

R para profesionales de los datos: una introducción

Carlos J. Gil Bellosta

2018-04-22

Índice general

1. Introducción	7
1.1. Contenido y plan del libro	7
1.2. RStudio y la instalación del entorno	8
2. Tablas (dataframes)	11
2.1. Inspección de una tabla	11
2.2. Selección de filas y columnas	13
2.3. Creación y eliminación de tablas y columnas	14
2.4. Ordenación	15
2.5. Lectura de datos externos	15
2.6. Gráficos básicos	17
2.7. Resumen y referencias	25
2.8. Ejercicios adicionales	26
3. Vectores	27
3.1. Creación de vectores	27
3.2. Inspección de vectores	28
3.3. Selecciones	29
3.4. Ordenación	30
3.5. Operaciones matemáticas y vectorización	31
3.6. Digresión: creación de funciones	32
3.7. La función tapply	33
3.8. Resumen y referencias	34
3.9. Ejercicios adicionales	34
4. Introducción a la estadística con R	37
4.1. El test de Student	37
4.2. Regresión lineal	39
4.3. Regresión logística	41
4.4. Resumen y referencias	43
4.5. Ejercicios adicionales	43
5. Listas	45
5.1. Algunos usos de las listas	45
5.2. Exploración y manipulación de listas	46
5.3. Las funciones lapply, sapply y split	47
5.4. Resumen y referencias	49
5.5. Ejercicios adicionales	49
6. Paquetes, rmarkdown y shiny	51
6.1. Paquetes	51
6.2. Rmarkdown	52

6.3. Shiny	53
6.4. Resumen y referencias	55
6.5. Ejercicios adicionales	55
7. Web scraping y manipulación básica de texto	57
7.1. Web scraping	57
7.2. Manipulación básica de texto	58
7.3. JSON y XML	60
7.4. Resumen y referencias	62
7.5. Ejercicios adicionales	62
8. Pivotación de tablas con reshape2	63
8.1. La función melt y datos en formato largo	63
8.2. La función dcast y datos en formato ancho	66
8.3. Resumen y referencias	69
8.4. Ejercicios adicionales	69
9. Introducción a ggplot2	71
9.1. Una primera toma de contacto	73
9.2. Elementos de un gráfico en ggplot2	74
9.3. Ejemplos	80
9.4. Resumen y referencias	86
9.5. Ejercicios adicionales	86
10. Introducción a ggmap	87
10.1. Funciones de ggmap	88
10.2. Ejemplos	91
10.3. Resumen y referencias	96
10.4. Ejercicios adicionales	97
11. Procesamiento de tablas por trozos con plyr	99
11.1. Resúmenes de datos por bloques	99
11.2. Otras funciones similares a ddply de plyr	101
11.3. Transformaciones de datos por bloques	101
11.4. Más allá de summarize y transform	102
11.5. Cruces de tablas	103
11.6. Resumen y referencias	105
11.7. Ejercicios adicionales	105
12. Programación en R	107
12.1. Programación imperativa en R	107
12.2. Programación funcional en R	113
12.3. Orientación a objetos	115
12.4. Resumen y referencias	117
12.5. Ejercicios adicionales	117
13. Estadística y ciencia de datos con R	119
13.1. Árboles de decisión	119
13.2. Igualdad de medias y t-test	122
13.3. Igualdad de medias mediante remuestreos	124
13.4. Regresión lineal	125
13.5. Regresión logística	129
13.6. Modelos no lineales	131
13.7. Clustering con k-medias	132
13.8. Resumen y referencias	133

13.9. Resumen y referencias	134
13.10 Ejercicios adicionales	134
14. Ejemplos y casos de uso	135
14.1. Una calculadora de hipotecas con R	135
15. Apéndice	137
15.1. Distribuciones de probabilidad	137

Capítulo 1

Introducción

Las herramientas ofimáticas habituales (y estamos hablando aquí esencialmente de Excel) para manipular datos no dan más de sí. Cada vez hay más datos, cada vez son más grandes y los análisis son, cada vez, más sofisticados. El profesional de hoy en día necesita renovar su utilaje.

R es una magnífica opción para llegar más allá, para ser más productivo. Uno de los ingredientes fundamentales de la productividad es la posibilidad de automatizar tareas: hace falta programar las que sean tediosas y repetitivas; hace falta familiarizarse con al menos un lenguaje de programación. Como R.

Existen otros, sin duda, con grandes méritos. Python es uno de ellos. Este libro propone R por varios motivos:

- Porque tiene un mercado real en la industria y la academia.
- Porque está especialmente indicado para la manipulación y el análisis de datos.
- Y porque, más específicamente, está especialmente indicado para el tipo de manipulaciones y análisis de datos que realizan los profesionales de los datos (p.e., responsables de marketing analítico, analistas de riesgos, etc.) en los organismos y empresas de hoy en día.

1.1. Contenido y plan del libro

Este libro está motivado por la experiencia del autor en entornos de trabajo similares a los descritos más arriba y cubre aquellos aspectos que ha visto que más demandaban sus colegas. Que son, esencialmente, cuatro:

- Crear visualizaciones de datos de alta calidad.
- Crear *dashboards* para visualizar y analizar datos.
- Crear informes automáticos.
- Disponer de herramientas de análisis estadístico para ahondar en el conocimiento de los datos.

Este libro se centra en los tres primeros, aunque hace una breve incursión en el último. El motivo para excluirlo es que exige no solo un par de capítulos en este libro sino una biblioteca entera que, seguro, desarrollarán mucho mejor otros autores.

Claro que son inalcanzables si el interesado no se familiariza con los prerrequisitos (tablas, vectores, programación) que son los mimbres con los que armar aquellos canastos. El plan del libro incluye el ubicar las recompensas lo más tempranamente posible para que el trayecto por los capítulos más áridos sean más llevaderos. Obviamente, eso deja muchos cabos sueltos que el autor espera que el lector, motivado por las aplicaciones, cubra de mejor grado. Por eso, además, cada sección termina con un apartado lleno de consejos y referencias para llegar más allá de lo que aquí se cuenta.

La obra deja fuera varios temas de interés como, por ejemplo:

- Las series temporales. Porque son objetos especiales que se manipulan y se analizan con herramientas específicas. Incluirías contribuiría a confundir por acumulación de conceptos nuevos antes que a facilitar la transición de lector desde completo ignorante hasta usuario capaz de manejar R autónomamente.
- La manipulación de datos grandes. R sufre una no del todo justificada fama de tener problemas para manipular datos grandes (o medianos, i.e., datos con muchos registros pero que aún caben en la RAM de un ordenador moderno). Aunque existen extensiones *ad hoc* para procesarlos eficazmente, por el mismo motivo que antes, no se tratarán aquí.
- La estadística. R es también, entre otras muchas cosas, una herramienta para el análisis estadístico de datos. Pero todo lo que tiene que ver con la estadística viene justo después de lo que el libro trata. Incluso cuando toque temas de interés estadístico (p.e., la regresión lineal), lo hará más desde la perspectiva de su encaje en R que desde la descripción del aparataje matemático subyacente.

El libro no está ordenado por materias. De hecho, comparado con otras obras más formales, está desordenado. Ni siquiera comienza por lo más básico (p.e., vectores) sino por lo más familiar para su audiencia potencial: las tablas. Esto es así porque el libro es una carrera contrarreloj cuyas metas son las recompensas anunciadas más arriba: gráficos, informes automatizados y *dashboards*.

El libro quiere dejar claro que R no es solo un lenguaje de programación: más bien, R es *también* un lenguaje de programación. El usuario habitual de R no programa propiamente sino que utiliza R interactivamente: ensaya, se equivoca y vuelve a probar. Solo cuando termina el ciclo y el resultado es satisfactorio, produce un resultado final. Que, usualmente, no es un programa sino, p.e., un informe. Por eso la parte relativa a la programación se relega a la parte final.

El libro lleva al lector hasta el punto en el que puede comenzar a aplicar métodos estadísticos (y de la llamada ciencia de datos) por su cuenta. El último capítulo es una introducción a la materia. En concreto, a cómo se aplicar esos métodos con R. La gran lección que aprender es que gran parte de los métodos estadísticos habituales no solo están disponibles, sino que tienen un tratamiento homogéneo y previsible en R.

Existe un principio director en Python: debería existir un método obvio (y preferiblemente solo uno) de hacer las cosas. En R no es así: existen tal vez demasiadas maneras alternativas de hacer las cosas y eso es un problema, un problema muy desconcertante, para el principiante. No obstante, por motivos pedagógicos, el libro tratará de presentar una y solo una de las formas de resolver un determinado problema: la que el autor, fiable, considere más natural. De todos modos, el autor espera que el lector sea capaz, al final de la obra, de dar con, aplicar, evaluar y comparar las distintas alternativas.

1.2. RStudio y la instalación del entorno

El libro es eminentemente práctico y exige que el lector no solo vaya ejecutando el código que se presenta de ejemplo sino que resuelva los problemas que a lo largo de la exposición se le plantean. Para eso es necesario tener R instalado.

1.2.1. Instalación de R y RStudio

La instalación de R está documentada en todas partes. Puede descargarse de <https://cran.r-project.org> para instalarlo luego como cualquier programa. Para muchos sistemas operativos eso no es siquiera necesario porque R está disponible en la mayor parte de los repositorios habituales de programas y bibliotecas. Por ejemplo, en Ubuntu y similares, suele bastar con ejecutar

```
sudo apt-get install r-base
```

Nota

Suele publicarse una nueva versión de R cada seis meses. Es conveniente utilizar siempre la última versión. Entre otros motivos, porque arregla los errores que los usuarios van detectando.

Hoy en día, la mejor plataforma para utilizar R (sobre todo para el principiante) es RStudio¹. RStudio es un IDE muy popular y que ofrece una entorno prácticamente idéntico en todos los sistemas operativos para utilizar R.

1.2.2. Brevísima introducción a RStudio

RStudio, por defecto, tiene cuatro paneles. El panel inferior izquierdo es una consola de R. En ella se puede escribir y ejecutar código. R muestra también en ella los resultados obtenidos.

El panel superior izquierdo es un editor de código. Los ficheros que se abran y se editen aparecerán en él dentro de sus correspondientes pestañas. Es imperativo aprender a usar algunos de los atajos de teclado más comunes². P.e., **Control-Enter** ejecuta la línea de código en la que se sitúa el cursor: la copia en la consola, la ejecuta y muestra los resultados obtenidos.

Los paneles de la derecha son menos importantes. El superior contiene un listado de las variables en el entorno y un histórico de comandos ejecutados. De entre todas las pestañas que aparecen en el inferior, las más usadas generalmente son:

- *Files*: da acceso al sistema de ficheros del disco duro
- *Plots*: aloja los gráficos que cree R
- *Help*: muestra la página de ayuda de las funciones cuando la solicite el usuario

¹Para descargarlo, busca en internet *download rstudio* y sigue las instrucciones.

²Pueden encontrarse listas de ellos buscando en Google *rstudio keyboard shortcuts*.

Capítulo 2

Tablas (dataframes)

Muy frecuentemente, los datos se disponen en tablas: las hojas de cálculo, las bases de datos, los ficheros `csv`, etc. contienen, esencialmente, tablas. Además, casi todos los métodos estadísticos (p.e., la regresión lineal) operan sobre información organizada en tablas. Como consecuencia, gran parte del día a día del trabajo con R consiste en manipular tablas de datos para darles el formato necesario para acabar analizándolos estadística o gráficamente.

Las hojas de cálculo son herramientas que prácticamente todos hemos usado para manipular tablas de datos. El objetivo de esta primera parte del libro es aprender a realizar operaciones habituales y bien conocidas por los usuarios de Excel sobre tablas de datos en R.

2.1. Inspección de una tabla

Nuestra primera tarea consistirá en inspeccionar los contenidos de una tabla. Esta es, de hecho, la primera tarea que uno realiza en la práctica cuando recibe un conjunto nuevo de datos: averiguar cuántas filas y columnas tiene, de qué tipo (numérico o no) son sus columnas, etc.

Para practicar y como ejemplo, usaremos una tabla que viene *de serie* con R, `iris`¹. Tiene 150 filas que corresponden a otras tantas iris ([una especie de flor] (http://es.wikipedia.org/wiki/Iris_%28planta%29)) y sus columnas contienen cuatro características métricas de cada ejemplar: la longitud y la anchura de sus pétalos y sépalos; y la subespecie: setosa, versicolor o virgínica, a la que pertenece. De momento y hasta que aprendamos cómo importar datos de fuentes externas, utilizaremos este y otros conjuntos de datos de ejemplo que incluye R.

Para inspeccionar una tabla utilizaremos una serie de funciones básicas que se usan muy frecuentemente en R.

Escribir en la consola de R²

```
iris
```

es lo mismo que ejecutar

```
print(iris)
```

y muestra la tabla en la consola. Cosa que no es particularmente útil si la tabla tiene muchas filas.

¹Este conjunto de datos tiene una larga historia (fue publicado originalmente en 1935) y se ha venido utilizando para ilustrar el uso de ciertos modelos estadísticos de clasificación.

²O, si la línea aparece en un programa, colocando el cursor en cualquier punto de ella y pulsando **Control-Enter**.

Este comportamiento no es exclusivo de las tablas. Aplica también a otros tipos de datos que veremos más adelante: *ejecutar* el nombre de un objeto *imprime* (muestra) directamente en la consola una representación textual del mismo.

Otras funciones útiles para inspeccionar tablas (y, como veremos más adelante, no solo tablas) son

```
plot(iris)      # la representa gráficamente
summary(iris)   # resumen estadístico de las columnas
str(iris)       # "representación textual" del objeto
```

Ejercicio 2.1.1 Trata de interpretar el gráfico generado con `plot(iris)`. ¿Qué nos dice, p.e., de la relación entre la anchura y la longitud de los pétalos?

Ejercicio 2.1.2 Trata de interpretar la salida del resto de las anteriores líneas de código. ¿Qué nos dicen del conjunto de datos?

Es impráctico mostrar tablas enteras en la consola, sobre todo cuando son grandes. Para mostrar solo parte de ellas,

```
head(iris)      # primeras seis filas
tail(iris)      # últimas seis filas
```

En R, la ayuda de una función (`summary` en el ejemplo siguiente), se consulta así³:

```
?summary
```

Si usas RStudio, el texto de la ayuda correspondiente a esa función aparecerá en la pestaña correspondiente del panel inferior derecho.

Ejercicio 2.1.3 Consulta la ayuda de la función `head` y averigua cómo mostrar las diez primeras filas de `iris` en lugar de las seis que aparecen por defecto.

La ayuda de una función en R siempre tiene la misma estructura:

- una breve descripción de la función,
- una descripción de los argumentos que admite,
- una sección que detalla el funcionamiento de la función y de los distintos argumentos y
- una sección final de ejemplos.

Habitualmente, los ejemplos suelen más instructivos que el resto: con un poco de suerte, el autor de la página de ayuda ha incluido uno que, con pequeños cambios, te puede servir.

Además de lo anterior, para inspeccionar una tabla es fundamental conocer su tamaño

```
dim(iris)      # filas x columnas
```

```
## [1] 150    5
```

```
nrow(iris)    # número de filas
```

```
## [1] 150
```

```
ncol(iris)    # número de columnas
```

```
## [1] 5
```

y el nombre de sus columnas

³También se puede escribir `help(summary)`, pero es más largo y nadie lo hace.

```
colnames(iris)

## [1] "Sepal.Length" "Sepal.Width"   "Petal.Length" "Petal.Width"
## [5] "Species"
```

Ejercicio 2.1.4 Consulta el tamaño, número de filas y el número y nombre de las columnas del conjunto de datos airquality; muestra también las primeras 13 filas de esa tabla.

Ejercicio 2.1.5 Examina el conjunto de datos `attenu`. Consulta su ayuda (`?attenu`) para averiguar qué tipo de información contiene. Finalmente, usa `summary` para ver si contiene algún nulo en alguna columna.

Ejercicio 2.1.6 Haz una lista con todas las funciones que aparecen nombradas en esta sección y trata de recordar lo que hace cada una de ellas.

2.2. Selección de filas y columnas

Igual que ocurre con una hoja de cálculo, frecuentemente queremos trabajar con un subconjunto de la tabla, i.e., con una selección de filas y/o columnas. Para este tipo de selecciones se usa el corchete:

```
iris[1:10,]      # diez primeras filas
iris[, 3:4]       # columnas 3 y 4
iris[1:10, 3:4]
```

Cuando se opera con tablas, los corchetes **siempre**⁴ tienen dos partes separadas por una coma:

- la que la precede se refiere a las filas
- la que la sigue, a las columnas.

No es necesario conocer índice de una determinada columna para seleccionarla: los corchetes admiten también el nombres de columnas:

```
iris[, "Species"]
```

Es tan habitual extraer y operar sobre columnas individuales de una tabla que R proporciona una manera más breve de hacerlo:

```
iris$Species
```

De hecho, se prefiere habitualmente (porque es más rápido y más legible) `iris$Species` a `iris[, "Species"]`.

— Nota —

R está diseñado fundamentalmente para ser usado interactivamente, es decir, para que el usuario utilice la consola para ensayar, probar distintas alternativas, etc. Por eso R está lleno de *atajos* pensados para abreviar el trabajo, dispone de operadores flexibles como los corchetes, etc. Eso lo diferencia de otros lenguajes, frecuentemente mucho más rígidos, que están orientados a crear programas más o menos complejos.

El corchete también permite seleccionar filas mediante condiciones lógicas:

```
iris[iris$Species == "setosa", ]
```

⁴No es ortotipográficamente correcto usar negritas de esta manera; sin embargo, es tan importante recordarlo que, por una vez, el autor se saltará la norma.

Ejercicio 2.2.1 Selecciona las filas de `iris` cuya longitud del pétalo sea mayor que 4.

Ejercicio 2.2.2 Selecciona las filas donde `cyl` sea menor que 6 y `gear` igual a 4 en `mtcars`. Nota: el operador AND en R es `&`.

Nota

La selección de filas mediante condiciones lógicas es muy útil y será muy necesaria posteriormente cuando queramos eliminar sujetos con edades negativas, detectar los pacientes con niveles de glucemia por encima de un determinado umbral, etc.

Los ejemplos anteriores dan cuenta de la versatilidad de `[]`. Dentro de él pueden indicarse:

- coordenadas (o rangos de coordenadas) de filas y columnas
- nombres de columnas
- condiciones lógicas para seleccionar filas que cumplan un criterio determinado

Más adelante, cuando tratemos los vectores⁵, veremos cómo:

- seleccionar no solo rangos de columnas (p.e., `1:4`), sino columnas determinadas (p.e., la 1, 4 y 7)
- seleccionar varias columnas por nombre
- construir condiciones lógicas no triviales

2.3. Creación y eliminación de tablas y columnas

Es posible crear otras tablas a partir de una dada mediante el operador `<-`, que sirve para asignar:

```
mi.iris <- iris # mi.iris es una copia de iris
head(mi.iris)
```

Nota

Los nombres de objetos siguen las reglas habituales en otros lenguajes de programación: son secuencias de letras y números (aunque no pueden comenzar por un número) y se admiten los separadores `_` y `:` tanto `mi.iris` como `mi_iris` son nombres válidos. Hay quienes prefieren usar *camel case*, como `miIris`. Es cuestión de estilo; y, en cuestiones de estilo, todo es discutible salvo la consistencia.

`iris`, que viene de *serie* con R, no es un *objeto visible*. Sin embargo, los que crees tú, sí:

```
ls()      # lista de objetos en memoria
rm(mi.iris) # borra el objeto mi.iris
ls()
```

Además de con la función `ls`, los objetos que crees aparecerán listados en el panel correspondiente, el superior derecho de RStudio (si es que lo utilizas).

Añadir una columna a una tabla es como crear una nueva variable dentro de ella⁶. Una manera de eliminarlas es asignarle el valor `NULL`.

```
mi.iris <- iris
mi.iris$Petal.Area <- mi.iris$Petal.Length * mi.iris$Petal.Width
mi.iris$Petal.Area <- NULL
```

Ten en cuenta que

- agregar una columna que existe la reemplaza,

⁵Porque dentro del corchete podremos utilizar un vector de nombres o números e columnas.

⁶Existe una función, `transform` que también puede usarse para transformar tablas, en el sentido de añadirle nuevas columnas, que veremos más adelante.

- agregar una columna que no existe la crea y
- asignar `NULL` a una columna existente la elimina.

Ejercicio 2.3.1 Crea una copia del conjunto de datos `airquality`. Comprueba con `ls` que está efectivamente creado y luego añádele una columna nueva llamada `temperatura` que contenga una copia de `Temp`. Comprueba que efectivamente está allí y luego, elimínala. Finalmente, borra la tabla.

Ejercicio 2.3.2 Usando el conjunto de datos `CO2`, selecciona los valores en los que el tratamiento sea `chilled`, y el valor de `uptake`, mayor que 15; devuelve únicamente las 10 primeras filas.

2.4. Ordenación

La reordenación de las filas de una tabla es fundamental para analizar su contenido y, frecuentemente, para detectar problemas en los datos. Por ejemplo, al ordenar los sujetos por edad y echar un vistazo a las primeras filas podemos identificar aquellos que, por error, tienen asociadas edades negativas.

R no dispone de ninguna función *de serie* para ordenar por una columna (o varias)⁷. En R, *ordenar* es *seleccionar ordenadamente*:

```
mi.iris <- iris[order(iris$Petal.Length),]
```

La función `order` aplicada a un vector devuelve otro vector de la misma longitud que tiene el valor 1 en el primer elemento del vector, 2 en el segundo, etc. Es decir, en el ejemplo anterior, `mi.iris` tiene como primera fila aquella que corresponde al valor más pequeño de `iris$Petal.Length`, etc.

Ejercicio 2.4.1 Verifica que `mi.iris <- iris[order(-iris$Petal.Length),]` ordena decrecientemente.

Ejercicio 2.4.2 Crea una versión de `iris` ordenando por especie y dentro de cada especie, por `Petal.Length`. Ten en cuenta que en R se puede ordenar por dos o más columnas porque `order` admite dos o más argumentos (véase `?order`). Por ejemplo, `iris[order(iris$Petal.Length, iris$Sepal.Length),]` deshace los empates en `Petal.Length` de acuerdo con `Sepal.Length`.

Ejercicio 2.4.3 Encuentra el día más frío de los que contiene `airquality`.

Ejercicio 2.4.4 Usando el mismo conjunto de datos, encuentra el día más caluroso del mes de junio.

2.5. Lectura de datos externos

Por supuesto, no nos vamos a limitar a trabajar con datos de ejemplo provistos por R: queremos utilizar también nuestras propias tablas. En R hay decenas de maneras de importar datos tabulares desde distintas fuentes (Excel, bases de datos, SPSS, etc.) pero la más habitual consiste en leer ficheros de texto con la función `read.table`.

Existen muchas variantes de ficheros de texto: `csv`, ficheros separados con tabulador, con y sin encabezamientos, con distintos códigos para representar los nulos, etc. La función `read.table` es muy flexible y tiene muchos parámetros, casi demasiados, para adaptarse a la mayor parte de ellos⁸.

⁷Aunque más adelante veremos extensiones que sí permiten hacerlo

⁸Existen versiones de `read.table`, como `read.csv` o `read.csv2`, que son la misma función pero con valores por defecto distintos para poder leer (a menudo) directamente ficheros con formato `csv` y otros. Pero, en el fondo, no dejan de ser *alias* de `read.table`.

Antes de comenzar a utilizarla, sin embargo, es necesaria una digresión acerca de directorios y directorios de trabajo. El directorio de trabajo es aquel en el que R busca y escribe ficheros por defecto. Las funciones `getwd` y `setwd` permiten, respectivamente, averiguar cuál es y cambiarlo si procede. De todos modos, la mejor manera de especificar el directorio de trabajo en RStudio es usando los menús: `Files > More > Set as working directory`.

```
getwd()
setwd(..)      # "sube" al directorio padre del actual
setwd("mi_proyecto/src")    # ruta relativa
setwd("c:/users/yo/proyecto") # ruta absoluta en Windows
setwd("/home/yo/proyecto")   # ruta absoluta en Linux y otros
dir()           # contenidos del directorio "de trabajo"
```

Una llamada típica a `read.table` para leer un fichero es así:

```
datos <- read.table("data_dir/mi_fichero.csv", sep = "\t", header = TRUE)
```

Examinémosla:

- `datos` es el nombre de la tabla que recibirá la información leída por `read.table`; de no hacerse la asignación, R se limitará a imprimirlas en la consola.
- "`data_dir/mi_fichero.csv`" (entrecomillado!) es la ruta del fichero de interés. Se trata de una ruta relativa al directorio de trabajo, que se supone que contiene el subdirectorio `data_dir` y, dentro de él, el fichero `mi_fichero.csv`.
- `sep = "\t"` indica que los campos del fichero están separados por tabuladores (sí, el tabulador es `\t`). Muchos ficheros tienen campos separados por, además del tabulador, caracteres tales como `,`, `;`, `|` u otros. Es necesario indicárselo a R y la manera de averiguar qué separador usa un fichero, si es que no se sabe, es abriendolo previamente con un editor de texto decente⁹.
- `header = TRUE` indica que la primera fila del fichero contiene los nombres de las columnas. Si olvidas especificarlo y la primera fila del fichero contiene efectivamente el nombre de las columnas, R interpretará estas, erróneamente, como datos.

Con una expresión similar a esa, tal vez cambiando el separador, se leen la mayoría de los ficheros de texto habituales. Otras opciones de las muchas que tiene `read.table` que pueden ser útiles en determinadas ocasiones son:

- `dec`, para indicar el separador de decimales. Por defecto es `.`, pero en ocasiones hay que cambiarlo a `dec = ","` para que interprete correctamente, p.e., el antiguo estándar español (p.e., `67,56` en lugar de `67.56`).
- `quote`, que indica qué carácter se usa para acotar campos de texto. En algunas ocasiones aparecen campos de texto que contienen apóstrofes (p.e., calle O'Donnell) y la carga de datos puede fracasar de no indicarse `quote = .` Esta expresión desactiva el papel especial de acotación de campos de texto que por defecto tienen las comillas.

Nota

Los siguientes ejercicios (y muchos más a lo largo de libro) hacen referencia a conjuntos de datos disponibles en <https://bit.ly/2qS7DvO>. Descarga ese fichero y descomprímelo en alguna carpeta.

Ejercicio 2.5.1 Lee el fichero `paro.csv` usando la función `read.table`. Comprueba que está correctamente importado usando `head`, `tail`, `nrow`, `summary`, etc. Para leer la tabla necesitarás leer con cierto detenimiento `?read.table`.

Ejercicio 2.5.2 Repite el ejercicio anterior eliminando la opción `header = TRUE`. Examina el resultado y comprueba que, efectivamente, los datos no se han cargado correctamente.

⁹Notepad no es un editor de texto decente.

Ejercicio 2.5.3 Lee algún fichero de datos de tu interés y repite el ejercicio anterior.

Ejercicio 2.5.4 En `read.table` y sus derivados puedes indicar, además de ficheros disponibles en el disco duro, la URL de uno disponible en internet. Prueba a leer directamente el fichero disponible en https://datanalytics.com/uploads/datos_treemap.txt. Nota: es un fichero de texto separado por tabuladores y con nombres de columna.

Ejercicio 2.5.5 Alternativamente, si quieres leer un fichero remoto, puedes descargarlo directamente desde R. Consulta la ayuda de `download.file` para bajarte al disco duro el fichero del ejercicio anterior y leerlo después.

2.6. Gráficos básicos

Esta sección es una introducción a los gráficos básicos en R orientada a la inspección visual y rápida de conjuntos de datos, que es fundamental en todo proceso de análisis y, particularmente, en sus fases iniciales.

No nos preocuparemos demasiado de los aspectos estéticos de estos gráficos. En primer lugar, porque más adelante trataremos otros tipo de gráficos más sofisticados y atractivos estéticamente. Pero también porque los detalles sobre cómo modificar el aspecto de los gráficos es tan prolífico y lleno de detalles que es mejor omitirlo en una primera aproximación. Además, internet contiene seguramente la respuesta a cualquier pregunta que se te ocurra sobre cómo modificar los valores por defecto: ejes, orientación de etiquetas, etc. Es un campo amplio y lleno de detalles pero que es más bien material de consulta puntual en un momento de necesidad que de exposición exhaustiva en un texto introductorio.

En particular, en esta sección trataremos la manera de representar:

- Una variable continua
- Una variable categórica
- La relación entre dos variables continuas
- La relación entre una variable continua y otra categórica

2.6.1. Representación gráfica de variables continuas: histogramas

Nuestros datos pueden contener una columna como, por ejemplo, `edad`. En apartados anteriores hemos aprendido:

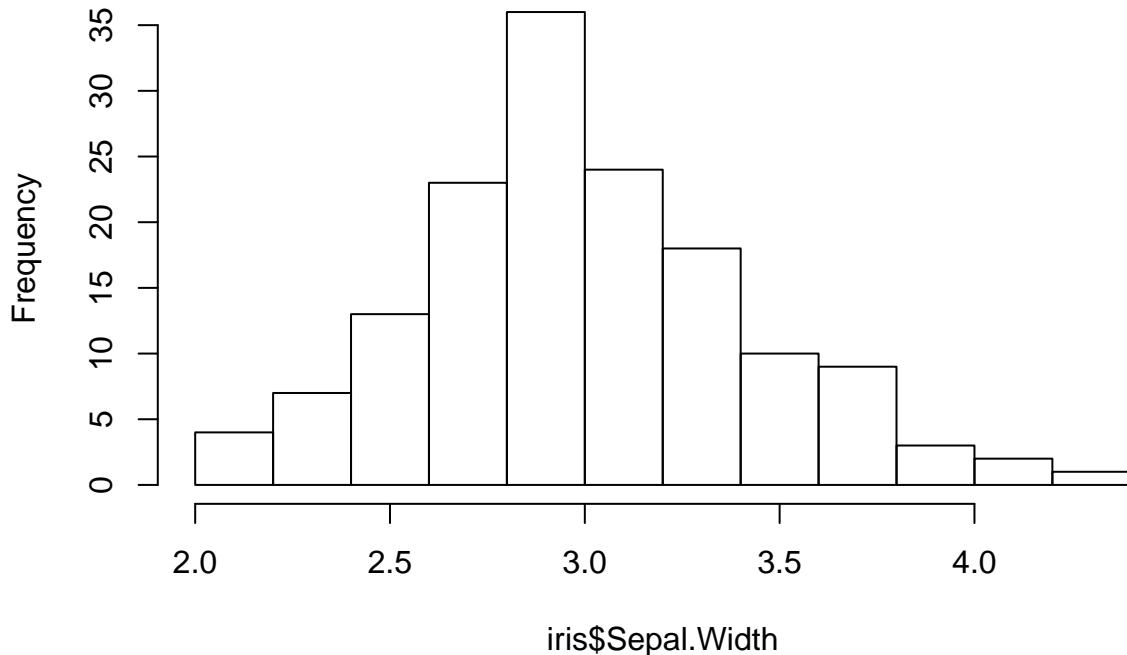
- cómo inspeccionar los valores extremos de esa variable (p.e., ordenando la tabla por edad y mostrando las primeras y las últimas filas con las funciones `head` y `tail`) por si, por ejemplo, existen edades negativas;
- cómo obtener algunos estadísticos básicos de esa columna (usando `summary` sobre la tabla).

Ejercicio 2.6.1 Inspecciona la columna `Temp` (temperatura) del conjunto de datos `airquality` de acuerdo con las sugerencias del párrafo anterior.

Sin embargo, es mucho más informativa una representación visual de los datos. La manera más rápida (y recomendada) de hacerse una idea de la distribución de los datos de una columna numérica es usando histogramas. En R, para representar el histograma de la columna `Sepal.Width` de `iris` se puede hacer:

```
hist(iris$Sepal.Width)
```

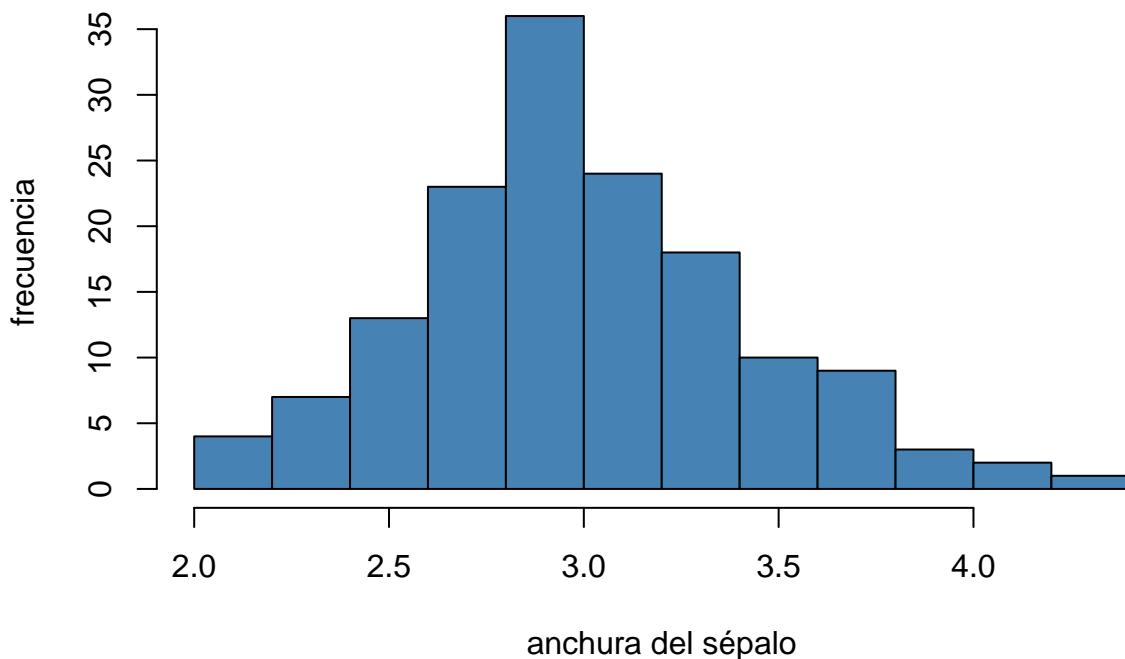
Histogram of iris\$Sepal.Width



Esa es la orden básica. Pero los gráficos pueden ser modificados para incluir títulos, etiquetas, colores, etc. Por ejemplo,

```
hist(iris$Sepal.Width, main = "iris: histograma de la anchura de los sépalos",
     xlab = "anchura del sépalo", ylab = "frecuencia",
     col = "steelblue")
```

iris: histograma de la anchura de los sépalos



Los argumentos `main`, `xlab`, `ylab` y `col`¹⁰ se pueden aplicar también a otros gráficos que veremos a continuación.

Ejercicio 2.6.2 Por defecto, el eje horizontal de un histograma muestra el número de observaciones en cada *bin*. Examina la ayuda de `hist` para ver cómo mostrar en, lugar de los números absolutos, la proporción.

Ejercicio 2.6.3 El número de *bins* también es parametrizable. Examina otra vez la página de ayuda para modificar el valor por defecto.

Ejercicio 2.6.4 Estudia la distribución de las temperaturas en Nueva York (usa `airquality`).

Para guardar el gráfico, puedes usar los menús de Rstudio. Pero también puedes hacerlo programáticamente. En la página de ayuda de la función `png` se explica cómo hacerlo¹¹.

Ejercicio 2.6.5 Usa las funciones `png` y `jpeg` para guardar alguno de los gráficos anteriores en tu disco duro.

2.6.2. Representación gráfica de variables categóricas: barras

En las tablas suelen coexistir variables continuas y categóricas. Por ejemplo, es interesante conocer la distribución (o frecuencia) de cada una de las categorías. Para eso se suelen usar los diagramas de barras; en particular, la función `barplot` de R.

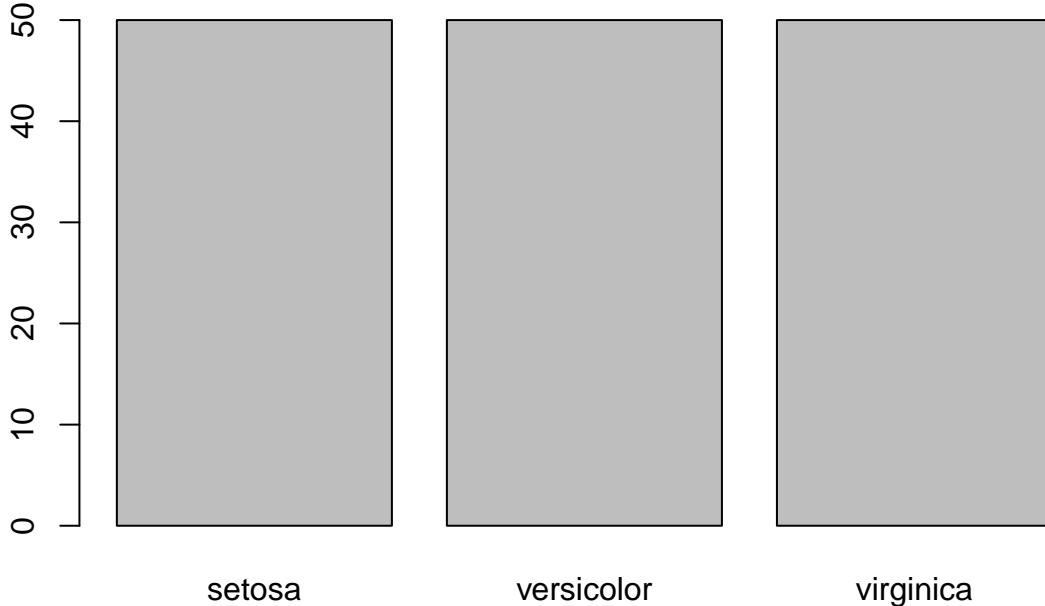
¹⁰En el ejemplo anterior se ha usado el color `steelblue`. Si buscas en internet encontrarás la lista completa de aquellos colores cuyos nombres entiende R o cómo usar sus representaciones RGB u otras.

¹¹Seguro, entenderás mejor los ejemplos de esa página que el mismo cuerpo de la documentación

Esta función no muestra directamente las frecuencias de una variable categórica. Es necesario calcular previamente dichas frecuencias, para lo cual usaremos la función `table` que se tratará con más detalle posteriormente.

Por ejemplo, la expresión siguiente muestra cómo en `iris` existe el mismo número de observaciones de cada especie:

```
barplot(table(iris$Species))
```

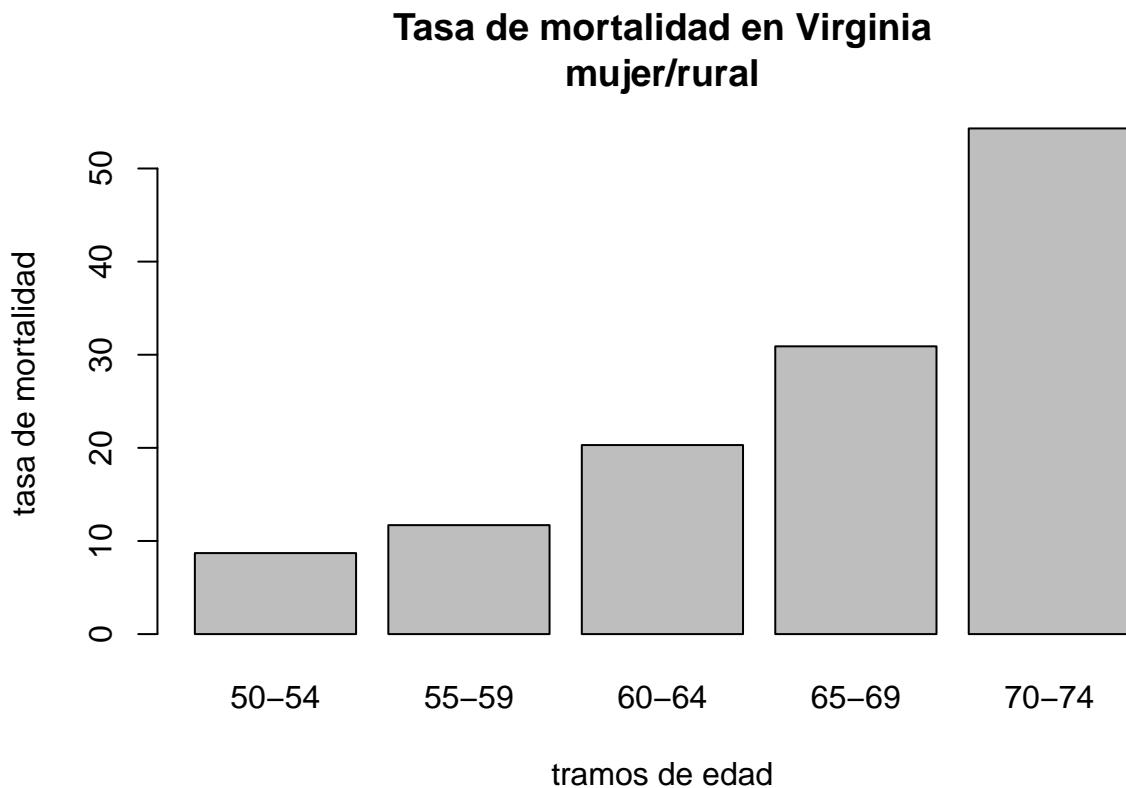


Ejercicio 2.6.6 Usa los parámetros `main`, `xlab`, `ylab` y `col` discutidos en la sección anterior para mejorar el aspecto de este gráfico.

Ejercicio 2.6.7 Investiga el argumento `horiz` de `barplot` para crear un gráfico de barras horizontales.

Los diagramas de barras también pueden usarse para mostrar datos contenidos en vectores etiquetados. De hecho, `table` crea un vector etiquetado: asocia a cada etiqueta su frecuencia en la columna. Algunas tablas contienen un registro por etiqueta y entonces podemos usar gráficos de barras para representar esa información. Por ejemplo:

```
barplot(VADeaths[, 2], xlab = "tramos de edad", ylab = "tasa de mortalidad",
        main = "Tasa de mortalidad en Virginia\nmujer/rural")
```

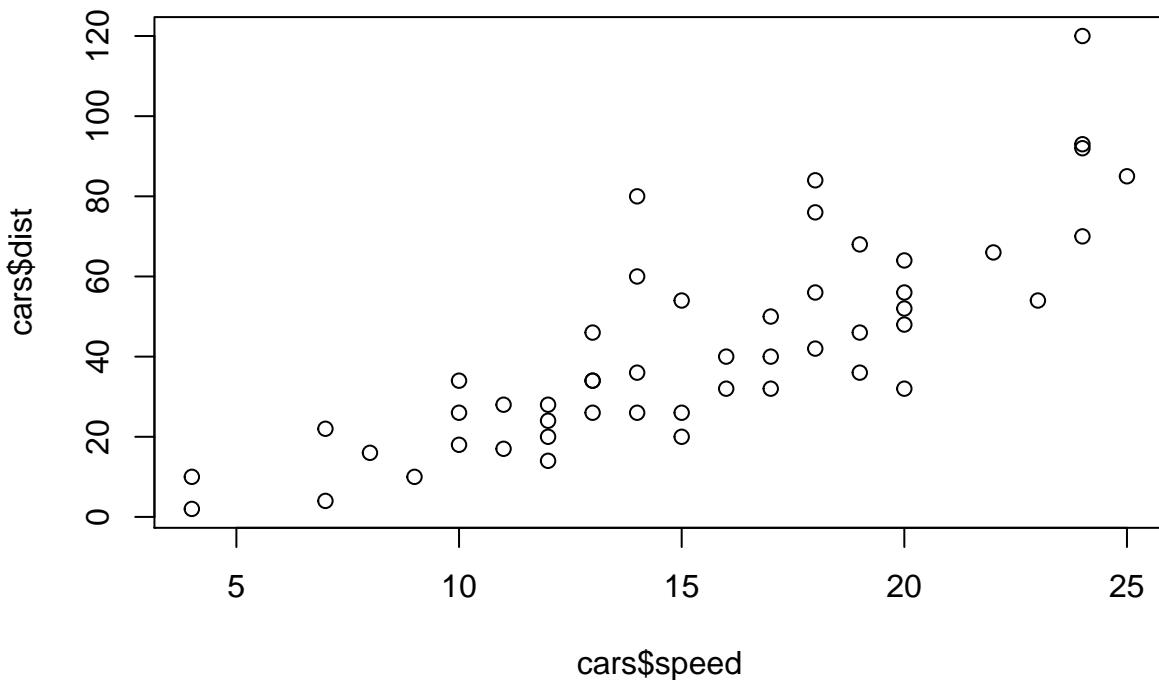
**Nota**

Los gráficos de barras son las representaciones más habituales para mostrar la distribución de vectores (entre ellos, las frecuencias de etiquetas). Sin embargo, existen alternativas modernas y superiores a ellos en algunos aspectos. Por ejemplo, los gráficos de puntos, implementados en R en la función `dotchart`.

