

Introducción a R y RStudio

Manejo de datos

Índice

Ideas clave	3
5.1. Introducción y objetivos	3
5.2. Importando datos desde un archivo	4
5.3. Leer datos desde un paquete	24
5.4. Guardar datos desde R	25
5.5. Manipulación de datos: una introducción al <i>tidyverse</i>	32
5.6. Referencias bibliográficas	43
5.7. Cuaderno de ejercicios	43

5.1. Introducción y objetivos

Un aspecto importante del uso de R y RStudio es el manejo de datos. Hasta ahora, hemos trabajado con datos ya existentes en R base o que hemos generado nosotros mismos, sin embargo, lo usual, es que usemos datos almacenados en archivos externos a R, siendo necesario importarlos. También es posible que esa información esté alojada en la web o algún otro tipo de recurso.

Por otra parte, también resultará necesario exportar nuestros objetos de R a archivos en nuestra computadora.

En este tema estudiaremos las funciones que necesitaremos utilizar para importar datos desde fuentes externas y para exportar datos en diferentes formatos, explicando los procedimientos a seguir en cada caso.

Luego, centraremos el estudio a la manipulación de los datos utilizando funciones de R base como también, de la colección de paquetes *tidyverse*.

Al finalizar este tema, esperamos alcanzar los siguientes objetivos:

- ▶ Ser capaces de importar y exportar ficheros de datos en R.
- ▶ Ser capaces de usar las principales acciones de manipulación de *data frames* con pipes en `{dplyr}`.
- ▶ Comprender cómo combinar `group_by()` y `summarize()` para obtener resúmenes de *datasets*.

5.2. Importando datos desde un archivo

Ya sabemos que R es un lenguaje de programación orientado al análisis de datos. Lo primero que tenemos que hacer para empezar un análisis con datos en R es, evidentemente, cargar los datos. Ya hemos usado datos en varios ejemplos, pero éstos siempre han estado ya almacenados como objetos R o los hemos generado nosotros mismos (por ejemplo utilizando `rnorm()`). Pero lo usual al trabajar con datos es importarlos a R desde un archivo, una base de datos u otras fuentes.

5.2.1. Importar archivos TXT

La función básica para leer archivos TXT es `read.table()`. De acuerdo a lo que encontramos en la documentación de esta función, `read.table()` lee un archivo en formato de tabla y crea un *data frame* a partir de él, con los casos, individuos u observaciones en las filas y las variables o atributos en las columnas.

Al importar archivos TXT en R rara vez necesitemos más argumentos de los que veremos a continuación, pero en caso de que quieras conocer todos los argumentos, puedes encontrarlos en la documentación de la función [read.table](#) o llamando a la ayuda de la función con `?read.table`.

La sintaxis básica de esta función es

```
read.table(file, header = FALSE, sep = "", dec = ".")
```

donde:

- `file`: nombre del archivo TXT del cual se van a leer los datos indicado como *string* (es decir, entre comillas). Si el archivo no se encuentra en la carpeta definida como directorio de trabajo, es necesario incluir la ruta hasta el mismo. Esta ruta puede especificarse en forma completa desde la raíz de la estructura de archivos o en forma relativa desde el directorio de trabajo.

- ▶ **header**: un valor lógico (TRUE o FALSE) que indica si el archivo contiene los nombres de las variables en su primera fila.
- ▶ **sep**: Separador de las columnas del archivo. Los separadores más comunes son la coma (","), punto y coma (";"), espacio en blanco (" ") y tabuladores ("\t").
- ▶ **dec**: Carácter utilizado para separar decimales de los números en el archivo. Frecuentemente será un punto ((".")), pero en algunos archivos de datos el punto decimal puede estar especificado con una coma (",").
- ▶ **stringsAsFactors**: este argumento lo utilizamos cuando queremos que, al leer el archivo, los datos de texto se conviertan automáticamente a factores. Si este no es el comportamiento que deseamos, definimos este argumento como FALSE (aunque esta es la opción por defecto, con lo que bastará no especificar nada).

Es importante señalar que el objeto obtenido al usar esta función es siempre un *data frame*.

Ejemplo: cargar datos desde un archivo TXT

Vamos a cargar el archivo de datos `medidas_cuerpo.txt`. Este fichero tiene la particularidad de que, en las primeras filas del archivo, y precedidos por el símbolo #, se incluyen comentarios explicativos de los datos. Ello implica que debemos indicarle a R que esas filas no corresponden a datos sino a comentarios. Esto lo hacemos utilizando el argumento `comment.char="#"`.

```








medidas_cuerpo.txt: Bloc de notas
Archivo  Editar  Ver

# A continuación un conjunto de medidas corporales tomadas
# a un grupo de estudiantes en una universidad de Medellín.
# A continuación la variables de la base de datos:
# edad: edad del estudiante en años
# peso: peso del estudiante en kilogramos
# altura: estatura del estudiante en centímetros
# sexo: género del estudiante
# muneca: diametro de la muñeca derecha en centímetros
# biceps: diametro del biceps derecho en centímetros
edad peso altura sexo muneca biceps
43 87.3 188.0 Hombre 12.2 35.8
65 80.0 174.0 Hombre 12.0 35.0
45 82.3 176.5 Hombre 11.2 38.5
37 73.6 180.3 Hombre 11.2 32.2
55 74.1 167.6 Hombre 11.8 32.9
33 85.9 188.0 Hombre 12.4 38.5
25 73.2 180.3 Hombre 10.6 38.3
35 76.3 167.6 Hombre 11.3 35.0
  
```

Figura 1: previsualización del archivo medidas_cuerpo.txt.

```
datos1 <- read.table("medidas_cuerpo.txt", comment.char = "#", header = TRUE, stringsAsFactors = TRUE)
```

Al ejecutar esta instrucción, estamos creando el objeto `datos1` en la sesión de trabajo actual, con los datos contenidos en el archivo.

Environment	History	Connections	Tutorial
   Import Dataset ▾  194 MiB ▾ 			
R ▾  Global Environment ▾			
Data			
 <code>datos1</code>		37 obs. of 6 variables	



Al ejecutar la función `read.table`, creamos el objeto `datos1`

Figura 2: Objeto que se crea al ejecutar `read.table`.

Notar que hemos utilizado además el argumento `stringsAsFactors = TRUE`. Esto hará que la variable `sexo` sea considerada como un factor (es decir, una variable cualitativa con niveles fijos). Podemos utilizar la función `str()` para ver la estructura del *data frame*. Veamos esto.

```
str(datos1)
## 'data.frame': 36 obs. of 6 variables:
## $ edad : int  43 65 45 37 55 33 25 35 28 26 ...
## $ peso : num  87.3 80 82.3 73.6 74.1 85.9 73.2 76.3 65.9 90.9 ...
## $ altura: num  188 174 176 180 168 ...
## $ sexo : Factor w/ 2 levels "Hombre","Mujer": 1 1 1 1 1 1 1 1 1 1 ...
## ...
## $ muñeca: num  12.2 12 11.2 11.2 11.8 12.4 10.6 11.3 10.2 12 ...
## $ biceps: num  35.8 35 38.5 32.2 32.9 38.5 38.3 35 32.1 40.4 ...
```

Como podemos observar de la salida de `str(datos1)`, el objeto creado es efectivamente un *data frame*, formado por 36 filas (observaciones) y 6 columnas (variables). Las variables son todas cuantitativas (esto en la salida está indicado por `int` (tipo *integer*) y `num` (tipo *numeric*) excepto la variable `sexo`, que es un factor con 2 niveles: Factor w/ 2 levels "Hombre", "Mujer".

Ejercicio: vuelve a cargar este archivo de datos, pero ahora sin especificar el argumento `stringsAsFactors = TRUE` y utilizando la función `str()` observa las diferencias.

Para leer archivos de datos registrados en ficheros de texto plano también podemos usar las funciones `read.delim` y `read.delim2`. Estas funciones permiten tratar

archivos delimitados de forma predeterminada. La sintaxis y argumentos predeterminados de estas funciones son:

```
read.delim(file, header = TRUE, sep = "\t", quote = "\"", dec = ".", fill = TRUE, comment.char = "", ...)
```

```
read.delim2(file, header = TRUE, sep = "\t", quote = "\"", dec = ",", fill = TRUE, comment.char = "", ...)
```

En la Tabla 1 se resumen los valores por defecto de los argumentos más utilizados de las tres funciones vistas.

Función	header	sep	dec
read.table	FALSE	" "	"."
read.delim	TRUE	"\t"	"."
read.delim2	TRUE	"\t"	","

Tabla 1: Valores por defecto de argumentos.

Saltar filas con el argumento skip

En ocasiones, los archivos TXT contienen algunas líneas de texto antes del conjunto de datos. En el ejemplo eso es precisamente lo que ocurría. Y lo solucionamos utilizando el argumento `comment.char="#"`. Otra posibilidad es utilizar el argumento `skip = filas_a_ignorar`, que por defecto está establecido en 0. En el ejemplo hay 9 líneas de texto antes de los datos, con lo que podemos leer el archivo de la siguiente manera:

```
datos2 <- read.table("medidas_cuerpo.txt", skip = 9, header = TRUE, stringsAsFactors = TRUE)
```

Y podemos utilizar la función `head()` para visualizar las primeras filas de `datos2`

```
head(datos2, 3)
##   edad peso altura  sexo muneca biceps
## 1   43 87.3  188.0 Hombre   12.2   35.8
## 2   65 80.0  174.0 Hombre   12.0   35.0
## 3   45 82.3  176.5 Hombre   11.2   38.5
```

Como podemos ver, efectivamente se ha saltado las primeras líneas de texto.

Importar TXT desde la web a R

En caso de querer importar un archivo TXT alojado en algún sitio web, podemos abrirlo sin necesidad de descargarlo. Solo necesitamos pasar la URL como cadena al argumento `file` de la función.

```
url <- "http://www.biostatisticien.eu/springeR/imcenfant.txt"
datos3 <- read.table(url, header = TRUE)
head(datos4, 3)
## SEXE zep poids an mois taille
## 1    F    0  16.0  3     5  100.0
## 2    F    0  14.0  3    10   97.0
## 3    G    0  13.5  3     5   95.5
```

Descargar los archivos desde R

Vale la pena señalar que también podemos descargar los archivos de internet usando R con la función `download.file()`.

