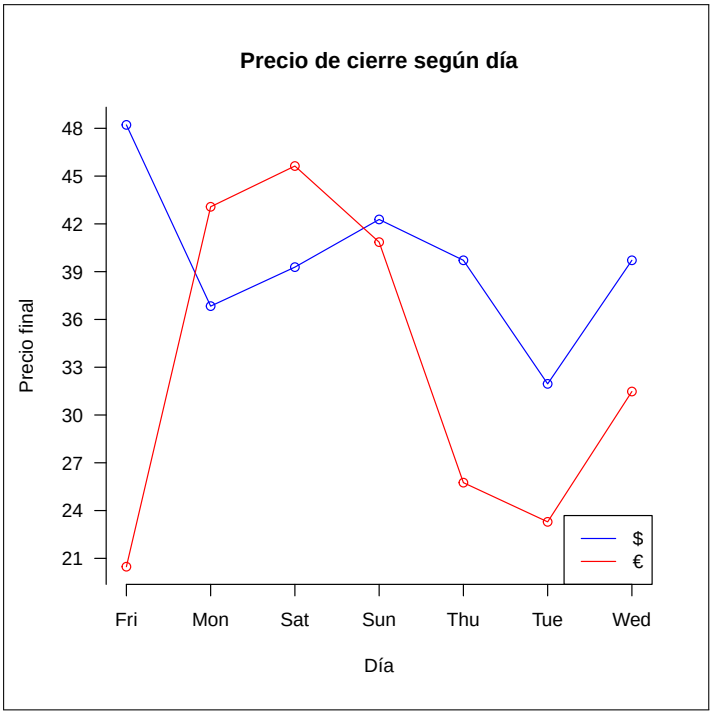

Figura 7.7: Comparación de precios de cierre por día y moneda

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()`.

Sintáxis 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')
```

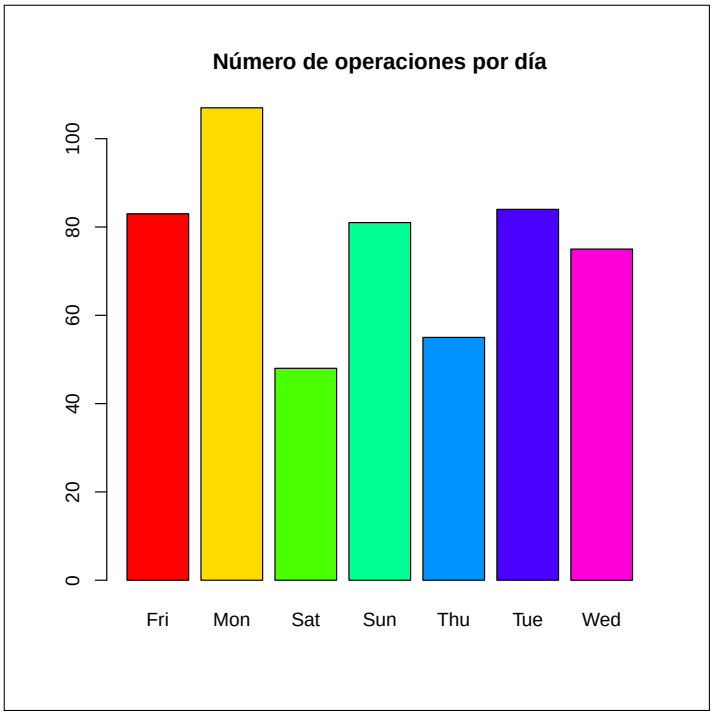

Figura 7.8: Gráfica de barras simple

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.

Sintáxis 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`.

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'))
```