2.6.3. Representación de la relación entre dos variables continuas: gráficos de dispersión

Los aspectos más interesantes de los datos se revelan no examinando las variables independientemente sino en relación con otras. Los gráficos de dispersión muestran la relación entre dos variables numéricas. En el ejemplo siguiente serán la velocidad y la distancia de frenado de un conjunto de coches recogidas en el conjunto de datos `cars`:

```
plot(cars$speed, cars$dist)
```



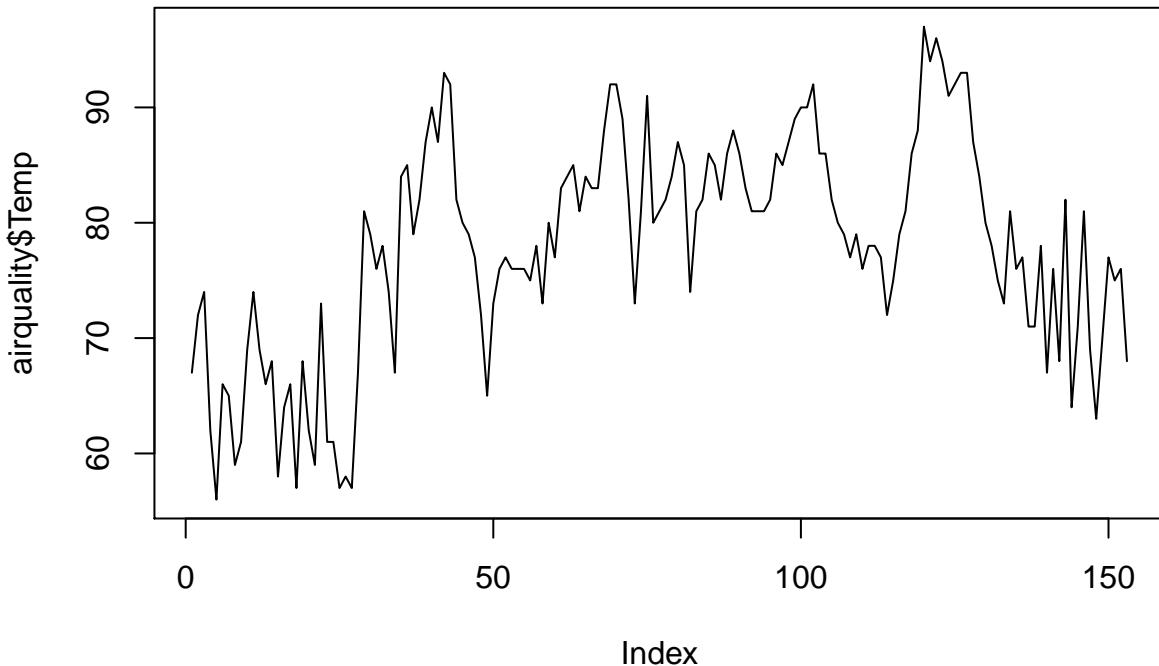
El gráfico muestra cómo aumenta dist en función de speed.

Ejercicio 2.6.8 Representa gráficamente la anchura del sépalo contra su longitud (usando `iris`). Interpreta el gráfico.

Ejercicio 2.6.9 De nuevo, usa los parámetros `main`, `xlab`, `ylab` y `col` discutidos en la sección anterior para mejorar el aspecto de los gráficos anteriores.

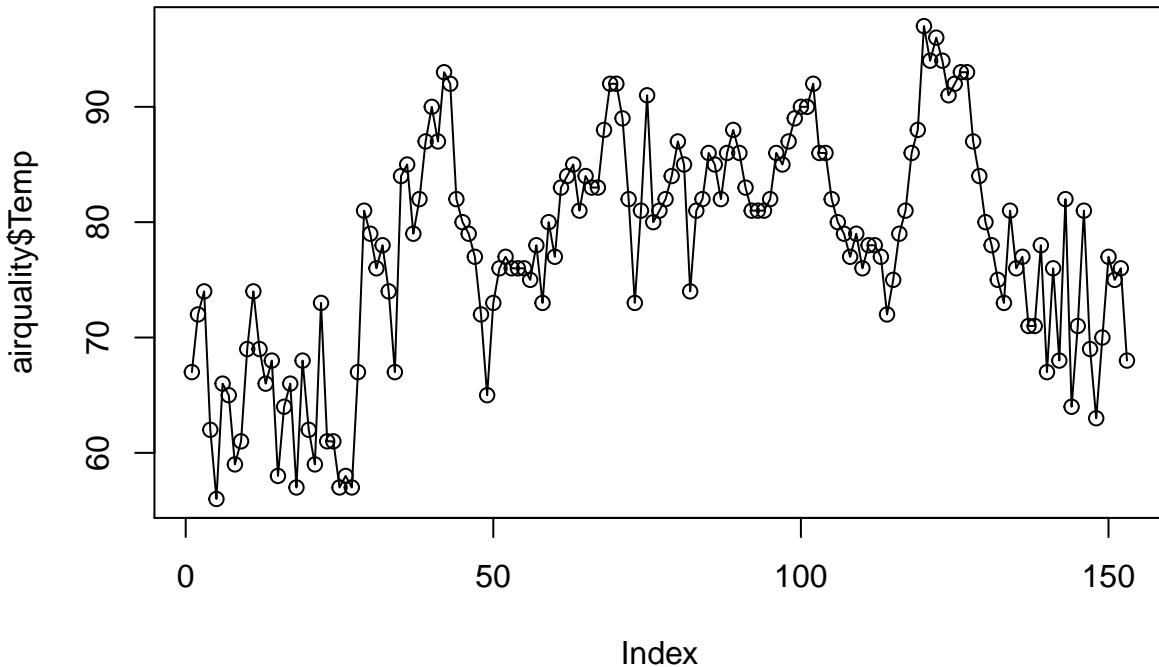
En ocasiones, cuando una de las variables tiene un orden determinado (por ejemplo, es una variable temporal) pueden utilizarse líneas para unir los puntos de un diagrama de dispersión (o, más habitualmente, reemplazarlos por ellas). Por ejemplo, utilizando el hecho de que las observaciones de `airquality` están ordenadas temporalmente, podemos representar la temperatura en periodo que comprende así:

```
plot(airquality$Temp, type = "l")
```



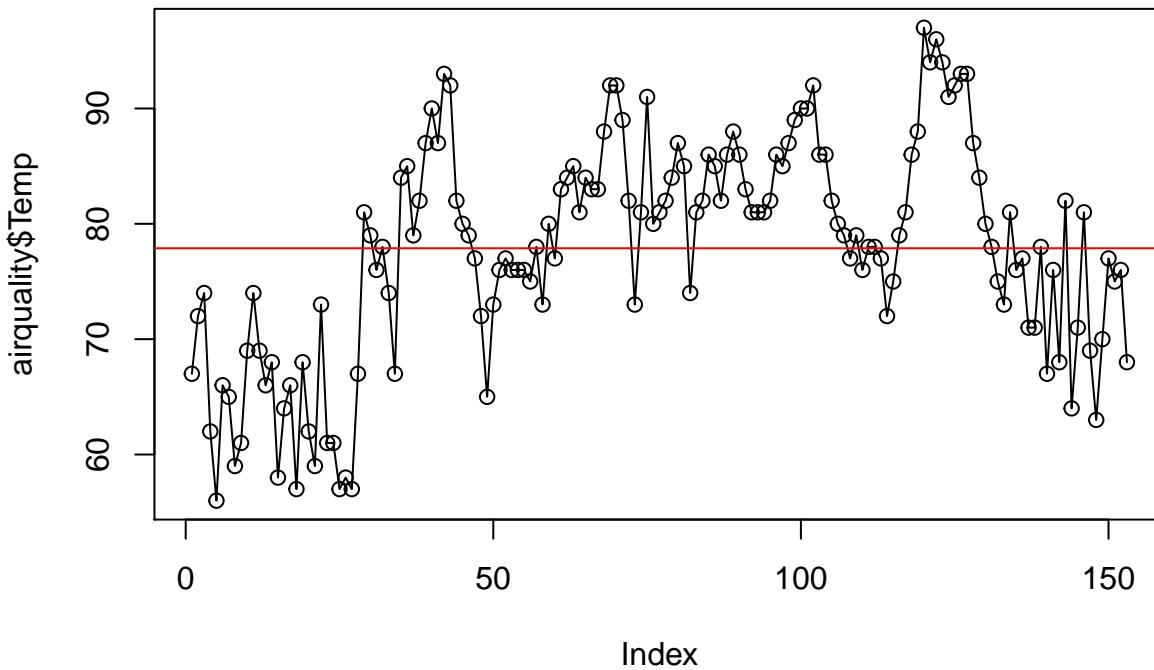
Incluso, se pueden combinar varios elementos gráficos sobre la misma representación gráfica: por ejemplo, combinar puntos y líneas como aquí:

```
plot(airquality$Temp)
lines(airquality$Temp)
```



El anterior es un ejemplo de una característica de los gráficos básicos de R: a un primer gráfico se le pueden añadir progresivamente capas adicionales. En el caso anterior, a un gráfico de puntos se le han añadido líneas. Pero podrían añadirse más elementos. Por ejemplo, al gráfico anterior se le puede añadir un elemento más, una línea horizontal roja a la altura de la temperatura media, usando la función (muy útil) `abline`:

```
plot(airquality$Temp)
lines(airquality$Temp)
abline(h = mean(airquality$Temp), col = "red")
```



Ejercicio 2.6.10 Consulta la ayuda de la función `abline` y úsala para añadir líneas (no solo horizontales) a alguno de los gráficos anteriores.

Ejercicio 2.6.11 Consulta `?par`, una página de ayuda en R que muestra gran cantidad de parámetros modificables en un gráfico. Investiga y usa `col`, `lty` y `lwd`. Nota: casi nadie conoce estos parámetros y, menos, de memoria; pero está bien saber que existen por si un día procede utilizarlos.

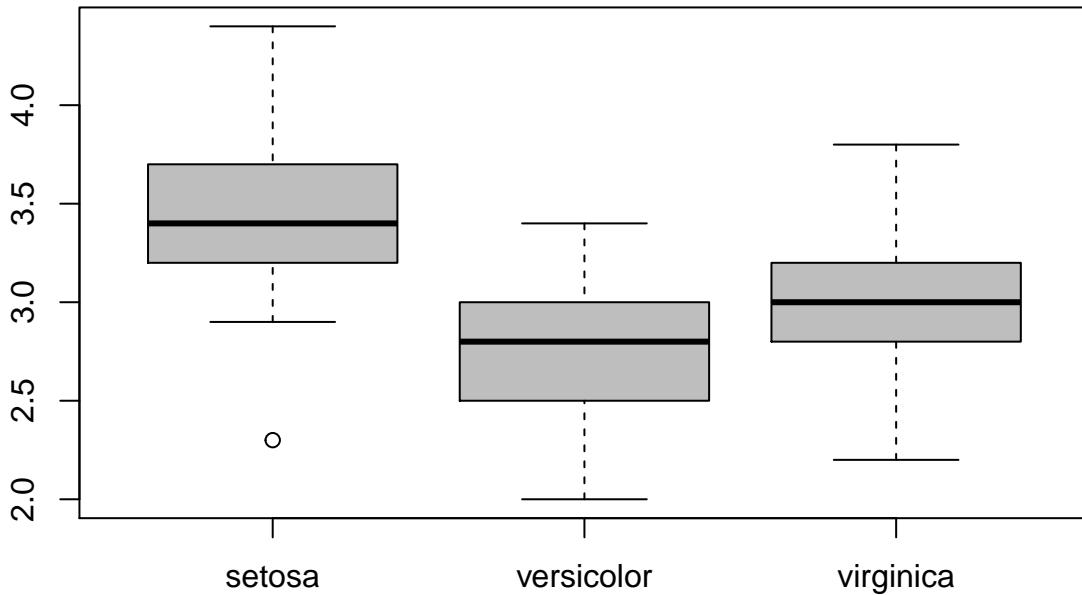
2.6.4. Representación de la relación entre una variable continua y otra categórica: diagramas de caja (boxplots)

Los diagramas de cajas (*boxplot*) estudian la distribución de una variable continua en función de una variable categórica. Están emparentados con los histogramas porque resumen la distribución de una variable continua. Para ello utilizan una representación todavía más esquemática que la de un histograma: una caja y unos segmentos que acotan las regiones donde la variable continua concentra el grueso de las observaciones.

Por ejemplo, podemos estudiar la distribución de la anchura del sépalos en `iris` en función de la especie usando diagramas de cajas así:

```
boxplot(iris$Sepal.Width ~ iris$Species, col = "gray",
       main = "Especies de iris\nsegún la anchura del sépalo")
```

Especies de iris según la anchura del sépalo



La notación $y \sim x$ es muy común en R y significa que vas a hacer *algo* con y en función de x ; en este caso, *algo* es un diagrama de cajas. Cuando construyamos modelos, querremos entender la variable objetivo y en función de una o más variables predictoras x y volveremos a hacer uso de esa notación¹².

El gráfico anterior ilustra la potencia de los diagramas de caja. Para las setosas, existe una observación atípica, mucho menor que el resto, pero cuya atipicidad queda oculta por otras observaciones *normales* correspondientes a virgínicas o versicolores. Esa observación atípica no llamaría la atención si se representan gráficamente todos los valores, independientemente de su tipo, pero se manifiesta claramente al segmentar la representación por especie.

Ejercicio 2.6.12 Identifica la observación atípica. ¿Es atípica también con respecto a otras variables?

Ejercicio 2.6.13 Muestra la distribución de las temperaturas en Nueva York en función del mes.

2.7. Resumen y referencias

Las tablas son las estructuras más habituales de la ciencia de datos. En R, a diferencia de otros lenguajes de programación, como Python o Java, son objetos nativos. En esta sección nos hemos familiarizado con los rudimentos del manejo de tablas en R: inspección de sus contenidos, ordenación, creación y borrado de columnas adicionales, etc. Esencialmente, aquellas que la mayoría hemos hecho ya previamente con herramientas tales como Excel.

Las tablas son fundamentales en R y en todas las secciones siguientes. Es muy importante adquirir una mínima soltura en su manejo. Gran parte de las secciones posteriores, de hecho, tratarán esencialmente sobre técnicas para procesar datos en tablas con extiendo las planteadas en esta sección.

¹²En R existe un tipo de datos muy especial: **formula**; sirve para especificar relaciones entre variables y aunque fue creado para especificar modelos estadísticos, se utiliza frecuentemente en otros contextos.

2.7.1. Referencias

- Python tiene tablas similares a las de R gracias a la librería Pandas. Puedes echarle un vistazo al tutorial *10 minutes to Pandas* para comparar las implementaciones. Debería resultarte familiar.
- Existen muchos tutoriales que profundizan en los detalles de los gráficos básicos de R. Por ejemplo, este.

2.8. Ejercicios adicionales

Ejercicio 2.8.1 Estudia el conjunto de datos `airquality` (información meteorológica de ciertos meses de cierto año en Nueva York, que también viene *de serie* en R) aplicando las funciones anteriores. En particular, responde las preguntas

- ¿cuál es la temperatura media de esos días?
- ¿cuál es la temperatura media en mayo?
- ¿cuál fue el día más ventoso?

Ejercicio 2.8.2 Crea una tabla adicional seleccionando todas las columnas menos mes y día; luego haz un plot de ella y trata de encontrar relaciones (cualitativas) entre la temperatura y el viento, o el ozono,...

Ejercicio 2.8.3 Usando el conjunto de datos `mtcars` (consulta `?mtcars`), averigua:

- cuál es el modelo que menos consume
- cuál es el consumo medio de los modelos de 4 cilindros

Ejercicio 2.8.4 Crea un histograma de la temperatura en Nueva York (usando `airquality`) y después usa `abline` para dibujar una línea vertical roja en la media de la distribución. Puedes obtener la media con `summary` o bien aplicando la función `mean` a la columna de interés.

Ejercicio 2.8.5 Carga el fichero `pisasci2006.csv` que contiene los promedios de los resultados en ciencia notas por país en las pruebas PISA de 2006. Con él:

- identifica y muestra la línea correspondiente a España
- haz una gráfica en que se muestre la nota en función de los ingresos
- identifica el *outlier* en la gráfica anterior: ¿a qué país corresponde?
- construye un histograma con la nota y añade una recta vertical roja en la correspondiente a España

Capítulo 3

Vectores

Los vectores en R son secuencias de objetos del mismo tipo (p.e., números). Los encontraremos, sobre todo, como columnas de una tabla; que son, efectivamente, vectores (p.e., `iris$Petal.Length`). En esta sección veremos cómo crear, inspeccionar y operar con vectores.

3.1. Creación de vectores

El código siguiente crea dos vectores, uno numérico y otro categórico (del tipo `factor`, que veremos más adelante).

```
x <- 1:10  
y <- iris$Species
```

Una vez creados estos vectores, aparecerán en el panel correspondiente de RStudio (o si ejecutas `ls()`).

Ejercicio 3.1.1 Crea el vector que numera las filas de `iris` (es decir, que contenga los números del 1 hasta el número de filas de `iris`). Nota: este es un ejercicio muy importante sobre el que abundaremos más adelante.

Previamente hemos usado el operador `:` para crear secuencias de números enteros. Para construir vectores arbitrarios, podemos usar la función de concatenación, `c`:

```
1:5  
5:1  
c(1:5, 5:1)  
c(1, 5, -1, 4)  
c("uno", "dos", "tres")
```

Nota

Recuerda: al no ser asignados a ninguna variable, los objetos anteriores, simplemente, se muestran en pantalla.

En realidad, `:` es una abreviatura de `seq`:

```
seq(1, 4)
```

```
## [1] 1 2 3 4
```

Esta función está estrechamente emparentada con `rep`:

```
rep(1:4, 4)

## [1] 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4
rep(1:4, each = 4)

## [1] 1 1 1 1 2 2 2 2 3 3 3 3 4 4 4 4
```

Ejercicio 3.1.2 Crea el patrón 1, 1.1, 1.2,..., 2. Existen varias maneras de hacerlo. Una de ellas es utilizar el argumento `by` de `seq`.

Los siguientes ejemplos ilustran cómo crear patrones más complejos usando `rep`. No es necesario dominar todas las opciones disponibles en la función `rep`, pero sí saber que, llegado el caso, la función dispone de ellas.

```
rep(1:4, 2)
rep(1:4, each = 2)
rep(1:4, c(2,2,2,2))
rep(1:4, times = 4:1)
rep(1:4, c(2,1,2,1))
rep(1:4, each = 2, len = 4)
rep(1:4, each = 2, len = 10)
rep(1:4, each = 2, times = 3)
```

Ejercicio 3.1.3 Selecciona las columnas 1, 2 y 5 de `iris`.

Ejercicio 3.1.4 Selecciona las filas 1:4 y 100:104 de `iris`.

Ejercicio 3.1.5 Usa un vector de texto para seleccionar las columnas `Wind` y `Temp` de `airquality`.

Ejercicio 3.1.6 Crea una tabla que sea la primera fila de `iris` repetida 100 veces.

3.2. Inspección de vectores

Las siguientes funciones, algunas de ellas ya conocidas, sirven para inspeccionar el contenido de un vector:

```
plot(x)
length(x)
table(y)          # ¡muy importante!
summary(y)
head(x)
tail(x)
```

La función `table` cuenta los elementos de un vector por valor. Es una función sumamente importante para examinar vectores categóricos¹:

```
table(iris$Species)
```

¹Por eso la usamos en la sección anterior al introducir los diagramas de barras.

`summary` ofrece un pequeño resumen del contenido de un vector. De hecho, cuando aplicábamos `summary` a una tabla, obteníamos el `summary` de cada una de sus columnas².

Ejercicio 3.2.1 En `C02`, cuenta cuántas filas corresponden a cada tipo de planta.

Ejercicio 3.2.2 Ejecuta e interpreta `table(table(C02$conc))`. Nota: `table(table(x))` es una operación muy frecuente y útil.

3.3. Selecciones

Para seleccionar elementos de un vector se usa el corchete `[]`. Pero, a diferencia de lo que ocurre con las tablas, como los vectores son objetos unidimensionales, sin coma. Obviamente, el corchete sigue admitiendo no solo los índices de los elementos que se quieren extraer, sino que además permite utilizar condiciones lógicas, etc.

```
x <- x^2
x[1:3]
x[c(1,3)]
x[x > 25]
x[3:1]
x[-(1:2)]
x[-length(x)]
```

Advierte en el ejemplo anterior el uso (y el efecto) del signo menos y de las condiciones lógicas dentro de los corchetes.

Ejercicio 3.3.1 Selecciona todos los elementos de un vector menos los dos últimos.

Ejercicio 3.3.2 Implementa la función `diff` (las diferencias entre cada valor de un vector y el que lo precede) a mano. Nota: la función `diff` existe en R; pruébala en caso de duda.

El corchete también permite seleccionar elementos de un vector por nombre. En efecto, en R se pueden asociar nombres a los elementos de un vector. Lo hace automáticamente, por ejemplo, `table` (para poder saber a qué etiqueta corresponde cada conteo):

```
z <- table(iris$Species)
z["setosa"]

## setosa
##      50
z[c("setosa", "virginica")]

##
##      setosa virginica
##      50          50
```

— Nota —

En cierto sentido, los vectores con nombre operan (aunque con algunas diferencias técnicas) como los diccionarios de Python o las tablas *hash* de otros lenguajes de programación: contienen parejas clave-valor.

²Esto es muy habitual en R: existen funciones, como `summary` que modifican su comportamiento según su argumento. `summary` siempre genera un resumen del objeto que se le pase, sea este de la naturaleza que sea; algo parecido sucede con `plot`, `str` y otras funciones que reciben por ello el nombre de *genéricas*.

Una propiedad muy importante del corchete es que permite, además de seleccionar, cambiar el contenido de los elementos seleccionados de un vector. Por ejemplo:

```
z <- 1:10
z[z < 5] <- 100
z
## [1] 100 100 100 100 5 6 7 8 9 10
```

Reemplazar subselecciones es muy útil: permitirá, por ejemplo, cambiar las edades negativas por un valor con sentido, sustituir los nulos por un determinado valor por defecto, reemplazar los valores que excedan un tope por dicho tope, etc.

El nombre de las columnas de una tabla también es un vector. Por eso, para cambiar el nombre de una columna podemos hacer lo siguiente:

```
mi.iris <- iris # una copia de iris
colnames(mi.iris)[5] <- "Especie"
```

Lo mismo ocurre con los nombres de un vector, aunque en este caso, la función correspondiente es `names`:

```
z <- table(iris$Species)
names(z)
names(z)[1] <- "A"
names(z)
```

Ejercicio 3.3.3 Cambia (en una sola expresión) los nombres de las dos primeras columnas de `mi.iris` por su traducción al español.

Frecuentemente se quiere muestrear un vector, es decir, obtener una serie de elementos al azar dentro de dicho vector. Para ello se utiliza la función `sample`:

```
sample(x, 4)
sample(x, 100)          # ¡falla!
sample(x, 100, replace = TRUE) # manera correcta
```

La función `sample` trata el vector como una urna y a sus elementos como bolas contenidas en ella que va extrayendo al azar. Obviamente, es incapaz de extraer más elementos de los que contiene la urna. Pero existe el la opción de que el muestreo se realice *con reemplazamiento*, i.e., de modo que cada vez que `sample` extraiga una bola, la reintroduzca en la urna.

Ejercicio 3.3.4 Muestrea `iris`, es decir, extrae (p.e., 30) filas al azar de dicha tabla. Pista: recuerda que *ordenar* era *seleccionar ordenadamente*; de igual manera, en una tabla, muestrear será...

El muestreo de vectores es fundamental en diversos ámbitos. Por ejemplo, a la hora de realizar los llamados tests A/B] y determinar qué observaciones van al grupo A y cuáles al B.

Ejercicio 3.3.5 Parte `iris` en dos partes iguales (75 observaciones cada uno) con las filas elegidas al azar (¡y complementarias!).

3.4. Ordenación

Existen tres funciones fundamentales relacionadas con la ordenación de vectores: `order`, `sort` y `rank`. `sort` ordena los elementos de un vector, i.e., crea una copia de dicho vector con sus elementos ordenados.

```
x <- c(4, 5, 3, 2, 1, 2)
sort(x)          # ordena los elementos del vector
```

```
## [1] 1 2 2 3 4 5
```

La función `order`, que ya vimos cuando ordenamos tablas, es tal que `sort(x)` es lo mismo que `x[order(x)]`. Es decir, devuelve los índices de los elementos del vector de menor a mayor:

```
x
```

```
## [1] 4 5 3 2 1 2
```

```
order(x)
```

```
## [1] 5 4 6 3 1 2
```

```
x[order(x)]
```

```
## [1] 1 2 2 3 4 5
```

El código anterior muestra cómo para ordenar `x` hay que tomar primero el elemento quinto elemento de `x`; luego, el cuarto; después, el sexto, etc.

La función `rank` indica la posición de los elementos de un vector: el primero, el segundo, etc.

```
rank(x)
rank(x, ties = "first")
```

Si `x` contuviese los tiempos de los velocistas en los 100 metros lisos, `rank(x)` nos indicaría quién es el primero en llegar a la meta, quién es el segundo, etc.

Nota

`rank(x)` es una transformación no lineal de `x` que puede ser útil para normalizar datos en algunos contextos.

Ejercicio 3.4.1 Si `x` contuviese el número de puntos obtenidos en la liga de fútbol por los distintos equipos, ¿cómo usarías `rank` para determinar cuál es el campeón?

Ejercicio 3.4.2 ¿Qué otros tipos de `ties` existen? ¿Qué hacen?

Ejercicio 3.4.3 Comprueba que `rank(x, ties = 'first')` es equivalente a `order(order(x))`.

Ejercicio 3.4.4 Comprueba que `order(order(order(x)))` es equivalente a `order(x)`.

Ejercicio 3.4.5 Ejecuta e interpreta `tail(sort(table(CO2$uptake)))`. ¿Qué utilidad le ves a la expresión anterior?

3.5. Operaciones matemáticas y vectorización

R puede ser usado como una calculadora:

```
2+2
```

```
## [1] 4
```

```
x <- 4*(3+5)^2
x / 10
```

```
## [1] 25.6
```

En R se puede operar sobre vectores igual que se opera sobre números. De hecho, en R, un número es un vector numérico de longitud 1:

```
c(length(2), length(x))
```

```
## [1] 1 1
```

Así que se pueden hacer cosas tales como:

```
x <- 1:10
2*x
2*x + 1
x^2
x*x
```

Operaciones como las anteriores están *vectorizadas*, i.e., admiten un vector como argumento y operan sobre cada uno de los elementos. Generalmente, las operaciones vectorizadas son muy rápidas en R. Muchos problemas de rendimiento en R se resuelven, de hecho, utilizando versiones vectorizadas del código ineficiente³.

Ejercicio 3.5.1 Comprueba que `log` es una función vectorizada aplicándosela a `x`.

Ejercicio 3.5.2 Calcula el valor medio de la longitud de los pétalos de `iris` usando `mean`.

Ejercicio 3.5.3 Repite el ejercicio anterior usando `sum` y `length`.

Ejercicio 3.5.4 Suma un millón de términos de la fórmula de Leibniz] para aproximar π . Pista: crea primero un vector con los índices (del 0 al 1000000) y transfórmalo adecuadamente.

Ejercicio 3.5.5 Si `x <- 1:10` e `y <- 1:2`, ¿cuánto vale `x * y`? ¿Qué pasa si `y <- 1:3`?

La operación a la que se refiere el ejercicio anterior se denomina *reciclado de vectores*. Cuando se opera con dos vectores (por ejemplo, para multiplicarlos) con longitudes distintas, el más corto se *recicla* (es decir, se repite) tantas veces como sea necesario hasta alcanzar la longitud del más largo. Si la longitud del más corto no divide exactamente la del más largo (p.e., uno tiene longitud 10 y otro, longitud 3), R recorta la parte reciclada sobrante y lanza un *warning*.

El lector está invitado a consultar cómo construir con R una calculadora de hipotecas, que ilustra el uso de la vectorización en un problema menos trivial que los anteriores, en el capítulo dedicado a ejemplos de uso.

3.6. Digresión: creación de funciones

Esta sección, por su importancia, altera el flujo del libro. Lo que contiene pertenece propiamente a la sección de programación. Pero, dada la importancia de la creación de funciones en lo que sigue (y, en particular, para la sección siguiente), presenta lo que será estrictamente necesario para los capítulos intermedios.

En R hay miles de funciones que pueden aplicarse a un vector. Unas cuantas de las más comunes que pueden aplicarse a vectores numéricos son:

³De hecho, uno de los errores más frecuentes de los novatos en R que tienen experiencia previa en lenguajes de programación tales como Java, C o Matlab es utilizar código no vectorizado: bucles `for` y similares. En R se prefiere casi siempre recurrir a la vectorización.

```
fivenum(x)    # los "cinco números característicos" de un vector
mean(x)
max(x)
median(x)
sum(x)
prod(x)
```

Sin embargo, por muchas funciones que tenga R para operar sobre vectores, seguramente falta aquella que resolvería un problema dado: por ejemplo, no existe una función de serie en R que sume los cuadrados de unos números. De necesitarla, podríamos crearla así:

```
x <- 1:10
suma_cuadrados <- function(x) sum(x*x)
suma_cuadrados(x)

## [1] 385
```

El código anterior crea una nueva función, `suma_cuadrados`, que suma los cuadrados de un vector. La nueva función es un objeto más de R⁴, que aparece en la pestaña **Environment** de RStudio y en los listados de `ls()`. Queda en memoria para ser utilizada posteriormente.

Ejercicio 3.6.1 Si no existiese `prod` se podría programar: usa `log`, `exp` y `sum` para crear una función, `mi.prod` que multiplique los elementos de un vector (de números, por el momento, > 0).

Ejercicio 3.6.2 Crea una función que cuente el número de valores negativos (< 0) del vector que se le pase como argumento.

Por supuesto, a menudo queremos crear funciones que no se resuelvan en una única línea, como la de arriba. Cuando el *cuerpo* de la función sea más complejo, hay que encerrarlo en llaves, {}.

```
media <- function(x){
  longitud <- length(x)
  suma <- sum(x)
  suma / longitud
}
```

Nota

En R, a diferencia de otros lenguajes como Python, no es necesario el `return`. Una función devuelve siempre el último valor definido en su cuerpo. No obstante, como veremos después, `return` existe y es muy útil en algunos contextos.

3.7. La función tapply

La función `tapply` *aplica* (de ahí parte de su nombre) una función a un vector en los subvectores que define otro *vector máscara*:

```
tapply(iris$Petal.Length, iris$Species, mean)
```

```
##      setosa versicolor  virginica
##      1.462     4.260     5.552
```

⁴Esta es una característica de R que lo distingue de algunos otros lenguajes de programación que distinguen estrictamente entre lo que son datos y lo que son funciones; en R, sin embargo, una función es un objeto más.

En el ejemplo anterior, `tapply` aplica la función `mean` a un vector, la longitud del pétalo, pero de especie en especie. Responde por tanto a la pregunta de cuál es la media de la longitud del pétalo por especie. Es una operación similar a la conocida en SQL como `group by`.

Ejercicio 3.7.1 Calcula el valor medio de la temperatura en cada mes de Nueva York (usando `airquality`).

`tapply` es una de las llamadas funciones *de orden superior* porque acepta como argumento otra función (`mean` en el caso anterior). Las funciones a las que llaman tanto `tapply` como el resto de las funciones de orden superior pueden tener parámetros adicionales. Por ejemplo, si el vector sobre el que se quiere calcular la media contiene `NA`, el resultado es `NA`. Pero `mean` admite el parámetro opcional `na.rm = TRUE` para ignorar los nulos. Para aplicar no la función `mean` sino `mean` con la opción `na.rm = TRUE` hay que cambiar la llamada a `tapply` así:

```
tapply(iris$Petal.Length, iris$Species, mean, na.rm = TRUE)
```

El resultado es el mismo (pero la sintaxis es mucho más limpia) que haciendo:

```
foo <- function(x) mean(x, na.rm = TRUE)
tapply(iris$Petal.Length, iris$Species, foo)
```

La particularidad de `tapply` y otras funciones de orden superior similares de R es que aquellas opciones adicionales que se incluyen en la llamada a `tapply` después de la función que aplican pasan a esta última.

Ejercicio 3.7.2 Calcula el valor medio del ozono en cada mes de Nueva York (usando `airquality`).

Ejercicio 3.7.3 Usando `mtcars`, calcula la potencia mediana según el número de cilindros del vehículo.

3.8. Resumen y referencias

En esta sección hemos repasado las operaciones más habituales que involucran vectores: cómo crearlos, inspeccionarlos, transformarlos, etc. Hemos prestado atención también a un tipo muy concreto y característico de R de vectores: los factores. Finalmente hemos introducido algunos conceptos avanzados y muy característicos de R para la manipulación de datos en vectores, como la vectorización y la función `tapply`.

Lo que no hemos hecho en absoluto es mencionar los bucles. En muchos otros lenguajes de programación, hablar de vectores es hablar de bucles. En R, aunque también pueden usarse bucles, son innecesarios para la casi totalidad de los fines: con las operaciones vectorizadas, basta.

3.9. Ejercicios adicionales

Ejercicio 3.9.1 Ejecuta `mean(sample(iris$Sepal.Length, replace = T))` varias veces. Comprueba que obtienes números que se parecen a la media la columna `Sepal.Length`. Nota: esto es el fundamento de una técnica estadística muy poderosa, el *bootstrap*, para estimar cómo puede variar una media (i.e., estimar la varianza de una media).

Ejercicio 3.9.2 El vector `letters` contiene las letras `a, b, ...` Hasta 26 en total. Crea otro vector que tenga la letra `a` repetida 26 veces; la `b`, 25, etc.

Ejercicio 3.9.3 Toma el vector que has creado en el ejercicio anterior y crea otro que cuente las veces que aparecen las cinco letras más frecuentes y que agrupe el resto en `otros`. Nota: este ejercicio se realiza

innumerables veces. Por ejemplo, cuando existe un vector con tantas categorías que representarlas todas es imposible; una solución consiste en agrupar las menos frecuentes en una sola, la del resto.

Ejercicio 3.9.4 En una provincia la población activa es de un millón de personas. El 10 % de ellas está en el paro. Periódicamente el INE hace una encuesta sobre 1000 personas para estimar la tasa de paro. Pero esta encuesta, por estar basada en 1000 personas, está sujeta a error. Puedes tratar de medir ese error de la siguiente manera: crea un vector de longitud 1M con cien mil valores iguales a 1 y el resto, a 0. Extrae una muestra de tamaño 1000 y calcula la proporción de unos. ¿Está cerca del 10 %?

Ejercicio 3.9.5 Repite el ejercicio anterior varias veces. ¿Cómo varían las estimaciones? ¿Qué pasa si encuestas a 10000 personas en lugar de a 1000? ¿Y si encuestas a 100?

Ejercicio 3.9.6 Lee la parte relevante de `?replicate`. ¿Para qué sirve esta función? ¿Puede ser útil para analizar el caso propuesto en los ejemplos anteriores? Nota: la página de ayuda de la función anterior documenta varias funciones relacionadas, pero puedes ignorar por el momento todo lo que no se refiera a la función en cuestión.

Capítulo 4

Introducción a la estadística con R

Esta sección muestra cómo aplicar una serie de técnicas estadísticas habituales con R como ejercicio de lo aprendido en los capítulos anteriores y como motivación para introducir otros conceptos más adelante. Probablemente, ayudará a satisfacer la curiosidad de quienes ya las conozcan y quieran ver cómo utilizarlas en R. En todo caso, esta sección renuncia absolutamente a profundizar en sus aspectos matemáticos.

4.1. El test de Student

El test de Student es la contraparte estadística al diagrama de cajas. Si estos permiten comparar gráficamente, i.e. cualitativamente, dos (o más) distribuciones, el test de Student permite cuantificar de alguna manera esa diferencia.

Para ilustrar el uso del test de Student vamos a analizar el conjunto de datos `sleep`:

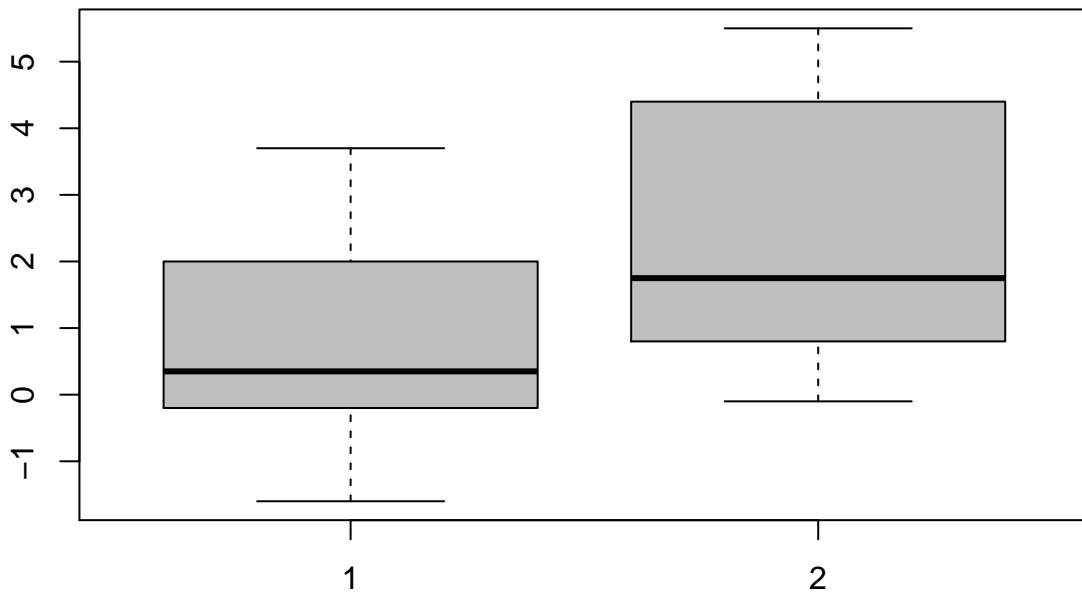
```
summary(sleep)
```

```
##      extra      group       ID
##  Min.   :-1.600  1:10    1   :2
##  1st Qu.:-0.025  2:10    2   :2
##  Median : 0.950      3    :2
##  Mean   : 1.540      4    :2
##  3rd Qu.: 3.400      5    :2
##  Max.   : 5.500      6    :2
##                  (Other):8
```

`sleep` contiene información sobre el número adicional de horas que durmieron una serie de sujetos después de recibir un determinado tratamiento médico. Los vamos a comparar con otros sujetos, los del grupo de control, que recibieron un placebo. El siguiente diagrama de cajas muestra la distribución del número de horas de sueño con y sin tratamiento.

```
boxplot(sleep$extra ~ sleep$group, col = "gray",
        main = "Diferencias por grupo")
```

Diferencias por grupo



Este gráfico muestra cómo el grupo tratado, el de la derecha, tiende a dormir más horas. Pero no responde fehacientemente a la pregunta de si el tratamiento funciona o no. El llamado test de Student ayuda a determinar si la diferencia observada es o no *estadísticamente significativa*. Nótese cómo su sintaxis es esencialmente la misma que la de boxplot:

```
t.test(sleep$extra ~ sleep$group)

##
##  Welch Two Sample t-test
##
## data: sleep$extra by sleep$group
## t = -1.8608, df = 17.776, p-value = 0.07939
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
## -3.3654832  0.2054832
## sample estimates:
## mean in group 1 mean in group 2
##           0.75          2.33
```

La salida de la expresión anterior es la que normalmente reproducen los artículos científicos cuando se aplica esta técnica. El indicador más importante (aunque cada vez más cuestionado) es el llamado p-valor y una regla de decisión habitual es dar por *estadísticamente significativo* el efecto del tratamiento cuando este es inferior a 0.05. En este caso, el p-valor es de 0.07 por lo que se consideraría que no existe diferencia significativa entre los grupos.

Ejercicio 4.1.1 El test de Student no se puede aplicar a cualquier tipo de datos. Estos tienen que cumplir una serie de condiciones. Cuando las pruebas previas parecen indicar que no dichas condiciones no se cumplen, los manuales recomiendan utilizar una alternativa no paramétrica, la del Wilcoxon. Aplícasela a estos datos comprueba el p-valor obtenido. Nota: en R, es el `wilcox.test`.

Existen muchas más pruebas estadísticas en R. Que la sintaxis de las dos anteriores (la de `t.test` y de `wilcox.test`) sea similar no es casualidad: la de todas las funciones que aplican pruebas estadísticas en

R es homogénea y predecible.

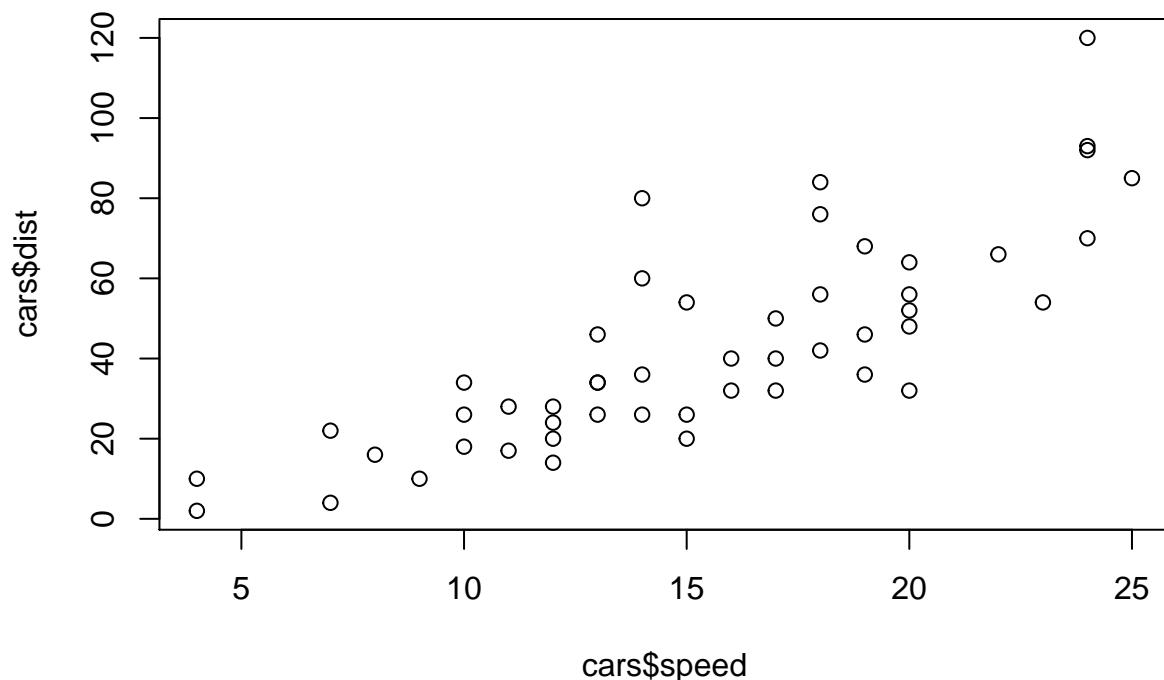
Ejercicio 4.1.2 Estudia los ejemplos de `prop.test`, que implementa una prueba estadística para comprobar si dos proporciones son o no *significativamente distintas*.

