

Información general

Exploración del contenido

Estadística descriptiva

Funciones básicas

Aplicación a estructuras complejas

La función `describe()`

Agrupamiento de datos

Tablas de contingencia

Discretización de valores

Agrupamiento y selección

Ordenación de datos

Generación de rankings

Particionamiento de los datos

6. Análisis exploratorio

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

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

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

6.1 Información general

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

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

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

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

Ejercicio 6.1 Obtención de información general de los datos

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

[1] "data.frame"

> typeof(iris) # Tipo del objeto

[1] "list"

>
> # Información sobre su estructura

> str(iris)

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

6.1.1 Exploración del contenido

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

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


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

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

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

Sintaxis 6.4 `summary(objeto)`

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

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

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

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

Ejercicio 6.2 Exploración del contenido de un *data frame*

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

Sepal.Length	Sepal.Width	Petal.Length	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

```
> head(iris) # Primeras filas
```

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

```
> tail(iris) # Últimas filas
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
145	6.7	3.3	5.7	2.5	virginica
146	6.7	3.0	5.2	2.3	virginica
147	6.3	2.5	5.0	1.9	virginica

```

148      6.5      3.0      5.2      2.0 virginica
149      6.2      3.4      5.4      2.3 virginica
150      5.9      3.0      5.1      1.8 virginica

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

[1] 7.0 6.4 6.9 5.5 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6
[16] 6.7 5.6 5.8 6.2 5.6 5.9 6.1 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7
[31] 5.5 5.5 5.8 6.0 5.4 6.0 6.7 6.3 5.6 5.5 5.5 6.1 5.8 5.0 5.6
[46] 5.7 5.7 6.2 5.1 5.7

```

6.2 Estadística descriptiva


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

6.2.1 Funciones básicas

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

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

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

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

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

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

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

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

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

Devuelve el valor promedio existente en el vector facilitado como parámetro. Si no se facilita el parámetro `trim` este tomará por defecto el valor 0 y el resultado será la media aritmética. Cualquier otro valor asignado a dicho parámetro, en

el rango (0, 0.5], provocará que antes de calcular el promedio se recorten de manera simétrica la porción de muestras indicada de cada extremo. Esto permite obviar la presencia de valores extremos (*outliers*), muy pequeños o muy grandes, que podrían sesgar la media.

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

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

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

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

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

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

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

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

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

Ejercicio 6.3 Funciones básicas de estadística descriptiva

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

	media	desviacion	varianza	minimo
	4.934783	2.280370	5.200089	1.000000
	maximo	mediana	rango1	rango2
	9.000000	4.967391	1.000000	9.000000
quartiles.0%	quartiles.25%	quartiles.50%	quartiles.75%	
	1.000000	4.000000	4.967391	6.750000
quartiles.100%				
	9.000000			

6.2.2 Aplicación a estructuras complejas

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

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

Ejercicio 6.4 Selección de una variable de un objeto complejo

```
> mean(iris$Sepal.Length) # Seleccionamos una variable del dataset  
[1] 5.843333
```

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

Sintaxis 6.13 `lapply(objeto, función)`

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

Sintaxis 6.14 `sapply(objeto, función)`

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

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

Ejercicio 6.5 Aplicar una función a múltiples variables de un objeto

```
> lapply(iris[,1:4], mean)  
$Sepal.Length  
[1] 5.843333  
  
$Sepal.Width  
[1] 3.057333  
  
$Petal.Length  
[1] 3.758  
  
$Petal.Width  
[1] 1.199333
```

La función entregada como segundo parámetro puede estar predefinida, como es el caso de `mean()` o cualquier otra de las antes enumeradas, pero también es posible defi-

nirla en ese mismo punto. Para ello se usará la sintaxis `function(par) operación-a-efectuar`, siendo `par` cada uno de los elementos generados por `lapply()/sapply()` a partir del objeto entregado como primer argumento.

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

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

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

[1] 5.006 5.936 6.588
```

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

Sintáxis 6.15 `which(vector)`

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

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

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

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

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

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

[1] 5.936

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

[1] 5.936
```

6.2.3 La función describe()

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

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

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

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

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

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

ebay