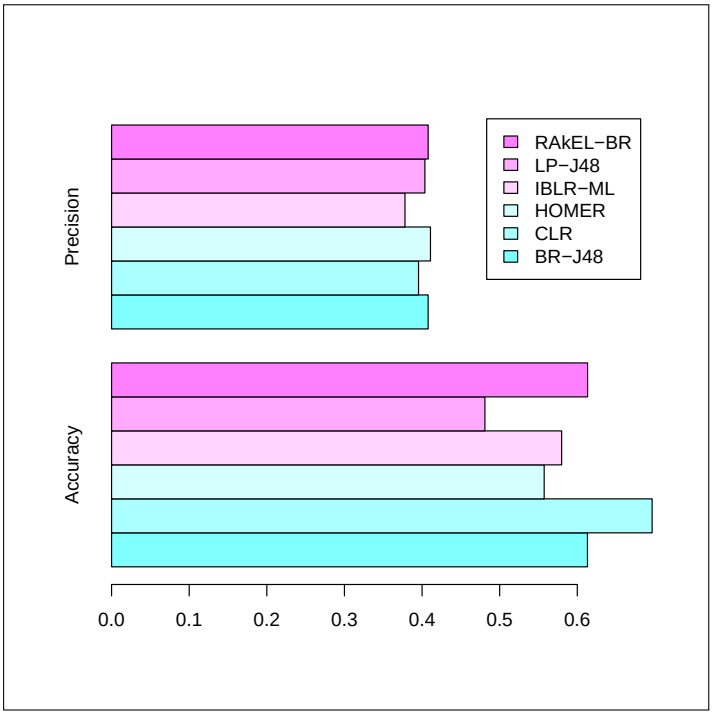

Figura 7.9: Gráfica de barras a partir de datos agregados

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()`.

Sintáxis 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:

- `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)
```

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.

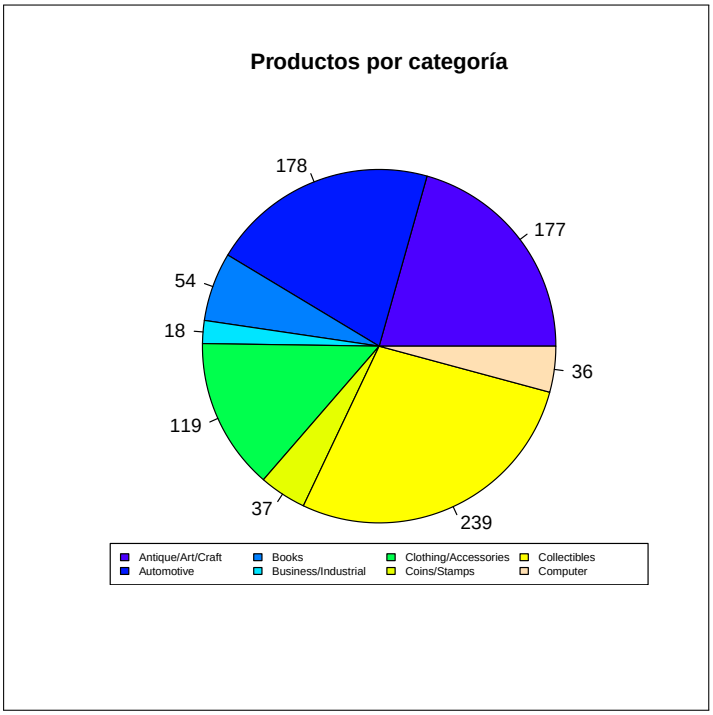

Figura 7.10: Proporción de productos por categoría en gráfica de sectores

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')
```