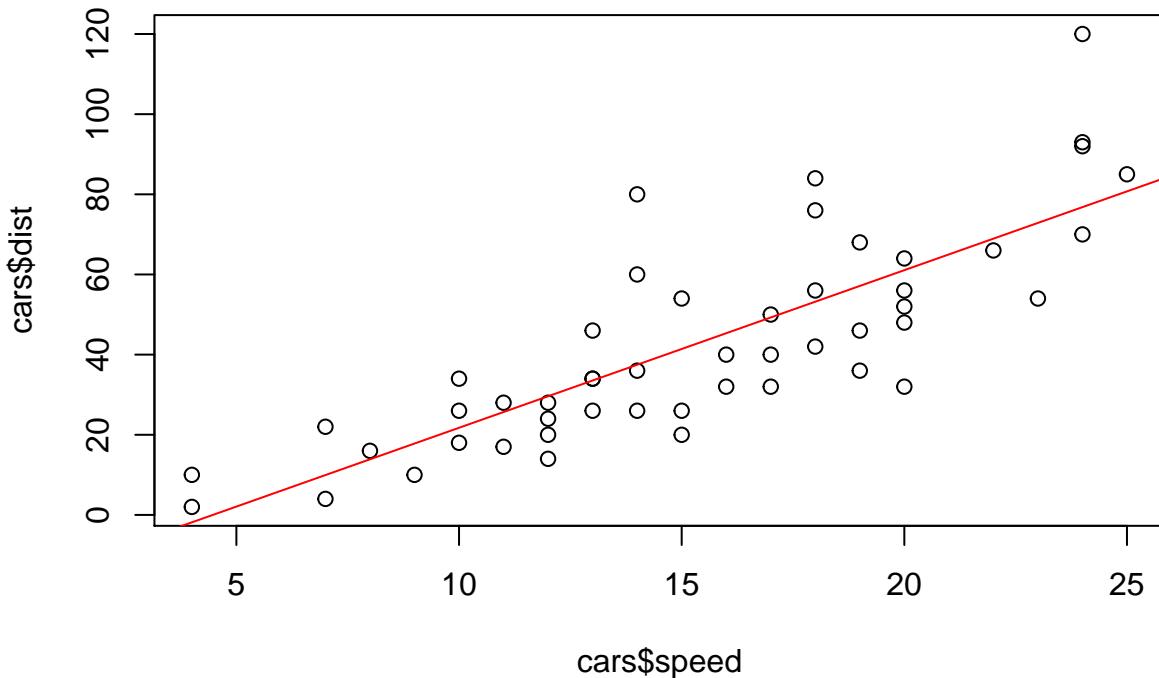
4.2. Regresión lineal

La regresión lineal expresa la relación entre una variable numérica con otras variables predictoras. En el caso más simple, el que vamos a explorar a continuación, se considera una única variable predictora.

Existe una relación creciente entre las variables velocidad y distancia en el conjunto de datos `cars`. En efecto, a mayor velocidad, mayor es la distancia de frenado de los vehículos:

```
plot(cars$speed, cars$dist)
```





El código precedente usa la función `lm` para ajustar el modelo de regresión lineal (*linear model*), muestra un pequeño resumen del modelo en pantalla y finalmente añade la recta de regresión, en rojo, al gráfico anterior.

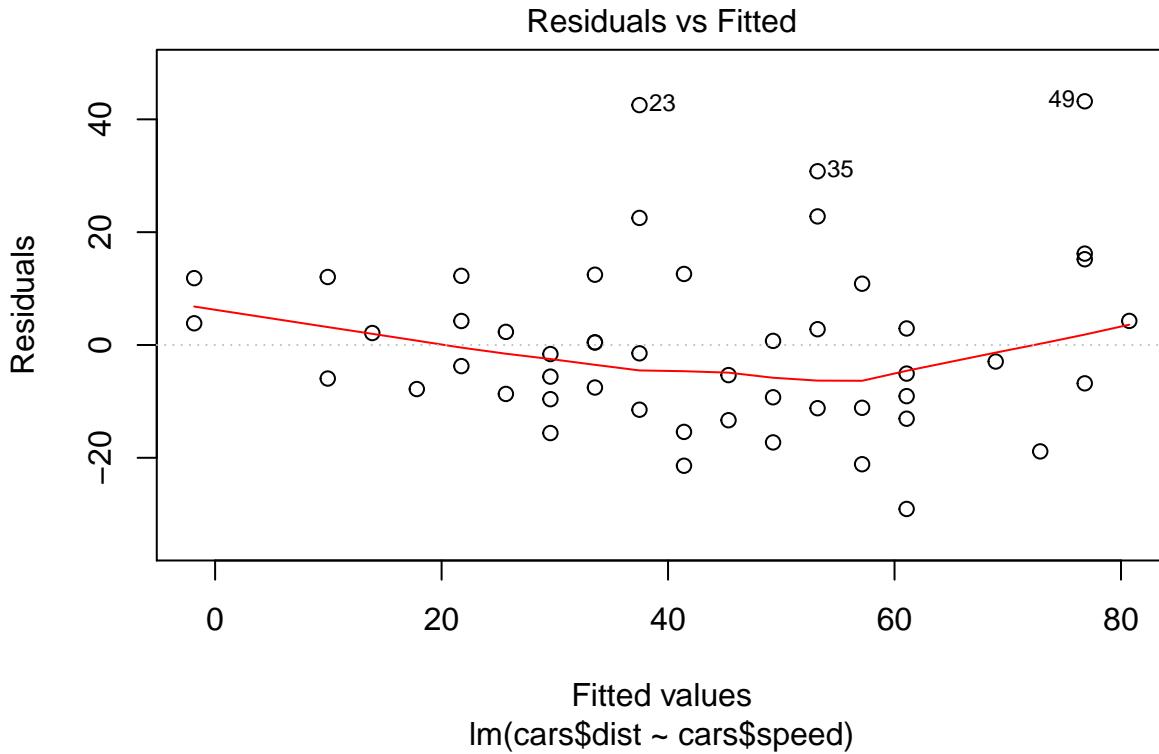
La función `summary`, aplicada en este caso no a un vector o a una tabla sino al objeto resultante de la regresión lineal, muestra un resumen del modelo similar al que aparece en las publicaciones científicas:

```
summary(lm.dist.speed)

##
## Call:
## lm(formula = cars$dist ~ cars$speed)
##
## Residuals:
##    Min     1Q   Median     3Q    Max 
## -29.069 -9.525 -2.272  9.215 43.201 
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept) -17.5791   6.7584  -2.601  0.0123 *  
## cars$speed    3.9324   0.4155   9.464 1.49e-12 *** 
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 15.38 on 48 degrees of freedom
## Multiple R-squared:  0.6511, Adjusted R-squared:  0.6438 
## F-statistic: 89.57 on 1 and 48 DF,  p-value: 1.49e-12
```

`summary` no es la única función que admite el modelo creado: `plot` crea una representación gráfica (de hecho, produce varias gráficas distintas de las que en el código que aparece a continuación representaremos solo la primera; para ver las restantes, cambia el índice 1 por otros valores entre 2 y 6) que ilustra algunos aspectos del modelo que los implementadores de la función `lm` encontraron útiles:

```
plot(lm.dist.speed, 1)
```



En general, muchos modelos creados en R admiten llamadas de las funciones `summary` y `plot`, que producen los resultados esperados.

Ejercicio 4.2.1 Haz una regresión del nivel de ozono sobre la temperatura en Nueva York. Crea el gráfico de dispersión y añádele la recta de regresión (en rojo u otro color distinto del negro) con `abline`.

4.3. Regresión logística

La regresión logística forma parte de los llamados modelos lineales generalizados y trata de estimar la probabilidad de ocurrencia de un evento binario (éxito/fracaso, cara/cruz) en función de una serie de variables predictoras.

En el ejemplo siguiente, usaremos el conjunto de datos `UCBAdmissions` y el evento binario es la admisión de un estudiante a un programa de doctorado en función otras variables.

```
datos <- as.data.frame(UCBAdmissions)
datos$Admit <- datos$Admit == "Admitted"
```

Este conjunto de datos se recogió para un estudio acerca de la discriminación contra las mujeres en este tipo de ámbitos publicado por *Science* en 1975. Por eso interesa conocer el efecto de la variable `Gender` en la probabilidad de resultar admitido.

Por lo tanto, en primer lugar, vamos a probar un modelo usando exclusivamente dicha variable:

```
modelo.sin.dept <- glm(Admit ~ Gender,
                        data = datos, weights = Freq,
                        family = binomial())
summary(modelo.sin.dept)

##
```

```

## Call:
## glm(formula = Admit ~ Gender, family = binomial(), data = datos,
##      weights = Freq)
##
## Deviance Residuals:
##      Min       1Q   Median       3Q      Max
## -20.336  -15.244    1.781   14.662   28.787
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) -0.22013   0.03879 -5.675 1.38e-08 ***
## GenderFemale -0.61035   0.06389 -9.553 < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
## Null deviance: 6044.3 on 23 degrees of freedom
## Residual deviance: 5950.9 on 22 degrees of freedom
## AIC: 5954.9
##
## Number of Fisher Scoring iterations: 4

```

Se aprecia en el resumen del modelo cómo el sexo es una variable predictora muy importante (el p-valor es ínfimo y el tamaño del coeficiente relativamente grande) y cómo da la impresión de que en el proceso de admisión existe discriminación contra las mujeres: el coeficiente es negativo.

El siguiente modelo incluye el departamento como variable predictora. En el resultado se aprecia cómo el efecto del sexo prácticamente desaparece.

```

modelo.con.dept <- glm(Admit ~ Gender + Dept,
                        data = datos, weights = Freq,
                        family = binomial())
summary(modelo.con.dept)

##
## Call:
## glm(formula = Admit ~ Gender + Dept, family = binomial(), data = datos,
##      weights = Freq)
##
## Deviance Residuals:
##      Min       1Q   Median       3Q      Max
## -25.3424  -13.0584   -0.1631   16.0167   21.3199
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)  0.58205   0.06899   8.436  <2e-16 ***
## GenderFemale 0.09987   0.08085   1.235    0.217
## DeptB        -0.04340   0.10984  -0.395    0.693
## DeptC        -1.26260   0.10663 -11.841  <2e-16 ***
## DeptD        -1.29461   0.10582 -12.234  <2e-16 ***
## DeptE        -1.73931   0.12611 -13.792  <2e-16 ***
## DeptF        -3.30648   0.16998 -19.452  <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
```

```
## (Dispersion parameter for binomial family taken to be 1)
##
## Null deviance: 6044.3 on 23 degrees of freedom
## Residual deviance: 5187.5 on 17 degrees of freedom
## AIC: 5201.5
##
## Number of Fisher Scoring iterations: 6
```

Lo que evidencia este segundo modelo es que no existía tal discriminación contra las mujeres. Lo que ocurría realmente es que los distintos departamentos tenían niveles de exigencia distintos y entre los candidatos a los más exigentes predominaban las mujeres. Las tasas de admitidos en cada departamento por sexo eran similares pero, globalmente, parecía haber un sesgo. Este es un ejemplo de libro de la llamada paradoja de Simpson.

De hecho, los autores del artículo mencionado más arriba lo resumieron así:

Measuring bias is harder than is usually assumed, and the evidence is sometimes contrary to expectation.

Ejercicio 4.3.1 El primer modelo, esencialmente, compara dos proporciones: el de hombres y mujeres admitidos. Utiliza la prueba de proporciones (`prop.test` en R) para compararlas de otra manera. ¿Coinciden los resultados?

4.4. Resumen y referencias

En esta sección hemos visto cómo aplicar algunos métodos estadísticos habituales en R. Son solo tres de los cientos disponibles en R. La buena noticia es que prácticamente todos tienen una interfaz similar y familiar en R. Las malas noticias son dos:

- Es arriesgado utilizar métodos estadísticos sin conocer bien sus fundamentos y tener claro qué hacen. Además, eso no se va a tratar en este libro.
- Antes de poder aplicar cualquier método estadístico existe un trabajo previo de selección, limpieza y tratamiento de datos que es tan poco vistoso como, a medio plazo, aburrido. Ese es, fundamentalmente, el objetivo del libro.

Existen muchas fuentes para ahondar en el estudio de la estadística y los métodos estadísticos en R. Algunas se ofrecen en la [sección de referencias del capítulo dedicado a la estadística][refs_statistics].

4.5. Ejercicios adicionales

TBA

Capítulo 5

Listas

Las tablas son contenedores de información estructurada: las columnas son del mismo tipo, todas tienen la misma longitud, etc. Gran parte de los datos con los que se trabaja habitualmente son *estructurados*, palabra que, en la jerga, significa que admiten una representación tabular.

Sin embargo, cada vez es más habitual trabajar directamente con información desestructurada. Particularmente, en ciencia de datos. Eso justifica el uso de las listas, que pueden definirse como contenedores genéricos de información desestructurada.

SQL, SAS, Matlab y otros entornos y lenguajes de programación usados en el análisis de datos utilizan fundamentalmente estructuras de datos pensados para información estructurada: tablas de distintos tipos, matrices, etc. Por eso, solo pueden utilizarse para procesar información desestructurada utilizando artificios *ad hoc*. Una de las ventajas (compartida con Python) de R sobre ellos es, precisamente, que no está sometido a esa restricción. Por eso puede aplicarse de manera natural en un rango más extenso de aplicaciones.

En esta sección investigaremos algunos de los usos de las listas y aprenderemos a crearlas, consultarlas y manipularlas.

5.1. Algunos usos de las listas

Las listas son objetos de R que ya hemos usado antes implícitamente. Por ejemplo, `iris`, como todas las tablas en R, es una lista.

```
is.list(iris)  
## [1] TRUE  
  
Es conveniente recordar que en R las tablas son esencialmente listas de vectores de la misma longitud1. Aunque el anterior es el principal uso de las listas en R, también se emplean como contenedores genéricos de información. Por ejemplo, en la sección anterior hemos construido un modelo,  
  
datos <- as.data.frame(UCBAdmissions)  
datos$Admit <- datos$Admit == "Admitted"  
modelo.con.dept <- glm(Admit ~ Gender + Dept,  
                        data = datos, weights = Freq,  
                        family = binomial())
```

La representación interna en R de este *modelo* es una lista:

¹En esto, R difiere de otras herramientas (p.e., SAS) que consideran las tablas una lista de filas.

```
is.list(modelo.con.dept)

## [1] TRUE

length(modelo.con.dept)

## [1] 30
```

Ejercicio 5.1.1 Explora `modelo.con.dept` con las funciones `names` y `str`.

En efecto, un modelo estadístico en R es una colección de objetos diversos (residuos, coeficientes, etc. representados en forma de valores atómicos, vectores o tablas) que lo resumen. Las listas son estructuras de datos que encapsulan toda esa información heterogénea.

Pero una lista también puede contener otras listas que, a su vez, pudieran contener otras, etc. Es decir, las listas permiten almacenar árboles de información. Los árboles de información son cada vez más importantes en las aplicaciones. De hecho, los ficheros `.xml` o `.json` disponen su información en árboles, como muestra el siguiente ejemplo de un fichero `.json` muy simple:

```
{
  "id": "0001",
  "type": "tarta",
  "name": "Tarta de zanahoria",
  "image": {
    "url": "images/tarta0001.jpg",
    "width": 200,
    "height": 200
  },
  "thumbnail": {
    "url": "images/thumbnails/tarta0001.jpg",
    "width": 32,
    "height": 32
  }
}
```

El fichero anterior tiene cinco elementos, dos de los cuales, `image` y `thumbnail`, contienen a su vez sendas listas de tres elementos (con identificadores `url`, `width` y `height`). Gráficamente, el fichero puede describirse así:

TODO: GRÁFICO DE UN ÁRBOL DE INFORMACIÓN QUE REPRESENTE EL ANTERIOR.

La representación natural en R de esta información es una lista con cinco elementos, dos de los cuales serán sublistas con tres elementos.

Ejercicio 5.1.2 Guarda la salida del test de Student de la sección anterior como un objeto. ¿De qué clase es? Examínalo con las funciones anteriores.

5.2. Exploración y manipulación de listas

La función `length` aplicada a una lista devuelve el número de elementos que contiene. Por eso

```
length(iris)
```

es el número de columnas de `iris`.

Nota

Esto es un poco contraintuitivo para quienes se han acostumbrado a considerar la longitud de una tabla como su número de filas.

Es el momento de introducir formalmente el operador \$, que ya utilizamos en capítulos previos para extraer columnas de una tabla. Ahora vamos a reconocerlo propiamente como un operador de listas.

```
modelo.con.dept$coefficients
```

```
## (Intercept) GenderFemale      DeptB      DeptC      DeptD
## 0.58205140  0.09987009 -0.04339793 -1.26259802 -1.29460647
##          DeptE      DeptF
## -1.73930574 -3.30648006
```

Además de \$, las listas disponen de otro operador para extraer elementos, los dobles corchetes, [[]], que funcionan de manera parecida a \$. Se parecen en que solo pueden extraer un único elemento y lo complementan porque:

- Admite un número como argumento (y \$ no); es decir, permite extraer el elemento n-ésimo de una lista. P.e., `iris[[3]]`.
- Permite extraer elementos cuyo nombre contiene un espacio (como ocurre en ocasiones con los nombres de columnas de tablas importadas de fuentes externas). Por ejemplo (hipotético), `iris[["Petal Length"]]`.

La función `list` permite crear listas. Una vez creada una lista se le pueden añadir campos adicionales².

```
mi.lista <- list(a = 1:3, b = c("holá", "adiós"))
mi.lista$z <- matrix(1:4, 2, 2)
```

Ejercicio 5.2.1 ¿Qué función serviría para concatenar dos listas? ¿Puedes poner un ejemplo?

Ejercicio 5.2.2 ¿Cómo borrarías un elemento de una lista? Recuerda cómo se hacía con tablas (que no dejan de ser listas).

Ejercicio 5.2.3 ¿Qué crees que pasaría si haces `mi.lista[1:2]`?

Ejercicio 5.2.4 Compara `mi.lista[2]` y `mi.lista[[2]]`. ¿Cuál es la diferencia? Pista: puede ayudarte a entender la diferencia el aplicarles la función `is.list`.

Ejercicio 5.2.5 ¿Cómo se crea una lista vacía?

Ejercicio 5.2.6 Si escribes `1m` en la consola de R, se muestra el código de dicha función. Hazlo y examina la última parte. Comprueba que no hace otra cosa que añadir progresivamente elementos a una lista, la que define el modelo, para devolverla al final.

5.3. Las funciones lapply, sapply y split

La función `lapply` está estrechamente relacionada con las listas. Aunque después, en la sección de programación, la trataremos de nuevo, es conveniente familiarizarse con ella cuanto antes.

²En el fondo, una lista en R es lo que en otros lenguajes se denominaría un *namespace*, i.e., una especie de entorno dentro del cual se pueden crear variables.

La función `lapply` es una función de orden superior, como `tapply`, que aplica una función a cada elemento de una lista o vector. Por ejemplo,

```
lapply(airquality[, 1:4], mean, na.rm = TRUE)
```

calcula la media de cada una de las cuatro primeras columnas de `airquality`. El argumento adicional, `na.rm = TRUE`, igual que como ocurría con `tapply`, se pasa a la función `mean`.

La función `lapply` opera sobre una lista o vector y devuelve, siempre, una lista. Está relacionada con la función `sapply`, que es una versión *simplificada* de `lapply`. Es decir, `sapply` aplica la función `lapply` y estudia la salida. Cuando entiende que dicha salida admite una representación menos aparatoso que una lista, la simplifica. Por ejemplo,

```
sapply(airquality[, 1:4], mean, na.rm = TRUE)
```

```
##      Ozone      Solar.R      Wind      Temp
##  42.129310 185.931507  9.957516 77.882353
```

En el caso anterior, en lugar de una lista, como hace `lapply`, `sapply` devuelve una representación más natural: un vector de números.

Tanto `lapply` como `sapply` son *maps*, i.e., funciones que aplican una función a cada elemento de una lista o vector. En muchos casos habituales, no hace falta usar estas funciones porque muchas, como ya hemos visto, están vectorizadas. Por ejemplo,

```
sqrt(1:5)
```

```
## [1] 1.000000 1.414214 1.732051 2.000000 2.236068
```

Pero si `sqrt` no estuviese vectorizada, aún podríamos aplicársela a un vector haciendo, por ejemplo,

```
sapply(1:5, sqrt)
```

```
## [1] 1.000000 1.414214 1.732051 2.000000 2.236068
```

Obviamente, `lapply` y `sapply` permiten aplicar cualquier función, incluidas las definidas por los usuarios.

Finalmente, también es muy útil la función `split`, que permite partir una tabla en trozos de acuerdo con un vector que define los grupos. Por ejemplo,

```
tmp <- split(iris, iris$Species)
```

Ejercicio 5.3.1 Investiga el objeto `tmp`: ¿qué longitud tiene? ¿qué contiene cada uno de sus componentes?

Ejercicio 5.3.2 Usa las funciones `lapply` y `sapply` para mostrar la dimensión de cada una de las tablas que contiene `tmp`.

La función `split` puede servir para operar en subtablas en combinación con `lapply`. Sin embargo, paquetes como `plyr`, que introduciremos más tarde, son más prácticos para este tipo de operaciones.

Ejercicio 5.3.3 Usa `split` para partir `iris` en dos subtablas al azar con el mismo número de filas.

Transformaciones como las que solicita el ejercicio anterior son comunes en ciencia de datos: partir un conjunto de datos en dos al azar para construir un modelo sobre una parte de los datos y evaluar su rendimiento sobre el complementario. El ejercicio siguiente está relacionado con una técnica similar, la de la validación cruzada, que es una alternativa más sofisticada a la anterior.

Ejercicio 5.3.4 Usa `split` para partir `iris` cinco subtablas, cada una de ellas con 30 filas distintas.

5.4. Resumen y referencias

Las listas permiten el almacenamiento, manipulación y análisis de información no estructurada en R. Los datos estructurados son un caso particular de los desestructurados. De ahí que las tablas sean también listas de columnas.

Las tablas de los sistemas de bases de datos han sido siempre colecciones de filas. Solo recientemente se están considerando las bases de datos *columnares*, en las que las columnas de una tabla se almacenan por separado. Esta arquitectura ofrece determinadas ventajas en algunos usos y desventajas en otros. Por lo tanto, como almacén de datos, R podría considerarse un ejemplo de una base de datos columnar.

Las listas están también muy relacionadas con la programación funcional. De hecho, las funciones `lapply` y `sapply` son ejemplos versiones de la operación `map`. La programación funcional admite muchas definiciones e interpretaciones, pero tal vez la más útil (a la vez que inconcreta) es que se trata de aquella que mejor separa el qué del cómo, es decir, qué operaciones queremos realizar para obtener un resultado del cómo realmente implementarlas. Eso lo consigue a través de conceptos y construcciones de orden superior, como `map` (y también `reduce`, `filter`, etc.) que se corresponden muy naturalmente a la manera en que las personas ideamos y construimos los algoritmos.

5.5. Ejercicios adicionales

Ejercicio 5.5.1 La función `strsplit` parte una cadena de texto por un carácter o conjunto de caracteres. Explora la salida de `strsplit(c("un día vi", "una vaca vestida de uniforme"), " ")`.

Ejercicio 5.5.2 Razona por qué la salida de la función `strsplit` tiene que ser una lista.

Ejercicio 5.5.3 Dada la lista `mi.lista <- list(a = 1:3, b = c("hola", "adiós"))`,

- suma los elementos de su primera componente
- añade a su segundo elemento el valor `hasta luego`
- usa `lapply` o `sapply` para calcular la longitud de cada elemento de la lista
- añade a la lista un tercer elemento (llamado `iris`) que contenga la tabla `iris`
- borra el elemento `a`

Capítulo 6

Paquetes, rmarkdown y shiny

Esta sección comienza con una discusión sobre los paquetes, que son extensiones de R. Luego nos centraremos en dos de ellos: **shiny** y **rmarkdown**. El primero de ellos permite crear cuadros de mando interactivos. El segundo, documentos automatizados que combinan texto con código, tablas y gráficos generados directamente por R.

Estos dos paquetes no son complejos sino, más bien, extensos y llenos de detalles. El objetivo de este breve capítulo no es, por supuesto, recorrer exhaustivamente todas sus opciones y posibilidades. Más bien, proporcionar al lector una base sobre la que poder completarlos todos por sí mismo cuando así lo exija una aplicación concreta.

6.1. Paquetes

R tiene paquetes (unos cuantos miles en la actualidad) que extienden sus funciones básicas. Al abrir R se cargan automáticamente una serie de paquetes básicos, como por ejemplo, **base**, donde están casi todas las funciones que hemos usado hasta el momento, o **stats**, que contiene las funciones **lm** o **glm** usadas previamente¹.

La lista completa de los paquetes oficiales puede consultarse en CRAN². Además de los oficiales, existen paquetes que pueden instalarse desde lugares como, por ejemplo, Github.

No es sencillo encontrar el paquete que puede ser útil para un determinado fin. Las *vistas* de CRAN, descripciones de paquetes usados en un determinado ámbito y mantenidas por un experto en la materia, pueden ser un buen punto de partida.

La instalación de paquetes (en este caso, los dos que usaremos en esta sección) puede realizarse o bien desde la consola,

```
install.packages("shiny")
```

o bien a través de los menús de RStudio (bajo *Tools*). Una vez instalados los paquetes es necesario cargarlos para que las funciones que contienen estén disponibles en la sesión:

```
library(shiny)
```

¹Puedes ver, aunque no es demasiado útil, la lista de los paquetes cargados en R con la función **search()**; en una pestaña del panel inferior derecho de RStudio también se muestra una lista de los disponibles y los actualmente cargados en la sesión.

²CRAN (*Comprehensive R Archive Network*) es el repositorio oficial de paquetes de R, el lugar donde se publican las nuevas versiones del programa, etc. Contiene la lista completa de paquetes oficiales

Ejercicio 6.1.1 Encuentra un paquete de R para acceder a datos de Google Analytics e instálalo. Nota: existen varios.

Muchos paquetes tienen *viñetas*, que son documentos que explican cómo usar el paquete. A diferencia de la ayuda de las funciones, que están pensadas para describir todos sus detalles de su uso, incluidos los más arcanos, las viñetas se parecen más a una guía de usuario: suelen indicar cómo utilizar las funciones del paquete y sus opciones más comunes para resolver algún caso sencillo. Leer las viñetas, cuando existen, es la mejor manera de familiarizarse con el funcionamiento de un paquete.

Ejercicio 6.1.2 Si tienes una cuenta en Google Analytics, sigue estas instrucciones para descargar datos de ese servicio y procesar los datos obtenidos.

6.2. Rmarkdown

Rmarkdown permite la creación de informes estáticos que combinan texto con código y resultados, gráficos incluídos, generados con R. Aprender Rmarkdown implica aprender dos cosas distintas:

- Markdown, un *formato* para escribir documentos simples en modo texto. Tiene la ventaja de ser fácilmente legible por humanos pero, a la vez, procesable programáticamente para volcarlos en otros formatos: pdf, html, etc.
- La integración entre R y markdown.

Para aprender markdown, una tarea que no debería durar más de 10-15 minutos, se recomiendan los dos siguientes ejercicios.

Ejercicio 6.2.1 Crea un fichero .Rmd usando File > New File > R Markdown.

Al crear un nuevo fichero de tipo **R Markdown**, RStudio proporciona, en lugar de uno vacío, una plantilla que muestra algunas de las opciones disponibles en este formato. Eso facilita el siguiente ejercicio:

Ejercicio 6.2.2 Aprende markdown modificando el fichero de ejemplo creado en el ejercicio anterior añadiéndole títulos de varios niveles, párrafos de texto, cursivas, negritas, enlaces, listas (numeradas y sin numerar), etc. usando como guía el tutorial <http://commonmark.org/help/>. Recuerda *compilar* el documento (p.e., pulsando el botón con la etiqueta *Knit HTML* situado encima del panel de edición de RStudio) para inspeccionar el resultado final.

También se pueden generar documentos en formato Word y PDF. Para estos tendrás que tener instalados los programas necesarios: MS Word, LibreOffice o similar para el primero y LaTeX para el segundo.

Nota

Para generar el documento en, por ejemplo, formato Word, despliega el menú debajo del icono **Knit** de RStudio y selecciona la opción **Knit to Word**.

Nota

No es necesario utilizar los menús para compilar el fichero de salida. Es posible hacerlo programáticamente.

El segundo de los componentes de Rmarkdown (y lo que lo diferencia de Markdown *a secas*) es la posibilidad de incorporar bloques de código en el hilo del documento. Estos bloques de código se procesan durante la compilación y los resultados que generan (tablas, gráficos, etc.) se integran en la salida. La plantilla de fichero Rmarkdown que genera RStudio incluye unos cuantos bloques de código de ejemplo.

Ejercicio 6.2.3 Crea sobre tu documento (o sobre una nueva plantilla) nuevos bloques de código que hagan alguna cosa.

Ejercicio 6.2.4 Los bloques de código incluyen opciones en su encabezamiento (p.e., para que un bloque se ejecute o no; para que el código se muestre o se oculte en el documento final, etc.). Consulta las opciones (y el tipo de opciones) disponibles en la página de documentación de Knitr.

Con Rmarkdown y sus extensiones se han escrito informes, artículos e, incluso, libros completos. Una de las grandes ventajas de integrar el texto y el código es que todo el proceso de generación del documento final es automático, sin necesidad de incorporar manualmente gráficos o tablas procedentes de terceros programas. Una consecuencia de creciente interés de la automatización del proceso es que evita los errores humanos (o cierto tipo de ellos) en la manipulación de datos. El documento, al contener el código, es en sí mismo la traza de las manipulaciones realizadas y un tercero, en caso de duda, puede reproducir³ el análisis completamente. O incluso aplicarlo a conjuntos de datos distintos.

6.3. Shiny

`shiny` es un paquete de R para la construcción de cuadros de mando *web* interactivos. Permite, por ejemplo, crear interfaces para algoritmos o acceder y manipular tablas de datos a través de controles de HMTL: *sliders*, botones, etc.

El paquete proporciona varias aplicaciones de ejemplo que usaremos para aprender los rudimentos de `shiny`. Por ejemplo, se puede hacer

```
library(shiny)
runExample(example = "01_hello")
```

para desplegar la aplicación de ejemplo `01_hello`. Esta aplicación pinta en el panel central un histograma y tiene en el lateral un *slider* con el que modular su granularidad (técticamente, para definir el número de pedazos, *breaks*, en los que partir el rango de valores del vector subyacente).

Para detener la aplicación, en RStudio, presiona sobre el icono de la señal de *stop* (en la parte superior de la ventana de la consola); en una terminal, usa Control-C para interrumpir la ejecución.

Ejercicio 6.3.1 Ejecuta `runExample()` (sin argumento); el mensaje de error indica qué otros ejemplos además de `01_hello` están disponibles por defecto. Échales un vistazo a algunos.

Ejercicio 6.3.2 Crea tu primera aplicación en `shiny`. Para ello, despliega `01_hello` como antes. Las aplicaciones de ejemplo incluyen su código, del que nos apropiaremos para desarrollar nuestra propia aplicación. Así que crea el fichero `app.R` y copia en él el contenido tal como se muestra en la aplicación. Guárdalo en un nuevo directorio vacío al que puedes llamar, por ejemplo, `prueba00`: recuerda, el nombre de tu aplicación será el nombre que des al directorio que la contiene. Finalmente, ejecuta `runApp("prueba00")` para desplegarla.

El ejercicio anterior muestra cómo construir aplicaciones en `shiny`. Una aplicación en `shiny` es un directorio que da nombre a la aplicación. Dentro de él tiene que haber, como mínimo, un fichero, `app.R`, que contiene dos bloques en que se definen los componentes `ui` y `server`. El primero define la interfaz de la aplicación. El segundo realiza los cálculos en segundo plano cada vez que el usuario manipula los controles de la interfaz.

Ejercicio 6.3.3 Compara el contenido de los bloques `ui` y `server` definidos en `app.R` y trata de relacionar los elementos que se definen en ellos con el resultado visual y funcional de la aplicación.

Nota

³De hecho, Rmarkdown nació para dar respuesta a un problema acuciante en la academia y fuera de ella: que los resultados de muchos artículos e informes eran imposibles de reproducir con otros datos, en otros lugares, por otras personas.

En versiones anteriores de `shiny` era obligatorio disponer los componentes `ui` y `server` en dos ficheros independientes, `ui.R` y `server.R`. Actualmente, es potestativo.

Además de este fichero, en aplicaciones más complejas, puede haber otros organizados o no en subdirectorios: datos, otros ficheros auxiliares de código, logos, CSSs, imágenes estáticas, etc. Tendrás que leer la documentación de `shiny` para averiguar cómo y dónde colocar estos recursos adicionales si es que los necesitas.

Los dos ejercicios siguientes, opcionales, bastante avanzados y que tienen más que ver con la administración de sistemas Linux que con R, muestran cómo una aplicación desarrollada en `shiny` puede ser puesta *en producción* y distribuida a los usuarios a través de la *web*.

Ejercicio 6.3.4 Instala *Shiny Server* (busca en internet la página de descarga de esa aplicación) en tu máquina Linux. Guarda tu fichero `app.R` en el directorio adecuado y accede a tu aplicación a través de tu navegador (puerto 3737, por defecto). Nota: existen tutoriales bastante detallados sobre cómo hacer esto.

Ejercicio 6.3.5 Como continuación del ejercicio anterior, da salida a `shiny` a través del puerto 80 vía Apache u otros.

Los bloques `ui` y `server` se comunican entre sí: `ui` tiene que pasarle parámetros a `server` y este resultados a aquél. Esto se hace a través de variables y estructuras de datos con una forma muy particular. El siguiente ejercicio está pensado para que descubras el mecanismo de comunicación. Se te va a pedir que traduzcas el nombre de las variables y los parámetros al español. Obviamente, al traducir las variables en uno de los bloques se romperá la aplicación. Relinear los nombres en el otro bloque te servirá para identificar los mecanismos de comunicación.

Ejercicio 6.3.6 Crea `prueba01` como una copia de `prueba00`. Entonces, traduce al español el nombre de todas las variables implicadas en la aplicación.

El siguiente ejercicio te enseñará a modificar la interfaz de una aplicación en `shiny`, incorporar nuevos controles y añadir el código subyacente para que responda adecuadamente.

Ejercicio 6.3.7 Añade a la aplicación `prueba01` otro *slider* que mueva una linea vertical roja que dibujes sobre el histograma. Es recomendable que comiences, y en este orden, añadiendo una línea roja en algún punto (del eje x) prefijado con la función `abline`, incorporando el *slider* y finalmente, vinculando el valor proporcionado por el *slider* al punto del eje x.

Nota: cuando modifiques `ui`, presta atención a la estructura del programa y cómo se corresponde a la de la interfaz web: qué es lo que va en la barra lateral, qué en el panel central, etc. Ten cuidado además con los paréntesis: ¡hay muchos y es fácil desemparejarlos!

Ejercicio 6.3.8 Inspecciona el tutorial de `shiny` para descubrir qué tipo *widgets* (además de *sliders*) existen, cómo procesan `server` esos *inputs*, etc. Recuerda que ese tutorial es la principal fuente de información sobre todo lo relacionado con `shiny`.

Ejercicio 6.3.9 Visita la galería de aplicaciones de Shiny) para investigar cómo implementar controles, qué tipos adicionales de paneles existen, etc.

Uno de los asuntos (avanzados) que se discuten en esas páginas es el de las reacciones: `shiny` está basado en un tipo de programación denominada *reactiva*, que es la que permite que las funciones de `server.r` simulen estar escuchando (y *reaccionen*) a los cambios que realiza el usuario en los controles de la aplicación. Puedes buscar en internet más información sobre la *programación reactiva* si te interesa el tema.

6.4. Resumen y referencias

Los paquetes de R extienden el lenguaje. Existen, literalmente, miles de ellos, aunque varían grandemente en extensión, calidad y utilidad. Los hay muy básicos: algunos desarrolladores, por ejemplo, han usado los paquetes para publicar un único conjunto de datos (i.e., han creado un paquete que solo contiene un conjunto de datos). En el extremo opuesto, los hay con centenares de funciones. En términos de calidad, los hay de todo tipo, desde los mantenidos de manera profesional hasta los francamente inadecuados para su uso en producción.

Finalmente, coexisten paquetes que implementan métodos estadísticos pensados para un fin sumamente concreto con otros para los que muchísimos usuarios han encontrado uso. De entre los más útiles para quienes analizan datos profesionalmente se encuentran los dos discutidos más ampliamente en este capítulo: `shiny` y `rmarkdown`.

Ambos se utilizan profusamente en estos entornos y son las dos herramientas que más se utilizan (y se recomienda utilizar) para comunicar y disseminar resultados analíticos.

Desarrollar interfaces en `shiny` puede ser engorroso. Existe un paquete, `flexdashboard`, que permite definir los componentes de una aplicación de `shiny` directamente usando RMarkdown, combinando así la sencillez de este último con la interactividad de Shiny. Puedes aprender más sobre `flexdashboard` aquí.

6.5. Ejercicios adicionales

Ejercicio 6.5.1 Busca dentro de las *vistas* de CRAN algún tema de tu interés. Échale un vistazo a los paquetes disponibles.

Ejercicio 6.5.2 Investiga cómo instalar paquetes directamente desde Github usando las herramientas del paquete `devtools`. Nota: muchos de estos paquetes no son oficiales (lo cual no es un juicio sobre su calidad); en otras ocasiones, desde Github pueden descargarse versiones más actualizadas de paquetes oficiales que las disponibles en CRAN.

Ejercicio 6.5.3 Investiga cómo representar tablas en documentos de RMarkdown. Ten en cuenta que existen varias maneras, desde las más rudimentarias a otras más sofisticadas y con un acabado más profesional que utilizan paquetes auxiliares.

Ejercicio 6.5.4 Investiga una aplicación de ejemplo desarrollada con `flexdashboard`. Aprópiate de ella y trata de introducir tus propias modificaciones.

Ejercicio 6.5.5 Reimplementa el ejemplo `01_hello` de Shiny en `flexdashboard`.

Capítulo 7

Web scraping y manipulación básica de texto

En esta sección vamos a realizar un pequeño desvío para tratar dos temas auxiliares pero muy importantes en la práctica. Lo hacemos precisamente ahora porque contamos con los conocimientos suficientes para sacarles partido y porque, además, nos serán útiles más adelante: el *rascado* de páginas web (*web scraping*) y la manipulación básica de texto.

7.1. Web scraping

En ocasiones interesa descargar datos directamente de páginas de internet recorriendo una, varias o, incluso, muchas de ellas. A eso, a falta de un nombre de consenso en español (*rascado*?), se lo denomina *web scraping*.

Para descargar datos de páginas web usaremos el paquete `rvest`.

```
library(rvest)
```

Con él podemos descargar, por ejemplo, las cotizaciones del IBEX 35 en *tiempo real*:

```
url.ibex <- "http://www.bolsamadrid.es/esp/aspx/Mercados/Precios.aspx?indice=ESI1000000000"
tmp <- read_html(url.ibex)
tmp <- html_nodes(tmp, "table")
```

La segunda línea descarga y preprocesa una página descargada de internet. El objeto `tmp` contiene una representación interna de la página. Una página *web*, en el fondo, no es otra cosa que un árbol del que penden nodos que son párrafos, imágenes, enlaces, tablas, etc. Sobre este árbol se pueden realizar distintos tipos de consultas, i.e., extraer distintos tipos de *nodos*.

La función `html_nodes` captura los nodos que tienen determinadas características. En este ejemplo, como ocurre con mucha frecuencia, nos interesan los identificados como tablas (`tables`). De hecho, las tablas son tan interesantes que el paquete `rvest` proporciona una función auxiliar para convertir los nodos de tipo `table` en tablas de R: `html_table`.

En nuestro ejemplo, la página contiene varias tablas. Como consecuencia, `tmp` es una lista de nodos:

```
length(tmp)
## [1] 5
sapply(tmp, class)
```

```
## [1] "xml_node" "xml_node" "xml_node" "xml_node" "xml_node"
```

La página, aunque no lo parezca, tiene varias tablas. Eso se debe a que en HTML las tablas se utilizan en ocasiones, por abuso, no para almacenar datos tabulares sino para dar formato a las páginas. Sin embargo, es fácil detectar estas *seudotablas* por inspección. Para identificar la tabla de interés, la que contiene las cotizaciones, podemos examinarlas todas ejecutando la función `html_table` sobre `tmp[[1]]`, `tmp[[2]]`, etc. hasta dar con ella: es la quinta y última.

Alternativamente, para evitar tener que examinar las tablas una a una se puede hacer

```
sapply(tmp, function(x) dim(html_table(x, fill = TRUE)))
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    46     1     1     1   35
## [2,]   351     1     7     9     9
```

que nos indica que la quinta tabla tiene 35 filas, un indicio sólido de que es la que va a contener las cotizaciones de las 35 empresas del IBEX.

Podemos entonces transformar este último nodo en una tabla de R:

```
ibex <- html_table(tmp[[5]])
```

Ejercicio 7.1.1 Inspecciona la tabla recién cargada. Presta atención al tipo de las columnas. ¿Observas algo raro?

La información colgada en internet está pensada para ser consumida por humanos, no máquinas. Como consecuencia, los números están decorados con separadores de miles, unidades, porcentajes, etc.; los nombres de columnas tienen espacios y otros caracteres extraños, etc. Por eso, la información directamente descargada mediante técnicas de *rascado* raramente se puede utilizar directamente: es necesario someterla a un proceso sencillo pero laborioso de limpieza.

Ejercicio 7.1.2 Dales nombres *razonables* a las columnas de `ibex`. Nota: usa `colnames`.

Es habitual tener que cambiar los nombres de columnas: utilizar nombres con caracteres no estándar es garantía de problemas en los análisis subsiguientes. También es habitual tener que manipular los valores de esas columnas; es frecuente que, por culpa de los separadores de miles, el uso de la coma como separador de los decimales o el uso de unidades, R no reconozca las columnas numéricas como tales y que las trate como texto. Entonces no queda otro remedio que tratar esas columnas para darles el formato correcto y convertirlas en numéricas usando `as.numeric`. De ahí la sección siguiente, que es una introducción a la manipulación básica de texto con R.

Además de las tablas, hay otros tipos de nodos que puede interesar extraer de una página *web*. La función `html_nodes` permite hacerlo utilizando XPath. Puedes echarle un vistazo a esta página, donde se muestra un ejemplo de *web scraping* en el que se descarga información no contenida en tablas de una serie de páginas.

Ejercicio 7.1.3 Examina la documentación del paquete `rvest` y busca aplicaciones a sus funciones.

7.2. Manipulación básica de texto

Las tablas bajadas de internet (y datos procedentes de otras fuentes) exigen frecuentemente un proceso de limpieza de datos. Por ejemplo, es típico que, al importar datos, las columnas que contienen números se interpreten como cadenas de caracteres por incluir símbolos *no estándar* como separadores de miles, etc. O tener que reinterpretar determinados campos como, p.e., fechas.

La función `gsub` se usa muy a menudo para dicha limpieza de datos. Una llamada a `gsub` tiene la forma

```
gsub("h", "H", c("hola", "búho"))
```

```
## [1] "Hola" "búHo"
```

donde el primer argumento, "h" es una expresión regular; la función `gsub` modifica las ocurrencias de esta expresión regular por el segundo argumento, "H" en este caso. El tercer argumento es un vector que contiene cadenas de texto en las que se realiza la sustitución.

Las expresiones regulares son muy útiles para manipular texto. Conviene aprender algunas de las más frecuentes, como por ejemplo, las que identifican caracteres que aparecen al principio de un texto,

```
gsub("^h", "H", c("hola", "búho"))
```

```
## [1] "Hola" "búho"
```

o al final del mismo,

```
gsub("o$", "os", c("hola", "búho"))
```

```
## [1] "hola"   "búhos"
```

Una función emparentada con `gsub` es `grep`, que busca cadenas en las que aparece una determinada expresión regular:

```
grep("^h", c("hola", "búho"))
```

```
## [1] 1
```

La salida de la expresión anterior nos indica que el patrón *cadena de texto que comienza con la letra h* aparece solo en la posición número 1 del vector.

Ejercicio 7.2.1 `colors()` es una función que devuelve el nombre de más de 600 colores en R. Usándolo,

- encuentra aquellos cuyo nombre contenga un número (posiblemente tengas que investigar cómo se expresa *cualquier número* como expresión regular)
- encuentra aquellos que comiencen con `yellow`
- encuentra aquellos que contengan `blue`
- reemplaza los números por `x` (p.e., `blue10` quedaría como `bluexx`)
- reemplaza secuencias de números por `x` (por ejemplo, `blue10` quedaría como `bluex`)

Ejercicio 7.2.2 Los números que aparecen en la tabla descargada en la sección anterior (y contenidos en `ibex`) no tienen formato numérico. Para convertirlos en números *de verdad*, transfórmalos adecuadamente:

- Usa `gsub` para cambiar `"."` por `"` (i.e., nada) en las columnas de interés. Ten en cuenta que `.` es el comodín de las expresiones regulares; el punto es `\\"..`
- Usa `gsub` para cambiar `,` por `.` en las columnas de interés.
- Finalmente, usa `as.numeric` para cambiar texto resultante por valores numéricos.
- ¿Te atreves a usar `as.Date` para cambiar texto por fechas donde proceda?