La función `download.file()` nos pide como argumento `url`, la dirección de internet del archivo que queremos descargar y `destfile` el nombre que tendrá el archivo en nuestra computadora. Ambos argumentos como cadenas de texto, es decir, entre comillas.

```
url <- "http://www.biostatisticien.eu/springeR/imcenfant.txt"
download.file(url, destfile = 'IMCenfant.txt')
```

Al ejecutar estas instrucciones, se descargará en el directorio de trabajo el archivo de nombre `IMCenfant.txt`, como podemos comprobar en la pestaña `file` de RStudio:

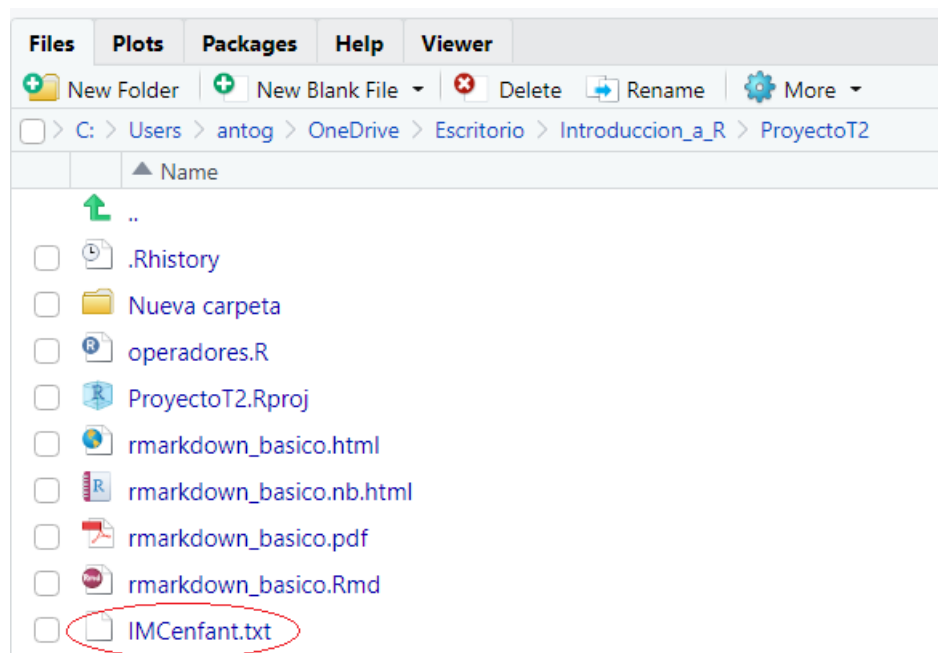


Figura 3: Vista del archivo descargado desde la pestaña Files de RStudio

Una vez descargado, podemos cargarlo a la sesión de trabajo como lo hicimos anteriormente:

```
datos4 <- read.table("IMCenfant.txt", header = TRUE)
head(datos4, 3)
##   SEXE zep poids an mois taille
## 1    F  0  16.0  3    5  100.0
## 2    F  0  14.0  3   10   97.0
## 3    G  0  13.5  3    5   95.5
```

5.2.2. Importar archivos CSV

Un caso particular de las tablas, son los archivos separados por comas, con extensión `.csv` (por *Comma Separated Values*). Este es un tipo de archivo comúnmente utilizado para compartir datos, pues es compatible con una amplia variedad de sistemas diferentes además de que ocupa relativamente poco espacio de almacenamiento.

Para ejemplificar, vamos a descargar los datos `gapminder_es.csv` desde el repositorio de [Github](#) de *R para ciencia de datos/datos-de-miercoles*.

Para importar estos datos podría utilizar la misma función `read.table()` que vimos antes, pero para documentos con extensión `.csv` tenemos la posibilidad de usar

alguna de las funciones `read.csv()` o `read.csv2()` que son versiones de `read.table()`, optimizadas para importar archivos `.csv`.

Estas funciones aceptan los mismos argumentos que `read.table()`, pero al usarla con un archivo `.csv`, en la mayoría de los casos, no hará falta especificar nada más que la ruta del archivo.

La sintaxis y argumentos predeterminados de estas funciones son:

```
# Por defecto coma (,) como separador y punto (.) como separador decimal
read.csv(file, header = TRUE, sep = ",", quote = "\"", dec = ".", fill =
TRUE, comment.char = "", encoding = "unknown", ...)

# Por defecto punto y coma (;) como separador y coma (,) como separador
decimal
read.csv2(file, header = TRUE, sep = ";", quote = "\"", dec = ",", fill =
TRUE, comment.char = "", encoding = "unknown", ...)
```

donde los argumentos son los que hemos especificado ya con la función `read.table()`. Como podemos observar la única diferencia entre estas funciones son los valores por defecto en los argumentos `sep` y `dec`. En países que utilizan comas como separador decimal (la mayoría de los países europeos, países de América del Sur, entre otros), para crear archivos CSV se necesita el punto y coma y, por lo tanto, para importar archivos con decimales, es necesario cambiar los argumentos por defecto de la función `read.csv`, o usar directamente la función `read.csv2`.

La Tabla 2 resume los valores predeterminados para los tres argumentos principales:

Función	header	sep	dec
<code>read.csv</code>	TRUE	“,”	“.”
<code>read.csv2</code>	TRUE	“;”	“,”

Tabla 2: Valores por defecto de argumentos principales.

```
download.file(
  url = "https://raw.githubusercontent.com/cienciadedatos/datos-de-
miercoles/master/datos/2019/2019-04-24/gapminder_es.csv",
  destfile = "gapminder_es.csv"
```

```
)
datos5 <- read.csv("gapminder_es.csv")
# Resultado
head(datos5)
##      pais continente anio esperanza_de_vida poblacion pib_per_capita
## 1 Afganistan      Asia 1952          28.801   8425333      779.4453
## 2 Afganistan      Asia 1957          30.332   9240934      820.8530
## 3 Afganistan      Asia 1962          31.997  10267083      853.1007
```

Tanto `read.csv()` como `read.csv2()` devuelven un *data frame* como resultado

5.2.3. Importar una hoja de Excel

R base no tiene una función para importar archivos almacenados en archivos con extensión `.xls` y `.xlsx`, creados con Excel.

Para importar datos desde este tipo de archivos, necesitamos instalar el paquete `{readxl}`, que contiene funciones específicas para realizar esta tarea.

Una vez instalado el paquete, lo cargamos en la sesión de trabajo actual con la instrucción `library(readxl)`.

De este paquete utilizaremos la función `read_excel()` para importar archivos `.xls` y `.xlsx` y `excel_sheets()` para obtener los nombres de las hojas del archivo Excel, en caso de ser necesario.

Para probar el uso de esta función, vamos a descargar una hoja de cálculo de prueba.

```
direccion <- "https://www.lock5stat.com/datasets3e/NutritionStudy.xlsx"
destino <- "NutritionStudy.xlsx"
download.file(
  url = direccion,
  destfile = destino,
  mode = "wb"
)
```

Notar que hemos establecido el argumento `mode = "wb"` para asegurar que el archivo se descargue correctamente (necesario al descargar archivos de Excel).

Una vez descargado el archivo, lo importamos a la sesión de trabajo mediante la función `read_excel()` y mostramos las primeras filas con la función `head()`

```
library(readxl)
datos6 <- read_excel(destino)
head(datos6)
> head(datos6)
# A tibble: 6 × 17
  ID Age Smoke Quetelet Vitamin Calories Fat Fiber Alcohol Choles...1
  <dbl> <dbl> <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 1 64 No 21.5 1 1299. 57 6.3 0 170.
2 2 76 No 23.9 1 1032. 50.1 15.8 0 75.8
3 3 38 No 20.0 2 2372. 83.6 19.1 14.1 258.
4 4 40 No 25.1 3 2450. 97.5 26.5 0.5 333.
5 5 72 No 21.0 1 1952. 82.6 16.2 0 171.
6 6 40 No 27.5 3 1367. 56 9.6 1.3 155.
# ... with 7 more variables: BetaDiet <dbl>, RetinolDiet <dbl>,
# BetaPlasma <dbl>, RetinolPlasma <dbl>, Sex <chr>, VitaminUse <chr>,
# PriorSmoke <dbl>, and abbreviated variable name 'Cholesterol'
# Use `colnames()` to see all variable names
```

Figura 4: Primeras filas de `datos6` (dataset de tipo `.xlsx` importado con `read_excel` del paquete `readxl`)

Como podemos ver, el objeto que se crea es un *tibble* (de hecho, todas las funciones del *tidyverse* que se usan para importar datos crean *tibbles*).

La función `read_excel()` detecta automáticamente el tipo de archivo que estamos leyendo (`xls` o `xlsx`) llamando internamente a la función `excel_format()` para determinar si la ruta es `xls` o `xlsx`, según la extensión del archivo y el archivo en sí. El paquete también cuenta con las funciones `read_xls()` y `read_xlsx()` directamente si sabemos el tipo de archivo que estamos leyendo y queremos evitar las conjeturas.

Los formatos de Microsoft Excel le permiten tener más de una hoja de cálculo en un archivo. Estos se conocen como hojas (*sheets* en inglés). Las funciones enumeradas anteriormente leen la primera hoja por defecto, pero también podemos leer las otras. La función `excel_sheets` nos da los nombres de todas las hojas en un archivo de Excel. Estos nombres entonces se pueden pasar al argumento `sheet` en las tres funciones anteriores para leer hojas distintas a la primera.

Veamos un ejemplo donde necesitamos indicar el argumento `sheet`. Para ello vamos a descargar una base de datos sobre consumo de alimentos en España por CCAA a partir de los ficheros que publica el ministerio competente [aquí](#).

```

direccion <- "https://www.mapa.gob.es/es/alimentacion/temas/consumo-
tendencias/2021datos anuales del panel de consumo alimentario en hogares v2_tc
m30-623605.xlsx"
destino <- "datos_consumo.xlsx"
download.file(
  url = direccion,
  destfile = destino,
  mode = "wb"
)

```

Si probamos a cargar el *dataset* como lo hicimos antes (solamente especificando el nombre del archivo), vemos que no obtenemos el resultado deseado:

```

datos_consumo <- read_excel(destino)
head(datos_consumo)
# A tibble: 4 × 1
##   `DATOS ANUALES DEL PANEL DE CONSUMO ALIMENTARIO EN HOGARES`
##                                     <dbl>
## 1                                     NA
## 2                                     NA
## 3                                     NA
## 4                                    2021

```

En caso de que tengamos instalado Excel o algún otro programa compatible con archivos de hoja de cálculo, como *LibreOffice Calc* o *Number*, podemos explorar su contenido.