■

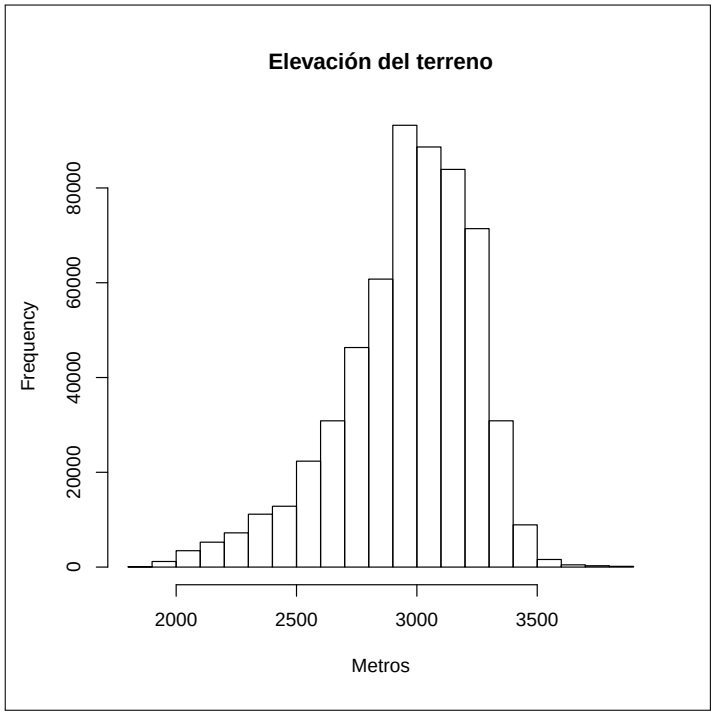

Figura 7.11: Histograma básico

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.

Sintáxis 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')
```

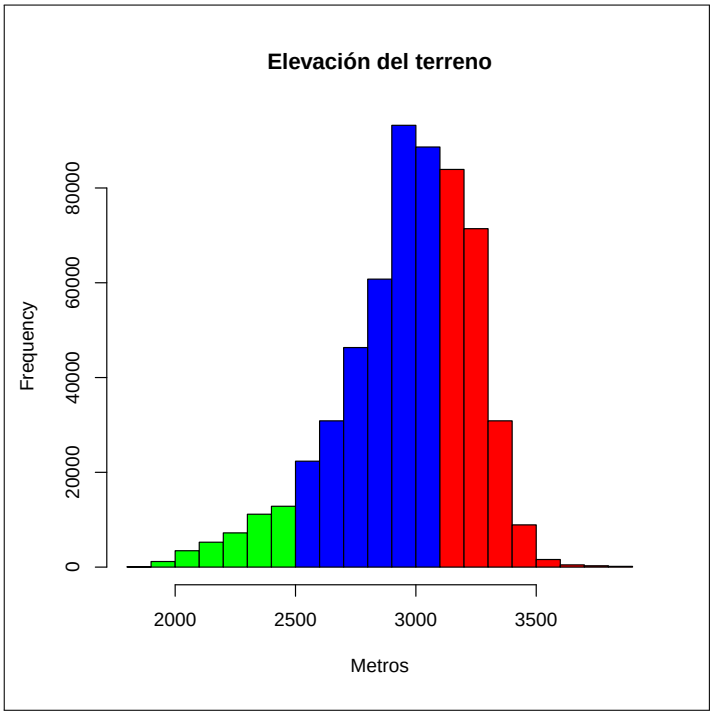

Figura 7.12: Personalización del histograma

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.

Sintáxis 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)
```