Otra función muy útil para procesar texto es `paste`, que tiene un comportamiento distinto según se use con el argumento `sep` o `collapse`.

```
paste("A", 1:6, sep = ",")
```

```
## [1] "A,1" "A,2" "A,3" "A,4" "A,5" "A,6"
```

```
paste("Hoy es ", date(), " y tengo clase de R", sep = "")
```

```
## [1] "Hoy es Sun Apr 22 21:45:23 2018 y tengo clase de R"
```

```
paste("A", 1:6, collapse = ",")  
  
## [1] "A 1,A 2,A 3,A 4,A 5,A 6"  
  
sep y collapse pueden combinarse:  
paste("A", 1:6, sep = "_", collapse = ",")  
  
## [1] "A_1,A_2,A_3,A_4,A_5,A_6"
```

Para la operación inversa, la de partir cadenas de texto, se usa la función `strsplit`:

```
strsplit("Hoy es martes", split = " ")  
  
## [[1]]  
## [1] "Hoy"      "es"       "martes"  
  
strsplit(c("hoy es martes", "mañana es miércoles"), split = " ")  
  
## [[1]]  
## [1] "hoy"      "es"       "martes"  
##  
## [[2]]  
## [1] "mañana"   "es"       "miércoles"
```

Advierte que esta función devuelve una lista de cadenas de texto (¿podría ser de otra manera?).

Ejercicio 7.2.3 Crea una función que tome los nombres de ficheros

```
ficheros <- c("ventas_20160522_zaragoza.csv", "pedidos_firmes_20160422_soria.csv")
```

y genere una tabla con una fila por fichero y tres columnas: el nombre del fichero, la fecha y y la provincia. Nota: puedes crear una función que procese solo un nombre de fichero y aplicársela *convenientemente* al vector de nombres.

Esas son las funciones fundamentales para la manipulación básica de texto en R. Existen otras que se encuentran también en otros lenguajes de programación (p.e., `sprintf`, `substr`, etc.) u otros paquetes, como `separate` y `unite` del paquete `tidyR`¹. No obstante, con las descritas en esta sección, se cubren la mayor parte de las necesidades de manipulación de datos corrientes.

7.3. JSON y XML

Existe una web para humanos (el mundo del HTML) y otra para máquinas. Se puede bajar información programáticamente de las primeras usando las técnicas de *rascado* (o *scraping*) discutidas más arriba y después limpiarla para su postproceso como hemos hecho en la sección anterior.

Pero cada vez son más populares los servicios web que proporcionan información a través de APIs que son consultadas directamente por ordenadores. Estos servicios suelen proporcionar resultados en formato JSON, XML o ambos. Los dos formatos son similares: organizan la información en forma de árbol.

Por ejemplo, el INE proporciona un API JSON del que se puede bajar información de interés estadístico. Tiene, además, un servicio que permite construir la consulta, i.e., obtener la URL con la que consultar una serie de datos en concreto. Usándola, encontramos que para obtener la población de cada provincia española por sexos durante los últimos cinco años tenemos que consultar esta.

Pero podemos realizar la consulta programáticamente así:

¹Véase esto

```
library(rjson)

pob <- readLines("http://servicios.ine.es/wstempus/js/ES/DATOS_TABLA/2852?nult=5&tip=AM")
pob <- paste(pob, collapse = " ")
pob <- fromJSON(pob)
```

En la primera línea cargamos el paquete necesario, `rjson`. En las dos siguientes leemos la URL y colapsamos todas las líneas en una única cadena de texto, una exigencia de `fromJSON`. Esta es la función que llamamos en última instancia para convertir el fichero JSON en una estructura arborescente en R, i.e., una lista que contiene, a su vez, otras listas.

```
class(pob)
```

```
## [1] "list"
```

```
length(pob)
```

```
## [1] 159
```

`pob` tiene longitud 159. Son tres elementos por cada provincia (más Ceuta, Melilla y el total nacional), los correspondientes a los dos sexos y el total. Cada uno de estos elementos tiene una serie de atributos y una sublistas de longitud 5, que almacena los datos anuales. Así,

```
pob[[89]]$Data[[5]]$Valor
```

```
## [1] 168013
```

es el valor (i.e, la población) correspondiente al quinto periodo (o año) del elemento 85 de la primera lista, i.e.,

```
pob[[89]]$Nombre
```

```
## [1] "Lugo. Hombres. Total habitantes. Personas. "
```

Idealmente, queríamos convertir nuestros datos (o la parte más relevante de ellos) en una tabla para su posproceso. Podremos hacerlo más adelante, cuando aprendamos más sobre manipulación de datos y programación en R.

Otras APIs proporcionan información en formato XML. Por ejemplo, la del Banco Mundial²:

```
library(xml2)
```

```
bm <- read_xml("http://api.worldbank.org/countries/all/indicators/NY.GDP.MKTP.CD?date=2009:2010&per_page=1000")
esp <- xml_find_all(bm, "//*/wb:data[wb:country[@id='ES']]//wb:value")
as.numeric(xml_text(esp))
```

```
## [1] 1.431617e+12 1.499100e+12
```

El código anterior proporciona el PIB español de los años 2009 y 2010 en dólares. La primera línea carga el paquete `xml2`, que es el que vamos a usar para leer ficheros XML. La función `read_xml` lee una URL de la API del Banco Mundial que extrae el indicador NY.GDP.MKTP.CD (PIB según la documentación de la API) y la procesa. A diferencia de lo que ocurría con JSON, el objeto `bm` no es una lista de R (aunque la podemos convertir en una usando la función `as_list`), sino un objeto de la clase `xml_document`. Sobre este documento se pueden realizar consultas usando XPath, como más arriba, y obtener los valores correspondientes al país con etiqueta `ES`, es decir, España.

El objeto `esp` contiene dos *nodos* y la función `xml_text` permite extraer su contenido (en formato de texto, que tenemos que convertir en números). En general, los nodos tienen atributos y el texto es uno más de ellos.

²Que también la ofrece en JSON.

7.4. Resumen y referencias

Hemos aprendido técnicas para descargar información disponible en internet en varios formatos. Para el más joven de todos ellos, JSON, hemos usado el paquete `rjson`. Este paquete tiene dos funciones: la que hemos usado, `fromJSON`, para leer ficheros JSON en R (y convertirlos en listas de listas...) y `toJSON`, para realizar el proceso inverso.

Existen otros paquetes similares en R, como `RJSONIO` o `jsonlite` muy similares a `rjson`. Curiosamente, los tres paquetes contienen las dos mismas funciones, `fromJSON` y `toJSON`, con pequeñas diferencias de implementación entre ellos. La mayor diferencia entre `jsonlite` y el resto es que su función `fromJSON` proporciona la opción de simplificar la lista que produce en algunos casos para construir una tabla u otras estructuras de datos cuando detecta que es posible. Esto puede ser ventajoso determinadas circunstancias.

También se pueden descargar datos de ficheros XML o HTML. Tradicionalmente, esto se hacía con el paquete `XML` de R, que aprovechaba el hecho de que HTML es un dialecto³ de XML. `XML`, sin embargo, era algo complicado. Así que con el tiempo apareció una alternativa, `xml2`, un tanto más amigable. En el fondo, tanto `XML` como `xml2` son interfaces de R para una librería desarrollada en C, `libxml2`. El paquete que hemos usado para *rascar* páginas, `rvest`, es a su vez, una interfaz para el paquete subyacente, `xml2`, adaptada al *web scraping*.

`XML` y `xml2` (y, como consecuencia, `rvest`) hacen uso de XPath, un lenguaje de consulta para documentos XML (de la misma manera que SQL es un lenguaje de consulta para datos relacionales). Es un lenguaje que pocos dominan con soltura y que, casi seguro, solo vas a necesitar esporádicamente. Es conveniente que sepas que existe y cuatro cosas básicas sobre él. Es fácil encontrar en internet ejemplos de consultas sencillas que pueden modificarse para un fin determinado. Lo más típico suele ser la extracción de valores encerrados en etiquetas `div` o `span` con determinados atributos `class` o `id`.

Finalmente, acerca de la manipulación de texto —que es un campo, por supuesto, mucho más extenso que el contexto con el que lo hemos motivado, i.e., el arreglo de datos *rascados* de internet— es conveniente hacer estas dos recomendaciones:

- Aprende bien los fundamentos de las expresiones regulares. Son muy útiles y se emplean en muchos más lenguajes de programación.
- Que aunque las funciones discutidas en este capítulo son suficientes para un uso esporádico, si vas a estar trabajando seriamente con texto, deberías familiarizarte con el paquete `stringr`, que contiene una selección amplia de funciones avanzadas de manipulación de texto con una interfaz homogénea y moderna.

7.5. Ejercicios adicionales

Ejercicio 7.5.1 Toma el vector `c("41°39'00''N", "0°53'00''O")` y convírtelo en *coordenadas decimales*. Recuerda que también tienes que asignar el signo (es menos para latitudes en el hemisferio sur y al oeste del meridiano de Greenwich).

³En sentido amplio, por supuesto.

Capítulo 8

Pivotación de tablas con reshape2

En contra de la opinión generalizada, una parte fundamental del trabajo con R consiste en manipular y transformar datos para darles la forma necesaria para su uso en un análisis posterior. En las secciones anteriores hemos explorado ya algunas técnicas básicas que pueden usarse para este fin. En esta y alguna de las siguientes vamos a presentar ciertas *metatransformaciones* de datos, i.e., operaciones genéricas que subyacen a muchas transformaciones específicas.

En concreto, en esta sección exploraremos el paquete `reshape2` que contiene, esencialmente, dos funciones: `melt` y `dcast`¹. La primera, a partir de una tabla, crea una versión en formato largo, arreglado² o *tidy*. La función `dcast` es una inversa —y algo más— de `melt`. Entre otras cosas, `dcast` incorpora a R operaciones análogas a las *pivot tables* de Excel.

8.1. La función melt y datos en formato largo

Una tabla está en formato largo cuando cada fila contiene un dato y el resto de las columnas son etiquetas que le dan contexto. Además de ser más claras conceptualmente, suele ser más sencillo procesar tablas en formato largo. A los interesados en saber más sobre este tipo de datos se les recomienda leer el artículo Tidy Data³.

Vamos a comenzar leyendo un fichero de datos que contiene la población en 2014 de las provincias aragonesas por sexo. Se trata de un conjunto de datos simple y minúsculo pero que, gracias a eso, permite visualizar las transformaciones que vamos a realizar.

```
pob.aragon.2014 <- read.table("data/pob_aragon_2014.csv",
                                header = T, sep = "\t")
pob.aragon.2014

##   Provincia Hombres Mujeres
## 1   Huesca  113840 111069
## 2   Teruel   71449  68916
## 3 Zaragoza  471675 488436
```

Es habitual encontrar conjuntos de datos (aunque típicamente más grandes) en formatos similares. Estos datos no están en formato largo o arreglado. En tablas arregladas, las columnas representan necesariamente variables. Sin embargo, en `pob.aragon.2014` existe una variable implícita, `sexo` de la que `Hombres` y `Mujeres` son *niveles* (o valores).

¹La traducción de estos dos verbos es, respectivamente, *fundir* y *moldear*; es conviente recordarlo.

²De acuerdo con el DRAE, arreglado significa *reducido o sujeto a regla*; efectivamente, veremos cómo los datos en formato largo siguen una serie de reglas que se echan mucho de menos cuando no se cumplen.

³Cuyo autor lo es también del paquete `reshape2`, el tema de esta sección

Podemos usar la función `melt` para pivotar la tabla y *arreglar* los datos, i.e., disponerlos en formato largo:

```
library(reshape2)
melt(pob.aragon.2014)
```

```
##   Provincia variable  value
## 1    Huesca   Hombres 113840
## 2    Teruel   Hombres  71449
## 3  Zaragoza   Hombres 471675
## 4    Huesca   Mujeres 111069
## 5    Teruel   Mujeres  68916
## 6  Zaragoza   Mujeres 488436
```

Es evidente que la información contenida en ambos conjuntos de datos es la misma. Pero en la nueva tabla cada fila tiene un único valor. El resto de las columnas, `Provincia` y `variable` —como R desconoce que el nombre ideal de la nueva variable sería `sexo`, por defecto, la denomina `variable`— contextualizan dicha cifra.

Un ejemplo un poco más interesante es este:

```
pob.aragon <- read.table("data/pob_aragon.csv",
                           header = T, sep = "\t")
pob.aragon
```

```
##   Provincia Periodo Hombres Mujeres
## 1    Huesca    2014 113840 111069
## 2    Huesca    2004 107961 104940
## 3    Teruel    2014  71449  68916
## 4    Teruel    2004  71073  68260
## 5  Zaragoza    2014 471675 488436
## 6  Zaragoza    2004 441840 455510
```

El nuevo conjunto de datos tiene una variable adicional, `Periodo`. Si se le aplica `melt` directamente, el resultado dista del esperado:

```
melt(pob.aragon)
```

```
##   Provincia variable  value
## 1    Huesca   Periodo  2014
## 2    Huesca   Periodo  2004
## 3    Teruel   Periodo  2014
## 4    Teruel   Periodo  2004
## 5  Zaragoza   Periodo  2014
## 6  Zaragoza   Periodo  2004
## 7    Huesca   Hombres 113840
## 8    Huesca   Hombres 107961
## 9    Teruel   Hombres  71449
## 10   Teruel   Hombres  71073
## 11  Zaragoza   Hombres 471675
## 12  Zaragoza   Hombres 441840
## 13    Huesca   Mujeres 111069
## 14    Huesca   Mujeres 104940
## 15    Teruel   Mujeres  68916
## 16    Teruel   Mujeres  68260
## 17  Zaragoza   Mujeres 488436
## 18  Zaragoza   Mujeres 455510
```

Efectivamente, R confunde `Periodo` con un nivel más de la nueva variable, junto a `Hombres` y `Mujeres`. Pero,

en realidad, tanto la provincia como el periodo forman la *clave* de la tabla. Esto se le puede indicar a `melt` así:

```
melt(pob.aragon, id.vars = c("Provincia", "Periodo"))
```

```
##   Provincia Periodo variable value
## 1     Huesca    2014   Hombres 113840
## 2     Huesca    2004   Hombres 107961
## 3     Teruel    2014   Hombres  71449
## 4     Teruel    2004   Hombres  71073
## 5   Zaragoza    2014   Hombres 471675
## 6   Zaragoza    2004   Hombres 441840
## 7     Huesca    2014 Mujeres 111069
## 8     Huesca    2004 Mujeres 104940
## 9     Teruel    2014 Mujeres  68916
## 10    Teruel    2004 Mujeres  68260
## 11   Zaragoza    2014 Mujeres 488436
## 12   Zaragoza    2004 Mujeres 455510
```

Los datos en formato largo están muy relacionados con los *arrays* (conocidos en algunos contextos empresariales como *cubos multidimensionales*). Un ejemplo de ellos es la tabla `Titanic` de R, que se refiere a los viajeros del Titanic, y que tiene cuatro dimensiones: la clase, el sexo, el grupo de edad y si sobrevivieron o no al hundimiento:

```
dimnames(Titanic)
```

```
## $Class
## [1] "1st"   "2nd"   "3rd"   "Crew"
##
## $Sex
## [1] "Male"  "Female"
##
## $Age
## [1] "Child" "Adult"
##
## $Survived
## [1] "No"    "Yes"
```

La función `as.data.frame` aplicada a un array multidimensional produce una tabla en formato largo:

```
tmp <- as.data.frame(Titanic)
rbind(head(tmp), tail(tmp))
```

```
##   Class   Sex   Age Survived Freq
## 1  1st   Male Child     No     0
## 2  2nd   Male Child     No     0
## 3  3rd   Male Child     No    35
## 4 Crew   Male Child     No     0
## 5  1st Female Child    No     0
## 6  2nd Female Child    No     0
## 27 3rd   Male Adult Yes    75
## 28 Crew   Male Adult Yes   192
## 29  1st Female Adult Yes   140
## 30  2nd Female Adult Yes    80
## 31  3rd Female Adult Yes    76
## 32 Crew   Female Adult Yes    20
```

Ejercicio 8.1.1 La función `melt` también funciona con matrices. Una de ellas es `VADeaths`. Usa `melt` para disponer esa matriz en formato largo. Verás que aparece una columna con valores del tipo `Rural Male`: construye a partir de ella dos columnas (`sex` y `residency`). **Nota:** existe una función auxiliar en `reshape2`, `colsplit` que realiza precisamente esa operación.

Ejercicio 8.1.2 Toma el conjunto de datos `airquality` y disponlo en formato largo. Ten cuidado con la definición de la clave de la tabla.

Ejercicio 8.1.3 Cambia el nombre a la variable `variable` creada automáticamente por `melt` en el ejercicio anterior.

Ejercicio 8.1.4 Calcula el valor mediano (`median`) de las variables de `airquality` (después de haberlo dispuesto en formato largo). Usa la función `tapply`.

En general, disponer datos en formato largo facilita muchas operaciones de manipulación de datos. En el ejercicio anterior, en lugar de calcular la mediana variable a variable, ha sido posible obtenerla en dos líneas de código. Que son independientes, además, del número de indicadores de la tabla.

8.2. La función `dcast` y datos en formato ancho

A partir del formato largo se puede pasar a distintos tipos de formatos no largos (o anchos) usando la función `dcast`. Por ejemplo, a partir de los datos

```
pob.aragon.2014.largo <- melt(pob.aragon.2014)
pob.aragon.2014.largo
```

```
##   Provincia variable value
## 1    Huesca   Hombres 113840
## 2    Teruel   Hombres  71449
## 3 Zaragoza   Hombres 471675
## 4    Huesca   Mujeres 111069
## 5    Teruel   Mujeres  68916
## 6 Zaragoza   Mujeres 488436
```

se pueden poner las provincias en filas

```
dcast(pob.aragon.2014.largo, Provincia ~ variable)
```

```
##   Provincia Hombres Mujeres
## 1    Huesca  113840  111069
## 2    Teruel   71449   68916
## 3 Zaragoza  471675  488436
```

o en columnas:

```
dcast(pob.aragon.2014.largo, variable ~ Provincia)
```

```
##   variable Huesca Teruel Zaragoza
## 1   Hombres 113840  71449  471675
## 2   Mujeres 111069  68916  488436
```

En su versión más simple, `dcast` necesita:

- la tabla sobre la que operar
- qué columnas disponer en filas (generalmente varias, separadas por +, delante de ~)

- qué columnas disponer en columnas (generalmente una, detrás de ~)

La función `dcast` utiliza una serie de reglas simples para determinar cuál es la columna de la que extraer los datos que incluir en la tabla resultante. Pero, a veces, como en el ejemplo que sigue, se equivoca. Por ejemplo, vamos a intentar medir el incremento en las horas de sueño (`extra`) producidas por el tratamiento (o `group`) 2 sobre el 1 en el conjunto de datos `sleep`. Este conjunto de datos se refiere a individuos (`ID`) que fueron sometidos a dos tratamientos distintos. Podemos hacer

```
dcast(sleep, ID ~ group)
```

```
## Using ID as value column: use value.var to override.
```

```
##   ID 1 2
## 1 1 1 1
## 2 2 2 2
## 3 3 3 3
## 4 4 4 4
## 5 5 5 5
## 6 6 6 6
## 7 7 7 7
## 8 8 8 8
## 9 9 9 9
## 10 10 10 10
```

para intentar colocar los dos valores correspondientes a cada individuo en la misma fila para poder restarlos después. Sin embargo, el efecto no es el deseado y el *warning* de R es muy informativo sobre el problema: que `dcast` ha elegido como variable *de valor* `ID` en lugar de `extra`. Además, indica cómo forzar una alternativa a la que usa por defecto, i.e., usando `value.var`:

```
dcast(sleep, ID ~ group, value.var = "extra")
```

```
##   ID 1 2
## 1 1 0.7 1.9
## 2 2 -1.6 0.8
## 3 3 -0.2 1.1
## 4 4 -1.2 0.1
## 5 5 -0.1 -0.1
## 6 6 3.4 4.4
## 7 7 3.7 5.5
## 8 8 0.8 1.6
## 9 9 0.0 4.6
## 10 10 2.0 3.4
```

Ahora es posible añadir la columna adicional de interés haciendo

```
tmp <- dcast(sleep, ID ~ group, value.var = "extra")
tmp$diff <- tmp[, "2"] - tmp[, "1"]
head(tmp, 3)
```

```
##   ID 1 2 diff
## 1 1 0.7 1.9 1.2
## 2 2 -1.6 0.8 2.4
## 3 3 -0.2 1.1 1.3
```

Ejercicio 8.2.1 `as.data.frame(Titanic)` es una tabla en formato largo que contiene información sobre los viajeros que sobrevivieron o perecieron el naufragio del Titanic. Calcula la proporción de viajeros que sobrevivieron por clase, sexo y grupo de edad. Nota: recuerda que para indicar varias filas, ya sea en filas o

columnas (aunque lo segundo no se recomienda), los nombres de columna tienen que indicarse separados del signo +.

No obstante, al usar `dcast` pueden ocurrir situaciones como que ilustra el ejemplo siguiente. En primer lugar construimos una versión *larga* de `iris`:

```
iris.long <- melt(iris)
head(iris.long, 4)

##   Species      variable value
## 1  setosa Sepal.Length   5.1
## 2  setosa Sepal.Length   4.9
## 3  setosa Sepal.Length   4.7
## 4  setosa Sepal.Length   4.6
```

Nota

Hemos hecho la operación `melt(iris)` para crear un conjunto de datos de práctica. Sin embargo, en esta operación se pierde información: ya no sabemos a qué fila de la tabla original corresponden las de la versión arreglada. Por lo tanto, la reconstrucción de los datos originales es imposible.

Ahora, al hacer

```
dcast(iris.long, Species ~ variable)

## Aggregation function missing: defaulting to length

##   Species Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1  setosa        50        50        50        50
## 2 versicolor     50        50        50        50
## 3 virginica      50        50        50        50
```

obtenemos un resultado extraño (además de un *warning*). Se debe a que en cada posición de la tabla resultante hay más de un valor candidato; de hecho, 50 de ellos. La función `dcast`, por defecto, combina esos valores usando la función `length`. Sin embargo, es posible especificar otras:

```
dcast(iris.long, Species ~ variable, fun.aggregate = mean)

##   Species Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1  setosa       5.006     3.428     1.462     0.246
## 2 versicolor    5.936     2.770     4.260     1.326
## 3 virginica     6.588     2.974     5.552     2.026
```

La función `dcast` admite otras invocaciones equivalentes. Por ejemplo, como hemos visto más arriba, se puede especificar explícitamente la variable sobre la que opera la nueva función:

```
dcast(iris.long, Species ~ variable, fun.aggregate = mean, value.var = "value")
```

No obstante, la opción más sucinta es

```
dcast(iris.long, Species ~ variable, mean)
```

Ejercicio 8.2.2 Calcula la media de cada variable meteorológica de `airquality` por mes.

Para los ejercicios siguientes vamos a utilizar una tabla descargada del INE que contiene información sobre la situación laboral de los ciudadanos de todas las provincias de España por periodo, provincia y sexo.

```
paro <- read.table("data/paro.csv", header = T, sep = "\t")
```

Los valores que aparecen en la tabla son valores absolutos: es el número de personas en miles.

Los datos están en formato largo, pero se pueden pivotar, p.e., así:

```
paro.alt <- dcast(paro, Gender + Provinces + Periodo ~ Situation)
```

Ejercicio 8.2.3 Añade a `paro.alt` una columna adicional con la tasa de paro (desempleados entre población activa).

El ejercicio anterior demuestra cómo en ocasiones es conveniente crear un determinado tipo de formato ancho para operar sobre los datos más fácilmente. En concreto, para colocar en la misma fila valores sobre los que realizar operaciones; en nuestro caso, un cociente.

Ejercicio 8.2.4 Identifica las provincias y periodos en los que la tasa de paro masculina supera a la femenina.

Si se utilizan todas las variables que forman parte de la clave de una tabla en la fórmula de `dcast`, en cada casilla de la tabla resultante aparece una única observación. Si se omite una o más, comienzan a aparecer valores repetidos y eso permite utilizar funciones de agregación para realizar operaciones similares a un `group by` de SQL.

Ejercicio 8.2.5 Obtén totales para toda España (i.e., ignorando la provincia y el sexo) usando `dcast`.

No obstante, en una sección posterior veremos una manera alternativa, la recomendada, para realizar este tipo de agregaciones.

8.3. Resumen y referencias

En esta sección hemos aprendido a distinguir entre datos en formato largo (o arreglados) y ancho. Los dos tipos tienen sus ventajas e inconvenientes. Los datos arreglados son conceptualmente más simples de describir y comprender. Además, facilitan ciertas transformaciones, como las operaciones por bloques o el filtrado (selección de un determinado subconjunto de las filas).

Muchas transformaciones de datos implican operar sobre valores que, en una tabla arreglada, estarían en filas distintas. Para ello es necesario *yuxtaponerlos*, i.e., colocarlos en una misma fila. Para ello es necesario pivotar la tabla. Suele ser habitual volver a arreglar los datos después de realizadas estas transformaciones.

De hecho, existen flujos de trabajo de manipulación de datos que consisten en una secuencia de pivotaciones entre el formato largo y distintas versiones de formatos anchos en los que se realizan determinadas operaciones.

Estas operaciones de pivotación no son sencillas con otras herramientas habituales, como SQL o Spark. En R existe una función *de serie*, `reshape`, que permite realizar estas transformaciones. Sin embargo, no se recomienda su uso: es muy mejorable, particularmente, en lo concerniente a su usabilidad.

Debido a esas limitaciones, se introdujo en R el paquete `reshape2` (que es, a su vez, una evolución de un paquete anterior, `reshape`). Este paquete contiene esencialmente dos funciones fundamentales, `melt` y `dcast`, que son las que hemos tratado aquí.

El autor de este paquete teorizó más tarde sobre los datos en formatos anchos y largos en el artículo, muy recomendable, Tidy Data. Además, a raíz de ese artículo, desarrolló un paquete adicional, `tidyR`. El paquete `tidyR` tiene una función similar a `melt`, `gather` y otra similar a `dcast`, `spread`. En <http://www.milanor.net/blog/reshape-data-r-tidyr-vs-reshape2/> se comparan ambos paquetes.

8.4. Ejercicios adicionales

Ejercicio 8.4.1 Identifica las provincias en las que más crece la tasa de paro en 2014.

Ejercicio 8.4.2 Lee este fichero (puedes hacerlo pasándole directamente la URL a `read.table`) y dispónlo en formato largo.

Ejercicio 8.4.3 Lee este fichero que contiene datos de casos de tuberculosis por país y año partido por sexos (m/f) y grupos de edad (u es *desconocido*). Ponlo en formato largo descomponiendo las variables que combinan edad y sexo convenientemente. Después, calcula el número de casos:

- por país y año
- por país, año y sexo
- por país, año y grupo de edad

Ejercicio 8.4.4 Usa el conjunto de datos de los casos de tuberculosis y repite el ejercicio de la agregación pero solo para algún país concreto.

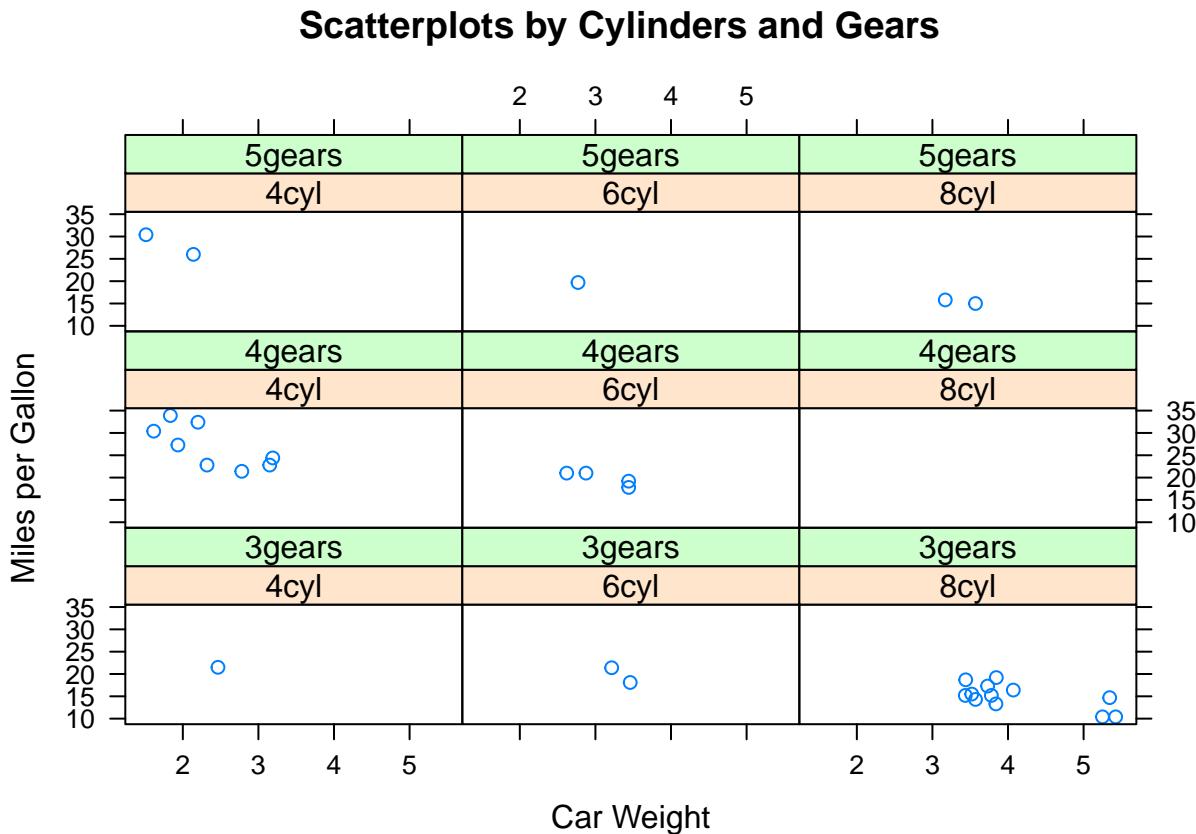
Capítulo 9

Introducción a ggplot2

R es un lenguaje para el análisis estadístico de datos. Como tal, destaca por sus gráficos. En la mayor parte de los lenguajes de programación, la capacidad de crear gráficos la proporcionan librerías adicionales ajenas a su núcleo. Sin embargo, en R, los gráficos son nativos.

Existen dos *motores* gráficos en R. Un motor gráfico es un conjunto de funciones que permiten realizar manipulaciones gráficas básicas: generar lienzos (o *canvas*), trazar líneas, dibujar puntos, etc. Un usuario de R no manipula generalmente esas funciones directamente: utiliza funciones *de alto nivel*, como `plot`. La función `plot` es la encargada de invocar esas funciones de bajo nivel que pintan los segmentos, círculos, etc. que conforman un gráfico estadístico, con sus ejes, sus etiquetas, etc.

Funciones de R tales como `plot`, `hist`, `barplot`, `boxplot` y otras se apoyan en el motor tradicional de R, que es suficiente para esos fines. Sin embargo, se queda corto para construir otro tipo de gráficos más avanzados. Por eso, en 2001, Paul Murrell desarrolló un motor gráfico alternativo, `grid`. Uno de sus objetivos era facilitar la generación en R de un tipo de gráficos conocidos como de Trellis, de celosía o de pequeños múltiples.



Los gráficos de Trellis permiten seguir el comportamiento de unas variables de interés a través de los distintos niveles de otras, disponiendo la información en una retícula que facilita el descubrimiento de patrones por inspección visual. El gráfico anterior muestra la relación entre el peso y el consumo de gasolina de una serie de vehículos en función de su número de cilindros y el número de marchas. La relación es más evidente usando un gráfico de Trellis que, por ejemplo, usando colores (o formas) para representar el número de cilindros o marchas en un único gráfico de dispersión.

Hay muchas funciones y paquetes que crean gráficos apoyándose en el motor gráfico tradicional. Otros, utilizan `grid`. Dos de los más conocidos son `lattice` (con el que está generado el gráfico anterior) y `ggplot2`. De hecho, `lattice` y `ggplot2` se solapan funcionalmente y la mayoría de los usuarios de R se decantan por uno u otro y lo usan predominantemente.

`ggplot2` es unos años posterior a `lattice` pero, a pesar de ello, más popular. `ggplot2` es una implementación de las ideas recogidas en el artículo de *The Language of Graphics* escrito por Leland Wilkinson y sus coautores en 2000. Ese artículo recogía una serie de ideas novedosas sobre qué es la representación gráfica de información estadística y cómo debería hacerse. Los gráficos tradicionales de R tienen un importante elemento de adhoquismo: las funciones para representar diagramas de dispersión, de cajas, de barras, etc. utilizan datos con distintos formatos, tienen parámetros no siempre coincidentes en nombre, etc. Lo revolucionario de planteamiento del artículo es la de poner de manifiesto que todos esos tipos de gráficos (y otros) pueden generarse mediante un lenguaje más o menos regular, con su sintaxis, su semiótica, etc. De la misma manera que el lenguaje natural organiza sonidos mediante ciertas reglas comunes, conocidas y regulares para generar mensajes con significado, es posible construir una serie de reglas comunes, conocidas y regulares para crear representaciones visuales de datos de interés estadístico.

De ese lenguaje, implementado en el paquete `ggplot2`, se ocupan las siguientes secciones.

9.1. Una primera toma de contacto

En primer lugar vamos a cargar los paquetes `ggplot2` y `reshape2`:

```
library(ggplot2)
library(reshape2)
```

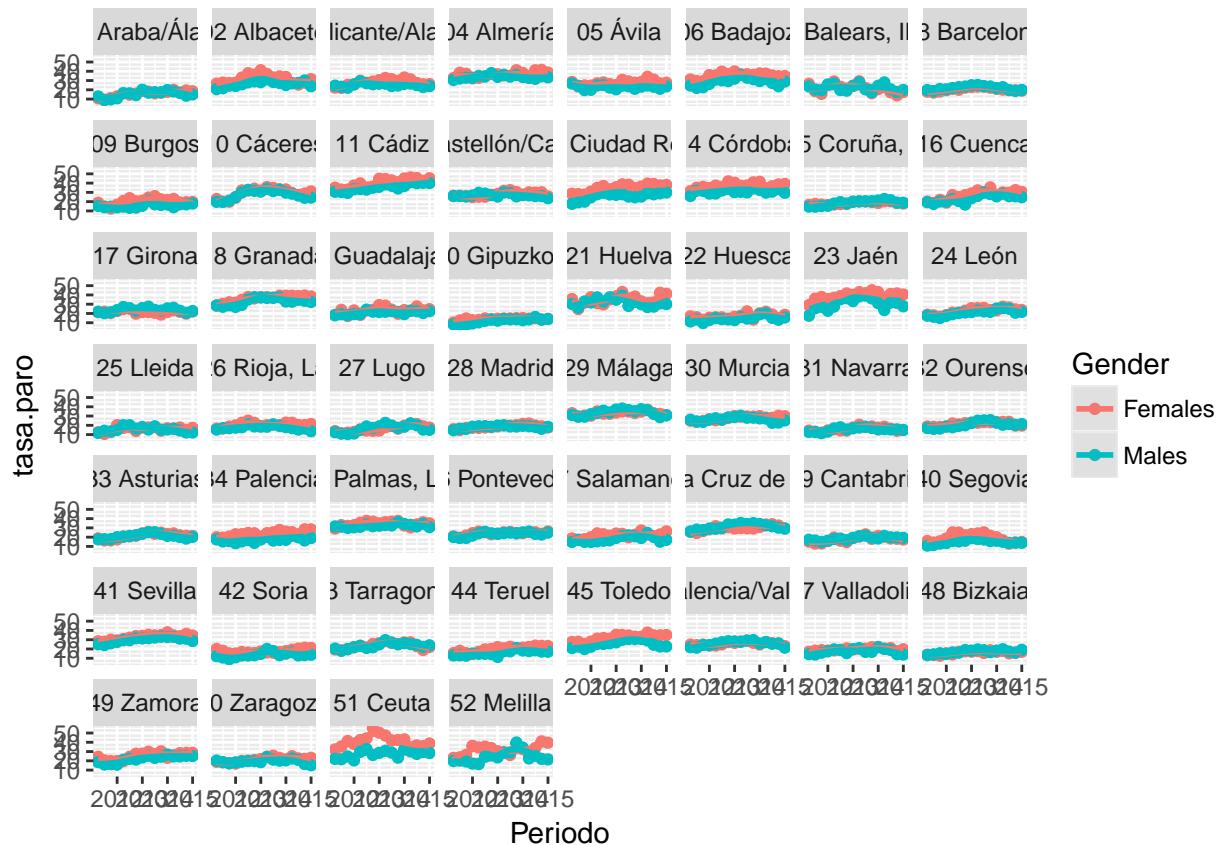
Para el primer gráfico con `ggplot2` vamos a leer y preprocesar de nuevo los datos del paro del INE:

```
paro <- read.table("data/paro.csv", header = T, sep = "\t")
paro <- dcast(paro, Gender + Provinces ~ Periodo ~ Situation)
paro$tasa.paro <- 100 * paro$unemployed / paro$active

paro$Periodo <- as.character(paro$Periodo)
paro$Periodo <- gsub("Q1", "-03-31", paro$Periodo)
paro$Periodo <- gsub("Q2", "-06-30", paro$Periodo)
paro$Periodo <- gsub("Q3", "-09-30", paro$Periodo)
paro$Periodo <- gsub("Q4", "-12-31", paro$Periodo)
paro$Periodo <- as.Date(paro$Periodo)
```

La expresión fundamental para crear un gráfico con `ggplot2` es:

```
ggplot(paro, aes(x = Periodo, y = tasa.paro, col = Gender)) +
  geom_point() + geom_smooth(alpha = 0.2) +
  facet_wrap(~ Provinces)
```



La expresión anterior combina varios elementos que discutiremos con detalle más adelante:

- Datos: siempre una tabla.

- Estéticas, que asocian a elementos representables gráficamente (la posición `x` e `y`, el color, etc.) columnas de la tabla de datos.
- Geometrías (o capas): puntos, rectas, histogramas, densidades, etc. También se llaman capas porque pueden superponerse.
- Facetas: parten un gráfico en sublienzo preservando las escalas (pequeños múltiplos)

9.2. Elementos de un gráfico en ggplot2

Un gráfico en `ggplot2` se construye combinando una serie de elementos básicos y comunes a muchos tipos de gráficos distintos mediante una sintaxis sencilla. Esta sección describe esa sintaxis y los elementos que articula.

9.2.1. Datos

Uno de los elementos más importantes de un gráfico son los datos que se quieren representar. Una particularidad de `ggplot2` es que solo acepta un tipo de datos: `data.frames`. Otras funciones gráficas (p.e., `hist`) admiten vectores, listas u otro tipo de estructuras. `ggplot2` no.

```
p <- ggplot(iris)
```

El código anterior crea un objeto, `p` que viene a ser un protográfico: contiene los datos que vamos a utilizar, los del conjunto de datos `iris`. Obviamente, el código anterior es insuficiente para crear un gráfico: aún no hemos indicado qué queremos hacer con `iris`.

9.2.2. Estéticas

En un conjunto de datos hay columnas: edad, altura, ingresos, temperatura, etc. En un gráfico hay, en la terminología de `ggplot2`, *estéticas*. Estéticas son, por ejemplo, la distancia horizontal o vertical, el color, la forma (de un punto), el tamaño (de un punto o el grosor de una línea), etc. Igual que al hablar asociamos a un conjunto de sonidos (p.e., m-e-s-a) un significado (el objeto que conocemos como mesa), al realizar un gráfico asociamos a elementos sin significado propio (p.e., los colores) uno: el que corresponde a una columna determinada de los datos.

En `ggplot2`, dentro del lenguaje de los gráficos que implementa, es muy importante esa asociación explícita de significados a significantes, es decir, de columnas de datos a *estéticas*.

En el código

```
p <- p + aes(x = Petal.Length, y = Petal.Width, colour = Species)
```

se están añadiendo a `p` información sobre las estéticas que tiene que utilizar y qué variables de `iris` tiene que utilizar:

- La distancia horizontal, `x`, vendrá dada por la longitud del pétalo.
- La distancia vertical, `y`, por su anchura.
- El color, por la especie.

Hay que hacer notar la sintaxis del código anterior, bastante particular y propia del paquete `ggplot2`. Al *protográfico* se le han sumado las estéticas. En las secciones siguientes se le *sumarán* otros elementos adicionales. Lo importante es recordar cómo la suma es el signo que combina los elementos que componen el lenguaje de los gráficos.

De todos modos, es habitual combinar ambos pasos en una única expresión

```
p <- ggplot(iris, aes(x = Petal.Length, y = Petal.Width, colour = Species))
```

El objeto `p` resultante aún no es un gráfico ni se puede representar. Le faltan capas, que es el objeto de la siguiente sección. No obstante, se puede inspeccionar así:

```
summary(p)
```

```
## data: Sepal.Length, Sepal.Width, Petal.Length, Petal.Width,
##   Species [150x5]
## mapping: x = Petal.Length, y = Petal.Width, colour = Species
## facetting: <ggproto object: Class FacetNull, Facet>
##   compute_layout: function
##   draw_back: function
##   draw_front: function
##   draw_labels: function
##   draw_panels: function
##   finish_data: function
##   init_scales: function
##   map: function
##   map_data: function
##   params: list
##   render_back: function
##   render_front: function
##   render_panels: function
##   setup_data: function
##   setup_params: function
##   shrink: TRUE
##   train: function
##   train_positions: function
##   train_scales: function
##   vars: function
##   super: <ggproto object: Class FacetNull, Facet>
```

Ahí están indicados los datos que va a utilizar y la relación (o *mapeo*) entre estéticas y columnas de los datos.

¿Cuántas estéticas existen? Alrededor de una docena, aunque se utilizan, generalmente, menos:

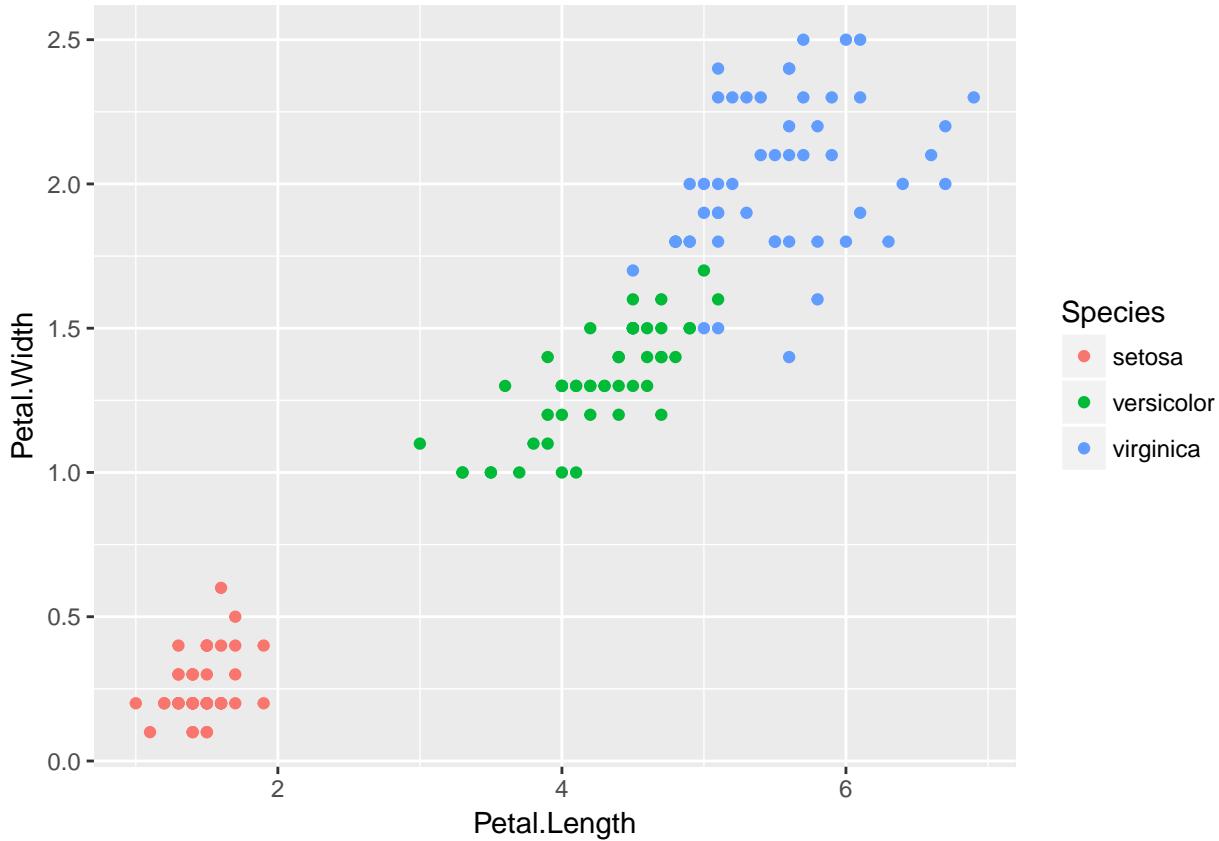
- `x` e `y`, distancias horizontal y vertical.
- `colour`, para el color.
- `size`, para el tamaño.
- `shape`, que indica la forma de los puntos (cuadrados, triángulos, etc.) de los puntos o del trazo (continuo, punteado) de las líneas.
- `alpha` para la transparencia: los valores más altos tendrían formas opacas y los más bajos, casi transparentes. De ahí la utilizad del canal alfa: da peso e importancia a las observaciones que la merecen.
- `fill`, para el color de relleno de las formas sólidas (barras, etc.).