La función `excel_sheets()` nos devuelve el nombre de las hojas como un vector.

```

excel_sheets(destino)
## [1] "PORTADA"          "VALOR"
## [3] "VOLUMEN"          "PENETRACIÓN"
## [5] "PRECIO"           "CONSUMOXCÁPITA"
## [7] "GASTOXCÁPITA"     "CANALES VALOR"
## [9] "CANALES VOLUMEN"  "CANALES PENETRACIÓN"
## [11] "SOCIOECONÓMICO VALOR" "SOCIOECONÓMICO VOLUMEN"
## [13] "SOCIOECONÓMICO PENETRACIÓN" "SOCIOECONÓMICO CONSUMOXCÁPITA"

```

Como podemos ver, el archivo cuenta con 14 hojas. Supongamos que queremos trabajar con la hoja "PRECIO". Entonces, al importar los datos utilizando la función `read_excel()`, vamos a indicarles el argumento `sheet = "PRECIO"`.

```

datos_consumo_precio <- read_excel(destino, sheet = "PRECIO")
head(datos_consumo_precio)
# A tibble: 6 × 19
  CONSUMO EN...1 ...2 ...3 ...4 ...5 ...6 ...7 ...8 ...9 ...10
...11
  <chr>          <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr>
<chr>
1 TAM PRECIO ... NA    NA    NA    NA    NA    NA    NA    NA    NA    NA

```

```

2 NA          T.ES... CATA... ARAG... ILLE... COMU... REGI... ANDA... COMU... CAST...
EXTR...
3 .TOTAL ALIM... 2.51... 2.70... 2.63... 2.36... 2.39... 2.39... 2.33... 2.69... 2.27...
2.24...
4 T.HUEVOS KGS 2.43... 2.56... 2.37... 2.43... 2.31... 2.32... 2.40... 2.32... 2.17...
2.24...
5 T.HUEVOS UN... 0.15... 0.16... 0.14... 0.15... 0.14... 0.14... 0.15... 0.14... 0.13...
0.14...
6 TOTAL HUEVO... 0.15... 0.16... 0.14... 0.15... 0.14... 0.14... 0.15... 0.14... 0.13...
0.14...
# ... with 8 more variables: ...12 <chr>, ...13 <chr>, ...14 <chr>,
# ...15 <chr>, ...16 <chr>, ...17 <chr>, ...18 <chr>, ...19 <chr>,
and
# abbreviated variable name 1`CONSUMO EN HOGARES`
# i Use `colnames()` to see all variable names

```

Aún seguimos teniendo un resultado que no nos gusta. Pero el problema ahora es otro: las primeras filas del archivo no contienen, sino que estos comienzan en la fila 3. Podemos solucionar esto indicándole a la función `read_excel()` que se salte las primeras 2 filas:

```

datos_consumo_precio <- read_excel(
  destino,
  sheet = "PRECIO",
  skip = 2
)
datos_consumo_precio
# A tibble: 681 × 19
  ...1      T.ESP...1 CATAL...2 ARAGÓN ILLES...3 COMUN...4 REGIÓN...5 ANDAL...6
COMUN...7
  <chr>      <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
<dbl>
1 .TOTAL ... 2.52     2.71     2.63     2.37     2.40     2.40     2.34     2.70
2 T.HUEVO... 2.44     2.56     2.38     2.44     2.31     2.32     2.41     2.33
3 T.HUEVO... 0.153     0.160     0.145     0.151     0.145     0.146     0.150
0.146
4 TOTAL H... 0.155     0.163     0.149     0.153     0.147     0.147     0.152
0.148
5 OTRAS A... 0.0806     0.0749     0.0733     0.0911     0.0807     0.0825     0.0833
0.0782
6 MIEL      6.89      7.08      7.12      6.92      6.30      7.24      6.28      6.90
7 GRANEL    8.49      9.12      7.73      8.35      7.43      8.98      8.11      9.09
8 ENVASADA  6.08      6.19      6.52      6.61      5.81      6.05      5.35      6.38
9 TOTAL C... 7.17      7.79      7.24      7.11      6.83      6.95      6.62      7.50
10 CARNE C... 8.45      9.32      9.01      8.04      8.19      8.27      7.39      9.10
# ... with 671 more rows, 10 more variables: `CASTILLA - LA MANCHA` <dbl>,
# EXTREMADURA <dbl>, `CASTILLA Y LEÓN` <dbl>, GALICIA <dbl>,
# `PRINCIPADO DE ASTURIAS` <dbl>, CANTABRIA <dbl>, `PAIS VASCO` <dbl>,
# `LA RIOJA` <dbl>, `C. FORAL DE NAVARRA` <dbl>, CANARIAS <dbl>, and
# abbreviated variable names 1`T.ESPAÑA`, 2`CATALUÑA`, 3`ILLES BALEARS`,
# 4`COMUNITAT VALENCIANA`, 5`REGIÓN DE MURCIA`, 6`ANDALUCÍA`,

```

```
# 7` COMUNIDAD DE MADRID`  
# i Use `print(n = ...)` to see more rows, and `colnames()` to see all  
variable names
```

Esta vez hemos tenido éxito y los datos importados son los correctos.

También podemos especificar otros argumentos de ser necesario. Los más frecuentes son:

- ▶ `range`: Cadena de texto con el rango de celdas a importar, escrito con el formato usado en Excel. Por ejemplo, "A1:B10".
- ▶ `col_names`: Con este argumento indicamos si la hoja que vamos a importar tiene encabezados para usar como nombres de columna. Por defecto su valor es `TRUE` y utiliza la primera fila como nombres de las columnas. Si no tenemos encabezados, podemos dar un vector con nombres para asignar a las columnas.

Como siempre, si quieres consultar todos los argumentos de la función, puedes recurrir a la ayuda ejecutando `?read_excel`.

5.2.4. Importar datos desde el menú de RStudio

Cuando trabajamos con RStudio es posible previsualizar el archivo de datos y cómo será importado. Esto suele ser útil en caso de necesitar comprobar la presencia de encabezados, separadores especiales entre columnas, etc. Podemos acceder a esta funcionalidad de RStudio de dos formas:

- ▶ Barra de herramientas general: `File -> Import Dataset`.

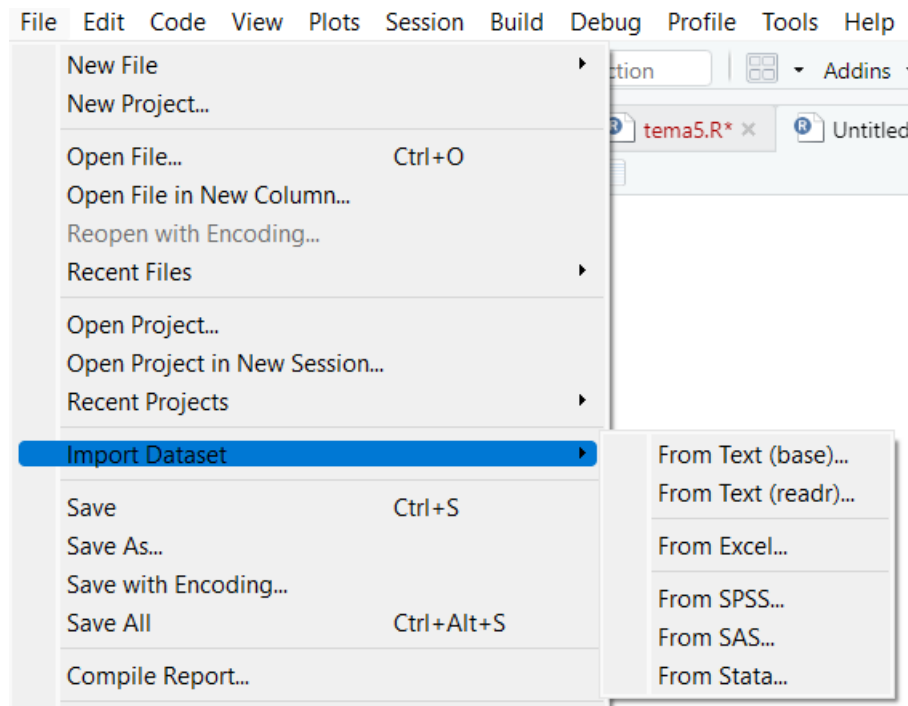


Figura 5: Importar datos usando barra de herramientas de RStudio

- Desde el panel *Environment*, Import Dataset

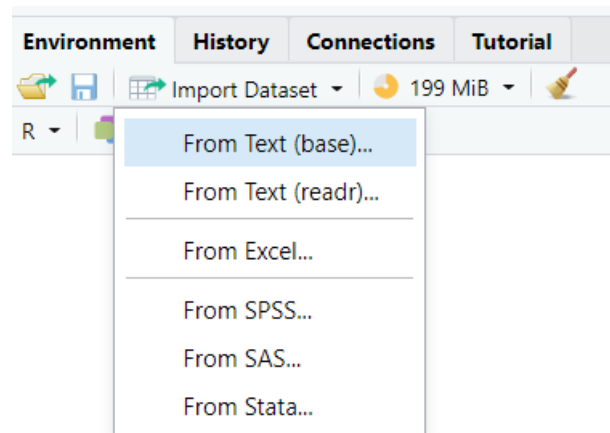


Figura 6: Importar datos usando el menú de la pestaña *environment* de RStudio

Al hacer clic allí, se nos pide seleccionar el tipo de archivo que queremos importar y si queremos usar las funciones de base o de otros paquetes. RStudio nos ofrece dos posibilidades: una de ellas usando las funciones disponibles en R base (seleccionando *From Text (base)*), y la segunda es usar los paquetes `{readr}`, `{readxl}` o `{haven}` del conjunto *tidyverse* (seleccionando *From text (readr)*, *From Excel*, *From SPSS*, *From SAS* o *From Stata*). Al activar una de estas opciones se abrirá un cuadro de dialogo

que nos permitirá en primer lugar seleccionar el archivo que queremos importar. Una vez seleccionado, podremos:

- ▶ Especificar el nombre asignado al objeto de R de tipo *data frame* en el que se almacenarán los datos leídos. Por defecto, tomará el nombre del archivo, colapsando espacios y algunos caracteres especiales.
- ▶ Especificar la presencia o no de encabezados, lo que se puede comprobar en el fragmento previsualizado.
- ▶ Especificar el separador usado entre columnas. Típicamente, si no hemos especificado el separador correcto, la previsualización del *data frame* como se verá una vez importado no tendrá la estructura de columnas correctas y toda la información se mostrará comprimida en una única columna. Es posible seleccionar separadores distintos hasta dar con el correcto.
- ▶ Especificar cómo están codificados los valores perdidos, en caso de existir cadenas de caracteres especiales para ello. Esto puede resultar especialmente útil en casos en que esta codificación no es uniforme a lo largo de todo el conjunto de datos. Los *strings* especificados son convertidos en NA al leer el archivo.