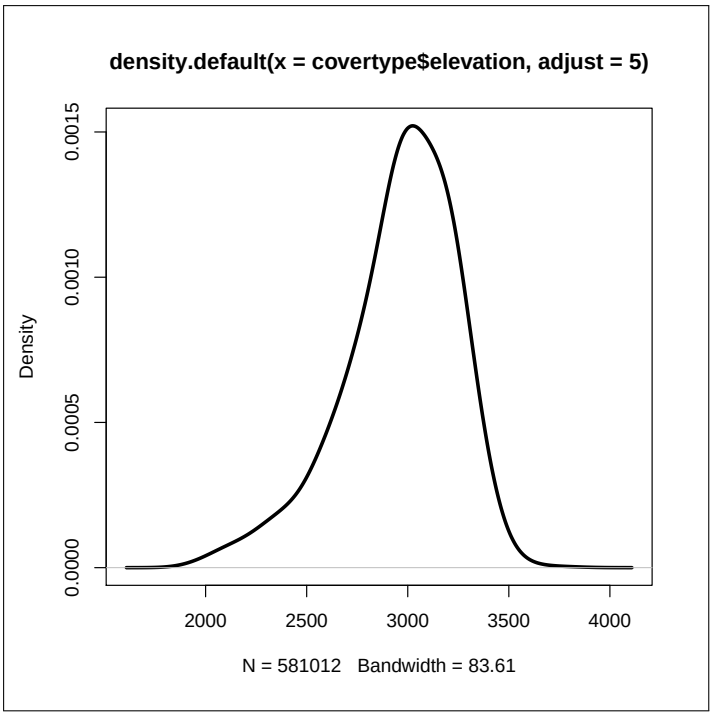

Figura 7.13: Curva de densidad sin histograma

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)
```

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.

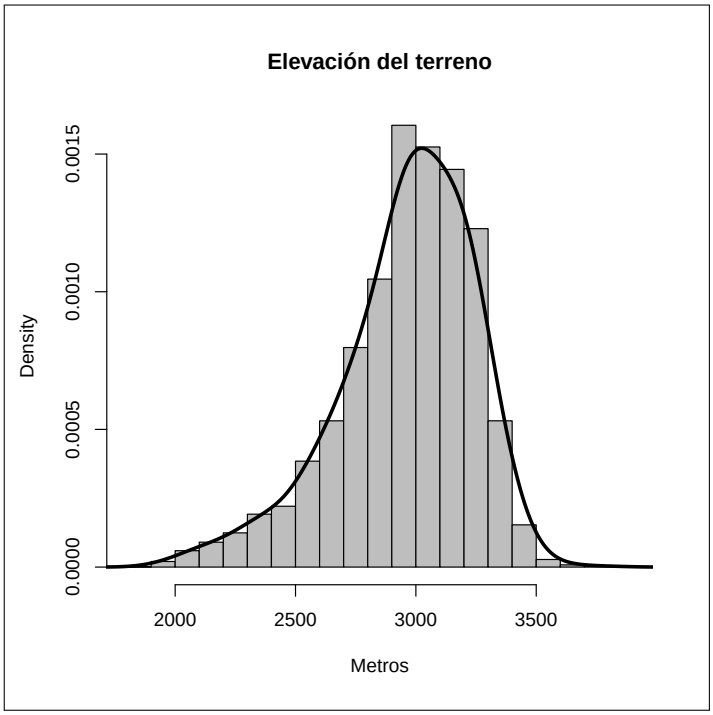

Figura 7.14: Histograma y curva de densidad

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.

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.

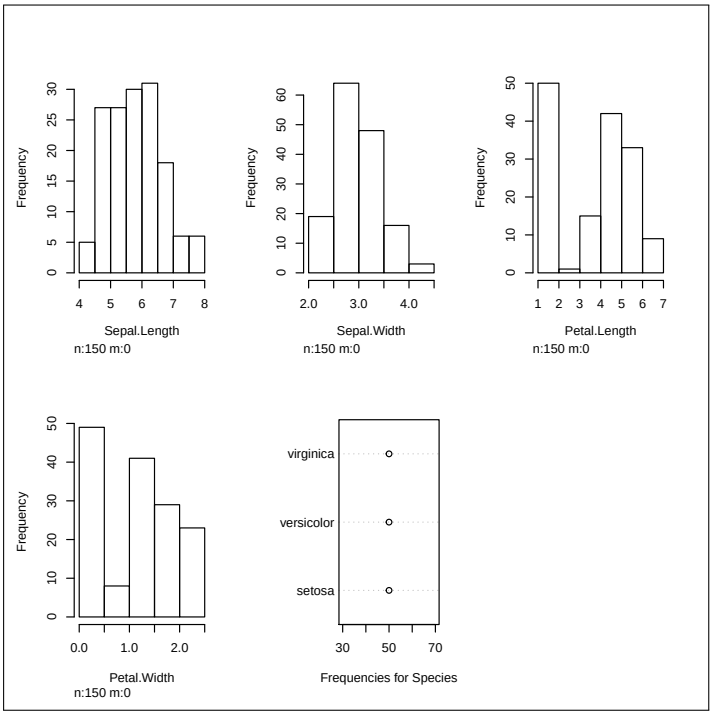

Figura 7.15: Histograma de objetos complejos

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