Hay que advertir que no todas las *estéticas* tienen la misma potencia en un gráfico. El ojo humano percibe fácilmente longitudes distintas. Pero tiene problemas para comparar áreas (que es lo que regula la estética `size`) o intensidades de color. Se recomienda usar las estéticas más potentes para representar las variables más importantes.

9.2.3. Capas

Las capas (o `geoms` para `ggplot2`) son los verbos del lenguaje de los gráficos. Indican qué hacer con los datos y las estéticas elegidas, cómo representarlos en un lienzo. Y, en efecto, el siguiente código crea el correspondiente gráfico.

```
p <- p + geom_point()
p
```

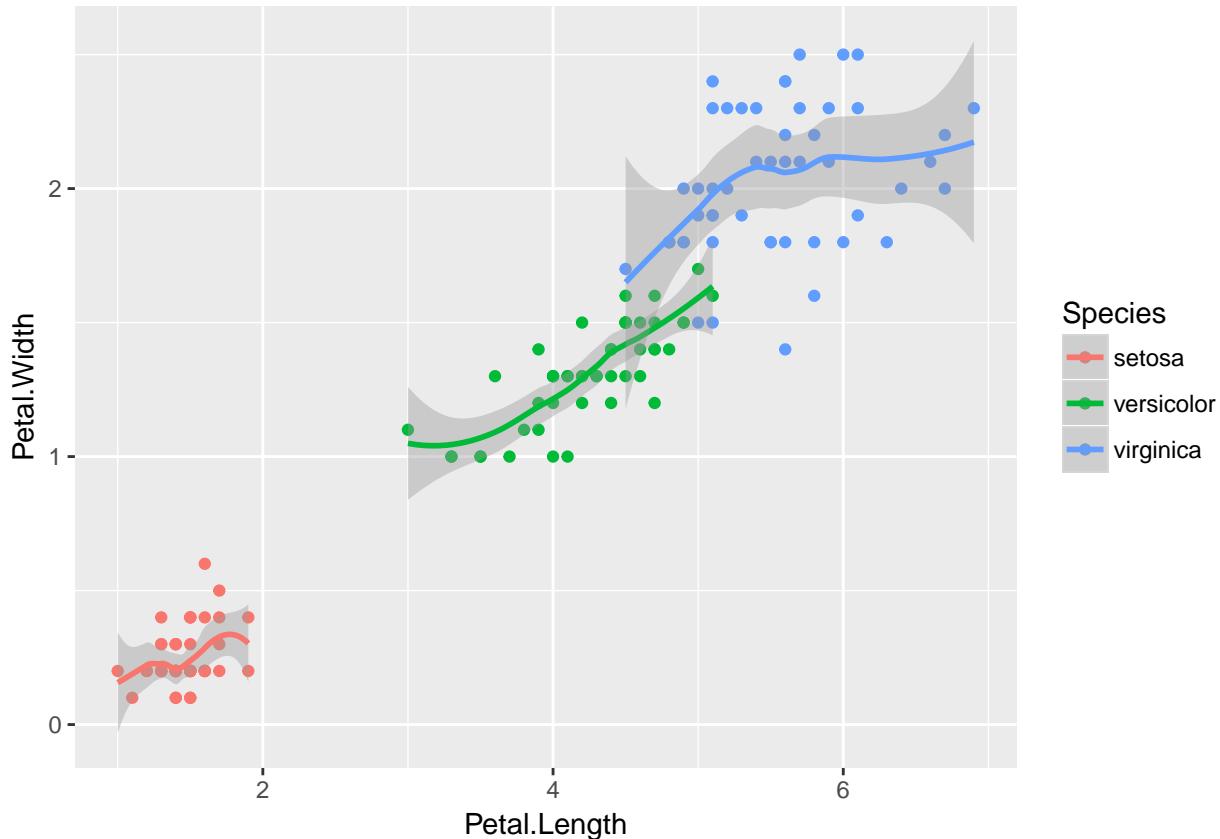


Una vez añadida una capa al gráfico, este puede pintarse (que es lo que ocurre al llamar a `p`). Se obtiene el mismo resultado haciendo, en una única línea,

```
ggplot(iris, aes(x = Petal.Length, y = Petal.Width, colour = Species)) + geom_point()
```

Una característica de las capas, y de ahí su nombre, es que pueden superponerse. Por ejemplo,

```
ggplot(iris, aes(x = Petal.Length, y = Petal.Width, colour = Species)) +
  geom_point() + geom_smooth()
```



añade al gráfico una curva suavizada (con sus intervalos de confianza en gris).

Existen muchos tipos de capas. Los más usuales son `geom_point`, `geom_line`, `geom_histogram`, `geom_bar` y `geom_boxplot`. Pero hay más. En la página <http://docs.ggplot2.org/current/> se muestra una lista de los disponibles (en la versión más actualizada de ggplot2). En esa página se indica qué `geom` hay que utilizar en función de una representación esquemática del tipo de gráfico que se quiere construir. Además, hay capas específicas que exigen estéticas especiales. Para algunas tiene sentido, p.e., `shape`. Para otras no. Esas especificidades están indicadas en dicha página, que es más útil que la ayuda general de R.

Una vez creado un gráfico, es posible exportarlo a png, jpg, etc. La función `ggsave` guarda en un fichero el último gráfico generado con `ggplot2`. Lo hace, además, en el formato indicado en el nombre del fichero que se quiere generar. Así,

```
ggplot(iris, aes(x = Petal.Length, y = Petal.Width, colour = Species)) + geom_point()
ggsave("mi_grafico.png")
```

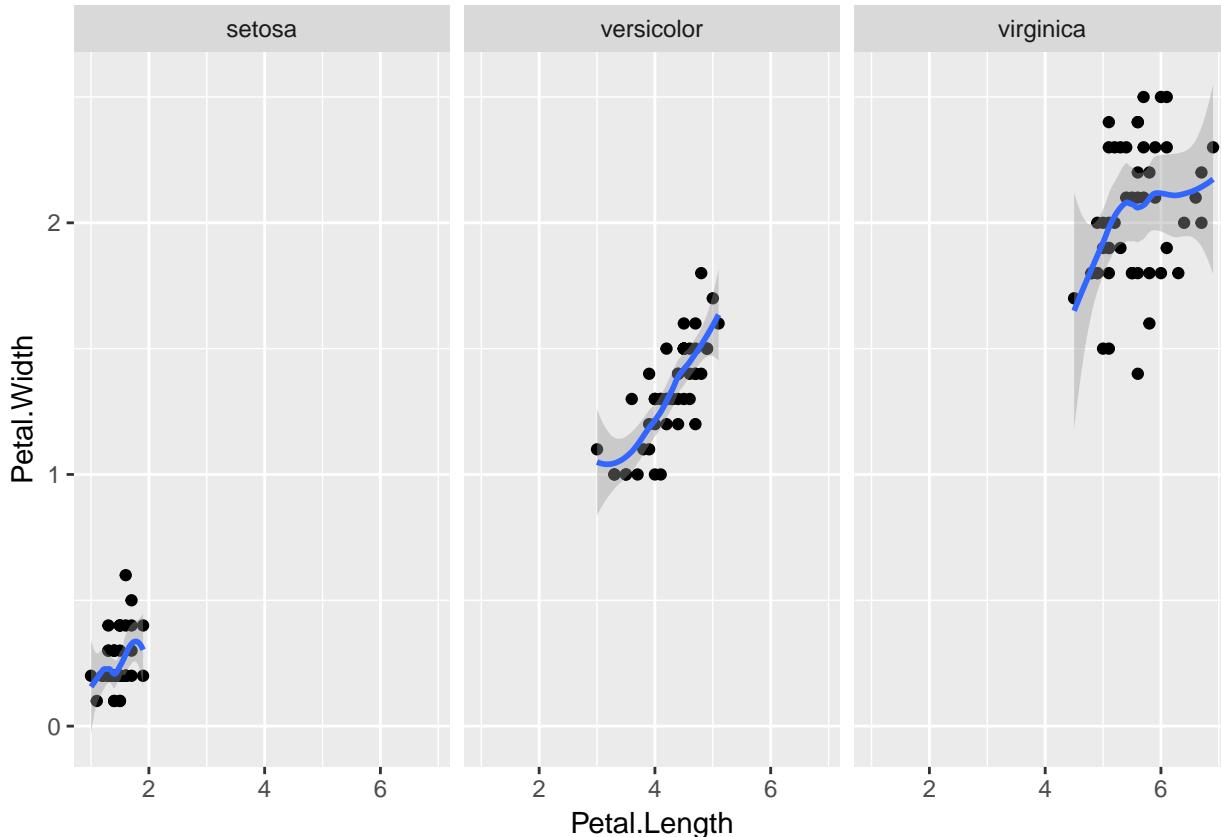
guarda la figura creada en la primera línea en formato png en el fichero `mi_grafico.png` del directorio de trabajo.

9.2.4. Facetas

Muchos de los gráficos que pueden generarse con los elementos anteriores pueden reproducirse sin mucho esfuerzo (exceptuando, tal vez, cuestiones de aspecto) usando los gráficos tradicionales de R, pero no los que usan facetas.

Las facetas implementan los gráficos de Trellis mencionados antes. Por ejemplo,

```
ggplot(iris, aes(x = Petal.Length, y = Petal.Width)) +
  geom_point() + geom_smooth() +
  facet_grid(~ Species)
```

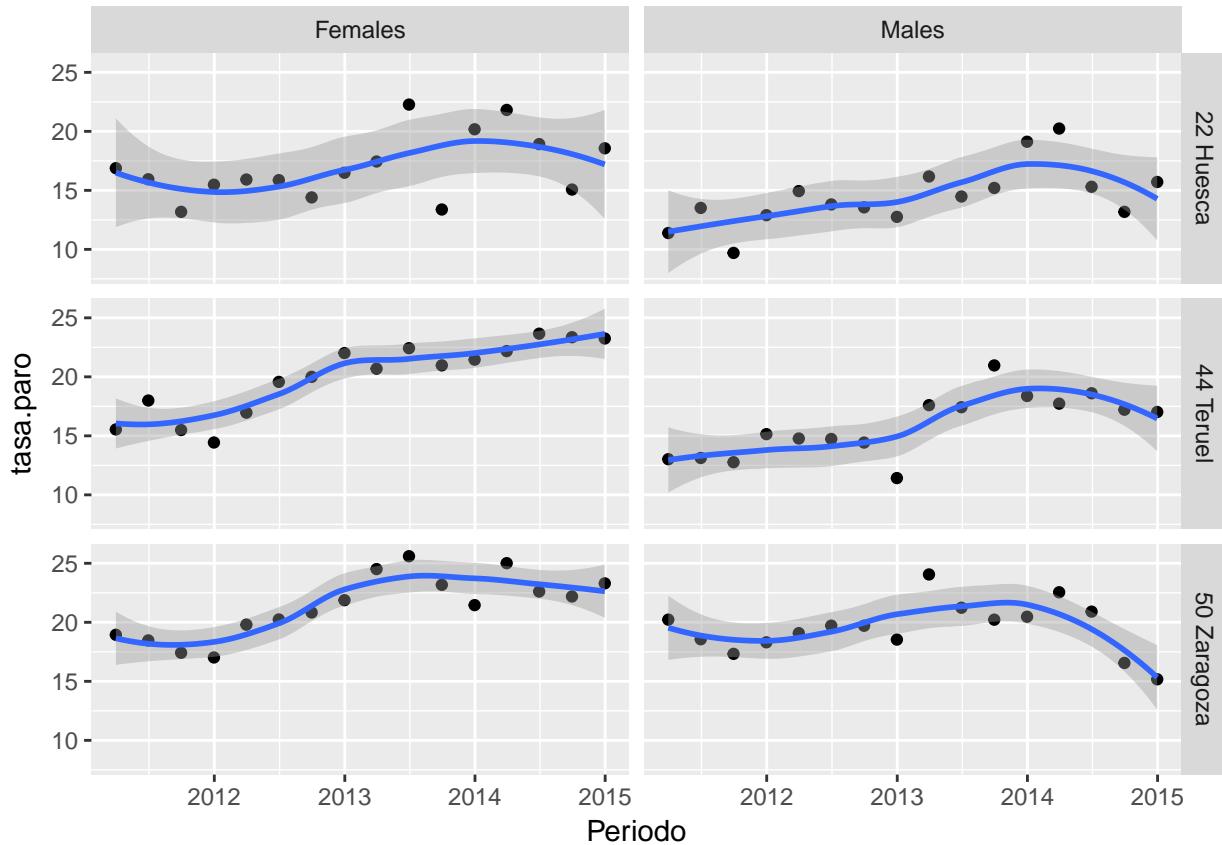


crea tres gráficos dispuestos horizontalmente que comparan la relación entre la anchura y la longitud del pétalo de las tres especies de iris. Una característica de estos gráficos, que es crítica para poder hacer comparaciones adecuadas, es que comparten ejes.

Los gráficos podrían disponerse verticalmente reemplazando `facet_grid(~ Species)` por `facet_grid(Species ~ .)` en el código anterior. Además, se puede subdividir el lienzo por dos (yo más!) variables así:

```
tmp <- paro[paro$Provinces %in% c("50 Zaragoza", "22 Huesca", "44 Teruel"),]
ggplot(tmp, aes(x = Periodo, y = tasa.paro)) +
  geom_point() + geom_smooth() +
  facet_grid(Provinces~Gender)
```

```
## `geom_smooth()` using method = 'loess'
```



En caso de haber muchas categorías (p.e., provincia), puede usarse la función `facet_wrap` para distribuir las subgráficas en una cuadrícula.

9.2.5. Más sobre estéticas

Las estéticas se pueden etiquetar con la función `labs`. Además, se le puede añadir un título al gráfico usando la función `ggtitle`. Por ejemplo, en el gráfico anterior se pueden reetiquetar los ejes y la leyenda haciendo

```
p <- p + ggtitle("Petal length and width") +
  labs(x = "Petal length",
       y = "Petal width",
       colour = "Species")
```

9.2.6. Temas

Los *temas* de `ggplot2` permiten modificar aspectos estéticos del gráfico que no tienen que ver con los datos en sí. Eso incluye los ejes, etiquetas, colores de fondo, el tamaño de los márgenes, etc. No es habitual (y se desaconseja a los usuarios menos expertos) tener que alterar los temas que `ggplot2` usa por defecto. Solo se vuelve necesario cuando los gráficos tienen que adecuarse a una imagen corporativa o atenerse a algún criterio de publicación exigente.

Un tema es una colección de elementos (p.e., `panel.background`, que indica el color, transparencia, etc. del lienzo sobre el que se representa el gráfico) modificables. El tema que usa `ggplot2` por defecto es `theme_grey`. Al escribir `theme_grey()` en la consola de R, se muestran alrededor de cuarenta elementos modificables y sus atributos tal y como los define dicho tema.

¿Qué se puede hacer con los temas? Una primera opción es elegir otro. Por ejemplo, se puede reemplazar el habitual por otros disponibles en el paquete como `theme_bw` (o `theme_classic`) haciendo

```
p <- p + theme_bw()
```

Es posible usar tanto los temas que incluye `ggplot2` por defecto como otros creados por la comunidad. Algunos, por ejemplo, tratan de imitar el estilo de publicaciones reconocidas como The Economist o similares. Algunos están recogidos en paquetes como, por ejemplo, `ggthemes`.

Alternativamente (o adicionalmente), es posible modificar un tema dado en un gráfico. Por ejemplo, haciendo

```
p <- p +
  theme_bw() +
  theme(
    panel.background = element_rect(fill = "lightblue"),
    panel.grid.minor = element_line(linetype = "dotted")
  )
```

se está modificando el atributo de color del lienzo de un gráfico y el tipo de la línea con que se dibuja la malla.

Finalmente, es posible construir temas propios y personalizados. Aunque no es un proceso complicado, los detalles quedan fuera del alcance de esta introducción.

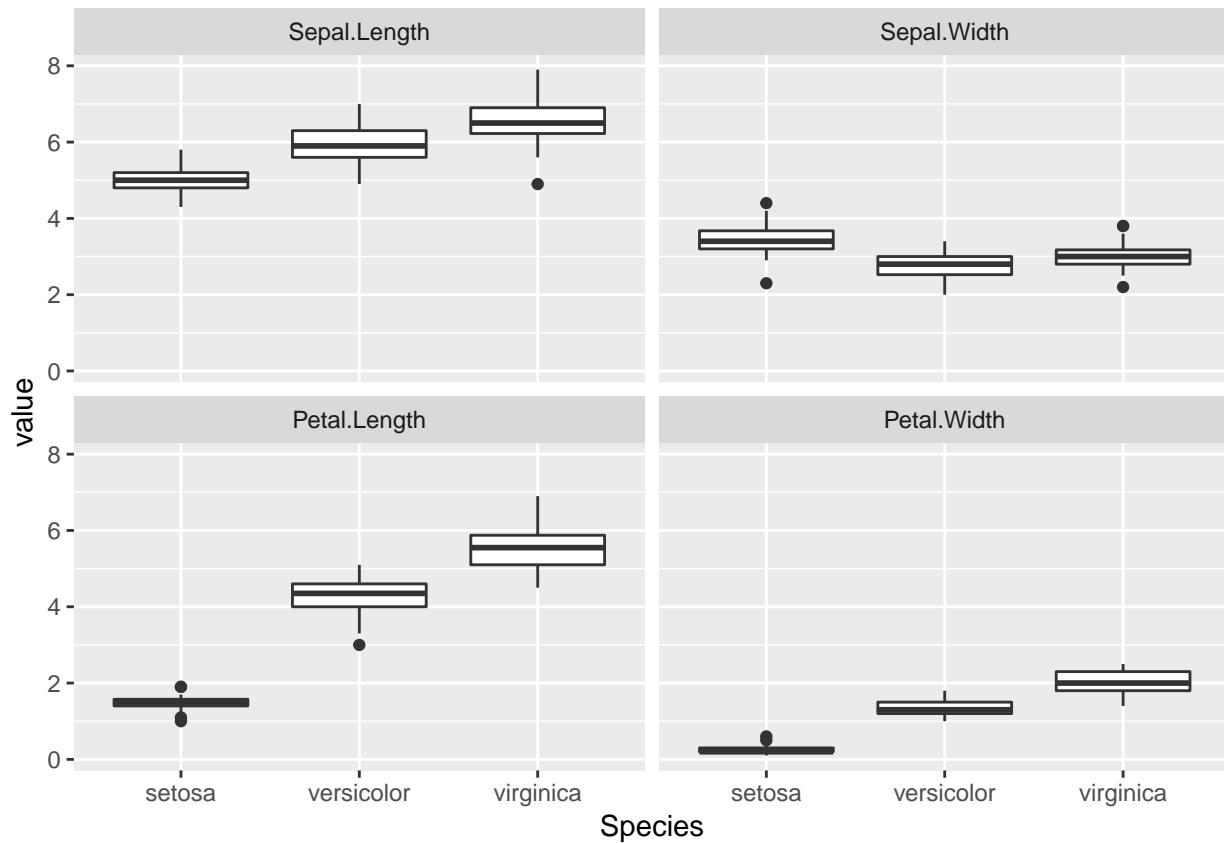
9.3. Ejemplos

En esta sección se van a explorar algunos de los gráficos estadísticos más básicos.

9.3.1. Diagramas de cajas (y de violín)

Los diagramas de caja (*boxplots*) describen de manera cruda la distribución de una variable continua en función de una discreta. En el ejemplo que aparece a continuación, se explora el conjunto de datos `iris` que contiene 50 observaciones de características métricas de cada una de las tres subespecies de iris, una flor. Es un conjunto de datos de larga tradición en estadística y se recopiló para ilustrar algoritmos de clasificación, es decir, cómo crear criterios para distinguir los tres tipos de iris.

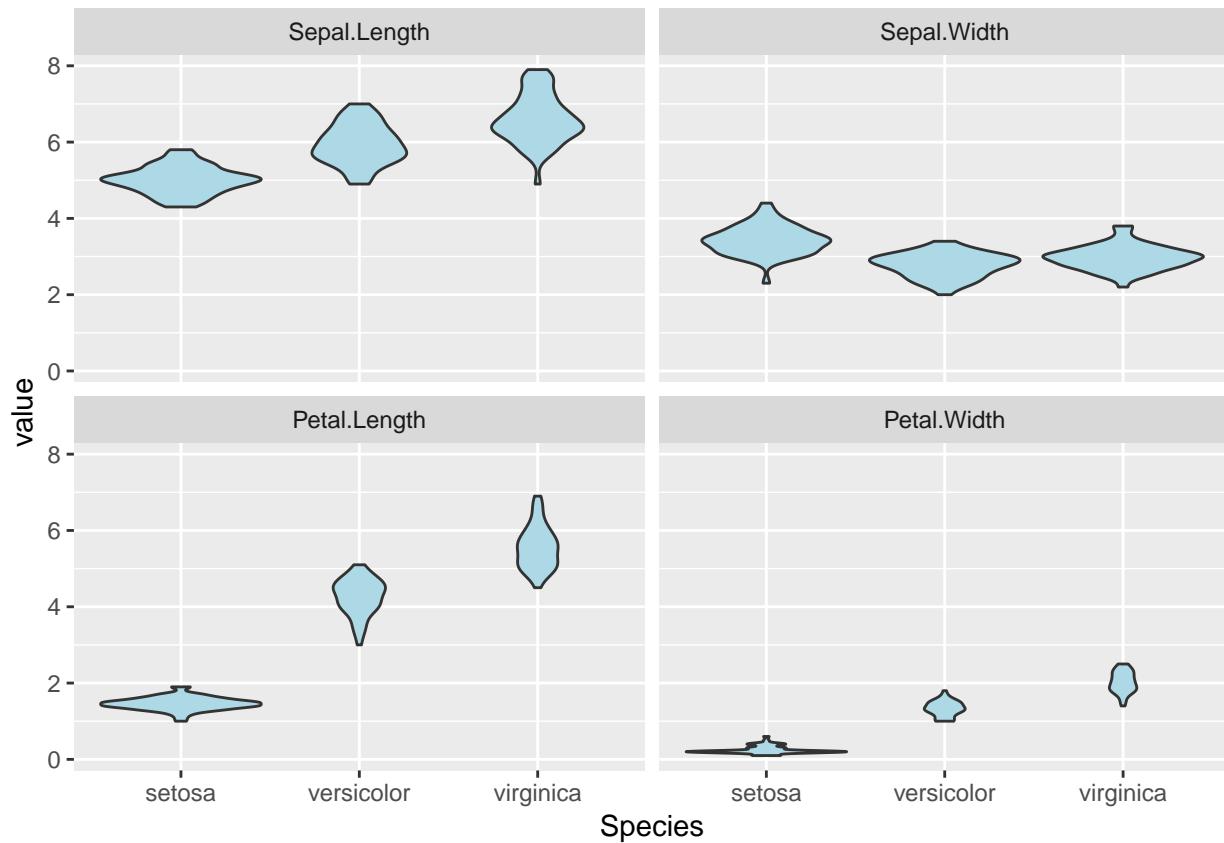
```
library(reshape2)
tmp <- melt(iris)
ggplot(tmp, aes(x = Species, y = value)) + geom_boxplot() + facet_wrap(~ variable)
```



El gráfico utiliza las facetas para crear cuatro paneles, uno por variable. Así resume rápidamente la información contenida en el conjunto de datos y revela eficazmente los patrones que encierra: por ejemplo, cómo la especie setosa tiene el pétalo sensiblemente más estrecho y corto que las otras dos. Este tipo de gráficos son fundamentales como herramienta exploratoria previa a la aplicación de técnicas de análisis estadístico (discriminante, en este caso).

Los gráficos de cajas son muy básicos: apenas muestran cinco puntos característicos de una distribución: la mediana, los extremos y los cuartiles. Son herencia de una época en que apenas había recursos, principalmente informáticos, para realizar representaciones más sofisticadas. Una versión moderna de los gráficos de cajas es la de gráficos de violín. Como los de cajas, resumen la distribución de las variables. Pero en lugar de una representación sucinta, tratan de dibujar la distribución real de los datos: son verdaderos gráficos de densidad, solo que dispuestos de otra manera para facilitar la comparación.

```
ggplot(tmp, aes(x = Species, y = value)) +
  geom_violin(fill = "lightblue") +
  facet_wrap(~ variable)
```



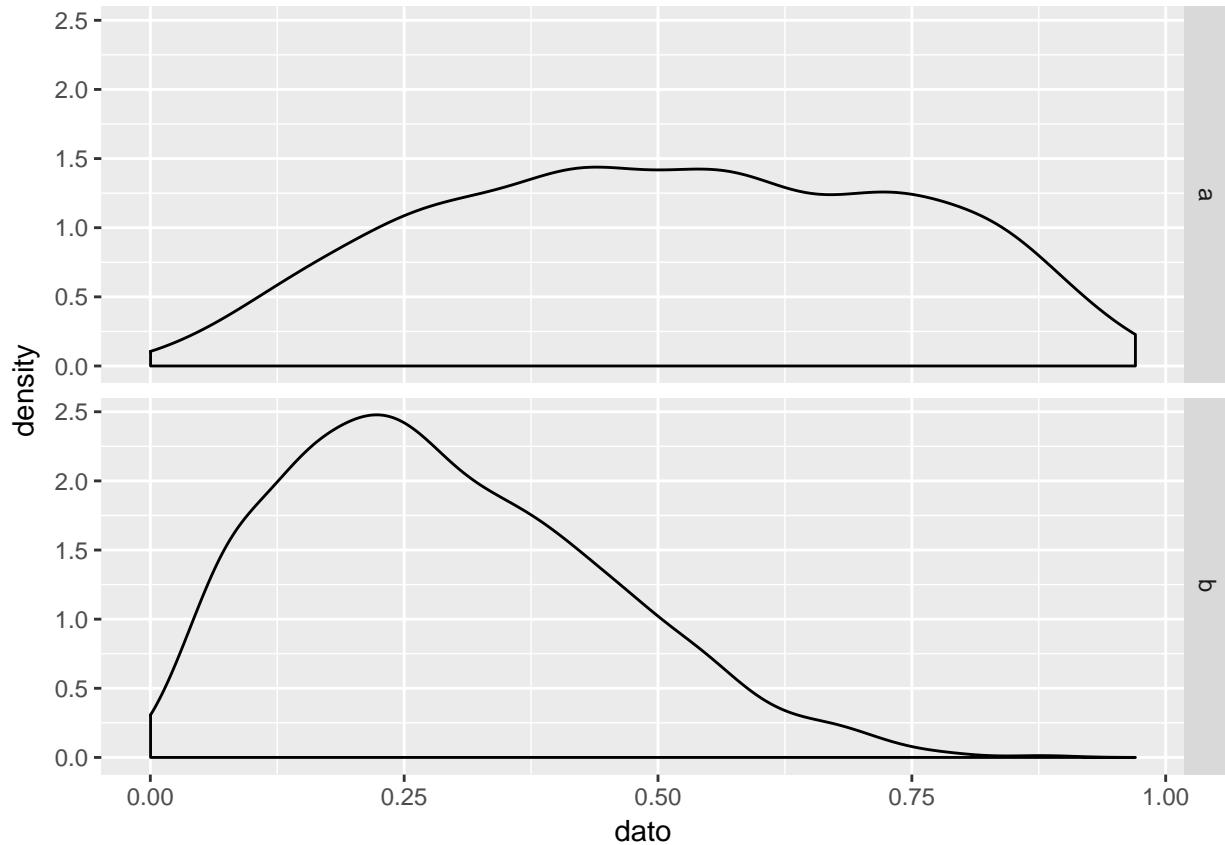
9.3.2. Comparación de dos densidades

El análisis de datos exige en ocasiones comparar dos distribuciones continuas. Se pueden usar gráficos de cajas o de violín, como arriba, pero también se puede dibujar la distribución completa como en el siguiente gráfico:

```
# datos (simulados)
a <- rbeta(1000, 2, 2)
b <- rbeta(2000, 2, 5)

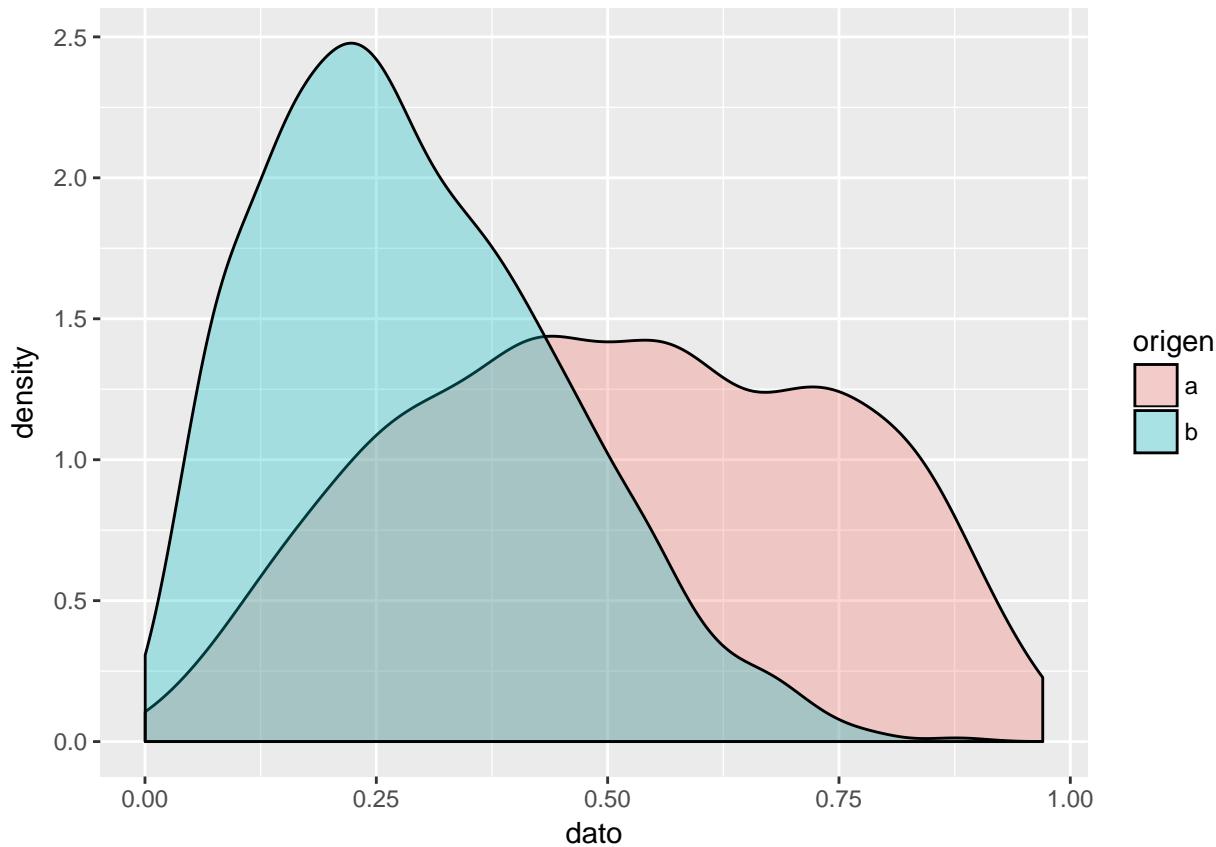
# construcción de un dataframe a partir de ellos
tmp <- rbind(data.frame(origen = "a", dato = a),
              data.frame(origen = "b", dato = b))

ggplot(tmp, aes(x = dato)) + geom_density() + facet_grid(origen ~ .)
```



Alternativamente, se pueden solapar ambas distribuciones. El uso del parámetro `alpha`, que controla la transparencia, es fundamental en este caso:

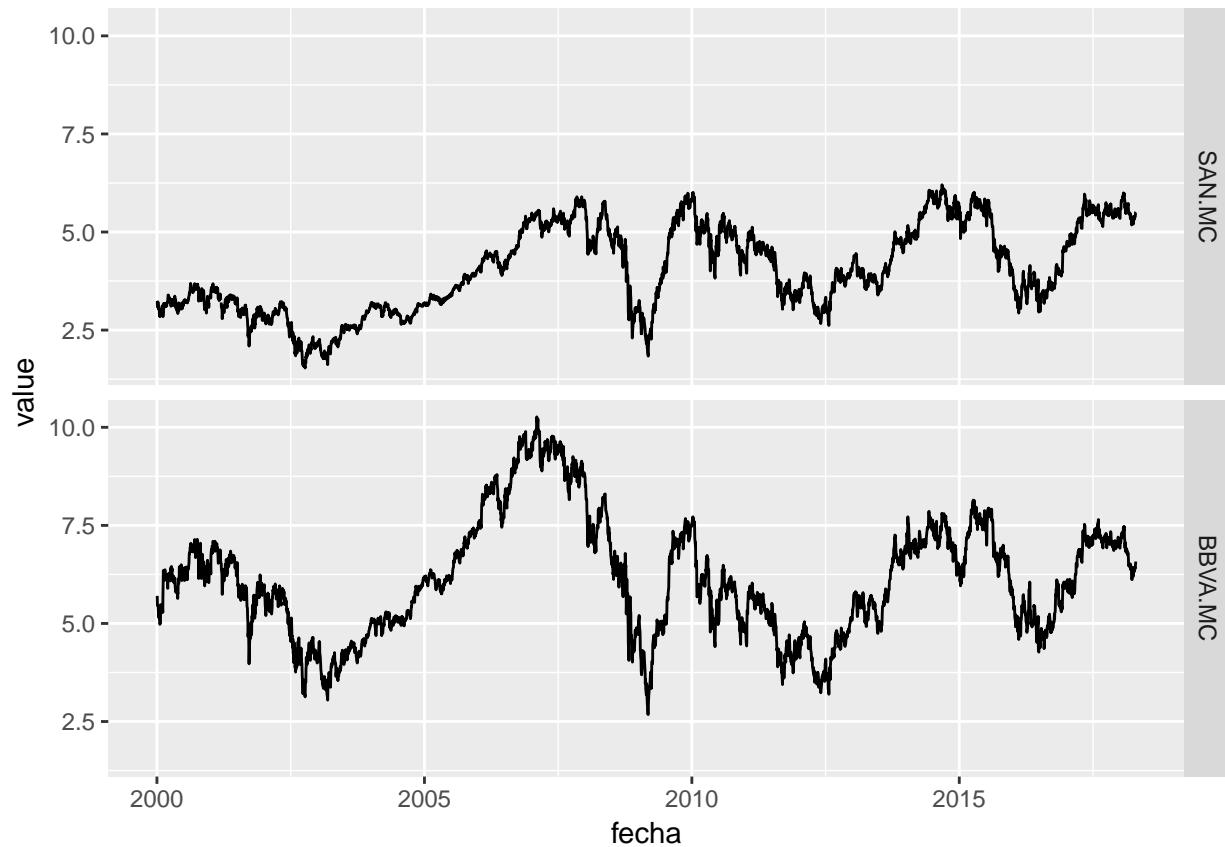
```
ggplot(tmp, aes(x = dato, fill = origen)) + geom_density(alpha = 0.3)
```



9.3.3. Series temporales

ggplot2 entiende ciertos tipos de datos particulares, como por ejemplo, series temporales. En el siguiente ejemplo se descargan las cotizaciones bursátiles de dos banco españoles directamente de Yahoo! Finance y se representan gráficamente:

```
library(tseries)
library(zoo)
library(reshape2)
# función para descargar las cotizaciones
cotizaciones <- function(valor){
  res <- get.hist.quote(valor, provider = "yahoo",
                        quote = "AdjClose", quiet = TRUE)
  colnames(res) <- valor
  res
}
# combinación de ambas series temporales
res <- merge(cotizaciones("SAN.MC"),
             cotizaciones("BBVA.MC"))
# construcción de un dataframe con un índice de tipo fecha
res <- data.frame(fecha = index(res), as.data.frame(res))
res <- melt(res, id.var = "fecha")
ggplot(res, aes(x = fecha, y = value)) + geom_line() + facet_grid(variable ~ .)
```



Una versión alternativa del gráfico es la que superpone las series usando colores para distinguirlas:

```
ggplot(res, aes(x = fecha, y = value, colour = variable)) + geom_line() +  
  labs(colour = "valor", y = "cotización")
```



9.4. Resumen y referencias

9.5. Ejercicios adicionales

Ejercicio 9.5.1 Construye un diagrama de cajas de las temperaturas en NY por mes (sin facetas). Nota: ten en cuenta que `ggplot` se confunde cuando la estética `x` del histograma no es categórica; por lo tanto, tendrás que convertir la variable mes en categórica (usando `factor`).

Ejercicio 9.5.2 Construye un histograma de las temperaturas en NY por mes (con facetas).

Ejercicio 9.5.3 Prueba con los gráficos de violín (que son una mezcla de los dos anteriores).

Ejercicio 9.5.4 Superpón las distribuciones de las temperaturas de NY por mes como en el ejemplo anterior de la superposición de dos densidades (o como aquí).

Ejercicio 9.5.5 Haz gráficos con tus propios datos.

Capítulo 10

Introducción a ggmap

Con `ggplot2` pueden construirse muchos tipos de gráficos de interés estadístico. Pero sus autores quisieron trasladar la arquitectura del paquete a otro ámbito: el de la representación de información georreferenciada. `ggplot2` permite representar información geográfica (puntos, segmentos, etc.): basta con que las estéticas `x` e `y` se correspondan con la longitud y la latitud de los datos. Lo que permite hacer `ggmap` es, en esencia, añadir a los gráficos ya conocidos una capa cartográfica adicional. Para eso usa recursos disponibles en la *web* a través de APIs (de Google y otros).

Un ejemplo sencillo ilustra los usos de `ggmap`. En primer lugar, se carga (si se ha instalado previamente) el paquete:

```
library(ggmap)
```

Existen varios proveedores que proporcionan APIs de geolocalización. Uno de ellos es Google: dado el nombre más o menos normalizado de un lugar, la API de Google devuelve sus coordenadas. Este servicio tiene una versión gratuita que permite realizar un determinado número de consultas diarias (2500 actualmente); para usos más intensivos, es necesario adquirir una licencia. La función `geocode` encapsula la consulta a dicha API y devuelve un objeto (un `data.frame`) que contiene las coordenadas del lugar de interés:

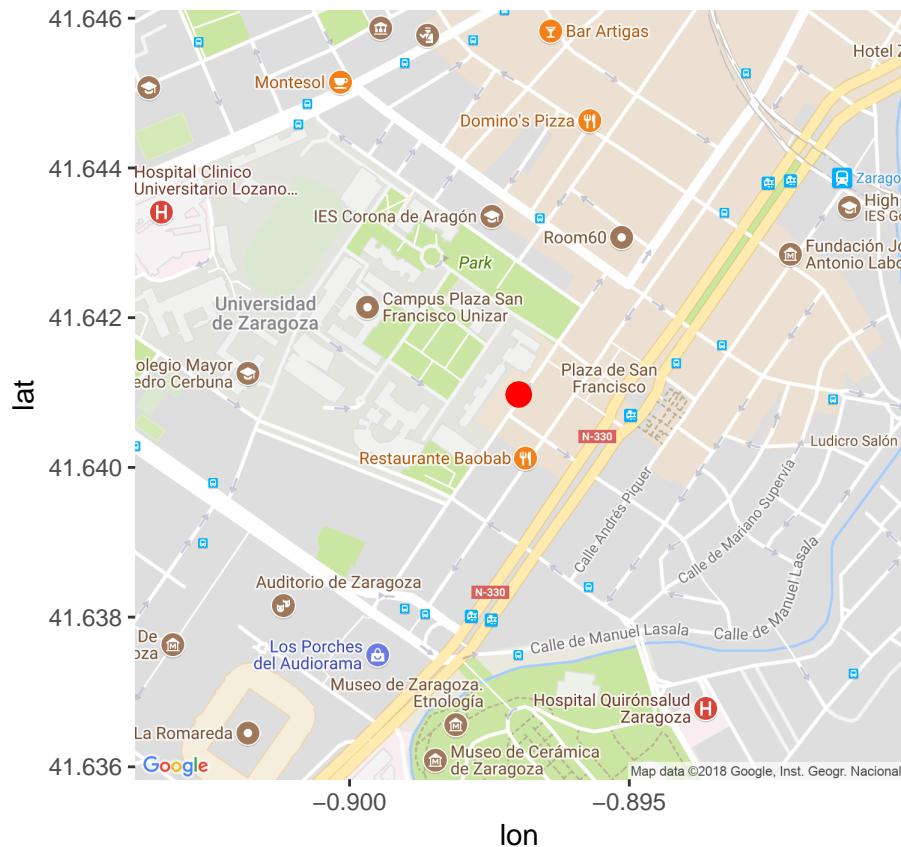
```
unizar <- geocode('Calle de Pedro Cerbuna 12, Zaragoza, España',
                   source = "google")
```

La función `get_map` consulta otro servicio de información cartográfica (GoogleMaps en el ejemplo siguiente) y descarga un mapa (que es, esencialmente, una imagen *raster*). La función exige una serie de argumentos: el nivel de `zoom`, si se quiere un mapa de carreteras o del terreno, etc. Son, de hecho, los parámetros que uno puede manipular con los controles de la interfaz habitual de GoogleMaps.

```
map.unizar <- get_map(location = as.numeric(unizar),
                       color = "color",
                       maptype = "roadmap",
                       scale = 2,
                       zoom = 16)
```

Es obvio que para poder invocar las dos funciones anteriores hace falta una conexión a internet. Sin embargo, el resto de las operaciones que se van a realizar se ejecutan localmente. Se puede, por ejemplo, representar el mapa directamente (haciendo `ggmap(map.unizar)`). O bien se puede marcar sobre él el punto de interés:

```
ggmap(map.unizar) + geom_point(aes(x = lon, y = lat),
                                 data = unizar, colour = 'red',
                                 size = 4)
```



Como veremos a continuación, no estamos limitados a representar un único punto: `unizar` podría ser una tabla con más de una fila y todos los puntos se representarían sobre el mapa.

Como puede apreciarse, la sintaxis es similar a la de `ggplot2`. Una diferencia notables que, ahora, los datos se pasan en la capa, es decir, en este caso, en la función `geom_point`.

10.1. Funciones de `ggmap`

`ggmap` incluye muchas funciones que pueden clasificarse en tres categorías amplias:

- Funciones para obtener mapas (de diversos tipos y de distintos orígenes: Google, Stamen, OpenStreetMap)
- Funciones que utilizan APIs de Google y otros. Por ejemplo, `geocode`, `reversegeocode` y `route` consultan la información que tienen distintos proveedores de servicios vía API sobre las coordenadas de un determinado lugar; indican el lugar al que se refieren unas coordenadas y, finalmente, encuentran rutas entre dos puntos. Es conveniente recordar que las consultas a los servicios de Google Maps exige la aceptación de las condiciones de uso y que existe un límite diario en el número de consultas gratuitas.
- Funciones que pintan mapas y que representan determinados elementos adicionales (puntos, segmentos, etc.) en mapas.

En esta sección se van a presentar los tres tipos de funciones de `ggmap`.