Al completar la selección de opciones en el cuadro de diálogo, RStudio copiará a la consola la traducción de nuestras opciones a un comando válido de R. Verificar esta línea suele ser una buena forma de familiarizarse con el uso de las funciones de importación de datos y sus distintos argumentos.

Por ejemplo, las selecciones realizadas en el cuadro de diálogo que muestra la Figura 6, al clicar sobre *Import*, produce la instrucción:

```
datos7 <- read.csv("C:/ruta_al_archivo/gapminder_es.csv", stringsAsFactors  
= TRUE)
```

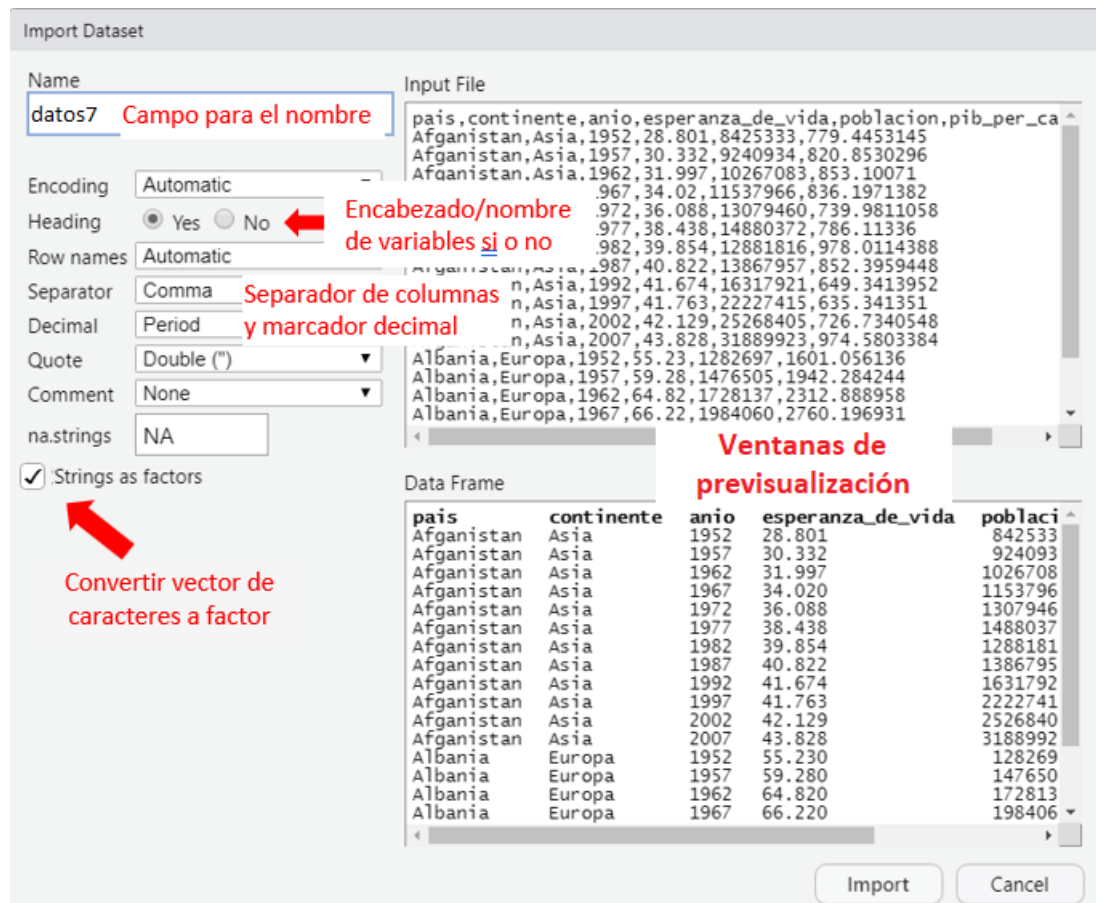


Figura 7: Cuadro de diálogo para importar datos desde el menú de RStudio utilizando las funciones de R base.

Finalmente, veamos un ejemplo de cómo importar datos de manera interactiva utilizando las funciones del *tidyverse*. Por ejemplo, veamos nuevamente como cargar los datos de consumo de alimentos en España por CCAA.

Ejemplo: importar un archivo de Excel utilizando el menú de RStudio
Comenzamos por clicar en *From Excel* en el panel *Environment* -> *Import Dataset*

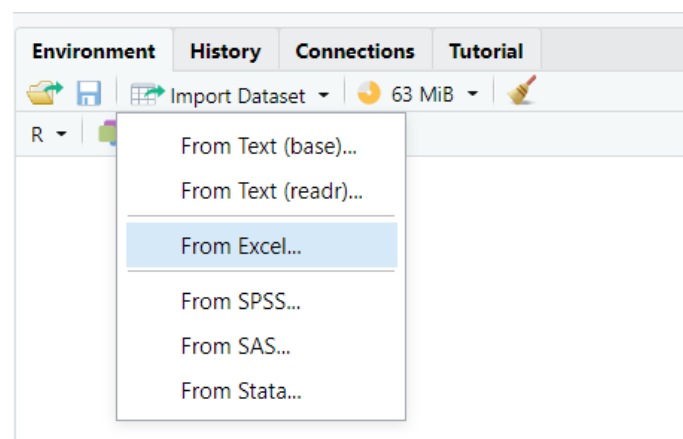


Figura 8: Importar archivo de Excel desde el menú de la pestaña *environment* de RStudio
Al clicar allí, se abrirá un cuadro de diálogo como el que se muestra en la Figura 9.

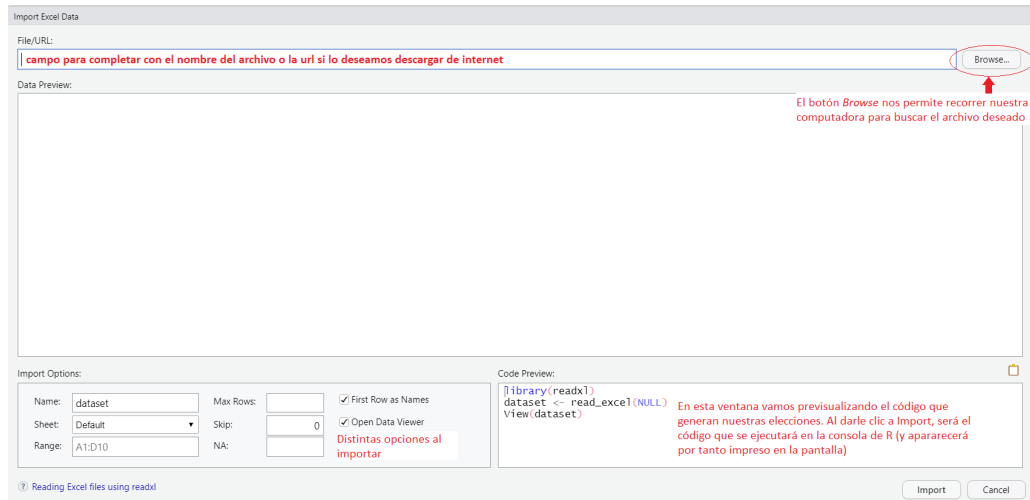


Figura 9: Cuadro de diálogo para importar datos utilizando funciones del tidyverse.

Vamos a probar importar el archivo introduciendo la url del mismo (recordar que al cargarlo utilizando las funciones desde un script, primero tuvimos que descargar el archivo con la función `download.file()` y luego leerlo con la función `read_excel`. Vamos a ver que, leyendo los datos de forma interactiva, los dos pasos (descarga y lectura) se resuelven de forma simultánea. Es importante destacar que la url debe introducirse sin utilizar comillas (de lo contrario dará error).

Lo primero que notamos es que, al introducir la url del archivo que debemos descargar para tener los datos, el botón *Browse* cambia automáticamente a *Update*. Hacemos clic en *Update*, e inmediatamente se actualizan las previsualizaciones tanto del archivo como del código.

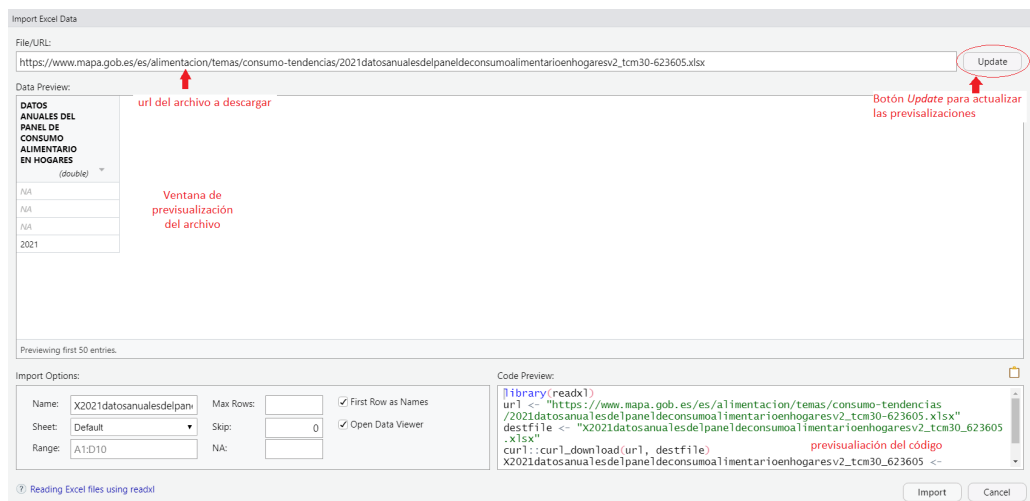


Figura 10: Cuadro de diálogo para importar datos desde la web.

En la ventana de opciones, podemos cambiar el nombre del archivo de destino, elegir la hoja "PRECIO", del menú desplegable *Sheet* e indicar que se ignoren las dos primeras filas especificando el argumento *Skip* en 2.