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

■

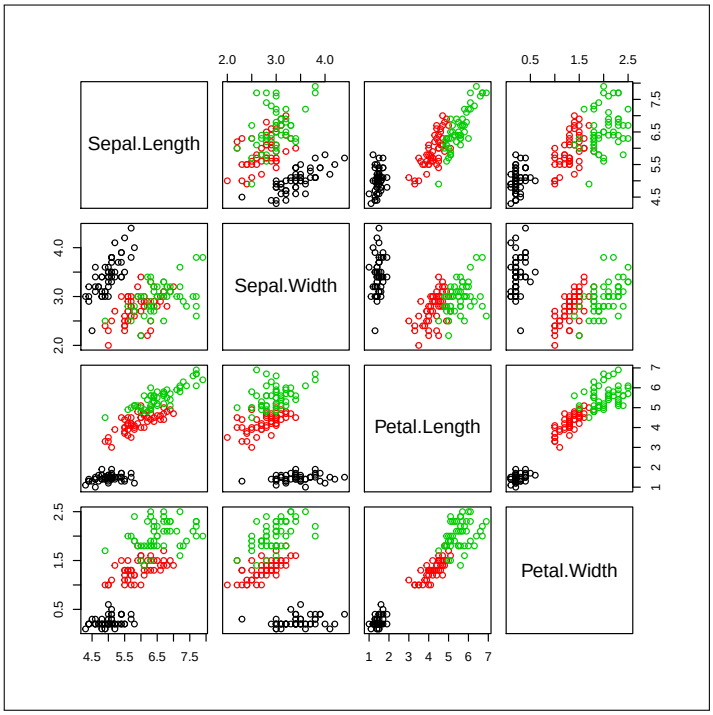

Figura 7.16: Combinaciones de los atributos en iris

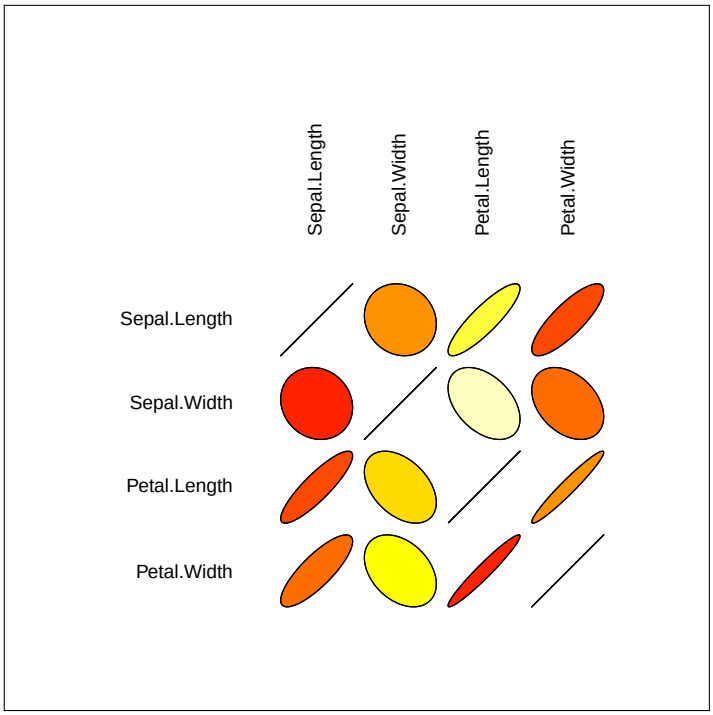

Figura 7.17: Gráfica de correlación entre los atributos de iris

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.

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

Sintáxis 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.

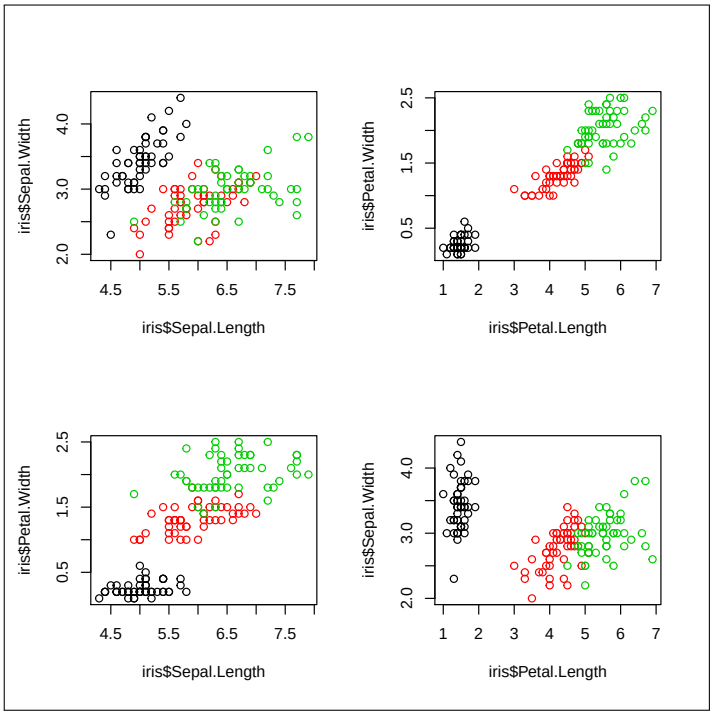

Figura 7.18: Composición de nXm gráficas usando `par()`

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)
```