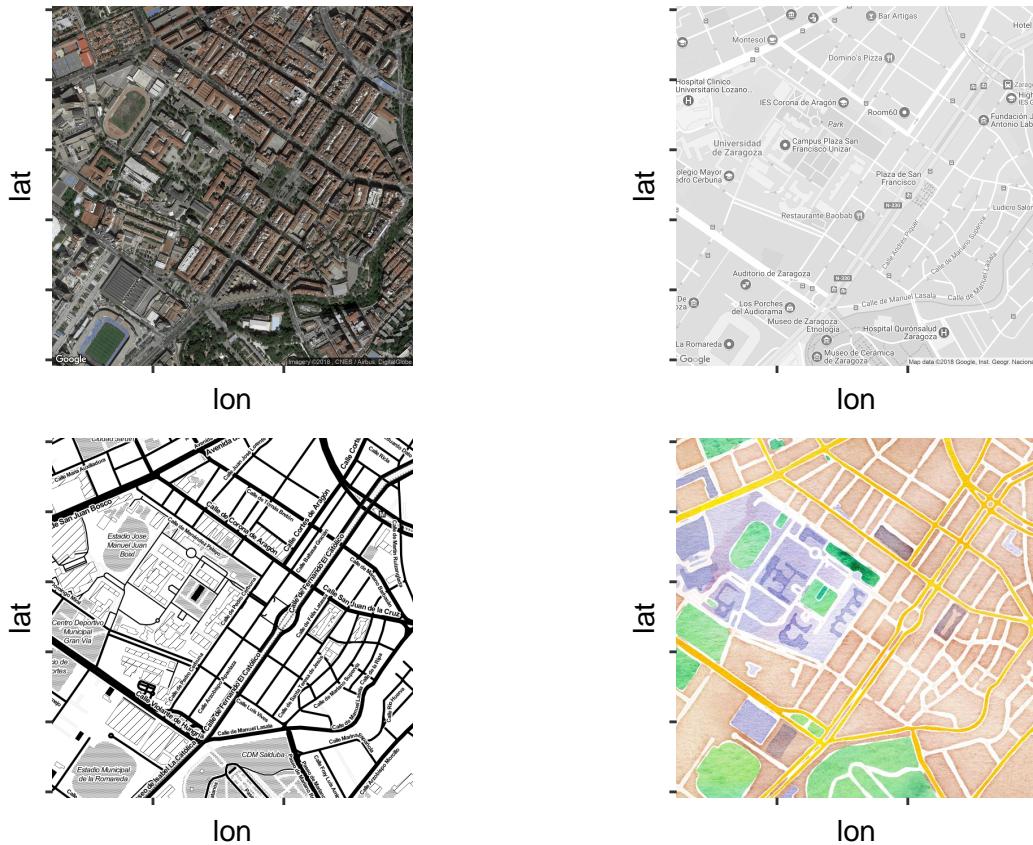
10.1.1. Funciones para obtener mapas

`ggmap` obtiene sus mapas, por defecto, de GoogleMaps. Sin embargo hay otros proveedores de mapas libres, como OpenStreetMap (OSM) o Stamen. Cada proveedor exige una serie de parámetros distintos y, por

ejemplo, un *zoom* de 8 puede significar una escala distinta en GoogleMaps que en OSM. Sin embargo, los autores de `ggmap` se han tomado la molestia de homogeneizar los argumentos de llamada para que sean aproximadamente equivalentes en todos los proveedores.

`ggmap` incluye funciones específicas para cada proveedor, como `get_googlemap` o `get_stamenmap`, pero salvo para usos avanzados, es recomendable usar la función `get_map`, que ofrece un punto de entrada único y homogéneo para el resto.

El gráfico siguiente muestra cuatro mapas obtenidos de diversos proveedores y con diversas opciones. En la fila superior, una capa de Google en modo imagen de satélite y otra estándar. En la inferior, dos mapas de Stamen, uno en modo *toner* y otro en modo *watercolor* o acuarela. Son solo cuatro de los muchos a los que la función `get_map` puede acceder.



Ejercicio 10.1.1 Representa la ubicación de la universidad de Zaragoza (u otra que prefieras) con otros tipos de mapas, otros zums, etc.

10.1.2. Funciones para consultar APIs cartográficas

Muchos servicios de información cartográfica proporcionan APIs para realizar consultas. Las APIs se consultan, típicamente, con URLs convenientemente construidas. Por ejemplo, la URL

<http://maps.googleapis.com/maps/api/geocode/json?address=Universidad+de+Zaragoza>

consulta el servicio de geolocalización de GoogleMaps y devuelve las coordenadas de la Universidad de Zaragoza (así como otra información relevante en formato JSON). La función `geocode` de `ggmap` facilita la consulta a dicho servicio: toma su argumento (el nombre de un lugar), construye internamente la URL, realiza la consulta (para lo que es necesario conexión a internet), lee la respuesta y le da un formato conveniente (en este caso, un `data.frame` de R).

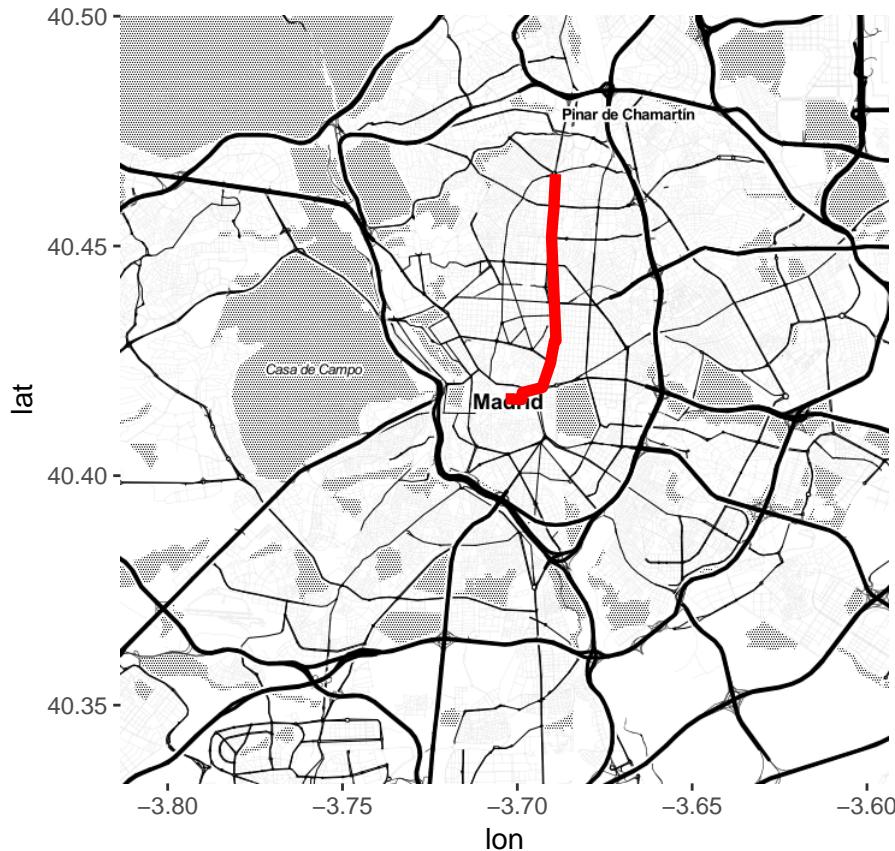
La de geolocalización no es la única API que permite consultar `ggmap`. También permite invertir la geolocalización, es decir, dadas unas coordenadas, devolver el nombre del lugar al que se refieren:

```
revgeocode(as.numeric(unizar))
```

```
## [1] "Historia Moderna y Contemporánea, 50009 Zaragoza, Spain"
```

Finalmente, `route` permite obtener la ruta entre dos puntos distintos:

```
mapa <- get_map("Madrid", source = "stamen", maptype = "toner", zoom = 12)
ruta <- route(from = "Puerta del Sol, Madrid", to = "Plaza de Castilla, Madrid")
ggmap(mapa) +
  geom_path(aes(x = startLon, y = startLat, xend = endLon, yend = endLat),
            colour = "red", size = 2, data = ruta)
```



En el mapa anterior la ruta elegida por GoogleMaps para ir de la Puerta del Sol hasta la plaza de Castilla (dos plazas de Madrid) está marcado en rojo sobre un mapa de Stamen de tipo *toner*.

10.1.3. Funciones para representar elementos sobre mapas

Como se ha visto en las secciones anteriores, la función `ggmap` permite representar un mapa descargado previamente. Además, a esa capa subyacente se le pueden añadir elementos (puntos, segmentos, densidades etc.) usando las funciones ya conocidas de `ggplot2`: `geom_point`, etc.

En `ggplot2` existe una función, `geom_path` que dibuja caminos (secuencias de segmentos). Se puede utilizar en `ggmap` para dibujar rutas, aunque este paquete proporciona una función especial, `geom_leg` que tiene la misma finalidad aunque con algunas diferencias menores: por ejemplo, los segmentos tienen las puntas redondeadas, para mejorar el resultado gráfico.

10.2. Ejemplos

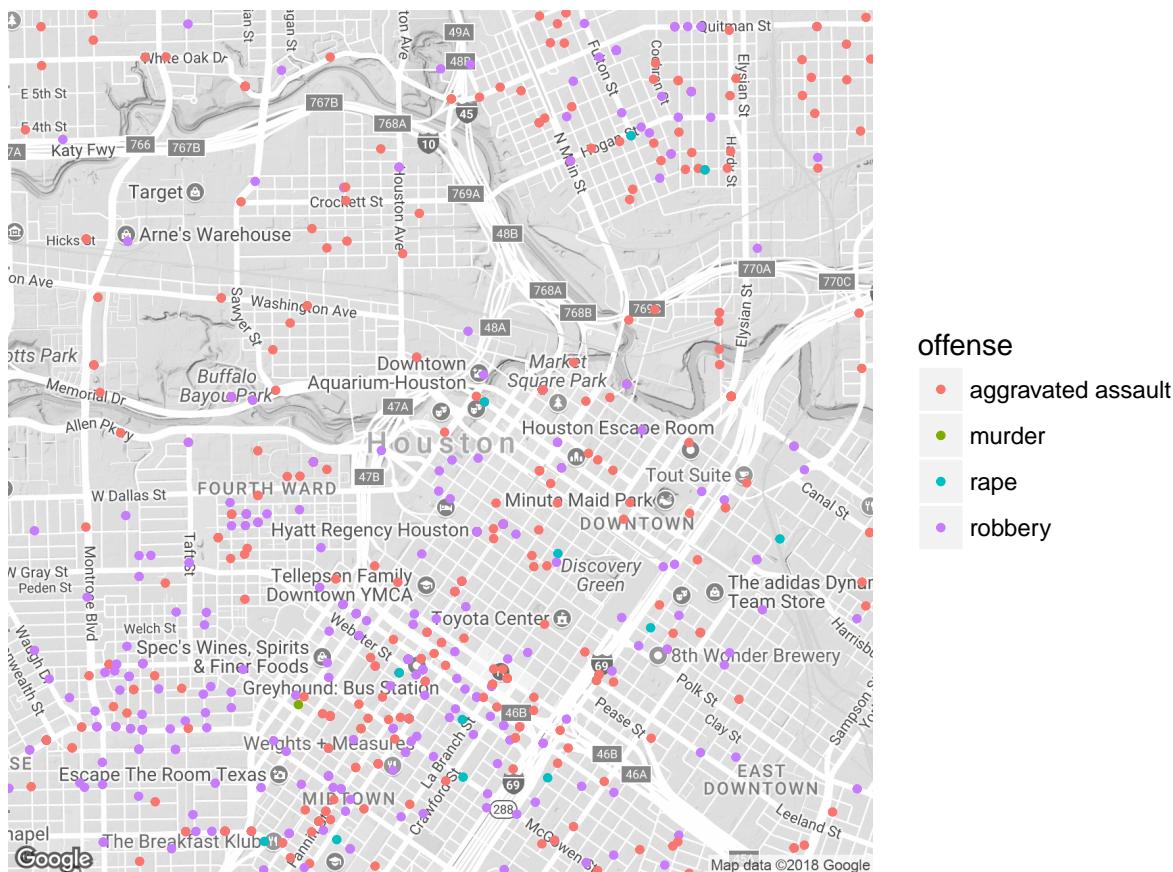
En los ejemplos que siguen se va a utilizar el conjunto de datos `crimes` que forma parte del paquete `ggmap` y que incluye información geolocalizada de crímenes cometidos en la ciudad de Houston. En realidad, solo consideraremos los crímenes *serios*, es decir,

```
crimes.houston <- subset(crime, ! crime$offense %in% c("auto theft", "theft", "burglary"))
```

10.2.1. Puntos sobre mapas

El tipo de mapas más simples son los que se limitan a representar puntos sobre una capa cartográfica.

```
HoustonMap <- qmap("houston", zoom = 14, color = "bw")
HoustonMap +
  geom_point(aes(x = lon, y = lat, colour = offense), data = crimes.houston, size = 1)
```



En el código anterior hemos usado la función `qmap`, una función auxiliar que internamente llama primero `get_map` y luego a `ggmap`.

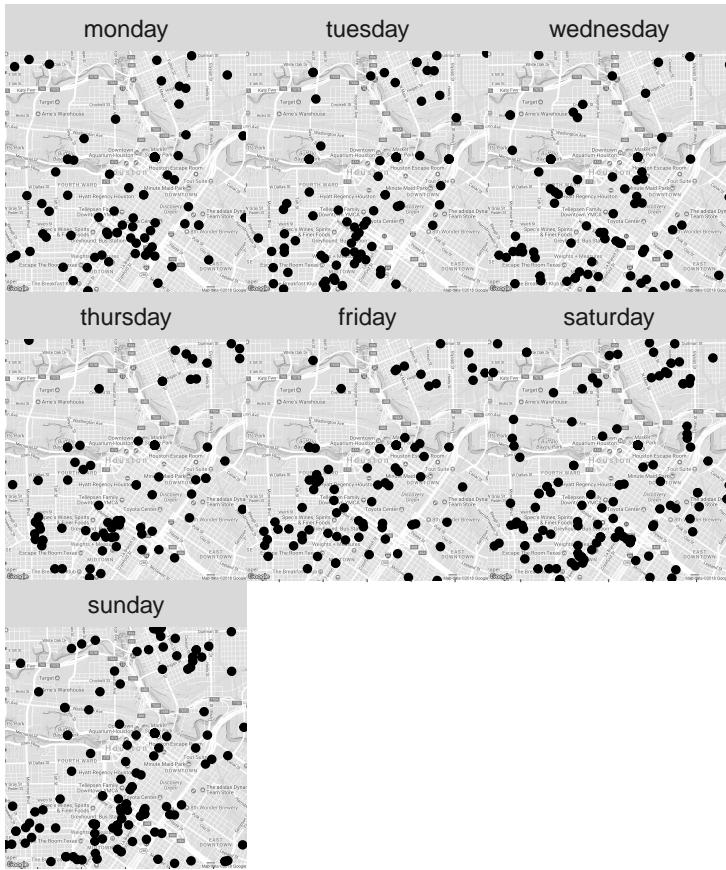
Los mecanismos conocidos de `ggplot2`, como las facetas, están disponibles en `ggmap`: es posible crear una retícula de mapas usando `facet_wrap`. En este primer caso, descomponiendo el gráfico anterior por tipo de crimen.

```
HoustonMap +
  geom_point(aes(x = lon, y = lat), data = crimes.houston, size = 1) +
  facet_wrap(~ offense)
```



O, alternativamente, por día de la semana.

```
HoustonMap +
  geom_point(aes(x = lon, y = lat), data = crimes.houston, size = 1) +
  facet_wrap(~ day)
```



Ejercicio 10.2.1 Prueba a pintar las gasolineras en el mapa de España (fichero dat/carburantes_20050222.csv).

Ejercicio 10.2.2 Baja datos georreferenciados del portal de datos abiertos del ayuntamiento de Madrid y represéntalos sobre un mapa.

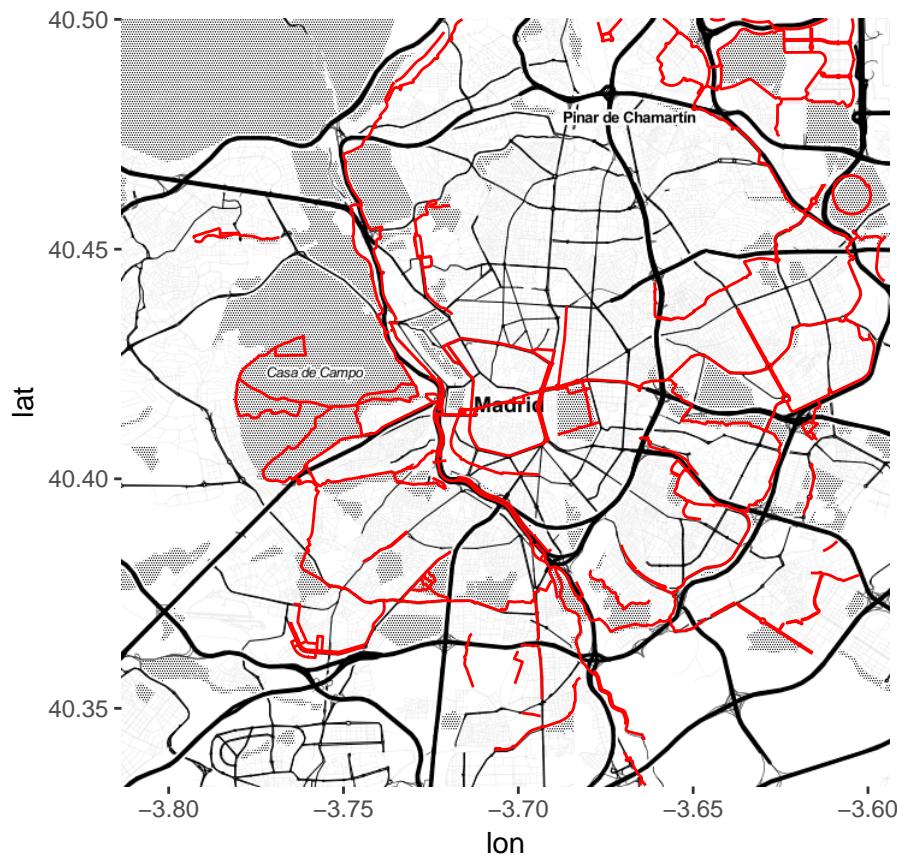
10.2.2. Rutas sobre mapas

Vamos a hacer una digresión para representar información geográfica contenida en ficheros .kml usando `ggmap` como ejemplo de la versatilidad del paquete.

```
library(maptools)
# un fichero bajado el Ayto. de Madrid
rutas <- getKMLcoordinates("data/130111_vias_ciclistas.kml")
```

El conjunto de datos anterior contiene una lista de rutas (inspecciónalo), que queremos convertir en una única tabla para poder representarlas gráficamente con `ggmap`.

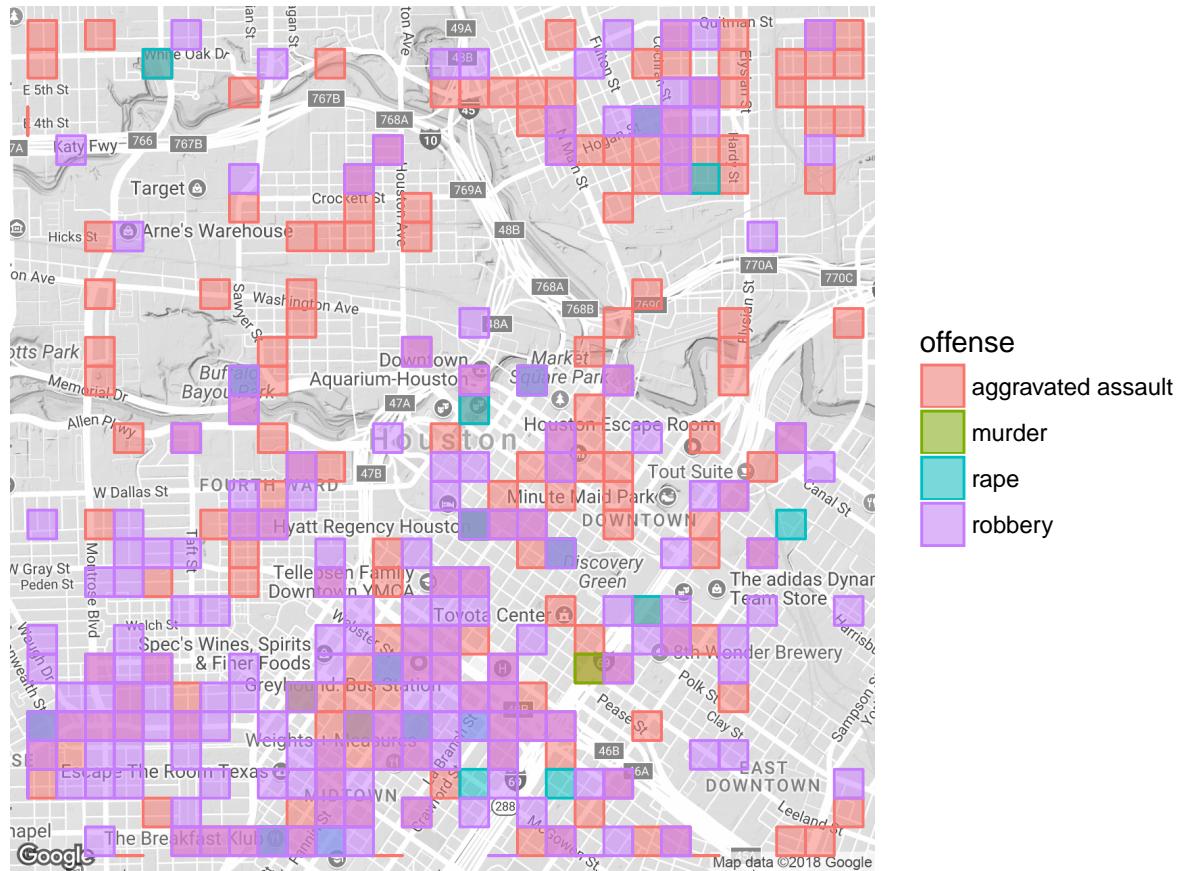
```
library(plyr)
rutas <- ldply(1:length(rutas), function(x) data.frame(rutas[[x]], id = x))
mapa <- get_map("Madrid", source = "stamen", maptype = "toner", zoom = 12)
ggmap(mapa) + geom_path(aes(x = X1, y = X2, group = id), data = rutas, colour = "red")
```



10.2.3. Más allá de los puntos: densidades y retículas

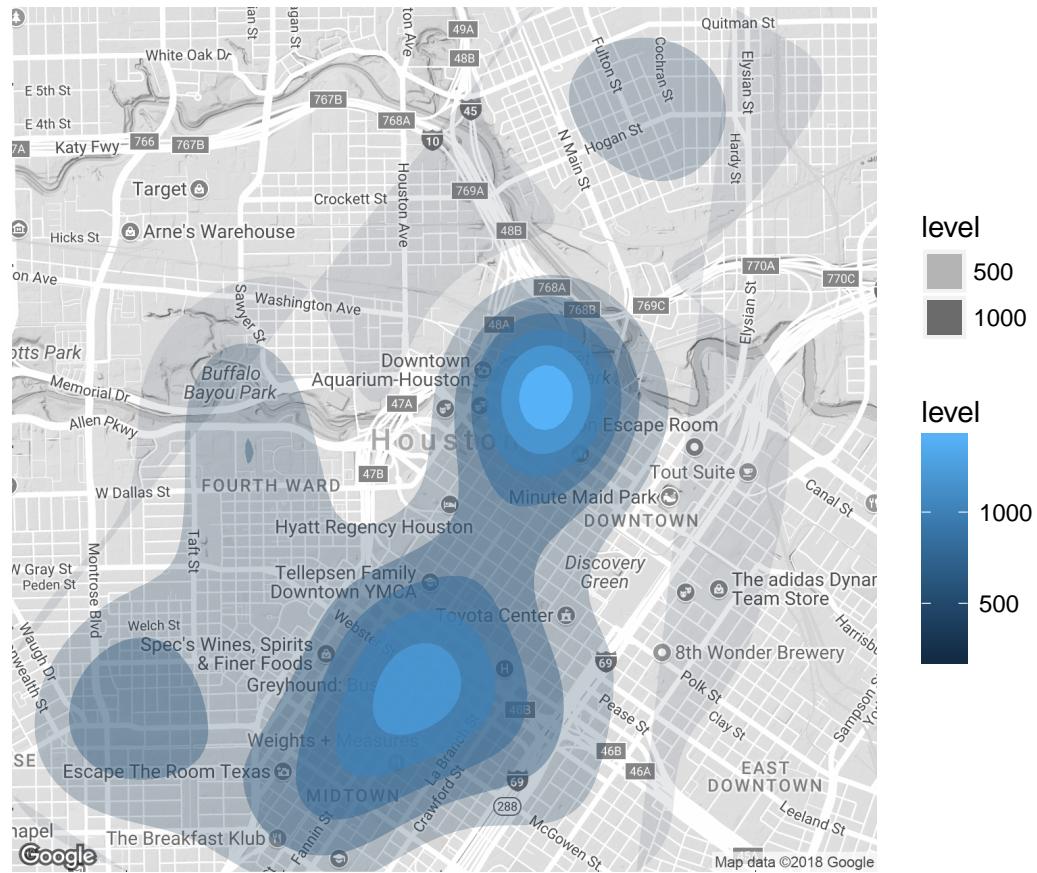
Además de `geom_point`, también están disponibles otros tipos de capas de `ggplot2`, como `stat_bin2d`, que cuenta el número de eventos que suceden en regiones cuadradas de un tamaño predefinido.

```
HoustonMap +
  stat_bin2d(
    aes(x = lon, y = lat, colour = offense, fill = offense),
    size = .5, bins = 30, alpha = 1/2,
    data = crimes.houston
  )
```



O se puede usar `stat_density2d`, que representa intensidades, para identificar las zonas de mayor criminalidad.

```
HoustonMap +
  stat_density2d(aes(x = lon, y = lat, fill = ..level.., alpha = ..level..),
                 size = 2, data = crimes.houston,
                 geom = "polygon")
```



Ejercicio 10.2.3 Lee este artículo y consulta qué otras cosas (además de puntos) pueden representarse sobre un mapa.

10.3. Resumen y referencias

`ggmap` extiende el paquete `ggplot2` para representar información geográfica. Sin embargo, en R existen otros paquetes para procesar información geográfica. De hecho, existe una colección de paquetes construidos alrededor del fundamental, `sp` que fue diseñado para aplicaciones en geoestadística y que permiten hacer de R, prácticamente, un GIS (*geographic information system*).

Estos paquetes permiten, por ejemplo, procesar *shapefiles*, que son ficheros usados en muchos GIS para almacenar información geográfica. El INE y otros organismos estadísticos suelen diseminar información con una componente geográfica en este formato.

En la actualidad, el paquete `sf` (*Simple Features*) es un intento por reorganizar los paquetes relacionados con la información geográfica alrededor de un nuevo estándar, *Simple Feature Access* (véase <http://www.opengeospatial.org/standards/sfa>).

Además de lo anterior, paquetes como `leaflet` permiten crear mapas dinámicos en los que es posible hacer zoom, desplegar marcadores, etc.

Finalmente, los usuarios interesados en información geográfica española, tienen a su disposición el paquete `caRtociudad`, que explota una API del Instituto Geográfico Nacional que proporciona servicios similares a los de `ggmap`: mapas, geolocalización, rutas, etc. Tiene algunas ventajas con respecto a `ggmap`, como que no tiene restricción en el número de llamadas y que, además, proporciona información administrativa adicional, como códigos postales, secciones censales o de referencias catastrales.

10.4. Ejercicios adicionales

Ejercicio 10.4.1 Busca rutas en tu localidad y represéntalas sobre un mapa.

Ejercicio 10.4.2 El fichero `carburantes_20050222.csv` contiene información sobre todas las gasolineras de España y los precios de los carburantes. Incluye las coordenadas geográficas de los surtidores. Representa esos datos sobre un mapa de España (o de tu provincia o municipio). Modifica el tamaño (o color) de los puntos en función de, por ejemplo, el precio de los carburantes.

Capítulo 11

Procesamiento de tablas por trozos con `plyr`

En esta sección vamos a aprender a realizar operaciones avanzadas de manipulación de datos con el paquete `plyr`. Muchas transformaciones de datos sobre tablas, conceptualmente, se reducen a:

- Partir una tabla en trozos definidos por provincias, tramos de edad, categorías de productos, etc.
- Realizar una operación sobre esos trozos individuales; p.e., en un caso muy simple, calcular unas medias.
- Recoger los resultados calculados en cada trozo y juntarlos en una única tabla final.

Eso es lo que se hace en SQL con operaciones como el `group by`. Solo que `plyr` es sustancialmente más poderoso.

11.1. Resúmenes de datos por bloques

Comenzaremos por cargando el paquete (y tendrás que instalarlo si no lo has hecho ya) y leer unos datos de ejemplo:

```
library(plyr)
paro <- read.table("data/paro.csv", header = T, sep = "\t")
```

La función fundamental del paquete `plyr` es `ddply` que, típicamente, se invoca así:

```
res <- ddply(paro, .(Gender, Periodo, Situation), summarize, total = sum(value))
```

La tabla resultante contiene el número de españoles (en miles) por sexo, periodo y situación laboral agregando los valores correspondientes a todas las provincias.

Los elementos que componen la expresión anterior son:

- `ddply`: es una función que transforma una tabla en otra tabla; de hecho, `ddply` *siempre* opera sobre tablas y *siempre* devuelve tablas.
- `paro`: la tabla sobre la que opera.
- `.(Gender, Periodo, Situation)`: variables de la tabla de entrada cuyos niveles definen los trozos sobre los que se opera individualmente. La tabla de entrada se parte en tantas subtablas como combinaciones distintas de valores de `Gender`, `Periodo` y `Situation` existan.
- `summarize`: una función que opera sobre tablas y devuelve tablas; veremos después ejemplos de otras funciones que pueden usarse en su lugar.
- `total = ...`: argumentos de la función anterior. Igual que ocurría con `tapply`, los argumentos que siguen a la función anterior se aplican a `ddply` sino sobre aquella.

La función `ddply` encuentra las subtablas que definen los niveles distintos de las variables indicadas en el segundo argumento y se las pasa una a una a la función que indica el tercer argumento. Esta función opera sobre cada uno de los trozos por separado y, finalmente, `ddply` apila los resultados en una única tabla final.

`ddply` es otro ejemplo de una función de orden superior. Anteriormente habíamos introducido `tapply`, que es similar pero mucho menos potente: `tapply` opera solo sobre vectores, pero `ddply` admite tablas como argumentos.

La función `summarize` es muy simple: aplicada sobre una tabla, devuelve otra con una única fila que contiene los resúmenes (medias, etc.) que se le pasan como argumentos adicionales. Por ejemplo,

```
summarize(iris, media.pl = mean(Petal.Length), media.pw = mean(Petal.Width))
```

```
##   media.pl media.pw
## 1      3.758 1.199333
```

devuelve las longitudes y anchuras medias de los pétalos de `iris`. Por sí sola, tal y como ilustra el ejemplo anterior, `summarize` no es una función particularmente útil. Pero en combinación con `ddply` sirve para implementar lo equivalente a los *group by* de SQL. Si quisieramos realizar esta operación *por especies*, podríamos envolver la expresión anterior en una llamada a `ddply` así:

```
ddply(iris, .(Species), summarize, media.pl = mean(Petal.Length), media.pw = mean(Petal.Width))
```

```
##       Species media.pl media.pw
## 1     setosa    1.462   0.246
## 2 versicolor    4.260   1.326
## 3  virginica    5.552   2.026
```

Ejercicio 11.1.1 Usa `summarize` para calcular la media y la mediana de la temperatura en `airquality`.

Ejercicio 11.1.2 Dispón `airquality` en formato largo y extrae la media y la mediana de cada variable por mes.

La función `summarize` no es, por supuesto, la única que admite `ddply`; le vale cualquier función que reciba una tabla y devuelva otra. Por ejemplo,

```
foo <- function(x) lm(Temp ~ Solar.R, data = x)$coefficients
ddply(airquality, .(Month), foo)
```

```
##   Month (Intercept) Solar.R
## 1      5    61.08143 0.02651712
## 2      6    73.64522 0.02868418
## 3      7    80.18086 0.01719467
## 4      8    77.68639 0.03528288
## 5      9    74.72485 0.01299114
```

crea una regresión lineal que modela la temperatura en función de la irradiación solar para cada mes y devuelve los coeficientes de dichos modelos en una tabla. Dentro de la expresión anterior hemos creado una función, `foo`, que toma unos datos, construye una regresión lineal con ellos y devuelve los coeficientes del modelo; `ddply` permite aplicar esa función a los subconjuntos de `airquality` correspondientes a cada uno de los meses de los que contiene datos.

Ejercicio 11.1.3 Usa `ddply` para crear la tabla que en SQL se construiría de la forma `select Species, mean(Petal.Length) from iris group by Species`.

Ejercicio 11.1.4 Carga el conjunto de datos `lmm_data.txt` y calcula el valor medio de las variables numéricas del fichero por colegio y clase.

Ejercicio 11.1.5 Repite el ejercicio anterior pero disponiendo previamente los datos en formato largo (usa la función `melt` del paquete `reshape2`).

Ejercicio 11.1.6 Obtén las tres filas de `iris` correspondientes a las observaciones de con la mayor longitud del pétalo dentro de cada especie. Pista: puedes crear una función que ordene una tabla por una columna y luego seleccione su primera fila.

11.2. Otras funciones similares a ddply de plyr

En el código

```
res <- dlply(airquality, .(Month), function(x) lm(Temp ~ Solar.R, data = x))
lapply(res, coefficients)
ldply(res, coefficients)
```

introducimos varias novedades. La primera, una función anónima: en lugar de crearla fuera de la llamada y asignarle un nombre, podemos simplemente utilizar una función *de usar y tirar*. En segundo lugar, hemos usado la función `dlply` que es similar a `ddply` y que solo se diferencia de ella en que devuelve una lista en lugar de una tabla. Finalmente hemos usado la función `ldply` que recorre esa lista y la recomponen en una tabla. Otras funciones de uso más esporádico son `llaply`, `laply`, `alply` e, incluso, `d_ply`.

Las funciones `xply` toman un elemento del tipo indicado por `x`, lo trocean convenientemente, aplican una función a cada trozo y tratan de recomponer el resultado en un objeto de tipo indicado por `y`. Por eso, `ddply` opera sobre tablas y devuelve tablas; `ldply` opera sobre listas (elemento a elemento) y devuelve tablas; `llply` opera sobre listas y devuelve listas, etc. El prefijo `a` corresponde a *arrays* (no cubiertos en este curso) y `_` a nada: se usa para funciones que no devuelven ningún objeto sino que, tal vez, escriben resultados en disco o generan gráficos.

Ejercicio 11.2.1 Usa `dlply` para partir `iris` por especies. Es decir, crear una lista que contenga los tres bloques de `iris` correspondientes a sus tres especies. Nota: la función `I`, la identidad, puede ser útil.

11.3. Transformaciones de datos por bloques

La función `summarize` devuelve un único registro por bloque. Sin embargo, en ocasiones interesa devolver la tabla entera transformada de alguna manera: por ejemplo, con alguna columna adicional calculada con el resto. Reemplazando `summarize` por `transform` se puede hacer, por ejemplo

```
tasa.paro <- dcast(paro, Gender + Provinces + Periodo ~ Situation)
tasa.paro <- transform(tasa.paro, tasa.paro = unemployed / active)
tasa.paro <- tasa.paro[, c("Gender", "Provinces", "Periodo", "tasa.paro")]

tmp <- ddply(tasa.paro, .(Gender, Provinces),
             transform, rank = rank(-tasa.paro, ties = "random"))
res <- tmp[tmp$rank == 1,]
```

que obtiene el periodo en el que la tasa de paro tocó techo por provincia y sexo. Para ello es fundamental que la función `rank` se aplique a cada uno de los bloques definidos por `Gender` y `Provinces`, es decir, que comience a contar desde 1 para cada uno de ellos.

Ejercicio 11.3.1 Selecciona, para cada periodo, las tres provincias en las que fue menor la tasa de paro tanto para hombres como para mujeres (por separado). Repite el ejercicio teniendo en cuenta la tasa global (o conjunta para hombres y mujeres) de paro.

Ejercicio 11.3.2 Dispón `airquality` en formato largo y para cada variable y cada mes, extrae el día en que cada variable tuvo el máximo valor.

En el siguiente ejemplo vamos a crear un modelo *a trozos*. Usando estos datos, vamos a

- crear un modelo lineal para cada colegio (con el que predecir la variable `extro` en función de otras)
- asignar a cada sujeto dicha predicción

```
dat <- read.table("data/lmm_data.txt", header = T, sep = ",")  
dat.preds <- ddply(dat, .(school), transform,  
                    pred = predict(lm(extro ~ open + agree + social + class)))
```

El código anterior funciona porque la función `transform` captura cada bloque (definido por la variable `school`) y pasa las variables `extro`, etc. a la expresión `predict(lm(extro ~ open + agree + social + class))`. Es posible realizar la misma operación sin `transform`, i.e., explícitamente, así:

```
foo <- function(x){  
  modelo <- lm(extro ~ open + agree + social + class, data = x)  
  res <- x  
  res$preds <- predict(modelo)  
  res  
}  
dat.preds <- ddply(dat, .(school), foo)
```

Es evidente que `transform`, aunque por sí sola no sea una función muy útil, usada en conjunción con `ddply` simplifica muchas operaciones.

Ejercicio 11.3.3 Usando los datos del paro (en formato largo), calcula para cada provincia, periodo y situación laboral el porcentaje de los sujetos que son hombres y mujeres.

Pista: Puedes hacerlo en una única línea con `ddply` en combinación con `transform`. La clave consiste en que puedes calcular el total (sumando hombres y mujeres) por bloques y dividir por esa cantidad.

Ejercicio 11.3.4 Usando de nuevo los datos del paro, crea una tabla en la que, para cada periodo, figure el porcentaje de los parados de España que viven en cada provincia.

11.4. Más allá de summarize y transform

Aunque es frecuente usar `ddply` junto con las funciones `summarize` o `transform`, también pueden usarse otras: la única condición es que acepten tablas y devuelvan igualmente tablas (con la misma estructura).

En esta sección vamos a construir un ejemplo para el que resultará útil (aunque no estrictamente necesaria) la función `arrange`. Esta función es una de las que, más allá de `ddply` y similares, contiene el paquete `plyr` y permite ordenar una tabla por una o varias columnas:

```
arrange(paro, Provinces, Periodo, Gender, Situation)
```

Ejercicio 11.4.1 Busca en `?arrange` cómo ordenar descendientemente.

El problema que queremos resolver es el siguiente: de cada especie de `iris` extraer los tres ejemplares con el pétalo más largo. Evidentemente, este proceso tiene que realizarse especie a especie. Para cada una de ellas, hay que ordenar por la longitud del pétalo y quedarse con las tres últimas filas. Podemos proceder así:

```

extract.top3 <- function(x)
  tail(arrange(x, Petal.Length), 3)

ddply(iris, .(Species), extract.top3)

##   Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
## 1          5.1        3.3         1.7       0.5    setosa
## 2          4.8        3.4         1.9       0.2    setosa
## 3          5.1        3.8         1.9       0.4    setosa
## 4          6.3        2.5         4.9       1.5 versicolor
## 5          6.7        3.0         5.0       1.7 versicolor
## 6          6.0        2.7         5.1       1.6 versicolor
## 7          7.7        3.8         6.7       2.2 virginica
## 8          7.7        2.8         6.7       2.0 virginica
## 9          7.7        2.6         6.9       2.3 virginica

```

En este caso `ddply` ha distribuido la función `extract.top3` que acepta una tabla, la ordena y selecciona sus tres últimas filas.

Ejercicio 11.4.2 En una sección anterior habíamos resuelto el ejercicio consistente en seleccionar, para cada periodo, las tres provincias en las que fue menor la tasa de paro tanto para hombres como para mujeres (por separado) usando `transform`. Repítelo utilizando una versión de la función introducida en esta sección.

11.5. Cruces de tablas

Con la función `merge` pueden hacerse cruces de tablas. Para ilustrar los distintos tipos de cruce, crearemos unos datos pequeños de ejemplo:

```

clientes <- data.frame(id = 1:3, nombre = c("Carlos", "Sara", "Raquel"))
ventas <- data.frame(fecha = c(1, 1, 1, 2, 2, 3, 3, 3, 4),
                      id = c(1, 2, 3, 2, 3, 1, 2, 3, 3),
                      total = 100 * runif(9))

```

Para cruzarlas por defecto, se usa la función `merge`:

```

merge(clientes, ventas)

##   id nombre fecha      total
## 1  1  Carlos     1  5.966848
## 2  1  Carlos     3 93.631642
## 3  2   Sara     3 74.718662
## 4  2   Sara     2  1.338995
## 5  2   Sara     1 28.164899
## 6  3 Raquel     2 77.764461
## 7  3 Raquel     1  6.645842
## 8  3 Raquel     3 17.461803
## 9  3 Raquel     4 13.648189

```

Esta función crea una nueva tabla juntando filas de otras dos. El resultado es el mismo que la consulta de SQL

```

select *
from clientes a, ventas b
where a.id = b.id

```

`merge` junta las filas que tienen el mismo valor en los campos de cruce. Por defecto, los campos de cruce son aquellos que se llaman igual en ambas tablas. Esto es a la vez cómodo (evita tener que especificarlos explícitamente) y peligroso. En el ejercicio siguiente se te pide que investigues cómo modificar este funcionamiento por defecto de `merge`.

Ejercicio 11.5.1 ¿Y si las variables de cruce no se llaman igual? ¿Y si no quieres cruzar por todas ellas?

`merge` también permite realizar *left joins*. Nótese la diferencia entre las dos tablas creadas por `merge` en el ejemplo siguiente.

```
ventas.2 <- ventas[ventas$fecha == 2,]
merge(clientes, ventas.2)
```

```
##   id nombre fecha      total
## 1  2    Sara     2  1.338995
## 2  3  Raquel    2 77.764461
merge(clientes, ventas.2, all.x = T)

##   id nombre fecha      total
## 1  1  Carlos     NA        NA
## 2  2    Sara     2  1.338995
## 3  3  Raquel    2 77.764461
```

La consulta de SQL equivalente a la última expresión es

```
select *
from clientes a left join ventas.2 b
on a.id = b.id
```

Para el ejercicio siguiente vamos a crear una tabla simulada,

```
n <- 10000
n.clientes <- 1000

contracts <- data.frame(
  customer = sample(n.clientes, n, replace = T),
  contract = sample(n, n),
  amount    = 1000 * exp(runif(n))
)
```

que contiene clientes, contratos y el importe de cada contrato.

Ejercicio 11.5.2 Calcula el porcentaje que cada contrato de un cliente supone dentro de su facturación total. P.e., si un cliente tiene dos contratos de 200 y 800 euros, a la primera fila le asociaremos un 20% y, a la segunda, 80%. Una manera de hacerlo consiste en:

- Crear una tabla que agregue el total de los importes por cliente
- Cruzar esta nueva tabla con la original (por cliente)
- Dividir el importe original por el total y multiplicarlo por 100.

`merge` es la función por defecto para cruzar tablas en R. Sin embargo, existe una colección de funciones (`left_join`, `inner_join`, etc.) en el paquete `dplyr` que pueden resultar más familiares a quienes tengan experiencia con SQL¹.

¹Puede consultarse un tutorial de estas funciones aquí

11.6. Resumen y referencias

11.7. Ejercicios adicionales

Ejercicio 11.7.1 Toma la tabla `contracts` definida en la sección anterior y calcula ese total en una única línea usando `ddply`.

Ejercicio 11.7.2 Extrae el mayor contrato (por importe) de cada cliente en `contracts`.

Ejercicio 11.7.3 El fichero `Olive.txt` contiene información sobre una muestra de aceites italianos. En particular, su región y área de procedencia y la concentración química de diversos ácidos.

- Calcula la media de esas concentraciones por región (nota: es probable que quieras trabajar sobre una versión arreglada o larga del fichero).
- Construye un gráfico que muestre qué compuesto distingue mejor las regiones (o, al menos, alguna de ellas del resto).
- Repite el ejercicio anterior normalizando (puedes usar `scale`) las concentraciones. De nuevo, es probable que quieras comenzar con los datos en formato largo.

Ejercicio 11.7.4 Lee la tabla <https://raw.githubusercontent.com/hadley/data-baby-names/master/baby-names.csv> con los nombres más comunes por sexos en EE.UU. durante los últimos años y extrae los tres más populares por año y sexo.

Ejercicio 11.7.5 Descarga ficheros de *actividad contractual* del portal de datos abiertos del ayuntamiento de Madrid, importa alguno de ellos a R y realiza agregaciones *de interés*: por departamento, por tipo de contrato, por empresa adjudicataria. Busca también cuál es el mayor contrato por departamento municipal. Trata de pensar en otras preguntas de interés sobre esos datos.

Capítulo 12

Programación en R

R, entre otras cosas, es también un lenguaje de programación, aunque, generalmente, no se utiliza para programar; más bien, se utiliza interactivamente: el usuario lee datos, los revisa, los manipula y genera gráficos, modelos, ficheros Rmd, etc. Además, típicamente, en este proceso hay ciclos: la revisión de los datos conduce a reescribir su lectura, la modelización a modificar su manipulación, etc. Es habitual usar R para desarrollar programas largos con R al estilo de los que se crean con Java, C++ u otros lenguajes.