MIEL	Default	701390	7.11731360
BRANEL	PORTADA	700035	7.73278584
INVASADA	VALOR	738516	6.51616174
TOTAL CARNE	VOLUMEN	879428	7.23649429
CARNE CERTI	PENETRACIÓN	387085	9.00979929
	PRECIO		
	CONSUMOXCÁPITA		
	GASTOXCÁPITA		
	CANALES VALOR		
	CANALES VOLUMEN		
	CANALES PENETRACIÓN		
	SOCIOECONÓMICO VALOR		
	SOCIOECONÓMICO VOLUMEN		
	SOCIOECONÓMICO PENETRACIÓN		
	SOCIOECONÓMICO CONSUMOXCÁPITA		

Max Rows:

Sheet: **PRECIO**

Skip: **2**

Range: A1:D10

NA:

Figura 11: Opciones para *Sheet* y *Skip*.

Hecho esto, haciendo clic en *Import*, se generará el código que dará como resultado la importación y lectura de los datos en nuestra sesión de trabajo. En efecto, veremos que se imprime en la consola el código:

```
library(readxl)
url <- "https://www.mapa.gob.es/es/alimentacion/temas/consumo-
tendencias/2021datos anuales del panel de consumo alimentario en hogares v2_tc
m30-623605.xlsx"
destfile <- "datos_consumo_precio.xlsx"
curl::curl_download(url, destfile)
datos_consumo_precio <- read_excel(destfile, sheet = "PRECIO", skip =
2)
```

Como podemos observar, el código que se genera es esencialmente el mismo que habíamos utilizado antes. La principal diferencia que encontramos es el uso de la función `curl_download()` del paquete `{curl}` en lugar de la función `download.file()` de R base.

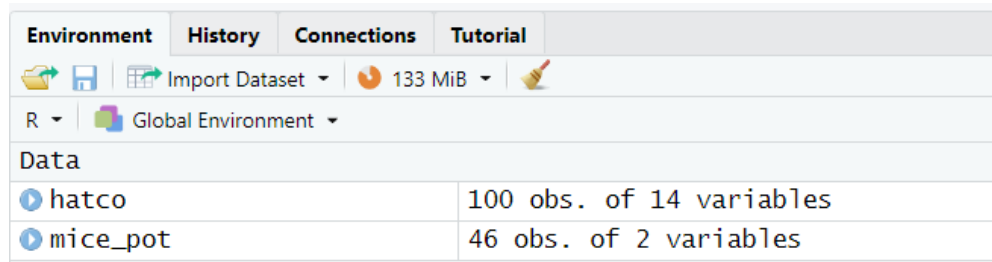
5.2.5. Leer un conjunto de datos propio de R

Otro tipo de conjunto de datos que podemos encontrar son los datos generados con el mismo R. Estos son conjuntos de datos creados en R y guardados con extensión `.RData` o `.rda`. R guarda los datos en formato binario. Este formato ahorra espacio de almacenamiento, pero restringe el uso exclusivamente al entorno R.

Como ejemplo, vamos a considerar los datos contenidos en los ficheros `hatco.RData` y `mice_pot.rda`. Para cargarlos a la sesión de trabajo, bastará con ejecutar las instrucciones:

```
# datos de R
load("./data/hatco.RData")
load("./data/mice_pot.rda")
```

Podemos comprobar que, al ejecutar estas líneas, se crean dos objetos de datos en la sesión actual, que podemos ver en la pestaña *environment*:



Environment	History	Connections	Tutorial
<div> <div>Import Dataset</div> <div>133 MiB</div> </div>			
R Global Environment			
Data			
hatco	100 obs. of 14 variables		
mice_pot	46 obs. of 2 variables		

Figura 12: Los objetos que se crean al ejecutar el load de los archivos `.rda` o `.RData`

Veamos que, efectivamente, los objetos creados son *data frames*:

```
class(hatco)
## [1] "data.frame"
class(mice_pot)
## [1] "data.frame"
```

y por lo tanto podemos comenzar a trabajar con ellos normalmente.

5.2.6. Importar datos de programas estadísticos comerciales (SPSS, SAS y STATA)

En ciertas disciplinas, el uso de determinados programas estadísticos comerciales es sumamente común. Por ejemplo, en Psicología el programa SPSS Statistics de IBM es el software estadístico comercial más utilizado. Archivos de datos generados por este software tendrán la extensión `.sav`, el tipo de archivo nativo de SPSS Statistics. Por otra parte, archivos generados con el software STATA, el programa comercial más utilizado en el campo de la economía, tendrán la extensión `.dta`

Por lo tanto, resulta conveniente ser capaces de importar y exportar datos almacenados en archivos compatibles con programas estadísticos comerciales, pues

esto nos permitirá usar datos ya existentes compatibles con ellos y colaborar con otras personas.

Para este fin, usamos el paquete `{haven}`, que cuenta con funciones para abrir bases de datos desde diferentes formatos. Este paquete forma parte del universo *tidyverse*, por lo cual, si ya has instalado el *tidyverse*, no hará falta que lo instales. De acuerdo a su documentación, `{haven}` actualmente soporta:

- ▶ SAS: `read_sas()` lee archivos `.sas7bdat` y `.sas7bcata`; mientras que `read_xpt()` lee el formato de archivo de transporte SAS (versión 5 y versión 8). La función `write_sas()` escribe archivos `sas7bdat` y `sas7bcata`, aunque es actualmente experimental y solo funciona para conjuntos de datos limitados.
- ▶ SPSS: `read_sav()` lee archivos `.sav` y `.zsav`; `read_por()` es para leer archivos `.por`. La función `read_spss()` utiliza las dos anteriores: `read_por()` y `read_sav()` de acuerdo a la extensión del archivo. `write_sav()` escribe archivos `.sav`.
- ▶ Stata: `read_dta()` y `read_stata()` leen archivos `.dta` (hasta la versión 15). `write_dta()` escribe archivos `.dta` (versiones 8-15).

Ejemplos

```
library(haven)
# SPSS
Ex_spss <- read_sav("../data/mtcars.sav")
Ex_spss
# A tibble: 32 × 11
  mpg   cyl  disp    hp  drat    wt  qsec    vs  am  gear  carb
  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1  21     6  160   110  3.9   2.62  16.5     0    1    4     4
2  21     6  160   110  3.9   2.88  17.0     0    1    4     4
3  22.8    4  108    93  3.85  2.32  18.6     1    1    4     1
4  21.4    6  258   110  3.08  3.22  19.4     1    0    3     1
5  18.7    8  360   175  3.15  3.44  17.0     0    0    3     2
6  18.1    6  225   105  2.76  3.46  20.2     1    0    3     1
7  14.3    8  360   245  3.21  3.57  15.8     0    0    3     4
8  24.4    4  147    62  3.69  3.19  20.0     1    0    4     2
9  22.8    4  141    95  3.92  3.15  22.9     1    0    4     2
10 19.2    6  168   123  3.92  3.44  18.3     1    0    4     4
# ... with 22 more rows
# i Use `print(n = ...)` to see more rows
# SAS
Ex_sas <- read_sas("../data/iris.sas7bdat")
iris_sas
# A tibble: 150 × 5
```

```

Sepal_Length Sepal_Width Petal_Length Petal_Width Species
      <dbl>      <dbl>      <dbl>      <dbl> <chr>
1         5.1         3.5         1.4         0.2 setosa
2         4.9         3         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         3.6         1.4         0.2 setosa
6         5.4         3.9         1.7         0.4 setosa
7         4.6         3.4         1.4         0.3 setosa
8          5         3.4         1.5         0.2 setosa
9         4.4         2.9         1.4         0.2 setosa
10        4.9         3.1         1.5         0.1 setosa
# ... with 140 more rows
# i Use `print(n = ...)` to see more rows

# Stata
Ex_stata <- read_dta("data/Milk_Production.dta")
Ex_stata
# A tibble: 199 × 7
  currentm previous   fat protein  days lactatio   i79
    <dbl>    <dbl> <dbl>   <dbl> <dbl>    <dbl> <dbl>
1      45      45  5.5    8.90   21        5     0
2      86      86  4.40   4.10   25        4     0
3      50      50  6.5     4     25        7     0
4      42      42  7.40   4.10   25        2     0
5      61      61  3.80   3.80   33        2     0
6      93      93  4.20    3     45        3     0
7      91      91  2.90   2.60   46        2     0
8      90      90  4.70   2.90   46        5     0
9      53      53  2.5     3.5   46        2     0
10     84      84  4.30   3.30   50        7     0
# ... with 189 more rows
# i Use `print(n = ...)` to see more rows

```

En el siguiente video, “*Importar datos*”, se presenta el paquete {readr} para importar datos en R:



Accede al vídeo

5.3. Leer datos desde un paquete

Aunque ya lo hemos utilizado para aportar ejemplos en temas anteriores, repasemos como hacemos para leer (cargar en la sesión de trabajo) datos que están contenidos en un paquete. Numerosos paquetes de R tienen conjuntos de datos integrados que pueden utilizarse para ejercitar el uso de las funciones del paquete.

5.3.1. Datasets de base

La instalación predeterminada de R viene con un paquete llamado `{datasets}` que se carga automáticamente cuando se inicia R. Podemos ver todos los conjuntos de datos disponibles en el paquete `{datasets}` ejecutando la instrucción

```
data(package = 'datasets')
```

Dicha instrucción abrirá una ventana donde se mostrarán todos los datos contenidos en el paquete (ver Figura 13).