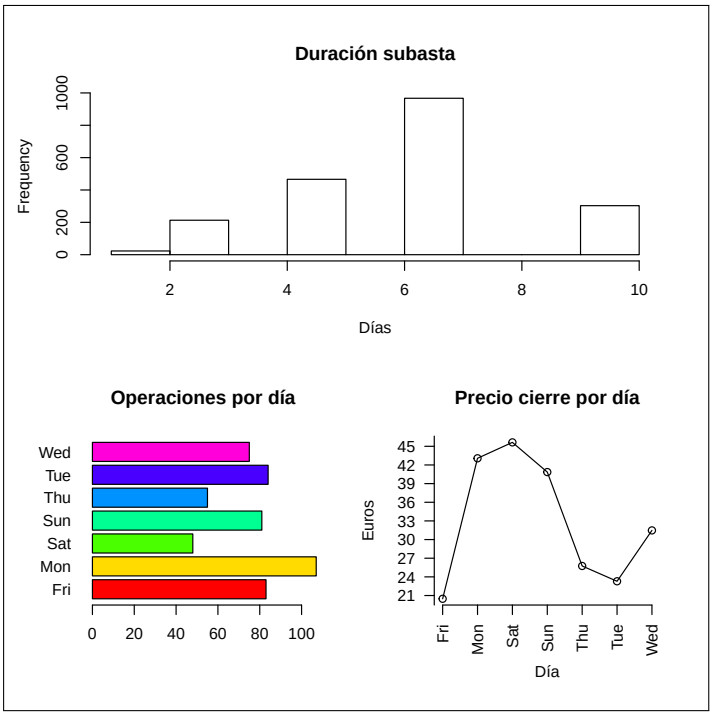

Figura 7.19: Composición libre de gráficas usando `layout()`

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 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.