Así que en R, normalmente, *programar* no consiste tanto en crear *programas* como en empaquetar código útil en bloques, i.e., funciones reutilizables.

Muchas de estas funciones son *de usar y tirar*, es decir, solo son útiles en un contexto determinado: pueden crearse para ser utilizadas en un único proyecto o en una parte muy concreta o pequeña del mismo. Otras son más generales y los usuarios pueden querer guardarlas para reutilizarlas en otros proyectos. No es infrecuente que los usuarios identifiquen determinadas necesidades, desarrollem funcións destinadas a satisfacerlas y acaben creando sus propios paquetes de funciones de R y redistribuyéndolas entre sus colegas. No obstante, la creación de paquetes, aunque no es complicada, queda fuera del alcance de este libro.

Existen dos grandes paradigmas de programación:

- Programación imperativa: variables, bucles, etc. Es la habitual en lenguajes como C, Fortran o Matlab.
- Programación funcional, donde las funciones son *ciudadanos de primera clase*. Lisp fue el lenguaje pionero en programación funcional y, actualmente, Haskell o Scala son lenguajes casi puramente funcionales; otros como Python, Java o C++, aunque imperativos, incorporan cada vez más elementos funcionales.

R permite combinar ambos. Y los combina, además, con la programación orientada a objetos. El objetivo de la sección será el de familiarizarnos con los aspectos imperativos y funcionales de la programación en R. Esta sección, de todos modos, no es una introducción a la programación. Se limita a mostrar la sintaxis que utiliza R para las construcciones (expresiones condicionales, bucles, definición de funciones, *maps*, etc.) habituales en otros lenguajes de programación, con alguno de los cuales se espera que esté familiarizado el lector.

12.1. Programación imperativa en R

La programación imperativa es aquella en la que el estado de un programa, definido por el valor de sus variables, se modifica mediante la ejecución de comandos. El énfasis recae, además, en *cómo* se modifica el estado del programa y es frecuente el uso de bucles y expresiones condicionales. Es el tipo de programación a la que están acostumbrados los programadores en C, BASIC, mucho de C++, Python o Java, o MatLab.

12.1.1. Variables

Las variables ya son conocidas. Se crean con el operador de asignación `<-`.

```
mi.iris <- iris[1:10,]
```

Las variables existentes en la memoria de R se pueden listar, borrar, etc.

```
ls()
rm(mi.iris)
ls()
```

Al programar, en algunas ocasiones, resulta necesario conocer el *tipo* de las variables. Existen lenguajes *tipados*, como Java o Scala, donde al declarar una variable es obligatorio especificar la naturaleza (número, texto, etc.) del valor que contendrá. En R (al igual que Python), sin embargo, no lo es. Pero eso no elimina la necesidad de conocer el tipo de las variables en algunas ocasiones: durante el curso hemos visto cómo determinadas funciones operan de manera distinta (¡o fallan!) dependiendo del tipo de la variable subyacente. Por eso, frecuentemente, es necesario comprobar que los datos con los que trabajamos son del tipo adecuado.

En el código que aparece a continuación inquiremos el tipo de las variables implicadas:

```
mi.iris <- iris[1:10,]
class(mi.iris)
```

```
## [1] "data.frame"
is.data.frame(mi.iris)
```

```
## [1] TRUE
x <- 1:10
is.vector(x)
```

```
## [1] TRUE
class(x)
```

```
## [1] "integer"
typeof(x)
```

```
## [1] "integer"
```

Este tipo de comprobaciones son importantes tras la lectura o la manipulación de datos para ver si se han procesado correctamente. También lo son cuando se crean funciones de propósito general y se quiere, por ejemplo, comprobar que el usuario proporciona a las funciones argumentos adecuados.

12.1.2. Funciones

Un análisis de datos consiste generalmente en una secuencia de comandos de R (posiblemente insertados dentro de un fichero .Rmd) que se ejecutan secuencialmente. En ocasiones, sin embargo, es conveniente crear funciones. Por ejemplo, cuando hay operaciones comunes que se realizan reiteradamente (incluso en análisis distintos).

Una función se define, por ejemplo, así:

```
calcular.cuota.hipoteca <- function(capital, anyos, interes){
  interes.mensual <- interes / 12 / 100
  meses <- 1:(anyos*12)
  return(capital / sum(1 / (1+interes.mensual)^meses))
}
```

Es decir, con `function`, seguido de la lista de argumentos y de un bloque de código (encerrado en llaves) que contiene el *cuerpo* de la función. Una función, típicamente, se asigna a una variable con `<-` (aunque veremos casos en que pueden usarse *funciones anónimas*). La función así creada puede *invocarse*:

```
calcular.cuota.hipoteca(100000, 20, 3)
```

```
## [1] 554.5976
```

La función, además, se convierte en un objeto *normal* de R; es decir, aparece en los listados de `ls`, que se puede borrar con `rm`, etc.:

```
calculadora.hipotecas <- calcular.cuota.hipoteca
calculadora.hipotecas(100000, 20, 3)
```

```
## [1] 554.5976
```

```
ls()
```

```
## [1] "calculadora.hipotecas"    "calcular.cuota.hipoteca"
rm(calculadora.hipotecas)
```

Ejercicio 12.1.1 Crea una función que, dado un número `n` calcule la suma de los `n` primeros términos de la serie de Leibniz para aproximar π . Nota: ya hemos realizado previamente este ejercicio; lo que se pide aquí es incluir aquel código dentro de una función.

Un bloque de código es un conjunto de líneas que se ejecutan secuencialmente. Para acotar los bloques de código se usan las llaves `{}`. Sin embargo, no son necesarias cuando el bloque consiste en una sola línea. Es decir,

```
cuadrado <- function(x){
  return(x^2)
}
```

```
y
```

```
cuadrado <- function(x) return(x^2)
```

son ambas definiciones válidas de la función `cuadrado`.

Hemos usado `return` para que la función devuelva un valor. En algunos lenguajes de programación es obligatorio el uso de `return`; sin embargo, en R no: una función de R devuelve el último valor calculado dentro de su cuerpo. Así que una tercera opción equivalente y más sucinta para definir la función `cuadrado` es

```
cuadrado <- function(x) x^2
```

En R, además, las funciones pueden tener argumentos con valores por defecto. Por ejemplo,

```
potencia <- function(x, exponente = 2) x^exponente
c(potencia(2), potencia(2, 3), potencia(2, exponente = 3))
```

Los valores por defecto permiten, por ejemplo, llamar a funciones como `read.table`, que admite muchos argumentos, especificando solo algunos de ellos.

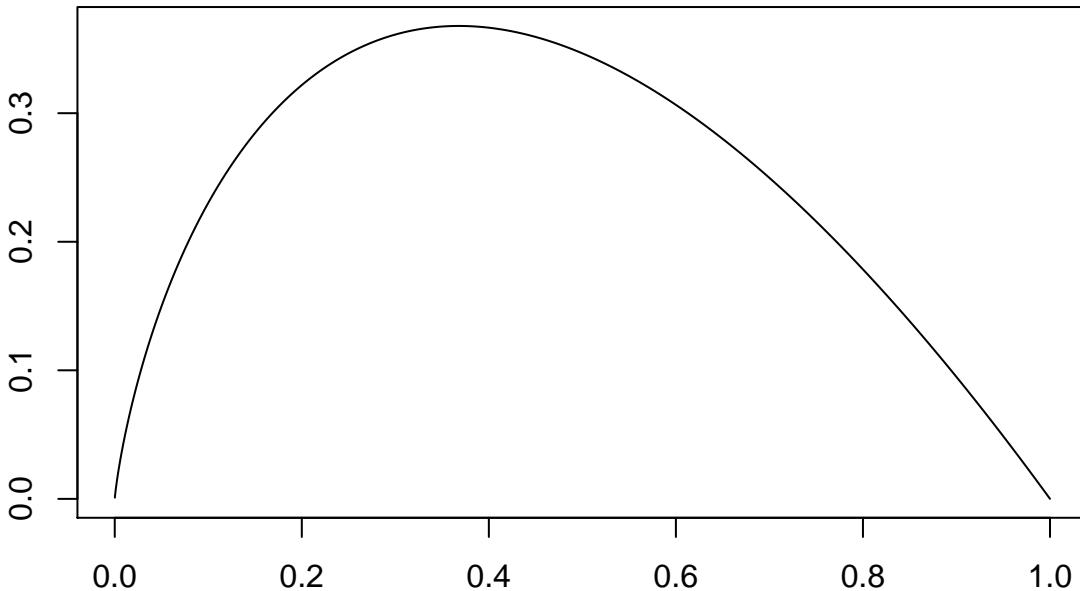
12.1.3. Expresiones condicionales

Las expresiones condicionales permiten ejecutar un código u otro en función de un criterio. Para ilustrar su uso en R, Comenzaremos definiendo y representando gráficamente la función `xln`, que aparece frecuentemente en estadística:

```
xln <- function(x){
  return(-x * log(x))
}

x <- 1:10000 / 10000
plot(x, xln(x), type = "l", xlab = "", ylab = "",
      main = "Función -x * log(x)")
```

Función $-x * \log(x)$



Diríase que su valor en 0 es cero; sin embargo, en contra de nuestra intuición,

```
xln(0)      # Nan cuando queremos cero!
```

```
## [1] NaN
```

Podemos, por lo tanto, *arreglarla* con una expresión condicional:

```
xln <- function(x){
  if (x == 0)
    return(0)
  return(-x * log(x))
}
```

Y ahora sí,

```
xln(0)
```

```
## [1] 0
```

Ejercicio 12.1.2 Modifica la función anterior para que dé un error cuando x sea menor que 0 o mayor que 1. Pista: la función `stop()` lanza un error. El argumento de `stop` es el texto que aparece en el mensaje de error.

Ejercicio 12.1.3 En la definición anterior hay dos `return`. Uno sobra y el otro no. ¿Cuál es cuál?

Como ilustra el ejercicio anterior, es muy común que una función resuelva al principio uno o más casos particulares (y salga de ellos mediante `returns` dentro de expresiones condicionales) y que, una vez solventados, se plantee al final el caso general y más complejo. La salida de este último, típicamente, no necesita `return`.

Frecuentemente, `if` va acompañado de `else`:

```
xln <- function(x){
  if (x == 0)
    return(0)
  else
    return(-x * log(x))
}
```

Nota: como antes, cuando el bloque de código que sigue a `if` (o `else`) contiene una única línea, se pueden ignorar las llaves. Si no, hay que usarlas.

Ejercicio 12.1.4 ¿Cuántos `return` sobran en la última definición de `xln`? ¿Por qué?

Ejercicio 12.1.5 Existe una función en R, `ifelse`, que permite escribir la función anterior de una forma más compacta. Búscalas en la ayuda y reescribe la función `xln`.

Ejercicio 12.1.6 Crea una función que tome como argumento un vector de texto (o un factor) cuyas entradas sean *números* de la forma "1.234,56" y devuelva el correspondiente número subyacente, es decir, que elimine el separador de miles, etc. Ten en cuenta que cuando el vector de entrada sea del tipo `factor` tendrás que convertirlo previamente en otro de tipo `character`.

Ejercicio 12.1.7 Modifica la función del ejemplo anterior de forma que si el usuario pasa como argumento un vector numérico devuelva ese número; si pasa un argumento del tipo `character` o `factor`, aplique la lógica descrita en ese ejercicio y, finalmente, si pasa un argumento de otro tipo, devuelva un error (usando `stop`) con un mensaje informativo.

12.1.4. Bucles

En la programación imperativa es habitual construir bucles dentro de los cuales se va modificando el valor de una expresión. Los bucles más habituales en R son los `for`. Su sintaxis es

```
for (var in vector){
  # expresión que se repite
}
```

Lo comentado más arriba sobre las llaves aplica también a los bucles: solo son obligatorias cuando el bloque de código contiene más de una línea.

Un ejemplo de libro para ilustrar el uso de los bucles es el del cálculo del factorial de un número:

```
mi.factorial <- function(n){
  factorial <- 1
  for (i in 1:n){
    factorial <- factorial * i
  }
  return(factorial)
}

mi.factorial(7)
```

```
## [1] 5040
```

Ejercicio 12.1.8 Modifica la función anterior para que devuelva explícitamente un error siempre que el argumento de la función no sea un entero positivo.

También existen (aunque se usan menos), los *bucles while*:

```
while (condicion){
  # expresión que se repite
}
```

Por ejemplo,

```
mi.factorial <- function(n){
  factorial <- n
  while (n > 1){
    n <- n - 1
    factorial <- factorial * n
  }
  return(factorial)
}

mi.factorial(7)
```

Los bucles se usan poco en R por varios motivos. Uno de ellos es que muchas funciones están **vectorizadas**, es decir, no es necesario explicitar el bucle:

```
x <- 1:5
sqrt(x)

## [1] 1.000000 1.414214 1.732051 2.000000 2.236068
sum(x)

## [1] 15
```

Para sumar un vector de números con bucles explícitos habría que hacer:

```
x <- 1:10
suma.x <- 0
for (i in x){
  suma.x <- suma.x + i
}
suma.x
```

Es evidente cómo la vectorización permite crear código más breve y, sobre todo, expresivo.

Ejercicio 12.1.9 Crea una función que calcule la raíz cuadrada de los elementos de un vector usando un bucle explícito.

Ejercicio 12.1.10 Crea una función que, dado un número **n** devuelva la lista de sus divisores.

Ejercicio 12.1.11 Modifica la función construida en un ejercicio anterior, la que devolvía los divisores de un número **n** para que compruebe que el argumento es un número entero positivo.

12.2. Programación funcional en R

La programación funcional, en un sentido amplio, es aquella en que determinadas funciones¹ admiten otras como argumento. Por ejemplo, la función `sapply`:

```
cuadrado.raro <- function(x) if(x < 5) x^2 else -x^2
sapply(1:10, cuadrado.raro)
```

```
## [1] 1 4 9 16 -25 -36 -49 -64 -81 -100
```

La programación funcional es sumamente poderosa y sugiere permitir de la siguiente manera:

- Crear funciones pequeñas y simples que resuelven un problema pequeño y acotado
- Aplicar esas funciones a grupos homogéneos de valores.

En el ejemplo de más arriba hemos construido una función, `cuadrado.raro`, y con la función `sapply` (`lapply`, como hemos visto previamente, es una alternativa) se las hemos aplicado a una lista de valores homogéneos, los números del 1 al 10.

Hay muchas funciones en R, algunas de las cuales son ya conocidas, que admiten otras como argumento. Algunas de las más corrientes son:

- `sapply` y `lapply` (que son casi la misma)
- `tapply`
- `apply` y `mapply`
- Las funciones `ddply`, `ldply`, etc. del paquete `plyr`

Dos ejemplos de usos muy habituales de estas funciones son

```
lapply(iris, class)
sapply(iris, length)
```

que permiten inspeccionar el tipo de columnas de una tabla. Aprovechan, precisamente, que una tabla es una lista de columnas y las recorren una a una.

Generalmente, el código que usa este tipo de funciones es más breve y legible.

Es conveniente recordar aquí que si consultas la ayuda de las funciones listadas más arriba verás que suelen incluir un argumento especial, ... que permite pasar argumentos adicionales a la función a la que llaman. En la sección en que introdujimos la función `tapply` discutimos un caso de uso.

12.2.1. Funciones anónimas

Las funciones que hemos usado son de dos tipos: o existen en R o las hemos definido previamente. Pero en ocasiones es conveniente usar funciones *anónimas*² de esta manera:

```
sapply(1:10, function(x) if(x < 5) x^2 else -x^2)
```

Conviene particularmente cuando la función solo se usa una única vez. Las funciones anónimas, debidamente usadas, confieren brevedad y expresividad al código.

Ejercicio 12.2.1 Crea el vector de nombres de ficheros de `data` usando `dir`; luego, aplícale una función que lea las líneas (`readLines`) y las cuente.

Ejercicio 12.2.2 Usa `nchar` para contar el número de caracteres de esos ficheros.

¹Se las conoce como *funciones de orden superior*.

²En otros lenguajes de programación se las conoce también como *funciones lambda*.

Ejercicio 12.2.3 Haz lo mismo usando la función `lapply` de `plyr`.

12.2.2. Map, reduce y más

Existen dos operaciones fundamentales en programación funcional. La primera es *map* y consiste en aplicar una función a todos los elementos de una lista o vector. Es, de hecho, la operación que hemos realizado más arriba:

```
sapply(1:10, function(x) if(x < 5) x^2 else -x^2)
```

Ese código aplica al vector `1:10` la función anónima que se le pasa a `sapply` como segundo argumento. Aunque en muchos lenguajes de programación existe una función *map* explícita (con ese nombre), en R hay varias: además de `sapply`, también están `lapply` o `apply`. La vectorización que hemos discutido previamente es un mecanismo implícito para realizar *maps*; p.e.,

```
sqrt(1:10)
```

aplica la función `sqrt` a cada elemento de su argumento.

La otra gran operación de la programación funcional es *reduce*. Consiste en aplicar una operación binaria (p.e., la que suma dos números) a una lista de valores iterativamente. En R se pueden realizar *reduces* explícitamente:

```
Reduce(function(a, b) a + b, 1:10)
```

La función anónima anterior admite dos argumentos, `a` y `b` y los suma. Dentro de `Reduce`, la función anónima suma los dos primeros elementos, al resultado le suma el tercero, al resultado el cuarto, etc., hasta proporcionarnos la suma total. De nuevo, es frecuente poder realizar estas operaciones implícitamente. Por ejemplo, usando `sum`:

```
sum(1:10)
```

Ejercicio 12.2.4 Vamos a crear el objeto `x <- split(iris, iris$Species)`, que es una lista de tres tablas. Usa `lapply` o `sapply` para examinarlas: dimensión, nombres de columnas, etc.

Ejercicio 12.2.5 Usa `Reduce` con la función `rbind` para apilar las tres tablas contenidas en `x` (véase el ejercicio anterior).

Nota: este ejercicio tiene aplicaciones prácticas importantes. Por ejemplo, cuando se leen tablas del mismo formato de ficheros distintos y es necesario juntarlas todas, i.e., apilarlas, en una única tabla final.

Operaciones tales como

```
sum(sqrt(1:10))
```

son, por lo tanto, los famosos *mapreduces*³ popularizados por las herramientas de *big data*.

Una operación relacionada con *map* y muy frecuente en R es `replicate`. Esta función permite llamar repetidamente a una función (o bloque de código) para, muy frecuentemente, realizar simulaciones:

```
simula <- function(n, lambda = 4, mean = 5){
  n.visitas <- sum(rpois(n, lambda))
  sum(rnorm(n.visitas, mean = mean))
}
```

```
res <- replicate(1000, simula(10, lambda = 7))
```

³ *Mapreduce* es una operación genérica que consiste en un *map* seguido de un *reduce*; muchas manipulaciones de datos se reducen en última instancia a un *mapreduce* o a una concatenación de ellos.

La diferencia fundamental con *map* es que no opera sobre un vector: crea uno *de la nada*.

Otra *metaoperación* básica en programación funcional es *filter*. Un filtro permite seleccionar aquellas observaciones (p.e., en una lista) que cumplen cierta condición. En R sabemos implementar esa operación usando los corchetes. Por ejemplo, para seleccionar los múltiplos de tres en un vector, podemos hacer

```
x <- 1:20
x[x %% 3 == 0]
```

```
## [1] 3 6 9 12 15 18
```

Pero en R también se puede usar⁴ la función *Filter*:

```
Filter(function(i) i %% 3 == 0, x)
```

```
## [1] 3 6 9 12 15 18
```

Muy importantes en R debido a lo habitual de operar con tablas son las operaciones basadas en otra operación funcional, el *groupby*. El *groupby* permite partir un objeto en trozos de acuerdo con un determinado criterio para operar a continuación sobre los subbloques obtenidos. *tapply* es una función que implementa una versión básica del *groupby*. Mucho más potente y versátil que ella es la función *ddply* del paquete *plyr*. Esta función, como ya sabemos, realiza tres operaciones:

- Parte una tabla convenientemente (*groupby*).
- Aplica (*map*) una función sobre cada subtabla.
- Recompone (*reduce*) una tabla resultante a partir de los trozos devueltos en el paso anterior.

12.2.2.1. Para saber más

La programación funcional proporciona una serie de operaciones genéricas, *map*, *reduce*, *filter*, *groupby* y algunas otras más que permiten modelar conceptualmente a los programas. Los programadores experimentados identifican frecuentemente un determinado algoritmo como, por ejemplo, un *map* seguido de un *filter* y un *reduce* final. Eso les facilita, por ejemplo, el poliglotismo: al final, desarrollar en cualquier lenguaje se reduce a expresar esas operaciones genéricas en la sintaxis específica.

La programación funcional, además, abre la puerta de muchas aplicaciones *big data*, donde impera desde sus inicios la programación funcional. El mismo Hadoop se concibió alrededor del concepto del *MapReduce* y la más popular de las herramientas actuales de *big data*, Spark, es una extensión del lenguaje funcional Scala.

12.3. Orientación a objetos

R está *orientado a objetos*. Los objetos son estructuras que combinan datos y funciones que operan sobre ellos y son muy útiles en un entorno, como R, pensado para el análisis estadístico de datos. Por ejemplo, el resultado de una regresión lineal es un objeto (de clase *lm*):

```
mi.modelo <- lm(dist ~ speed, data = cars)
class(mi.modelo)
```

```
## [1] "lm"
```

El objeto *mi.modelo* contiene información relevante acerca del modelo lineal: coeficientes, residuos, p-valores, etc. Se puede consultar esta información haciendo, por ejemplo,

```
str(mi.modelo)
```

Pero, además, existen funciones como *summary*, *plot* o *predict* que saben cómo operar sobre un objeto de la clase *lm* como el anterior proporcionando los resultados esperados.

⁴Aunque no se recomienda: el corchete es más sucinto.

12.3.1. Polimorfismo

El polimorfismo es una característica del lenguaje que se logra en R gracias a la orientación a objetos. Permite que una única función, p.e., `summary`, opere de manera distinta dependiendo de su argumento. La siguiente discusión pone de manifiesto la utilidad del polimorfismo y cómo la orientación a objetos de R es fundamental para conseguirla.

Nótese que el objeto `mi.modelo` construido más arriba es también una lista:

```
is.list(mi.modelo)
```

```
## [1] TRUE
```

En realidad, nuestro modelo es una lista a la que se ha añadido un atributo `class` que lo identifica como un modelo lineal, i.e., `lm`. Por contra, `iris` es una tabla, es decir, otro tipo especial de lista, una con atributo `class data.frame`. En el fondo, ambos son listas; sin embargo, la función `summary` proporciona información distinta sobre ellos:

```
summary(iris)
```

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

```
summary(mi.modelo)
```

```
##
## Call:
## lm(formula = dist ~ speed, data = cars)
##
## Residuals:
##   Min   1Q   Median   3Q   Max
## -29.069 -9.525 -2.272  9.215 43.201
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -17.5791    6.7584  -2.601  0.0123 *
## speed        3.9324    0.4155   9.464 1.49e-12 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 15.38 on 48 degrees of freedom
## Multiple R-squared:  0.6511, Adjusted R-squared:  0.6438
## F-statistic: 89.57 on 1 and 48 DF,  p-value: 1.49e-12
```

Es precisamente a través del atributo `class` que R implementa el *polimorfismo*, i.e, que una misma función opere de manera distinta sobre objetos distintos. Lo mismo ocurre con otras funciones *genéricas* como `plot`,

`print` o `predict`.

En realidad, *detrás* de `summary` existen muchas funciones que son versiones específicas (o *métodos*) suyas. Las disponibles para `summary`, son

```
methods(summary)
```

```
## [1] summary,ANY-method           summary.aov
## [3] summary.aovlist*            summary.aspell*
## [5] summary.check_packages_in_dir* summary.connection
## [7] summary.data.frame           summary.Date
## [9] summary.default              summary.ecdf*
## [11] summary.factor               summary.ggplot*
## [13] summary.glm                 summary,GridTopology-method
## [15] summary.infl*               summary.lm
## [17] summary.loess*              summary.loglm*
## [19] summary.manova              summary.matrix
## [21] summary.mlm*                summary.negbin*
## [23] summary.nls*                summary.packageStatus*
## [25] summary.PDF_Dictionary*    summary.PDF_Stream*
## [27] summary.polr*               summary.POSIXct
## [29] summary.POSIXlt             summary.ppr*
## [31] summary.prcomp*              summary.princomp*
## [33] summary.proc_time            summary.rlm*
## [35] summary.shingle*             summary,Spatial-method
## [37] summary.srcfile              summary.srcref
## [39] summary.stepfun              summary.stl*
## [41] summary.table                summary.trellis*
## [43] summary.tukeysmooth*
## see '?methods' for accessing help and source code
```

Cuando `summary` se aplica sobre un objeto de la clase `lm`, la *versión* de `summary` que se le aplica es `summary.lm`; cuando se aplica sobre `iris`, se utiza `summary.data.frame`, etc. Obviamente, es responsabilidad de los autores de `lm` definir las funciones `summary.lm`, `print.lm`, `plot.lm`, etc.

12.3.1.1. Para saber más

Para saber más sobre clases y orientación a objetos en R, puedes leer <https://www.datanalytics.com/2011/08/04/desarrollo-de-paquetes-con-r-iv-funciones-genericas/>. Ahí se discute cómo asignar atributos de clase a objetos y cómo crear métodos y, si procede, nuevas funciones genéricas.

En R existen varios mecanismos para dotarlo de la orientación a objetos. En esta sección hemos explorado superficialmente la más simple, la conocida como S3 (que corresponde con la tercera especificación del lenguaje). Existen otros mecanismos más formales de clases (p.e., las de las clases S4), que utilizan algunos paquetes de R. Por ejemplo, gran parte de los que se usan en geoestadística.

12.4. Resumen y referencias

12.5. Ejercicios adicionales

Ejercicio 12.5.1 En R nunca implementaríamos el factorial de la manera en que lo hemos hecho en esta sección, i.e., con bucles `for` o `while`. Reescribe la función `mi.factorial` teniendo en cuenta que el factorial de 7, por ejemplo, puede calcularse haciendo `prod(2:7)`.

Ejercicio 12.5.2 En este ejercicio y el siguiente, vamos a realizar una simulación para estimar la probabilidad del evento que se describe a continuación. En un avión viajan n personas. Cada una de ellas tiene asignado su asiento y entran al aparato en cualquier orden. El vuelo, además, está lleno: no hay plazas libres. Sin embargo, la primera persona tiene *necesidades especiales* y le conceden el primer asiento. El resto de los pasajeros ocupa los suyos así:

- Si su asiento está libre, se sientan en él.
- Si está ocupado, se sientan en cualquiera de los que está libre.

La probabilidad que se pide estimar es la de que el último pasajero encuentre libre su asiento.

Para ello, crea una función que tome como parámetro el número de asientos en el avión y devuelva TRUE o FALSE según si el último pasajero tiene disponible o no su asiento.

Ejercicio 12.5.3 Una vez terminado el ejercicio anterior, usa la función `replicate` (consulta su ayuda) para ejecutar la función anterior muchas veces. El promedio del número de valores TRUE obtenidos será la estimación de la probabilidad del evento descrito arriba.

Capítulo 13

Estadística y ciencia de datos con R

Esta sección es un paseo por algunos de los algoritmos que dispone R para resolver problemas estadísticos o de la llamada *ciencia de datos*. Se limitará a ilustrar el uso de estos algoritmos con R sin profundizar en su sustancia.

13.1. Árboles de decisión

Los árboles de decisión son útiles para entender la estructura de un conjunto de datos. Sirven para resolver problemas tanto de clasificación (predecir una variable discreta, típicamente binaria) como de regresión (predecir una variable continua). Se trata de modelos excesivamente simples pero, y ahí reside fundamentalmente su interés, fácilmente interpretables.

Existen varios paquetes en R para construir árboles de decisión. De entre todos ellos, vamos a usar los de la librería `party`:

```
library(party)
```

Vamos a analizar los datos de `Olive.txt`, que describen una muestra de diversos aceites de oliva italianos. Para cada aceite se ha anotado su procedencia (`Region` y `Area`) y una serie de características químicas, las concentraciones de determinados ácidos (como el eicosenoico). Este conjunto de datos se compiló para poder crear un modelo que distinguese la procedencia del aceite a través de pruebas químicas y evitar así fraudes de reetiquetado. Nuestro objetivo será, por tanto, predecir la zona de procedencia a partir de la *huella química* de los aceites, es decir, usando esas concentraciones químicas como potenciales predictores del origen de los distintos aceites.

En primer lugar vamos a leer y preparar los datos. Vamos a tratar de predecir la `Region`, por lo que eliminaremos las variables `Area` y la auxiliar `Test.Training`.

```
olive <- read.table("data/Olive.txt", header = T, sep = "\t")
olive.00 <- olive
olive.00$Area <- olive.00$Test.Training <- NULL
```

A continuación, crearemos el modelo.

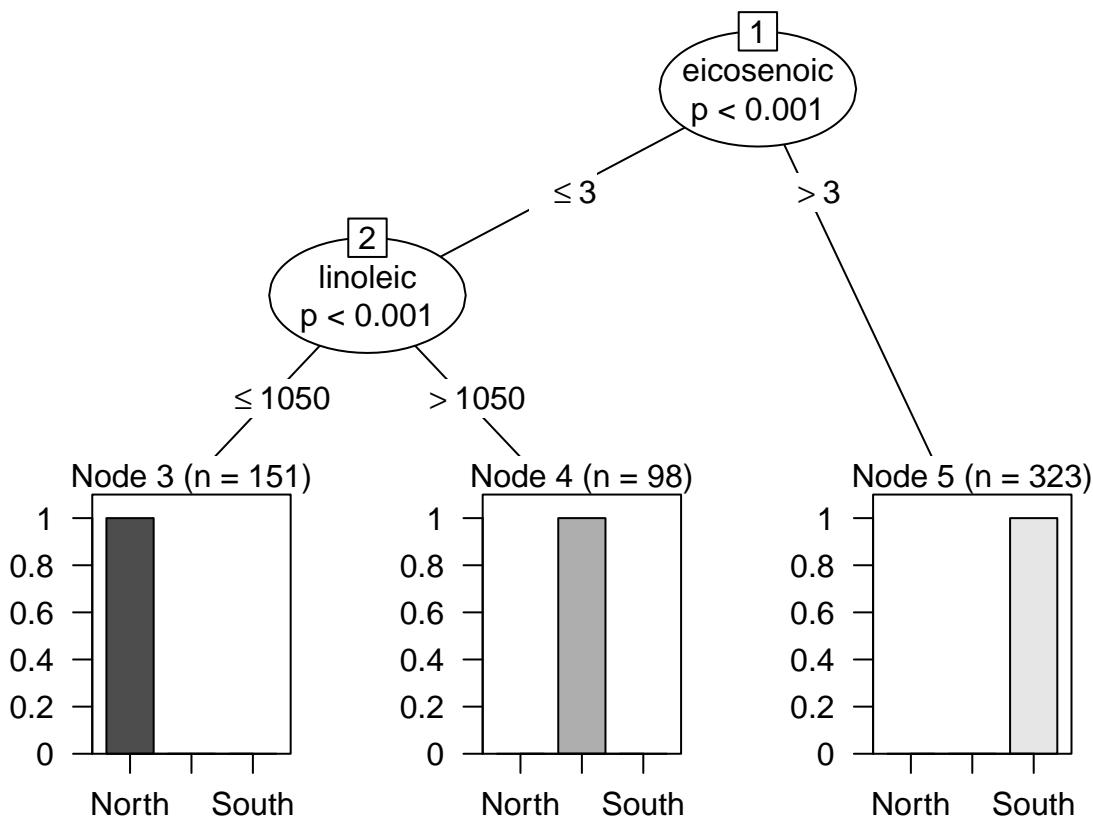
```
modelo <- ctree(Region ~ ., data = olive.00)
```

La fórmula `Region ~ .` indica que queremos modelar `Region` en función de `.`, es decir, el resto de las variables disponibles en nuestros datos. Alternativamente, se pueden indicar las variables predictoras a la derecha de `~` separadas por el signo `+`: `y ~ x1 + x2 + x3`. También es posible seleccionar todas menos algunas de ellas con el signo `-`, p.e., `y ~ . - x1`. Desafortunadamente, la función `ctree` no interpreta correctamente expresiones

que usan el signo $-$. Ese es el motivo por el que hemos creado un conjunto de datos auxiliar `olive.00`; lo natural hubiese sido especificar `Region ~ . - Area - Test.Training`.

El objeto `modelo` construido con `ctree` contiene toda la información relativa al árbol que hemos construido. La mejor manera de inspeccionar el modelo obtenido es representándolo gráficamente haciendo:

```
plot(modelo)
```



Como puede apreciarse, la predicción es *perfecta*: en los nodos finales no hay confusión de regiones. Por ejemplo, el nodo terminal de la derecha, el que se construye con la regla `eicosenoic > 3`, contiene únicamente aceites procedentes de la zona sur de Italia. Por lo tanto, siempre que un aceite cumpla dicha regla, se categorizará como procedente de esa región.

Nota

Si en el gráfico resultante no puedes apreciar correctamente los nodos terminales o sus etiquetas, trata de ampliar la ventana gráfica. Si aun ampliéndola siguiesen sin verse adecuadamente, guarda el gráfico con una resolución alta en disco y ábrelo con un programa de visualización de imágenes.

Ejercicio 13.1.1 Describe las reglas que definen los otros dos nodos terminales y las decisiones que se tomarán con los aceites que *caigan* en ellos.

Normalmente, para medir el poder predictivo de un modelo, se utiliza un conjunto de datos para entrenar el modelo y otro distinto para evaluarlo. Nuestro conjunto de datos sugiere realizar la partición de acuerdo con la columna (que suponemos generada al azar) `Test.Training`. Para ello usaremos la función `split`:

```
olive.01 <- olive
olive.01$Area <- NULL
olive.01 <- split(olive.01, olive.01$Test.Training)
```

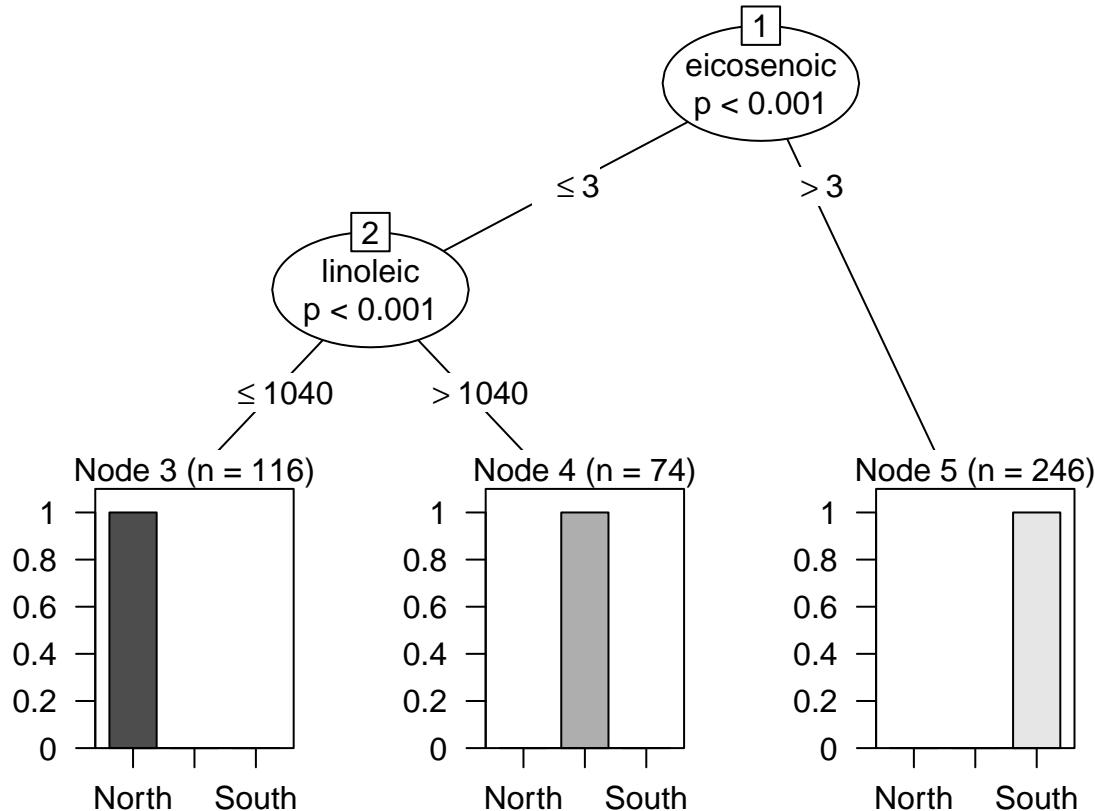
En el código anterior hemos partido el conjunto de datos en dos trozos en función de la columna

Test.Training.

Ejercicio 13.1.2 Inspecciona el objeto olive.01.

Ahora usaremos la parte Training para ajustar el modelo:

```
modelo <- ctree(Region ~ ., data = olive.01$Training)
plot(modelo)
```



El modelo resultante es muy similar construido con todos los datos. Pero ahora podemos hacer predicciones usando la parte Test¹ con la función predict.

```
predicciones <- predict(modelo, olive.01$Test)
```

La práctica totalidad de los modelos estadíticos en R implementan una versión de predict que siempre admite dos argumentos²: el modelo ajustado previamente y un nuevo conjunto de datos (obviamente, con la misma estructura que el usado para crearlo).

Una vez obtenidas las predicciones, se pueden comparar con los valores originales. La expresión

```
mean(predicciones == olive.01$Test$Region)
```

```
## [1] 0.9926471
```

calcula la proporción de aciertos.

Ejercicio 13.1.3 ¿Por qué?

Ejercicio 13.1.4 Cuenta los aciertos y los fallos.

¹Es muy importante no utilizar los datos usados para validar modelos en el ajuste de los mismos.

²Aunque en algunos casos puede admitir algunos más

Para ver dónde se han cometido los errores, se puede hacer

```
table(predicciones, olive.01$Test$Region)
```

```
##  
## predicciones North Sardinia South  
##   North      34       0     0  
##   Sardinia    1      24     0  
##   South       0       0    77
```

que construye una tabla que, idealmente, debería tener una estructura diagonal. Eso correspondería al caso en que cada aceite se asigna a la región a la que realmente pertenece. Sin embargo, el modelo comete un (único) error al asignar a Cerdeña un aceite que, en realidad, es de la región norte.

Ejercicio 13.1.5 Repite el ejercicio anterior (predicción) usando la variable `Área` en lugar de `Region`. En este caso el modelo no es tan bueno porque hay más áreas que regiones y algunas están tan próximas entre sí que es natural que las diferencias entre sus aceites no sean tan acusadas como entre regiones.

Ejercicio 13.1.6 Los bosques aleatorios son más potentes que los árboles que hemos visto arriba. Busca en internet cuál es el paquete y la función que sirve para crear un modelo de bosques aleatorios y repite el ejercicio anterior con él. ¿Funcionan mejor?

El ejercicio anterior sirve de ilustración de cómo en R muchos modelos tienen una interfaz similar. Solo en algunos casos particulares es necesario utilizar sintaxis especiales.

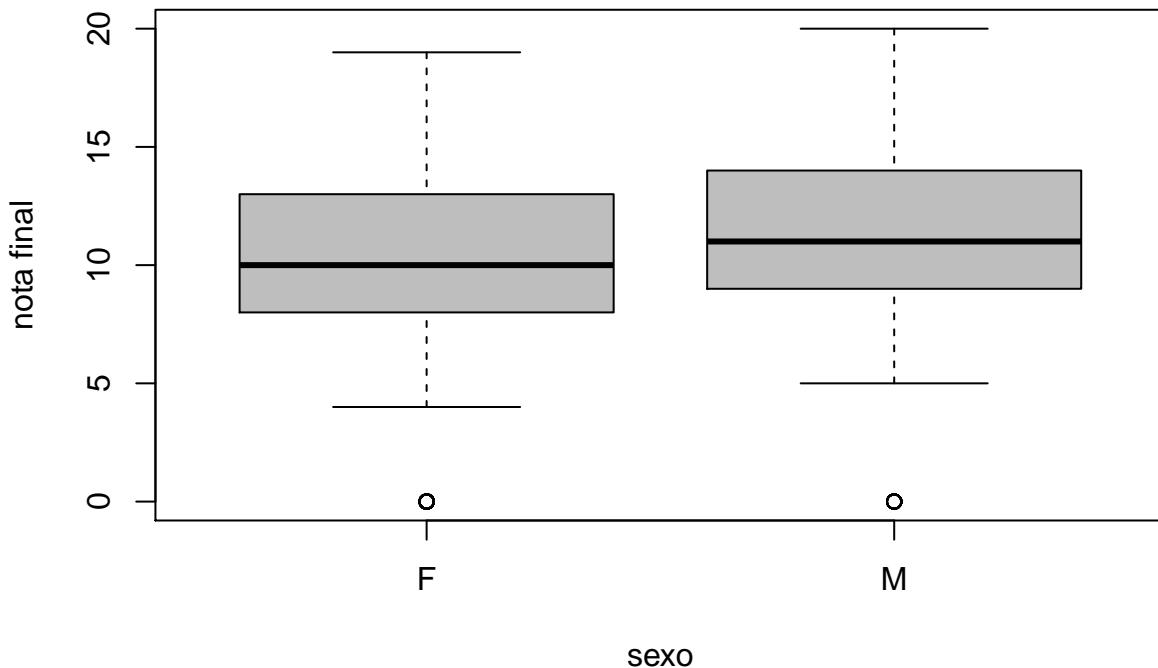
13.2. Igualdad de medias y t-test

Los datos que vamos a usar están bajados de <http://archive.ics.uci.edu/ml/datasets/Student+Performance>. Se refieren al desempeño académico de una serie de alumnos portugueses en función de una serie de indicadores sociodemográficos.

```
mat.por <- read.table("data/student-mat.csv", header = T, sep = ";")
```

Las variables `G1`, `G2` y `G3` se refieren a las notas obtenidas en tres exámenes distintos y nuestro objetivo será analizar las diferencias por sexos en el último examen, `G3`. Primero, gráficamente, con un diagrama de cajas:

```
boxplot(mat.por$G3 ~ mat.por$sex, col = "gray", xlab = "sexo", ylab = "nota final")
```



Se aprecia una pequeña diferencia en favor de los chicos. Es la diferencia observada, que puede deberse al azar. Por eso se puede aplicar una prueba estadística (en este caso, el llamado test de Student o t-test) para determinar si esa diferencia observada es o no *significativa*:

```
res <- t.test(mat.por$G3 ~ mat.por$sex)
res

## 
## Welch Two Sample t-test
##
## data: mat.por$G3 by mat.por$sex
## t = -2.0651, df = 390.57, p-value = 0.03958
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
## -1.85073226 -0.04545244
## sample estimates:
## mean in group F mean in group M
##         9.966346        10.914439
```

El objeto `res` contiene la información relevante sobre la prueba realizada y al ser imprimido muestra un resumen, el típico resumen que suele mostrarse en los estudios en el que se aplica: el intervalo de confianza para la diferencia de medias de interés y el p-valor correspondiente. En este caso el p-valor es ligeramente inferior al umbral habitual de 0.05, por lo que en muchos contextos se habría de *rechazar* la hipótesis de igualdad de medias efectivamente *aceptando* un peor desempeño académico de las chicas en este examen con respecto a sus homólogos masculinos.

El p-valor, por si interesase utilizarlo para otros fines, puede extraerse así:

```
res$p.value
```

```
## [1] 0.039577
```

Ejercicio 13.2.1 `res` es un objeto, ¿de qué clase? También es una lista: examínala. ¿Puedes identificar en ella todos los elementos que forman parte del resumen mostrado más arriba?

Ejercicio 13.2.2 Busca en Google *r non parametric alternative to t-test* e identifica un *test no paramétrico* alternativo al t-test. Úsalo para determinar si las notas finales entre sexos “son o no iguales”.

Pruebas estadísticas similares a las anteriores se realizan habitualmente para, por ejemplo, comparar el desempeño (en términos, p.e., de conversiones) entre dos versiones de una página *web*, de dos campañas publicitarias distintas, de dos tratamientos médicos alternativos, etc.