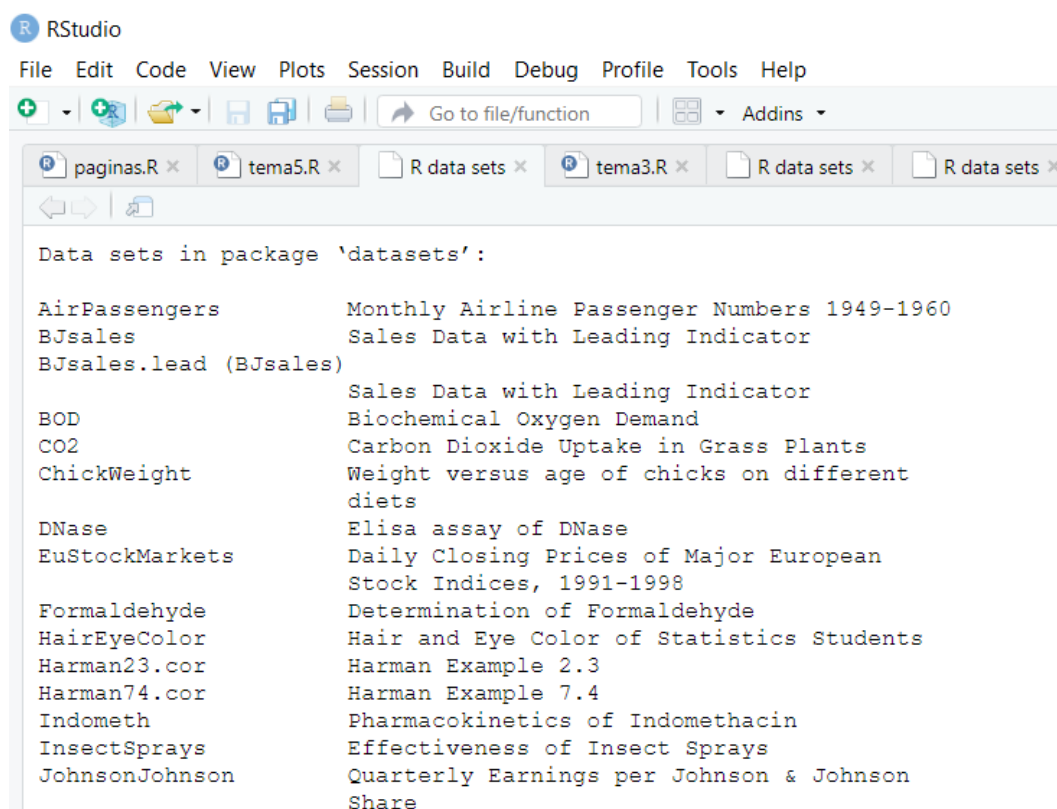


Figura 13: Ventana emergente de la instrucción `data(package = 'datasets')`

En realidad la función `data()` ejecutada sin argumentos nos mostrará todos los conjuntos de datos contenidos en todos los paquetes que están cargados en la sesión actual de trabajo.

Para utilizar un conjunto de datos particular, bastará con llamarlo con su nombre (siempre y cuando ya hayamos cargado el paquete que contiene los datos en la sesión actual).

Por ejemplo, el *dataset* `InsectSpray`, que contiene datos de conteos de insectos sobrevivientes en unidades experimentales agrícolas tratadas con diferentes insecticidas (para estudiar la efectividad del insecticida).

```
dim(InsectSprays)
## [1] 72 2
str(InsectSprays)
## 'data.frame': 72 obs. of 2 variables:
## $ count: num 10 7 20 14 14 12 10 23 17 20 ...
## $ spray: Factor w/ 6 levels "A","B","C","D",...: 1 1 1 1 1 1 1 1 1 1
## 1 ...
class(InsectSprays)
## [1] "data.frame"
with(InsectSprays, tapply(count, spray, sum))
## A B C D E F
## 174 184 25 59 42 200
```

5.4. Guardar datos desde R

Como hemos visto, R nos permite trabajar con datos y almacenarlos en variables en el espacio de trabajo (sesión actual). Sin embargo, a veces necesitaremos **exportar** o **guardar** los datos y/o resultados para compartirlos o trabajar con ellos en otro software. En este apartado veremos cómo exportar datos en R o RStudio. Debemos tener en cuenta que podemos exportar datos de R a varios formatos, como csv, sav, xls, xlsx, txt o incluso xml.

5.4.1. Guardar objetos de R

Todos los objetos almacenados en tu espacio de trabajo pueden guardarse. Hay tres opciones principales, dependiendo de si quieres guardar el espacio de trabajo completo, algunos objetos o solo uno.

Guardar todo el espacio o entorno de trabajo

Para esto utilizamos la función `save.image()`. El archivo resultante será de tipo RData.

```
# Exportar todos los objetos (la imagen del entorno de trabajo)
save.image(file = "Mi_espacio_en_R.RData")
```

Al ejecutar la instrucción se creará el objeto `Mi_espacio_en_R.RData` en el directorio de trabajo (ver Figura 14). Haciendo clic sobre el archivo R te preguntará si deseas cargar el contenido del archivo en el entorno de trabajo actual (ver Figura 15) y si clicamos en Yes, se cargará al entorno actual y así nos evitamos tener que volver a ejecutar el script para tener los objetos en memoria. Como el objeto que se crea al guardar el espacio de trabajo es un archivo RData, también podemos cargar el espacio utilizando la función `load()`, como vimos anteriormente (sección 5.2.5. Leer un conjunto de datos propio de R).

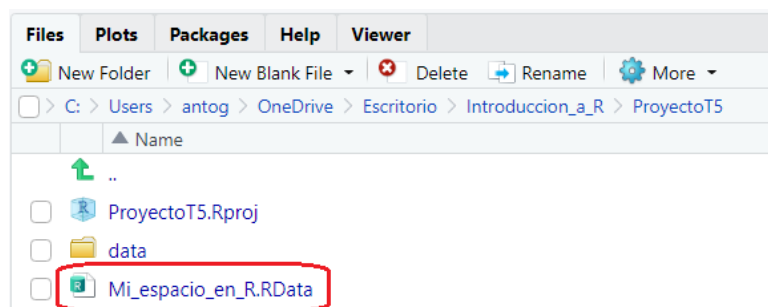


Figura 14: Guardando el espacio de trabajo.

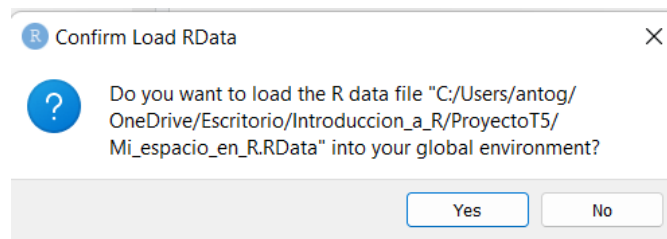


Figura 15: Cuadro de diálogo para confirmar o no la carga del entorno de trabajo guardado.

Si solo necesitas exportar algunos objetos, puedes especificarlos separados por comas con la función `save`. Como ejemplo, vamos a crear dos objetos para luego guardarlos:

```
# Exportar algunos objetos de R
# primero creamos los objetos
tabla <- with(InsectSprays, tapply(count, spray, sum))
vector <- rnorm(200)
```


Esto crea los objetos en el entorno de trabajo:


Environment


History

Connections


Tutorial









Import Dataset



171 MiB



R



Global Environment

Data

hatco

100 obs. of 14 variables

mice_pot

46 obs. of 2 variables

Values

tabla

num [1:6(1d)] 174 184 25 59 42 200

vector

num [1:200] 0.801 -1.18 -2.221 0.925 0.19 ...

Figura 16: Objetos en el entorno de trabajo que podemos guardar

```
# y ahora los guardamos
save(tabla, vector, file = "Mis_objetos.RData")
```

Finalmente, para guardar solo un objeto, se recomienda guardarlo como archivo con extensión `rds` utilizando la función `saveRDS`:

```
# Exportar solo un objeto de R
saveRDS(tabla, file = "Unico_objeto.rds")
```

Este es un tipo de archivo nativo de R que puede almacenar cualquier objeto a un archivo en nuestro disco duro.

Además, RDS comprime los datos que almacena, por lo que ocupa menos espacio en disco duro que otros tipos de archivos, aunque contengan la misma información.

5.4.2. *Data frames* y matrices

Si nuestros datos se encuentran contenidos en una estructura de datos rectangular, lo usual es exportarlos como archivos de texto plano TXT o como un archivo CSV. Esto podemos hacerlo con distintas funciones.

Exportar datos desde R a un archivo TXT

La función `write.table()` nos permite exportar matrices o *data frames*, como archivos de texto con distintas extensiones. Los argumentos más usados de esta función son:

- ▶ `x`: el nombre del *data frame* o matriz a exportar.
- ▶ `file`: el nombre, extensión y ruta del archivo que se crea. Si no especificamos la ruta, el archivo será creado en el directorio de trabajo actual.
- ▶ `sep`: el caracter que se usará como separador de columnas.
- ▶ `row.names`: un valor lógico que indica si los nombres de las filas de `x` se escribirán junto con `x`, o un vector de caracteres con los nombres de las filas que se escribirán. En general, es recomendable fijarlo como `FALSE`, para conservar una estructura tabular más fácil de leer.
- ▶ `col.names`: Si deseamos que el archivo incluya los nombres de las columnas en nuestro objeto, establecemos este argumento como `TRUE`. Es recomendable fijarlo como `TRUE` para evitar la necesidad de almacenar los nombres de columna en documentos distintos.

Como siempre, puedes consultar todos los argumentos de esta función ejecutando `?write.table`.

Ejemplo: exportando objeto a archivo de texto plano
Retomemos el *data frame* `df` que creamos en el Tema 3, cuando estudiamos la estructura de datos *data frames*.

Por simplicidad de lectura, vamos a repetir el código de creación de df:

```
# creamos los vectores de datos
altura <- c(167, 192, 173, 174, 172, 167, 171, 185, 163, 170)
peso <- c(86, 74, 83, 50, 78, 66, 66, 51, 50, 55)
fuma <- c("No", "Si", "No", "Si", "No", "No", "Si", "Si", "Si", "No")
sexo <- c("M", "M", "F", "M", "M", "M", "F", "M", "F", "F")
# creamos el df
df <- data.frame(altura, peso, habito = fuma, sexo)
```

Vamos a probar a exportar el data frame df a un documento de texto llamado df.txt a nuestro directorio de trabajo, usando como separador tabuladores ("\t"), con nombres de columnas y sin nombre de renglones:

```
write.table(
  x = df,
  file = "df.txt",
  sep = "\t",
  row.names = FALSE,
  col.names = TRUE
)
```

Al ejecutar la instrucción se creará el objeto df.txt en nuestro directorio de trabajo. Importemos el archivo que hemos creado usando read.table()

```
df_from_txt <- read.table(
  file = "df.txt",
  header = TRUE,
  sep = "\t"
)
# Resultado
head(df_from_txt, 3)
##   altura peso habito sexo
## 1   167   86    No    M
## 2   192   74    Si    M
## 3   173   83    No    F
```

Exportar datos desde R a un archivo CSV

Para exportar un *data frame* o matriz como CSV en R, podemos utilizar las funciones write.csv o write.csv2. Al igual que las funciones read.csv() y read.csv2(), el uso de una u otra dependerá del formato de los datos: write.csv() genera un CSV separado por comas, mientras que write.csv2() crea un archivo CSV utilizando el punto y coma como separador de los datos.