```
8 Variables      1972 Observations
```

```
-----
```

Category

	n missing	unique
1972	0	18

```
Antique/Art/Craft (177, 9%)
Automotive (178, 9%), Books (54, 3%)
Business/Industrial (18, 1%)
Clothing/Accessories (119, 6%)
Coins/Stamps (37, 2%)
Collectibles (239, 12%)
Computer (36, 2%), Electronics (55, 3%)
EverythingElse (17, 1%)
Health/Beauty (64, 3%)
Home/Garden (102, 5%), Jewelry (82, 4%)
Music/Movie/Game (403, 20%)
Photography (13, 1%)
Pottery/Glass (20, 1%)
SportingGoods (124, 6%)
Toys/Hobbies (234, 12%)
-----
```

currency


```

      n missing  unique
1972         0        3

```

EUR (533, 27%), GBP (147, 7%), US (1292, 66%)

sellerRating

```

      n missing  unique   Mean   .05   .10   .25   .50
1972         0    461  3560  50.0  112.1  595.0 1853.0
.75   .90   .95
3380.0 5702.8 22501.0

```

```

lowest :    0    1    4    5    6
highest: 25433 27132 30594 34343 37727

```

Duration

```

      n missing  unique   Mean
1972         0        5  6.486

```

```

          1  3  5  7 10
Frequency 23 213 466 967 303
%         1 11 24 49 15

```

endDay

```

      n missing  unique
1972         0        7

```

```

          Fri Mon Sat Sun Thu Tue Wed
Frequency 287 548 351 338 202 171  75
%         15 28 18 17 10  9  4

```

ClosePrice

```

      n missing  unique   Mean   .05   .10   .25   .50
1972         0    852  36.45  1.230  2.241  4.907  9.995
.75   .90   .95
28.000 80.999 153.278

```

```

lowest :   0.01   0.06   0.10   0.11   0.17
highest: 820.00 860.00 863.28 971.00 999.00

```

OpenPrice

```

      n missing  unique   Mean   .05   .10   .25   .50
1972         0    569  12.93   0.01   0.99   1.23   4.50
.75   .90   .95
9.99  24.95  49.99

```

```

lowest :   0.01000  0.01785  0.10000  0.25000  0.50000
highest: 300.00000 359.95000 549.00000 650.00000 999.00000

```

Competitive.

n	missing	unique	Sum	Mean
1972	0	2	1066	0.5406

6.3 Agrupamiento de datos

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

6.3.1 Tablas de contingencia

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

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

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

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

Ejercicio 6.9 Generación de tablas de contingencia de datos

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

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

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

	setosa	versicolor	virginica
4.3	1	0	0
4.4	3	0	0
4.5	1	0	0
4.6	4	0	0

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

6.3.2 Discretización de valores

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

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

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

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

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

Ejercicio 6.10 Tabla de contingencia sobre valores discretizados

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

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

6.3.3 Agrupamiento y selección

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

Sintaxis 6.20 `split(objeto, variableFactor)`

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



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

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

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

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

Ejercicio 6.11 División de un dataset en grupos

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

> str(bySpecies)

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

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

[1] 5.006
```

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

Ejercicio 6.12 Agrupamiento y selección de datos

```
> str(coverttype)
```

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

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

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

> # Selección aleatoria
> subcovertypes <- covertypes[sample(1:nrow(covertypes),
+                                   nrow(covertypes)*.1),]

> str(covertypes)

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

```
30 29 ...
$ class : Factor w/ 7 levels "1","2","3","4",...: 5 5 2 2 5 ...
```

6.4 Ordenación de datos

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

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

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

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

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

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

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

Ejercicio 6.13 Ordenación de los datos

```
> valores # Vector original

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



```
> sort(valores) # Vector ordenado

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

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

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

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

Ejercicio 6.14 Ordenación de los datos

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

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
23	4.6	3.6	1.0	0.2	setosa
14	4.3	3.0	1.1	0.1	setosa
15	5.8	4.0	1.2	0.2	setosa
36	5.0	3.2	1.2	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
17	5.4	3.9	1.3	0.4	setosa