13.3. Igualdad de medias mediante remuestreos

No es necesario utilizar pruebas estadísticas *de libro* para resolver el problema de la igualdad de medias (entre otros). Es más instructivo (e incluso tiene un alcance más largo) usar los ordenadores para crear pruebas *ad hoc* mediante remuestreos. La idea subyacente es la siguiente:

- El desempeño de dos grupos elegidos al azar nunca va a ser idéntico: tiene una variabilidad natural.
- Creando grupos al azar y midiendo la diferencia de desempeño puede medirse esa variabilidad natural.
- Cuando la diferencia entre dos grupos definidos por una determinada variable excede esa variabilidad natural, podrá decirse que los grupos son distintos. Si no, no habría motivos para ello.

Podemos implementar esa estrategia en R. Para ello, primero, debemos calcular la diferencia entre chicos y chicas:

```
g3 <- mat.por$G3
sex <- mat.por$sex

medias.sexo <- tapply(g3, sex, mean)
diferencia.original <- medias.sexo["M"] - medias.sexo["F"]
```

`diferencia.original` es la diferencia de medias observada en los datos. En el código que sigue vamos a asignar a los alumnos etiquetas al azar y medir las diferencias entre los grupos definidos por ellas. Lo haremos varias veces y luego veremos si la diferencia original es *normal* o si *se sale de rango*.

Para ello, primero, construiremos la función de remuestreo. Esta función aleatoriza el sexo. Eso quiere decir que asigna a los individuos una etiqueta M o F aleatoria y mide las diferencias obtenidas entre ambos grupos. Como las etiquetas se distribuyen al azar, no se esperarían diferencias de comportamiento entre los grupos. Obviamente, los promedios nunca van a ser exactamente iguales y esas diferencias (que son puro ruido estadístico) nos ayudan a determinar mediante su comparación en qué medida la diferencia entre los grupos definidos por el sexo encierran un efecto real.

```
remuestrea <- function(){
  tmp <- tapply(g3, sample(sex), mean)
  tmp["M"] - tmp["F"]
}
```

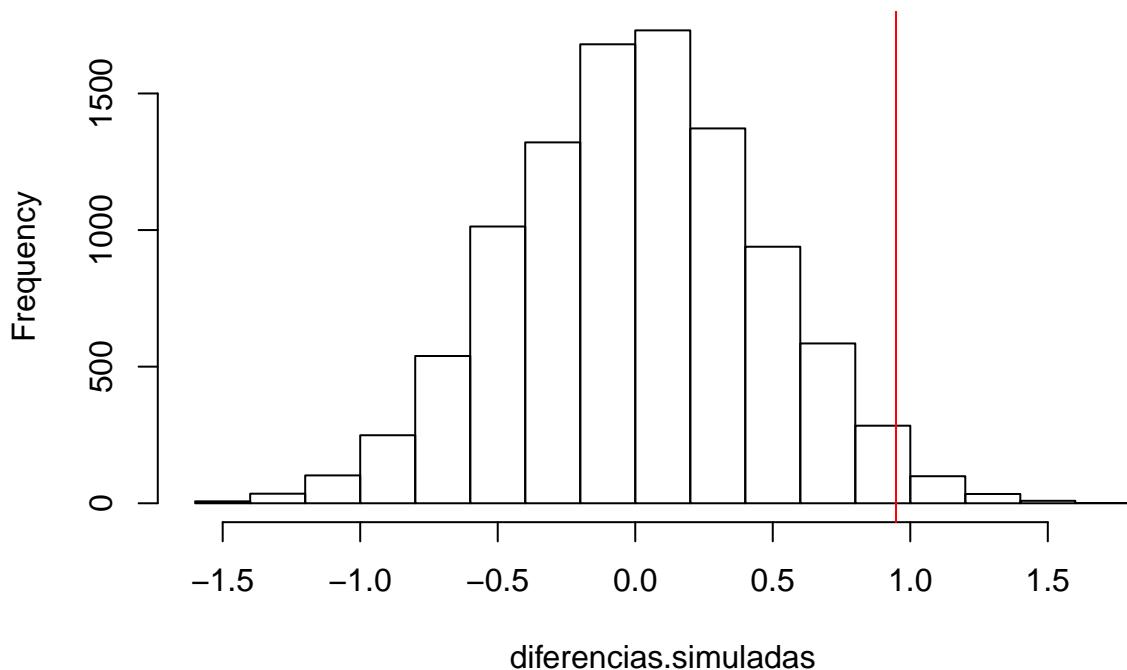
Luego podemos llamar a esa función muchas veces para obtener otras tantas simulaciones:

```
diferencias.simuladas <- replicate(10000, remuestrea())
```

En este momento disponemos de una muestra de 10000 diferencias entre grupos *sin señal*. De otra manera, hemos identificado el nivel de *ruido* del conjunto de datos, que podemos representar con un histograma. Sobre él podemos representar la diferencia real (con una línea vertical roja):

```
hist(diferencias.simuladas)
abline(v = diferencia.original, col = "red")
```

Histogram of diferencias.simuladas



La gráfica obtenida representa la diferencia original (por sexos) dentro del universo de diferencias sin señal creadas mediante remuestreos. El p-valor (que hemos obtenido antes mediante una prueba estadística clásica) no es otra cosa que la proporción de observaciones que caen a la derecha de la línea roja. Lo podemos calcular explícitamente así:

```
mean(diferencias.simuladas > diferencia.original)
```

```
## [1] 0.0186
```

Como se ve, aunque no exactamente igual, es similar al calculado más arriba.

Nota

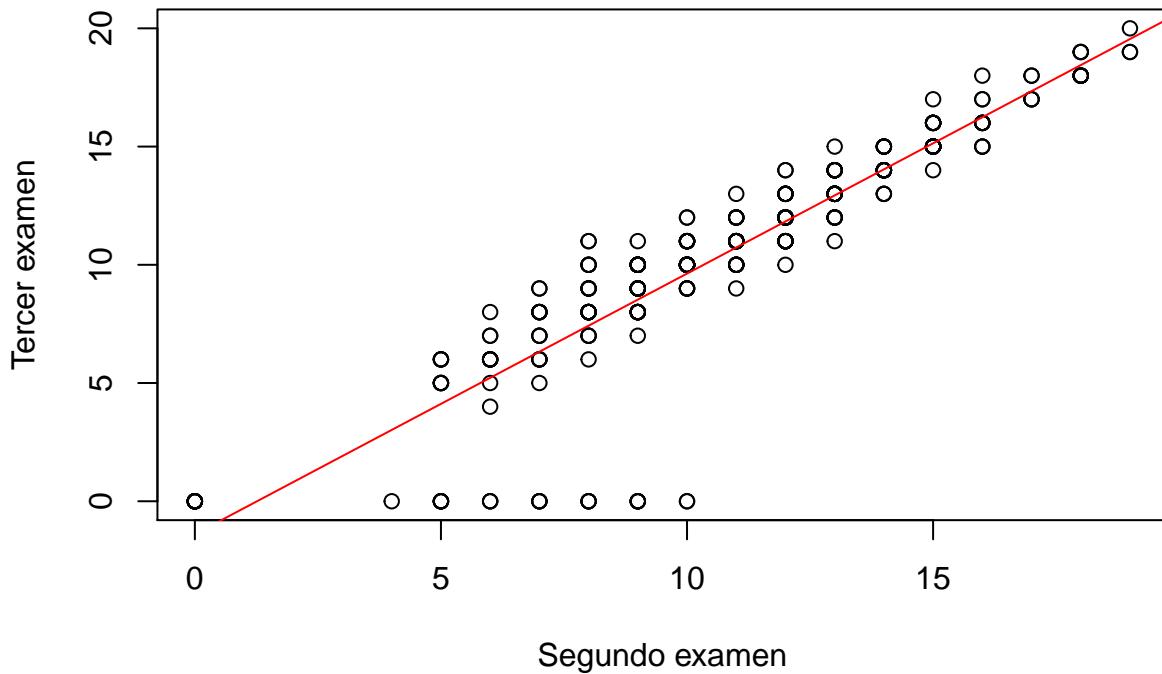
En realidad, está más próxima a la mitad del p-valor calculado arriba. Esto se debe a que antes se ha calculado el p-valor correspondiente a la prueba de que la diferencia de medias sea *igual a cero*. En las simulaciones hemos calculado el de que esta diferencia sea *menor que cero*. En un caso se trata de una de las llamadas pruebas bilaterales y en el otro, de las unilaterales. No vamos a adentrarnos en las diferencias entre ambas.

13.4. Regresión lineal

La regresión lineal (al menos en el caso más simple de una sola variable predictora) identifica la recta óptima que atraviesa una nube de puntos. En el caso que se muestra a continuación, la construida representando gráficamente la nota de los alumnos en los exámenes segundo y tercero, i.e., G2 y G3:

```
plot(mat.por$G2, mat.por$G3,
     main = "Notas en el segundo y tercer examen",
     xlab = "Segundo examen", ylab = "Tercer examen")
modelo.00 <- lm(G3 ~ G2, data = mat.por)
abline(modelo.00, col = "red")
```

Notas en el segundo y tercer examen



En el código anterior hemos representado nuestros datos con una gráfica de dispersión y hemos creado un modelo lineal para encontrar la recta que *mejor* se ajusta a los datos. Luego la hemos representado con la función `abline` en rojo. Advierte cómo `abline` es capaz de interpretar los coeficientes del modelo para representar la recta: no es necesario indicárselos explícitamente.

En las publicaciones científicas se suele resumir un modelo lineal mediante una tabla y algunos indicadores adicionales (como la R^2). La función `summary` proporciona dicha información:

```
summary(modelo.00)
```

```
##
## Call:
## lm(formula = G3 ~ G2, data = mat.por)
##
## Residuals:
##    Min     1Q   Median     3Q    Max 
## -9.6284 -0.3326  0.2695  1.0653  3.5759 
## 
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept) -1.39276   0.29694  -4.69 3.77e-06 ***
## G2          1.10211   0.02615  42.14 < 2e-16 ***
## ---        
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## 
## Residual standard error: 1.953 on 393 degrees of freedom
## Multiple R-squared:  0.8188, Adjusted R-squared:  0.8183 
## F-statistic: 1776 on 1 and 393 DF,  p-value: < 2.2e-16
```

Cabría esperar que la nota del segundo examen fuese muy similar a la del primero. Es decir, que la recta de regresión fuese aproximadamente $y = x$; dicho de otra manera, que los coeficientes del modelo fuesen 0 y 1.

Sin embargo, no ocurre así: por cada punto en el primer examen, los alumnos sacan 1.1 puntos en el segundo; pero comienzan con un *hándicap* de -1.39 puntos. Eso es extraño y por eso se plantea el siguiente ejercicio.

Ejercicio 13.4.1 Las observaciones con valor `G3 == 0`, fácilmente identificables en el gráfico de dispersión mostrado más arriba, parecen estar sesgando el modelo. Elimínalas y repite el análisis anterior.

El ejercicio anterior pone de manifiesto lo fundamental de un análisis visual de los datos previo a su modelización estadística. No obstante, los diagnósticos de la regresión, es decir, el estudio de sus resultados, también permite identificar *a posteriori* este tipo de problemas.

Ejercicio 13.4.2 Estas observaciones pudieran haber afectado también al t-test anterior. Repítelo después de eliminar esas observaciones.

La regresión lineal puede utilizarse con más de una única variable predictora, como a continuación:

```
modelo.10 <- lm(G3 ~ ., data = mat.por)
summary(modelo.10)

##
## Call:
## lm(formula = G3 ~ ., data = mat.por)
##
## Residuals:
##    Min      1Q  Median      3Q     Max 
## -7.9339 -0.5532  0.2680  0.9689  4.6461 
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)    
## (Intercept) -1.115488  2.116958 -0.527  0.598573  
## schoolMS      0.480742  0.366512  1.312  0.190485  
## sexM          0.174396  0.233588  0.747  0.455805  
## age           -0.173302  0.100780 -1.720  0.086380 .  
## addressU       0.104455  0.270791  0.386  0.699922  
## famsizeLE3     0.036512  0.226680  0.161  0.872128  
## PstatusT      -0.127673  0.335626 -0.380  0.703875  
## Medu          0.129685  0.149999  0.865  0.387859  
## Fedu          -0.133940  0.128768 -1.040  0.298974  
## Mjobhealth    -0.146426  0.518491 -0.282  0.777796  
## Mjobother      0.074088  0.332044  0.223  0.823565  
## Mjobservices   0.046956  0.369587  0.127  0.898973  
## Mjobteacher    -0.026276  0.481632 -0.055  0.956522  
## Fjobhealth     0.330948  0.666601  0.496  0.619871  
## Fjobother      -0.083582  0.476796 -0.175  0.860945  
## Fjobservices   -0.322142  0.493265 -0.653  0.514130  
## Fjobteacher    -0.112364  0.601448 -0.187  0.851907  
## reasonhome     -0.209183  0.256392 -0.816  0.415123  
## reasonother     0.307554  0.380214  0.809  0.419120  
## reasonreputation 0.129106  0.267254  0.483  0.629335  
## guardianmother  0.195741  0.252672  0.775  0.439046  
## guardianother    0.006565  0.463650  0.014  0.988710  
## travelttime     0.096994  0.157800  0.615  0.539170  
## studytime       -0.104754  0.134814 -0.777  0.437667  
## failures        -0.160539  0.161006 -0.997  0.319399  
## schoolsupyes    0.456448  0.319538  1.428  0.154043  
## famsupyes       0.176870  0.224204  0.789  0.430710
```

```

## paidyes      0.075764  0.222100  0.341  0.733211
## activitiesyes -0.346047  0.205938 -1.680  0.093774 .
## nurseryyes   -0.222716  0.254184 -0.876  0.381518
## higheryes     0.225921  0.500398  0.451  0.651919
## internetyes   -0.144462  0.287528 -0.502  0.615679
## romanticyes   -0.272008  0.219732 -1.238  0.216572
## famrel        0.356876  0.114124  3.127  0.001912 **
## freetime       0.047002  0.110209  0.426  0.670021
## goout          0.012007  0.105230  0.114  0.909224
## Dalc           -0.185019  0.153124 -1.208  0.227741
## Walc           0.176772  0.114943  1.538  0.124966
## health          0.062995  0.074800  0.842  0.400259
## absences         0.045879  0.013412  3.421  0.000698 ***
## G1              0.188847  0.062373  3.028  0.002645 **
## G2              0.957330  0.053460  17.907 < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.901 on 353 degrees of freedom
## Multiple R-squared:  0.8458, Adjusted R-squared:  0.8279
## F-statistic: 47.21 on 41 and 353 DF,  p-value: < 2.2e-16

```

En este modelo hemos incluido todas las variables predictoras. La tabla obtenida muestra cómo dos de la más importantes para predecir la nota final son G1 y G2. Lo cual tiene mucho sentido. Pero, a la vez, bajo cierto punto de vista, es poco instructivo. Nos gustaría más saber cómo afectan las variables sociodemográficas a la nota obtenida en un examen (y no constatar que quienes sacan buenas notas tienden a seguir sacándolas). Por eso podemos repetir el ejercicio eliminando dichas variables.

```

modelo.11 <- lm(G3 ~ . - G1 - G2, data = mat.por)
summary(modelo.11)

```

```

##
## Call:
## lm(formula = G3 ~ . - G1 - G2, data = mat.por)
##
## Residuals:
##      Min       1Q   Median       3Q      Max 
## -13.0442 -1.9028  0.4289  2.7570  8.8874 
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept) 14.07769   4.48089   3.142  0.00182 ** 
## schoolMS    0.72555   0.79157   0.917  0.35997    
## sexM        1.26236   0.50003   2.525  0.01202 *  
## age         -0.37516   0.21721  -1.727  0.08501 .  
## addressU    0.55135   0.58412   0.944  0.34586    
## famsizeLE3   0.70281   0.48824   1.439  0.15090    
## PstatusT    -0.32010   0.72390  -0.442  0.65862    
## Medu        0.45687   0.32317   1.414  0.15833    
## Fedu        -0.10458   0.27762  -0.377  0.70663    
## Mjobhealth   0.99808   1.11819   0.893  0.37268    
## Mjobother   -0.35900   0.71316  -0.503  0.61500    
## Mjobservices 0.65832   0.79784   0.825  0.40985    
## Mjobteacher  -1.24149   1.03821  -1.196  0.23257    
## Fjobhealth   0.34767   1.43796   0.242  0.80909  

```

```

## Fjobother      -0.61967   1.02304  -0.606   0.54509
## Fjobservices   -0.46577   1.05697  -0.441   0.65972
## Fjobteacher     1.32619   1.29654   1.023   0.30707
## reasonhome      0.07851   0.55380   0.142   0.88735
## reasonother     0.77707   0.81757   0.950   0.34252
## reasonreputation 0.61304   0.57657   1.063   0.28839
## guardianmother   0.06978   0.54560   0.128   0.89830
## guardianother    0.75010   0.99946   0.751   0.45345
## travelttime     -0.24027   0.33897  -0.709   0.47889
## studytime        0.54952   0.28765   1.910   0.05690 .
## failures         -1.72398   0.33291  -5.179   3.75e-07 ***
## schoolsupyes     -1.35058   0.66693  -2.025   0.04361 *
## famsupyes        -0.86182   0.47869  -1.800   0.07265 .
## paidyes          0.33975   0.47775   0.711   0.47746
## activitiesyes    -0.32953   0.44494  -0.741   0.45942
## nurseryyes       -0.17730   0.54931  -0.323   0.74706
## higheryes        1.37045   1.07780   1.272   0.20437
## internetyes      0.49813   0.61956   0.804   0.42192
## romanticyes      -1.09449   0.46925  -2.332   0.02024 *
## famrel            0.23155   0.24593   0.942   0.34706
## freetime          0.30242   0.23735   1.274   0.20345
## goout             -0.59367   0.22451  -2.644   0.00855 **
## Dalc              -0.27223   0.33087  -0.823   0.41120
## Walc              0.26339   0.24801   1.062   0.28896
## health            -0.17678   0.16101  -1.098   0.27297
## absences          0.05629   0.02897   1.943   0.05277 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 4.108 on 355 degrees of freedom
## Multiple R-squared:  0.2756, Adjusted R-squared:  0.196
## F-statistic: 3.463 on 39 and 355 DF,  p-value: 3.317e-10

```

Ejercicio 13.4.3 Examina los R^2 de ambos modelos: ¿cómo han cambiado? ¿Por qué? Nota: el indicador R^2 es una medida de la bondad del ajuste, i.e., de la diferencia entre los datos y sus predicciones. Tiene el valor 1 si el ajuste es perfecto y 0 cuando el modelo no dice absolutamente nada acerca de la variable de interés.

La tabla de coeficientes mostrada más arriba se utiliza para interpretar el modelo: ver qué variables son las más importantes, en qué medida influyen en la variable de interés, etc. Sin embargo, es evidente que la interpretación es complicada cuando el número de coeficientes crece. Modelos como los presentados más arriba, los árboles, son más fáciles de interpretar. De ahí el siguiente ejercicio.

Ejercicio 13.4.4 Usa árboles para tratar de entender mejor el conjunto de datos y las variables que afectan a las notas finales. ¿Ves alguna variable que pueda ayudar a explicar la diferencia de desempeño entre chicos y chicas?

13.5. Regresión logística

La regresión logística se usa para predecir (y explicar) una variable binaria. A continuación estudiaremos un conjunto de datos que trata de explicar los factores que afectan a la admisión de alumnos en determinadas

universidades estadounidenses como la nota en una serie de exámenes previos o la categoría de su escuela de educación secundaria.

Primero, vamos a leer los datos:

```
admitidos <- read.table("data/admitidos.csv", header = T, sep = "\t")
admitidos$rank <- factor(admitidos$rank)
```

Luego, vamos a ajustar el modelo usando la función `glm` (para modelos lineales generalizados). La sintaxis es similar a la usada más arriba con `lm`; cambia, esencialmente, la familia. El modelo logístico corresponde a `family = binomial`; las opciones `gaussian` y `poisson` corresponden al modelo lineal habitual (para modelar variables continuas) y al de Poisson (para modelar conteos).

La función `summary` genera una tabla similar a la obtenida con `lm`.

```
modelo.logistico <- glm(admit ~ ., data = admitidos,
                         family = binomial)
summary(modelo.logistico)

##
## Call:
## glm(formula = admit ~ ., family = binomial, data = admitidos)
##
## Deviance Residuals:
##      Min       1Q   Median       3Q      Max
## -1.6268  -0.8662  -0.6388   1.1490   2.0790
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) -3.989979  1.139951 -3.500 0.000465 ***
## gre          0.002264  0.001094  2.070 0.038465 *
## gpa          0.804038  0.331819  2.423 0.015388 *
## rank2        -0.675443  0.316490 -2.134 0.032829 *
## rank3        -1.340204  0.345306 -3.881 0.000104 ***
## rank4        -1.551464  0.417832 -3.713 0.000205 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
## Null deviance: 499.98 on 399 degrees of freedom
## Residual deviance: 458.52 on 394 degrees of freedom
## AIC: 470.52
##
## Number of Fisher Scoring iterations: 4
```

La variable `rank` es categórica. El coeficiente `rank2` (negativo) muestra cómo la probabilidad de ser admitido decrece en las escuelas del segundo nivel con respecto a las del primero, que es implícito. La misma interpretación tienen los coeficientes `rank3` y `rank4`. R elige el nivel de referencia, en este caso el 1, por estricto orden alfabético salvo que se especifique explícitamente³ lo contrario.

Ejercicio 13.5.1 ¿Cuántos puntos en el GRE son necesarios para compensar el haber estudiado la secundaria en una escuela de segundo nivel y no del primero?

Ejercicio 13.5.2 El coeficiente de la variable `gre` es muy pequeño con respecto al de `gpa`. Eso se debe a la distinta escala en la que se puntúan ambos exámenes. Una de las maneras de comparar la importancia

³La manera excede el alcance del curso.

relativa de los dos exámenes consiste en normalizar los datos. Hazlo, crea otro modelo y discute los resultados.

13.6. Modelos no lineales

En ocasiones un modelo lineal es insuficiente para explicar un fenómeno: la variable objetivo puede no depender linealmente de una variable explicativa. Aunque este tipo de modelos no suelen abordarse en cursos introductorios (ni siquiera de estadística), es ilustrativo mostrar cómo pueden aplicarse de manera sencilla en R.

En el siguiente trozo de código, vamos a cargar un conjunto de datos y ajustador un modelo lineal idéntico al que hubiésemos obtenido con la función `lm`, aunque usando la función `gam` del paquete `mgcv`.

```
library(mgcv)
pisa <- read.table("data/pisasci2006.csv", header = T, sep = ",")
mod.gam.0 <- gam(Overall ~ Income, data = pisa)
summary(mod.gam.0)

##
## Family: gaussian
## Link function: identity
##
## Formula:
## Overall ~ Income
##
## Parametric coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 204.32     35.37   5.777 4.32e-07 ***
## Income      355.85     46.79   7.606 5.36e-10 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## R-sq.(adj) =  0.518 Deviance explained = 52.7%
## GCV = 1504.5 Scale est. = 1448.8 n = 54
```

Ejercicio 13.6.1 Compara los resultados del modelo anterior con los que se obtendrían con `lm`.

Lo interesante es que `gam` permite introducir términos especiales para modelar efectos no lineales. Por ejemplo, en estos datos, los ingresos. El gráfico generado ilustra el impacto (no lineal) del ingreso sobre la variable objetivo.

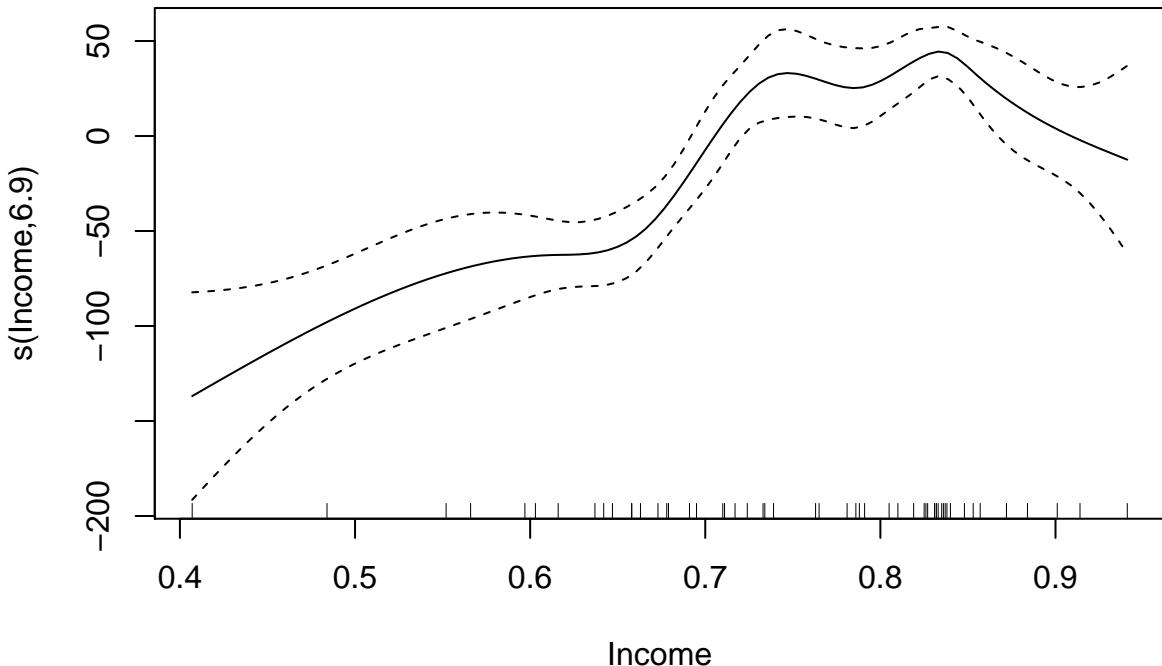
```
mod.gam.1 <- gam(Overall ~ s(Income, bs = "cr"), data = pisa)
summary(mod.gam.1)
```

```
##
## Family: gaussian
## Link function: identity
##
## Formula:
## Overall ~ s(Income, bs = "cr")
##
## Parametric coefficients:
##             Estimate Std. Error t value Pr(>|t|)
```

```

## (Intercept) 470.444      4.082   115.3   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Approximate significance of smooth terms:
##          edf Ref.df    F p-value
## s(Income) 6.895 7.741 16.67 1.59e-14 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## R-sq.(adj) =  0.7  Deviance explained = 73.9%
## GCV = 1053.7  Scale est. = 899.67 n = 54
plot(mod.gam.1)

```



Ejercicio 13.6.2 Usa predict con ambos modelos para estimar su correspondiente error cuadrático medio (la media de la diferencia entre los valores reales y estimados al cuadrado).

13.7. Clústering con k-medias

La función `kmeans` de R aplica el algoritmo de las k-medias para encontrar grupos de observaciones similares. Puedes visitar <https://www.datanalytics.com/2016/04/18/visualizacion-de-k-medias-y-dbscan/> para ver una animación que muestra cómo funciona el algoritmo.

Vamos a ilustrar el uso de este algoritmo con R usando `iris` para tratar de agrupar los registros en tres grupos por sus variables numéricas (una limitación de k-medias):

```

dat <- iris[,1:4]
res <- kmeans(dat, 3)

```

Ejercicio 13.7.1 Investiga el objeto `res`. Busca en particular el grupo al que el algoritmo asocia cada observación.

Ejercicio 13.7.2 Compara los clústers obtenidos con las especies (conocidas). Nota: puede servirte hacer `table(iris$Species, res$cluster)`.

Ejercicio 13.7.3 Repite el ejercicio anterior 3 o 4 veces: ¿obtienes los mismos (o equivalentes) resultados cada vez?

Ejercicio 13.7.4 Haz lo mismo pero usando 4 clústers.

13.8. Resumen y referencias

13.8.1. Árboles

En este capítulo hemos usado árboles en un problema de clasificación. Los árboles tienen una historia que se remonta a finales de los años setenta y fueron *presentados en sociedad* en el libro de 1984 *Classification and Regression Trees*, de Leo Breiman y otros coautores. Desde entonces se han creado muchas versiones de los árboles (CART, C4.5, etc.). Los árboles originales, los descritos en el libro anterior, están implementados en el paquete `rpart` de R. Nosotros, sin embargo, hemos usado los de `party`, que están basados en desarrollos teóricos posteriores.

Los árboles, en cualquiera de sus versiones, tienen las características indicadas en este capítulo: son simples, excesivamente simples y su poder predictivo es limitado, pero son fáciles de interpretar. Existen dos estrategias para mejorar el poder predictivo de los árboles. La primera es crear muchos de ellos y promediarlos. Es la que siguen los bosques aleatorios sugeridos más arriba. La segunda es crear iterativamente listas de árboles de manera que el i-ésimo corrija los errores cometidos por los anteriores. Es la estrategia subyacente a `xgboost` y sus versiones.

En ambos casos se pierde en gran medida la interpretabilidad de los modelos. Sin embargo, como contrapartida, tanto los bosques aleatorios como los basados en la técnica del *boosting* permiten construir modelos con un poder predictivo muy alto.

13.8.2. Pruebas de hipótesis

En R están implementadas infinidad de pruebas estadísticas: `t.test`, `wilcox.test`, `prop.test`, etc. De hecho, la inmensa mayoría de las que se enseñan en los cursos de introducción a la estadística. Sin embargo, R nos permite ir más allá de las limitaciones inherentes a este tipo de pruebas. Por ejemplo, tal y como hemos ilustrado en esta sección, nos permite crear nuestras propias pruebas mediante remuestreos.

Pero también, y aunque no nos hemos ocupado de eso, también nos permite subsumir pruebas estadísticas en modelos estadísticos que permiten considerar fuentes alternativas de información, i.e., el efecto *confusor* de otras variables observadas sobre el fenómeno. Y esto tanto bajo un punto de vista estrictamente frecuentista como desde otro, más moderno, bayesiano que da respuesta a los problemas metodológicos asociados a las pruebas de hipótesis, los p-valores, los errores de tipo I y II, etc.

13.8.3. Modelos lineales y sus extensiones

El modelo lineal ilustrado en esta sección tiene una larga historia y aunque hay pocos contextos en los que hoy en día sea la opción de modelado preferible, mucha de la estadística y la ciencia de datos moderna consiste en generalizaciones suyas. En esta sección hemos explorado algunas de ellas, como el modelo logístico (como ejemplo de los llamados modelos lineales generalizados) y los modelos lineales generalizados (GAM), pero podríamos haber tratado las regresiones penalizadas (*ridge*, *lasso*, *elasticnet*, etc.).

Una ventaja de R es que proporciona una interfaz relativamente homogénea y previsible para todos estos tipos de modelos: todos cuentan con una función `predict`, `summary`, etc.

13.8.4. Clústering

En esta sección hemos presentado el algoritmo de clústering más utilizado, k-medias. No obstante, existen alternativas como las implementadas en el paquete `cluster`. Una alternativa cada vez más popular a los métodos clásicos anteriores es DBSCAN (véase <https://cran.r-project.org/web/packages/dbscan/index.html>).

13.9. Resumen y referencias

TBA

13.10. Ejercicios adicionales

XXX

Capítulo 14

Ejemplos y casos de uso

Este capítulo contiene algunos ejemplos más largos y elaborados del uso de R para resolver determinados problemas. Ilustran el uso de algunos conceptos presentados en el hilo del libro, pero resultan demasiado prolijos para ser incluidos en flujo principal, que se ha tratado de mantener lo más sencillo posible.

14.1. Una calculadora de hipotecas con R

Esta sección incluye una discusión que ilustra el uso de la vectorización en un contexto menos trivial que los presentados anteriormente. Como tal, puede omitirse en una primera lectura.

Al cabo de 4 años, 1000 euros depositados al 3% valen

```
1000 * (1 + 3 / 100)^4
```

Es sencillo deducir que para recibir 1000 euros dentro de 4 años hay que invertir

```
1000 / (1 + 3 / 100)^4
```

euros ahora al 3%. Esa cantidad es el llamado *valor presente neto* de esos 1000 euros futuros. El valor presente neto de una determinada cantidad futura depende de tres factores: de la cantidad, del plazo y del tipo de interés.

El precio actual de una casa es el *valor presente neto* de todas las cantidades que se pagarán mensualmente a lo largo de la vida de la hipoteca. Si pagas 500 euros al mes durante treinta años y tu hipoteca está al 3%, su valor presente es

$$\sum_{i=1}^{12*30} \frac{500}{(1 + 0,03/12)^i}.$$

En R,

```
interes.mensual <- 3 / 12 / 100
meses <- 1:(12*30)
deflactor <- (1 + interes.mensual)^meses      # un vector
deflactor <- 1 / deflactor                      # otro vector
valor.actual <- sum(500 * deflactor)
```

En la expresión anterior hemos construido un vector de meses (desde 1 hasta $12 * \text{plazo}$), hemos operado vectorialmente sobre él para obtener los valores presentes de cada mensualidad y, finalmente, los hemos

sumado. Técnicamente, es la suma de lo que en bachillerato se conoce como *progresión geométrica*. Ahora vamos a hacer lo contrario: dado un capital, un plazo y un interés, calcular la cuota.

```
capital <- 100000
anyos <- 20
interes <- 3
interes.mensual <- interes / 12 / 100
meses <- 1:(anyos*12)
cuota <- capital / sum(1 / (1+interes.mensual)^meses)
```

Capítulo 15

Apéndice

Esta sección contiene algún material adicional

15.1. Distribuciones de probabilidad

Esta sección puede omitirse en una primera lectura y no es esencial para lo que sigue. No obstante, puede resultar interesante para aquellos lectores con una formación matemática más sólida.

Es frecuente querer crear vectores que sigan una determinada distribución de probabilidad. Por ejemplo, pueden obtenerse vectores con una distribución uniforme (en $[0, 1]$) o normal (estándar):

```
x.uniforme <- runif(10)  
x.normal    <- rnorm(13)
```

Ejercicio 15.1.1 Casi todas las distribuciones admiten parámetros adicionales. Por ejemplo, la media y la desviación estándar para la distribución normal. Consulta la ayuda de `rnorm` para ver cómo muestrear una variable aleatoria normal con media 1 y desviación estándar 3. Extrae una muestra de 10000 elementos de ella y comprueba que lo has hecho correctamente usando las funciones `mean` y `sd`.

Para entender la forma de esas distribuciones, podemos construir el histograma de una muestra, como a continuación:

```
hist(rnorm(1000))  
hist(runif(1000))  
hist(rpois(1000, 5))
```

Ejercicio 15.1.2 Busca cómo muestrear la distribución gamma.

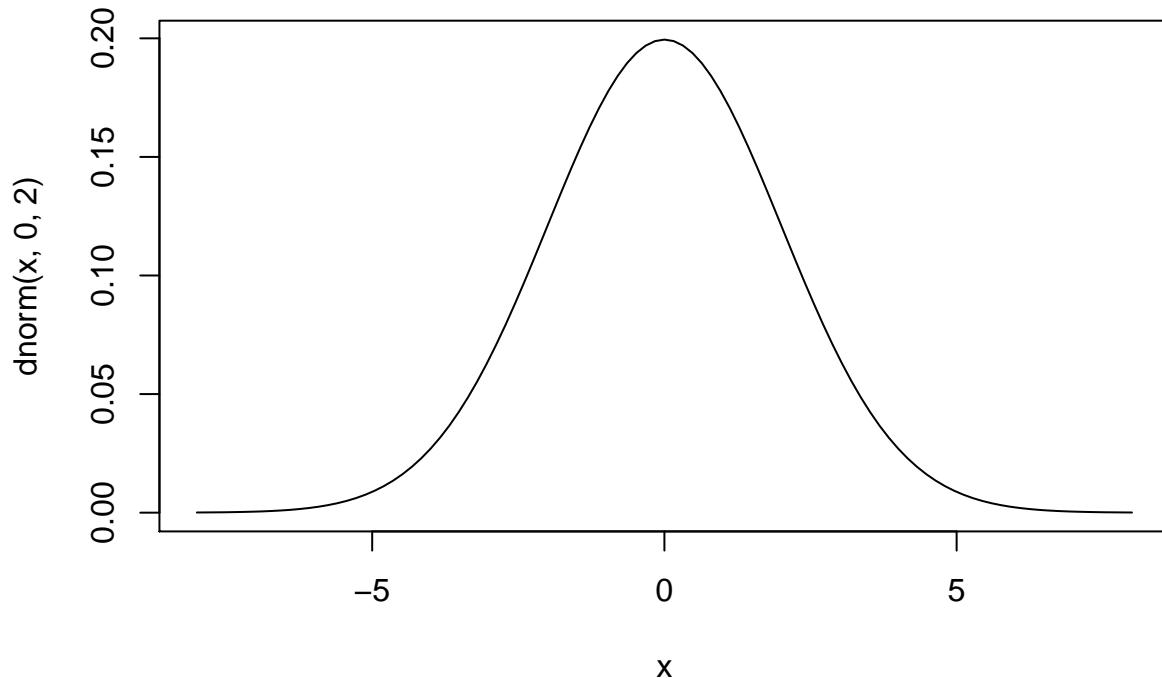
Ejercicio 15.1.3 Consulta la ayuda de `rnorm`, `runif` y `rpois`. ¿Qué tienen en común?

El ejercicio anterior debería poner de manifiesto cómo en R, asociada a cada distribución de probabilidad hay cuatro funciones cuyos nombres comienzan por las letras `r`, `p`, `d` y `q`. La función que comienza por `r` sirve para muestrear la distribución, como en los ejemplos anteriores.

La función que comienza por `d` es la densidad¹ de la distribución. Por ejemplo, la función `dnorm` es la famosa campana de Gauss. Se puede representar con la función `curve`:

¹ Los conceptos de función de densidad, de probabilidad, etc. son muy importantes, aunque no en el resto del libro. Si estás familiarizado con ellos, te servirá lo que sigue; si no, puedes ignorarlo en una primera lectura.

```
curve(dnorm(x, 0, 2), -8, 8)
```



Nota

La función de densidad tiene la forma *ideal* a la que convergen los histogramas de las muestras de la distribución. Estos histogramas serán más parecidos a ella conforme mayor sea el tamaño de la muestra. Usando términos matemáticos, aunque con cierto abuso del lenguaje, la función de densidad es el *límite* de los histogramas.

Ejercicio 15.1.4 Usa `curve` para representar la densidad de la distribución beta para diversos valores de sus parámetros. Lee `?curve` para averiguar cómo sobreimpresionar curvas y representa la densidad para diversas combinaciones de los parámetros con colores distintos. Puedes comparar el resultado con los gráficos que aparecen en la página de la distribución beta en la Wikipedia.

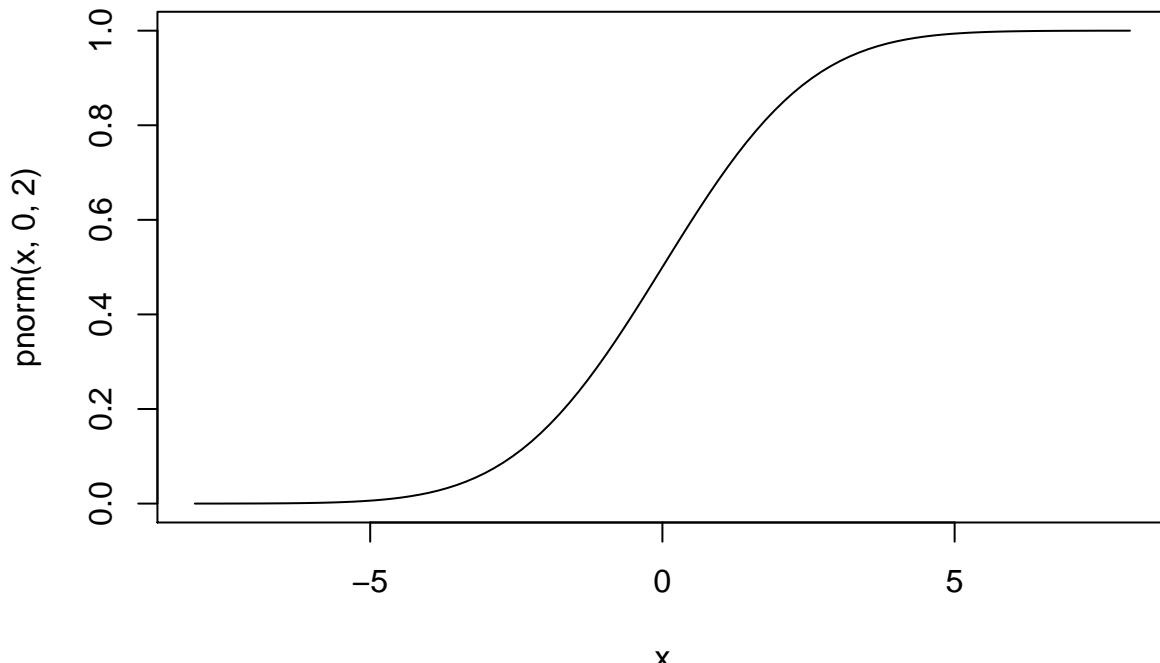
Ejercicio 15.1.5 Toma una distribución cualquiera y representa el histograma. Usa el ejercicio anterior para sobreimpresionar la función de densidad sobre el histograma. Nota: recuerda que el histograma puede representar frecuencias o proporciones; usa las segundas.

La función que comienza por `p` es la función de probabilidad, que es la integral de la densidad. En concreto, si la función de densidad es f , la función de probabilidad, F , es

$$F(x) = \int_{-\infty}^x f(x)dx.$$

Como consecuencia, es una función que crece más o menos suavemente de 0 a 1.

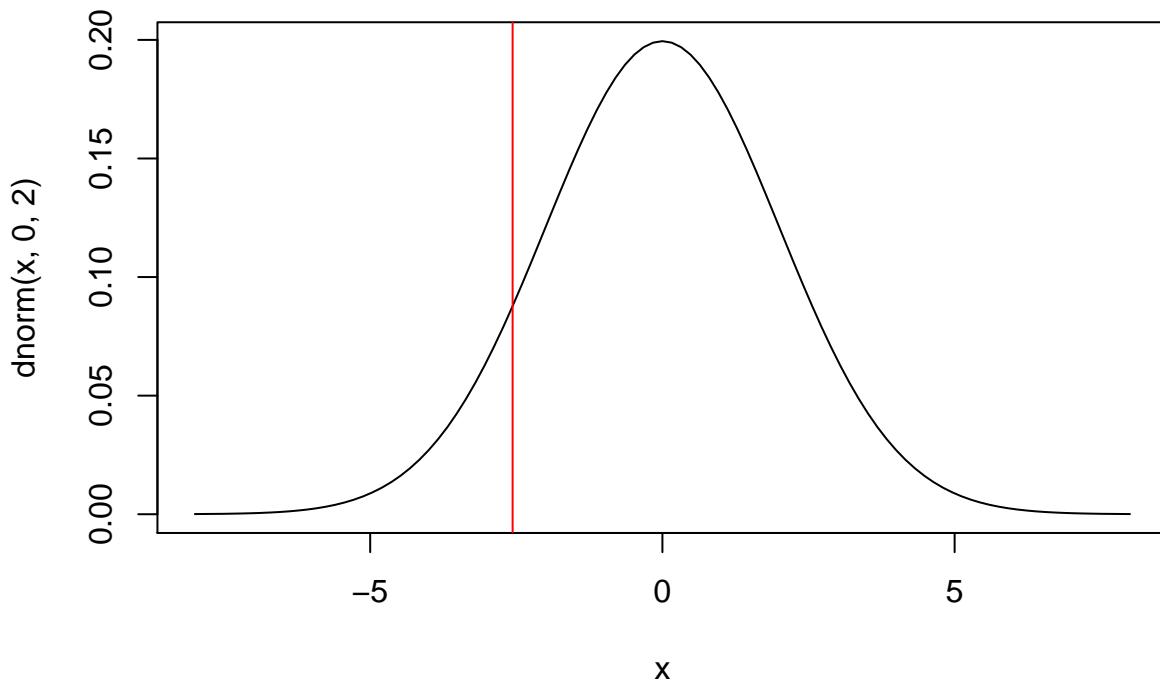
```
curve(pnorm(x, 0, 2), -8, 8)
```

**Nota**

Si X es una variable aleatoria con una función de probabilidad F , entonces $F(x) = P(X < x)$. Los eventos $X < x$ son tan importantes que sus probabilidades se *precalculan* en F .

Finalmente, la función cuyo nombre comienza por `q` es la que calcula los cuantiles. Es decir, es la inversa de la función de probabilidad. Por ejemplo, en el gráfico

```
curve(dnorm(x, 0, 2), -8, 8)
abline(v = qnorm(0.1, 0, 2), col = "red")
```



la probabilidad que asigna la normal a la zona que queda a la izquierda de la recta vertical roja es del 10 %, valor indicado por el primer argumento de `qnorm`, 0.1.