Ejemplo: exportando objeto a CSV

Exportemos el *data frame* df a un archivo CSV df.csv a nuestro directorio de trabajo:

```
# Coma como separador y punto como separador decimal
write.csv(
  x = df,
  file = "df.csv",
  row.names = FALSE
)
# Resultado
df_from_csv <- read.csv(
  file = "df.csv"
)
head(df_from_csv, 3)
##   altura peso habito sexo
## 1    167   86    No    M
## 2    192   74    Si    M
## 3    173   83    No    F

# Punto y coma como separador y coma como separador decimal
write.csv2(
  x = df,
  file = "df_2.csv",
  row.names = FALSE
)
# Resultado
df_from_csv2 <- read.csv2(
  file = "df_2.csv"
)
head(df_from_csv2, 3)
##   altura peso habito sexo
## 1    167   86    No    M
## 2    192   74    Si    M
## 3    173   83    No    F
```

5.4.3. Listas

La manera más sencilla de exportar listas es guardarlas en archivos RDS.

Como ya vimos, para exportar un objeto a un archivo RDS, usamos la función `saveRDS()`.

Ejemplo: exportando listas.

Vamos a utilizar la lista `lista1` que creamos como ejemplo en el Tema 3, cuando estudiamos esta estructura de datos.

```
# Primero creamos los elementos que conformarán nuestra lista
v <- runif(10,2,10)
m <- matrix(1:4, ncol = 2)
df <- data.frame("Nombres" = c('Lucia', 'Bianca', 'Fernando', 'Carlos'),
  "edad" = c(19,20,34,56))
# creamos la lista
lista1 <- list(v,m,df)
```

Podemos intentar exportar la lista anterior como un archivo .txt. utilizando `write.table()`, pero por lo general obtendremos un error como resultado.

```
write.table(x = lista1, file = "mi_lista.txt")
Error in (function (..., row.names = NULL, check.rows = FALSE,
check.names = TRUE, :
  arguments imply differing number of rows: 10, 2, 4
```

Esencialmente, trata de forzar `lista1` a *data frame* y falla porque elementos de `lista1` no son vectores e igual longitud.

Lo mismo ocurre si intentamos exportar `lista1` como archivo CSV.

Usamos la función `saveRDS()` para exportar al archivo `mi_lista.rds`.

```
saveRDS(object = lista1, file = "mi_lista.rds")
```

Si deseamos importar un archivo RDS a R, podemos utilizar la función `readRDS()`, indicando la ruta en la que se encuentra el archivo que deseamos. Intentemos importar el archivo `mi_lista.rds`.

```
mi_lista_imp <- readRDS(file = "mi_lista.rds")
mi_lista_imp
## [[1]]
## [1] 6.045428 9.628226 6.490316 7.819402 5.589335 7.836142 9.038665
## [8] 6.201006 3.781061 6.050708
##
## [[2]]
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## [[3]]
##      Nombres edad
## 1      Lucia   19
## 2     Bianca   20
## 3 Fernando   34
## 4     Carlos   56
```

También podemos importar el archivo de forma interactiva, haciendo clic sobre su nombre en la pestaña *Files* de RStudio. Esto abrirá un cuadro de dialogo (ver Figura 17) donde podemos introducir el nombre con el que deseamos importar el archivo y clicando en OK se cargará el archivo al entorno de trabajo.

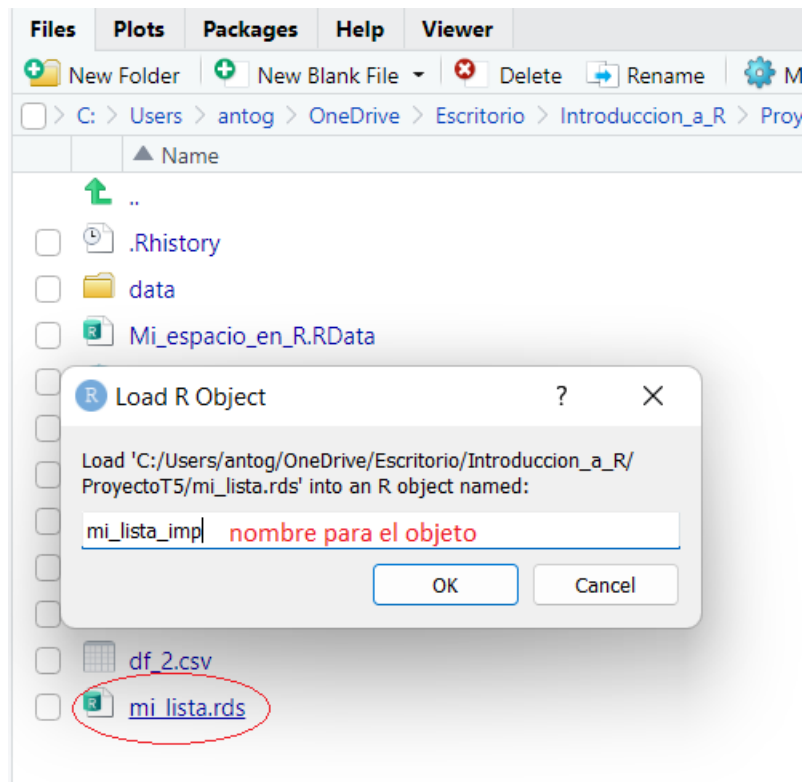


Figura 17: Importar archivo RDS de forma interactiva

5.5. Manipulación de datos: una introducción al *tidyverse*

Una vez que importamos datos a R es conveniente *ordenarlos*, esto implica darles la forma necesaria o aplicarles las transformaciones que nos permitan enfocarnos en responder preguntas con los datos en lugar de estar luchando con ellos. La manipulación de datos involucra distintas operaciones que podemos realizar con los datos y puede significar distintas cosas en diferentes momentos de un análisis. A veces queremos seleccionar ciertas observaciones (filas) o variables (columnas), otras veces deseamos agrupar los datos en función de una o más variables, o queremos calcular valores estadísticos de un conjunto.

La transformación de datos será mucho más fluida si trabajamos con conjunto de datos ordenado. Transformar los datos no significa en modo alguno alterarlos,

significa estrictamente cambiarles la forma: modificar su estructura, pero no la información que contienen.

Los principios de datos ordenados (*Tidy Data*, Wickham, H. (2014)) proveen una manera estándar de organizar la información:

- ▶ Cada variable forma una columna
- ▶ Cada observación forma un renglón.
- ▶ Cada tipo de unidad observacional forma una tabla.

país	año	casos	poblacion
Afganistán	1999	745	19987071
Afganistán	2000	2666	20595360
Brasil	1999	37737	17206362
Brasil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	213766	1280428583

variables

país	año	casos	poblacion
Afganistán	1999	745	19987071
Afganistán	2000	2666	20595360
Brasil	1999	37737	17206362
Brasil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	213766	1280428583

observaciones

país	año	casos	poblacion
Afganistán	1999	745	19987071
Afganistán	2000	2666	20595360
Brasil	1999	37737	17206362
Brasil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	213766	1280428583

valores

Figura 18: Representación visual de las reglas que hacen que un conjunto de datos sea tidy. Fuente: Wickham, H. and Grolemund, G. (2019)

Como ejemplo, los datos en el archivo `AirPassengers` del paquete `datasets` de R, no son *tidy*. Si visualizamos los datos veremos que se trata de una estructura rectangular de datos, pero no verifica las tres reglas de datos ordenados.

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
1949	112	118	132	129	121	135	148	148	136	119	104	118
1950	115	126	141	135	125	149	170	170	158	133	114	140
1951	145	150	178	163	172	178	199	199	184	162	146	166
1952	171	180	193	181	183	218	230	242	209	191	172	194
1953	196	196	236	235	229	243	264	272	237	211	180	201
1954	204	188	235	227	234	264	302	293	259	229	203	229
1955	242	233	267	269	270	315	364	347	312	274	237	278
1956	284	277	317	313	318	374	413	405	355	306	271	306
1957	315	301	356	348	355	422	465	467	404	347	305	336
1958	340	318	362	348	363	435	491	505	404	359	310	337
1959	360	342	406	396	420	472	548	559	463	407	362	405
1960	417	391	419	461	472	535	622	606	508	461	390	432

Figura 19: Visualización en consola del *dataset* `AirPassengers`

Para ser *tidy* tendríamos que cambiarle la forma para tener tres columnas (año, mes y cantidad de pasajeros), y entonces cada observación de `AirPassengers` tendría una fila.

Por el contrario, los datos `ChickWeight` si son *tidy*: cada observación (un peso) está representada por una fila. El pollito de donde provino esta medida es una de las variables.

En general, la forma en que representaríamos una base de datos ordenados en R es usando un *data frame* o un *tibble*.

5.5.1. ¿Qué es esto del tidyverse?

Con la palabra *tidyverse* se hace referencia a una nueva forma de afrontar el análisis de datos en R. Se hace uso de un grupo de paquetes que trabajan en armonía porque comparten ciertos principios como, por ejemplo, la forma de estructurar los datos.

La mayoría de estos paquetes han sido desarrollados por (o al menos con la colaboración de) [Hadley Wickham](#). Esta es la [página web del tidyverse](#). Y este es el [The tidy tools manifesto](#).

5.5.2. El operador pipe `%>%`

Este operador es básico en el *tidyverse*, ya que permite encadenar llamadas a funciones para así realizar de forma sencilla transformaciones de datos complejas. El operador pipe `%>%`, al que leemos como “después” o “luego”, se lo debemos a [Stefan Bache](#) en su paquete `{magrittr}`.

En palabras, lo que hace este operador es pasar el elemento que está a su izquierda como un argumento de la función que tiene a la derecha. Como ya lo habrán notado, es lo mismo que explicamos en el Tema 3 con el operador pipe `|>` de R base.

Un ejemplo sencillo:

```
# Sin utilizar un operador pipe
x <- 1:10
logx <- log(x)
meanlogx <- mean(logx)
sqrt(meanlogx) # resultado buscado
## [1] 1.229

# usando el operador pipe %>%
x <- 1:10 %>%
  log() %>%
  mean() %>%
  sqrt()
x
## [1] 1.229
```

Este código podría leerse de la siguiente manera:

primero crear un vector de números consecutivos desde 1 hasta 10, **luego** aplicar el logaritmo natural, **luego** calcular el promedio, **luego** sacar la raíz cuadrada

5.5.3. Manipulación de datos con {dplyr}

En temas anteriores hemos visto como realizar algunas tareas de manipulación de datos utilizando funciones de la librería base de R. Por ejemplo [,] para extraer subconjuntos, subset() para operaciones agrupadas, cbind() o rbind() para combinar múltiples fuentes de datos. A pesar de ser funciones sumamente flexibles y aplicables a múltiples estructuras de datos no son las más convenientes. Están pensadas para usarlas bajo un paradigma procedimental, más que con uno funcional, además generan un código difícil de leer y nos obligan a repetir los nombres de las estructuras de datos en cada llamada.