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

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
132	7.9	3.8	6.4	2.0	virginica
136	7.7	3.0	6.1	2.3	virginica
118	7.7	3.8	6.7	2.2	virginica
123	7.7	2.8	6.7	2.0	virginica

119	7.7	2.6	6.9	2.3 virginica
106	7.6	3.0	6.6	2.1 virginica

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

6.4.1 Generación de rankings

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

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

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

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

Ejercicio 6.15 Generación de rankings a partir de los datos

```
> valores    # Valores a tratar

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

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

[1] 3.5 17.0 34.5 17.0 3.5 28.5 17.0 10.0 34.5 23.5 40.5 23.5
[13] 10.0 7.0 28.5 45.0 48.5 17.0 10.0 3.5 48.5 17.0 3.5 34.5
```

```
[25] 3.5 34.5 28.5 17.0 48.5 45.0 28.5 10.0 17.0 40.5 34.5 40.5
[37] 17.0 40.5 23.5 28.5 23.5 45.0 48.5 40.5 10.0 3.5 28.5 34.5
[49] 17.0 40.5

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

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

6.5 Particionamiento de los datos

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

En realidad, puesto que sabemos cómo seleccionar parte de las filas de un *data frame*, no tendremos problema alguno en usar la función `nrow()` para obtener el número de filas y, a partir de ahí, dividir el conjunto de observaciones en las partes que nos interese. Es lo que se hace en el siguiente ejercicio, en el que se obtiene una partición de entrenamiento y otra de test:

Ejercicio 6.16 Particionamiento de datos en conjuntos disjuntos

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

[1] 112

> nrow(test)

[1] 38

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

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

Ejercicio 6.17 Particionamiento de datos en conjuntos disjuntos

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

```
$training
[1] 112
```

```
$test
[1] 38
```

```
> particion$test      # Contenido del conjunto de test
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.1	3.5	1.4	0.2	setosa
13	4.8	3.0	1.4	0.1	setosa
15	5.8	4.0	1.2	0.2	setosa
16	5.7	4.4	1.5	0.4	setosa
19	5.7	3.8	1.7	0.3	setosa
27	5.0	3.4	1.6	0.4	setosa
30	4.7	3.2	1.6	0.2	setosa
31	4.8	3.1	1.6	0.2	setosa
32	5.4	3.4	1.5	0.4	setosa
33	5.2	4.1	1.5	0.1	setosa
36	5.0	3.2	1.2	0.2	setosa
37	5.5	3.5	1.3	0.2	setosa
47	5.1	3.8	1.6	0.2	setosa
48	4.6	3.2	1.4	0.2	setosa
49	5.3	3.7	1.5	0.2	setosa
55	6.5	2.8	4.6	1.5	versicolor
61	5.0	2.0	3.5	1.0	versicolor
67	5.6	3.0	4.5	1.5	versicolor
73	6.3	2.5	4.9	1.5	versicolor
75	6.4	2.9	4.3	1.3	versicolor
76	6.6	3.0	4.4	1.4	versicolor
79	6.0	2.9	4.5	1.5	versicolor
81	5.5	2.4	3.8	1.1	versicolor
85	5.4	3.0	4.5	1.5	versicolor
88	6.3	2.3	4.4	1.3	versicolor
104	6.3	2.9	5.6	1.8	virginica
118	7.7	3.8	6.7	2.2	virginica
120	6.0	2.2	5.0	1.5	virginica
123	7.7	2.8	6.7	2.0	virginica
133	6.4	2.8	5.6	2.2	virginica
139	6.0	3.0	4.8	1.8	virginica
140	6.9	3.1	5.4	2.1	virginica
142	6.9	3.1	5.1	2.3	virginica

143	5.8	2.7	5.1	1.9	virginica
144	6.8	3.2	5.9	2.3	virginica
146	6.7	3.0	5.2	2.3	virginica
147	6.3	2.5	5.0	1.9	virginica
149	6.2	3.4	5.4	2.3	virginica