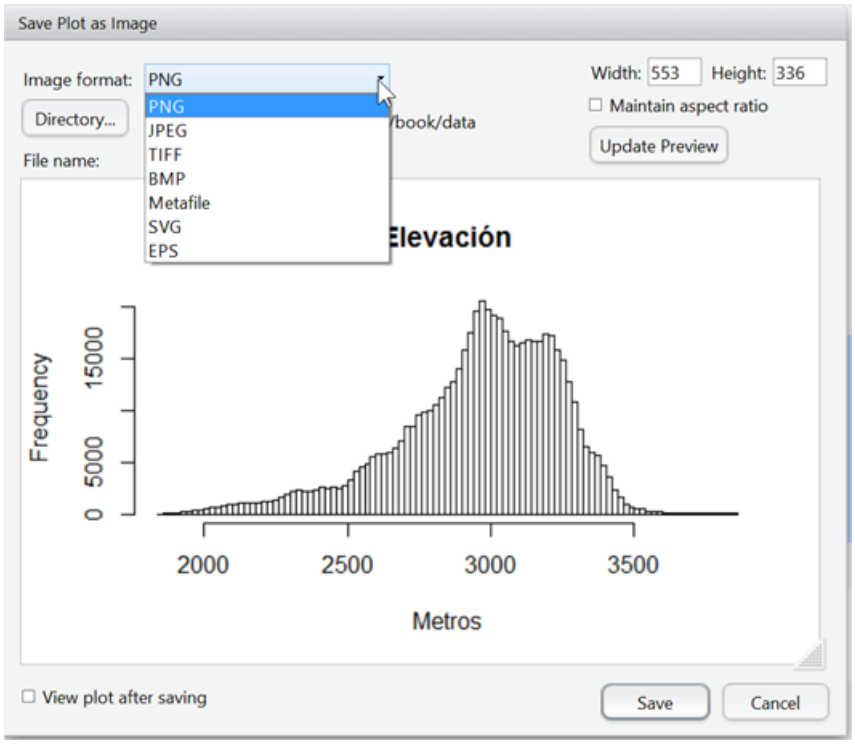

Figura 7.20: RStudio nos ofrece múltiples opciones de exportación de gráficos

Sintáxis 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.

Sintáxis 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)
```

■

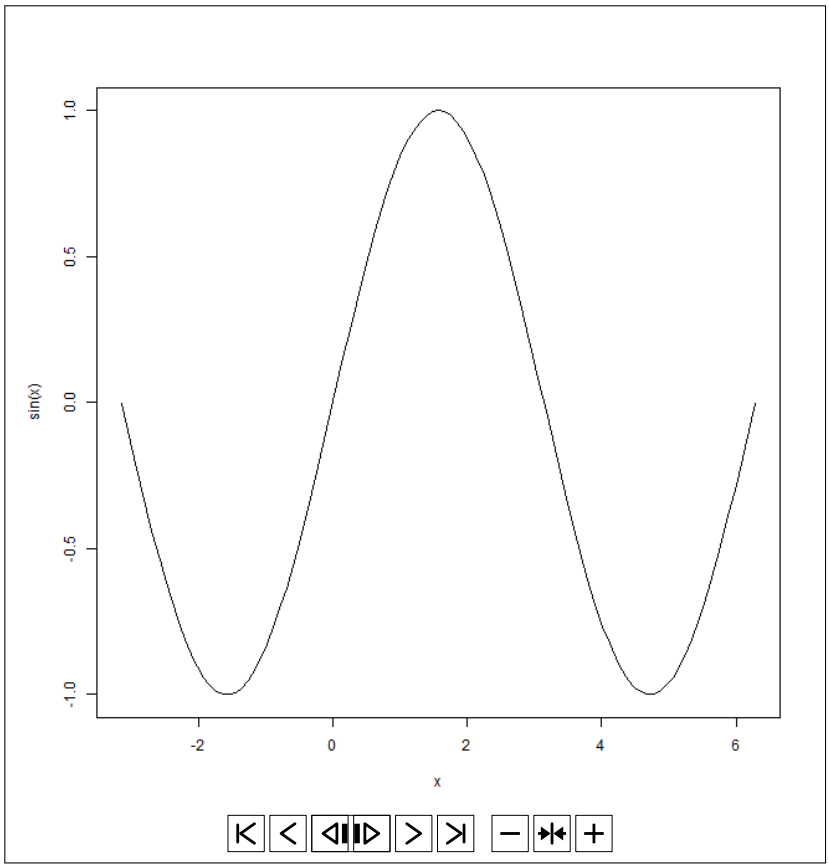

Figura 7.21: Animación de una función generada con saveLatex()