El paquete {dplyr} es definido por sus autores como una gramática para la manipulación de datos. De ahí que sus funciones son conocidas como *verbos*, considerando que el paquete incluye un conjunto de comandos que coinciden con las acciones más comunes que se realizan sobre un conjunto de datos (seleccionar filas filter(), seleccionar columnas select(), ordenar arrange(), añadir nuevas

variables `mutate()`, resumir mediante alguna medida numérica `summarise()`). Un resumen útil de la mayoría de estas funciones puedes encontrarlo [aquí](#).

Abarcar la totalidad de funciones que contiene el paquete resulta muy ambicioso en una introducción, así que nos concentraremos en las funciones más utilizadas:

- ▶ `group_by` (agrupa datos)
- ▶ `summarize` (resume datos agrupados)
- ▶ `mutate` (genera variables nuevas)
- ▶ `filter` (encuentra filas con ciertas condiciones)
- ▶ `select` junto a `where`, `starts_with`, `ends_with` o `contains`
- ▶ `inner_join`, `left_join`, `right_join`, `full_join` (unir conjuntos de datos con variables comunes)

Seleccionar un subconjunto de columnas

Para seleccionar un subconjunto de columnas de un *data frame* utilizamos la función `select()` y derivadas. Esto es especialmente cierto en el caso de conjuntos de datos con un número grande de variables, ya que nos ofrece algunas formas de “encontrar” las variables deseadas sin necesidad de especificarlas una a una. Veamos algunos ejemplos:

Ejemplo: seleccionar columnas.

Para los ejemplos que siguen utilizaremos los datos `imcenfant.txt`, provistos como material de este apunte.

```
library(tidyverse)
# Cargamos los datos
imcenfant <- read_table("data/imcenfant.txt", skip = 13)

# Seleccionamos las columnas SEXE y poids
imcenfant %>% select(SEXE, poids)

# Si ahora queremos todas las columnas excepto SEXE y poids
imcenfant %>% select(-c(SEXE, poids))
# o, equivalentemente
imcenfant %>% select(-SEXE, -poids)

# where aplica una función a todas las variables y selecciona
# aquellas para las que la función devuelve TRUE
imcenfant %>% select(where(is.numeric))
```

```

# select_if: este verbo permite usar un predicado en las columnas
# de un data frame.
# Solo se seleccionarán aquellas columnas para las que se devuelve
# el predicado TRUE.
imcenfant %>% select_if(is.numeric)

# Estos ayudantes seleccionan variables haciendo coincidir patrones
# en sus nombres:
# seleccionar todas las columnas cuyos nombres comiencen
# con determinado prefijo
imcenfant %>% select(starts_with("moi"))
# seleccionar todas las columnas cuyos nombres terminen con determinado
# sufijo
imcenfant %>% select(ends_with("lle"))
# seleccionar todas las columnas cuyos nombres contengan con
# determinada cadena
imcenfant %>% select(contains("oi"))

```

Seleccionar un subconjunto de filas

La forma más sencilla es mediante el uso de la función `filter()`. Debemos especificar como argumento una condición lógica (o más genéricamente un vector de valores TRUE o FALSE para cada una de las filas del *data frame*). Algunos ejemplos:

```

# Seleccionar a las niñas (SEXE == 'F')
imcenfant %>% filter(SEXE == "F")
# Seleccionar a las niñas (SEXE == 'F') que pesen más de 18 Kg
imcenfant %>% filter(SEXE == "F", poids > 18)

```

Modificar un subconjunto de columnas

Una forma de modificar los valores de algunas columnas (por ejemplo, escalar los valores) es mediante la función `mutate()`. Puede asociarse con la función `across()` del mismo paquete `{dplyr}`, para seleccionar en forma fácil de leer las variables a modificar. Algunos ejemplos:

```

# Pasemos la altura de cm a m
imcenfant %>% mutate(taille = taille/100)

```

La función `across()` facilita la aplicación de la misma transformación a varias columnas. Pueden ser funciones definidas por el usuario (incluso anónimas) o funciones incorporadas de R

```

# Estandarizamos las variables

```

```

imcenfant %>%
  mutate(across(where(is.numeric), function(x){(x-mean(x))/sd(x)}))
# o, equivalentemente
imcenfant %>%
  mutate(across(where(is.numeric), scale))

# estandarizamos las columnas que cumplan cierta condición
imcenfant %>%
  mutate(across(contains("oi"), scale))

# Podemos recombrar los niveles de un factor
imcenfant %>%
  mutate(SEX = ifelse(SEXE=="F", "FEMALE", "MALE"))

```

En el último ejemplo, en lugar de reemplazar los valores de una variable existente hemos creado una nueva variable. En este caso, `SEX` y `SEXE` tienen información redundante, ya que sólo hemos cambiado la codificación. No obstante, podemos usar `mutate()` para crear nuevas variables derivadas de las originales.

```

# creamos la variable imc que es el cociente entre peso y altura^2
# (media en m^2)
imcenfant %>%
  mutate(taille = taille/100) %>%
  mutate(imc = poids/taille^2)

```

Notar que, en este ejemplo, primero utilizamos `mutate()` para convertir la altura a metros y luego lo volvimos a usar para crear a variable `imc`.

Unir conjuntos de datos con variables comunes

En muchas ocasiones nos encontraremos con información sobre los mismos individuos registrada en archivos de datos separados y será de interés integrarla en un mismo *data frame* para proceder a su análisis. Podemos simular esta situación con el conjunto de datos que estábamos usando. Para ello, primero crearemos dos archivos de datos, agregaremos un identificador para cada individuo y luego los integraremos para recuperar un archivo idéntico al original:

```

# creamos los dos data frames con información sobre los niños
datosA = imcenfant %>%
  select(SEXE, zep) %>%
  rownames_to_column("ID")

datosA

```

```
## # A tibble: 152 × 3
##   ID     SEXE  zep
##   <chr> <chr> <chr>
## 1 1      F    0
## 2 2      F    0
## 3 3      G    0
## 4 4      F    0
## 5 5      G    N
## 6 6      G    0
## 7 7      G    N
## 8 8      G    0
## 9 9      G    0
## 10 10     G    0
## # ... with 142 more rows
## # i Use `print(n = ...)` to see more rows

datosB = imcenfant %>%
  select(where(is.numeric)) %>%
  rownames_to_column("ID")

datosB
## # A tibble: 152 × 5
##   ID     poids  an  mois  taille
##   <chr> <dbl> <dbl> <dbl> <dbl>
## 1 1      16      3    5    100
## 2 2      14      3   10    97
## 3 3     13.5     3    5   95.5
## 4 4     15.4     4    0   101
## 5 5     16.5     3    8   100
## 6 6      16     4    0   98.5
## 7 7      17     3   11   103
## 8 8     14.8     3    9    98
## 9 9      17     4    1  102.
## 10 10     16.7    3    3   100
## # ... with 142 more rows
## # i Use `print(n = ...)` to see more rows
```

Como podemos observar, hemos desagregado el *data frame* original en dos: las variables cualitativas forman *datosA*, mientras que las cuantitativas forman *datosB*.

Ahora vamos a unir estos dos *datasets* en uno solo que contenga toda la información disponible de cada niño.

```
datos <- datosA %>%
  left_join(datosB, by = "ID")
datos
## # A tibble: 152 × 7
##   ID     SEXE  zep  poids  an  mois  taille
##   <chr> <chr> <chr> <dbl> <dbl> <dbl> <dbl>
## 1 1      F    0    16      3    5    100
## 2 2      F    0    14      3   10    97
```

```
## 3 3 G O 13.5 3 5 95.5
## 4 4 F O 15.4 4 0 101
## 5 5 G N 16.5 3 8 100
## 6 6 G O 16 4 0 98.5
## 7 7 G N 17 3 11 103
## 8 8 G O 14.8 3 9 98
## 9 9 G O 17 4 1 102.
## 10 10 G O 16.7 3 3 100
## # ... with 142 more rows
## # Use `print(n = ...)` to see more rows
```

Las diferentes funciones que encontramos dentro de este tipo de tareas son (supongamos que las aplicamos a `datos1` y `datos2`):

`inner_join(datos1, datos2)`: incluye todas las filas en común (es decir, que corresponden a los mismos individuos u observaciones) en `datos1` y `datos2`.

`left_join(datos1, datos2)`: incluye todas las filas de `datos1` (si hay filas en `datos2` que no están en `datos1`, se pierden).

`right_join(datos1, datos2)`: incluye todas las filas de `datos2`, (si hay filas en `datos1` que no están en `datos2`, se pierden).

`full_join()`: incluye todas las filas en `datos1` o `datos2`.

En [este enlace](#) encontrarás una serie de animaciones que te ayudarán a entender mejor el funcionamiento de estas funciones.

Obtener medidas resúmenes

La función (o verbo) `summarize()` (o `summarise()` si prefieres la gramática americana) sirve para **RESUMIR** (o “colapsar filas”). Coge un grupo de valores como input y devuelve un solo valor; por ejemplo, encuentra la media aritmética (o el mínimo, o el máximo ...) de un grupo de valores. Veamos algunos ejemplos:

```
Calculemos la media y desvío estándar de la altura:
imcenfant %>%
  summarize(media = mean(taille), desvio = sd(taille))
## # A tibble: 1 × 2
##   media desvio
```



```
##      <dbl>  <dbl>
## 1   101.    4.23
```

También podemos combinar `summarize()` con `across()` para calcular la media y desvío estándar de todas las variables numéricas:

```
summarize(across(where(is.numeric), list(media = mean, desvio = sd)))
## # A tibble: 1 × 8
##   poids_media poids_desvio an_media an_desvio mois_media mois_desvio
##   <dbl>      <dbl>      <dbl>      <dbl>      <dbl>      <dbl>
## 1   16.3      1.93      3.30      0.461      5.62      3.55
## 101.    4.23
## # ... with abbreviated variable name 1taille_desvio
```

Resúmenes por grupos

Una funcionalidad muy útil viene de agrupar los datos en función de alguna variable categórica. Esto se consigue con la función `group_by()`. A menudo nos interesará obtener estadísticas que resuman estos grupos. Como vimos, esto se consigue con la función `summarize()`, usualmente asociada a `across()`. Por ejemplo, siguiendo con los datos que veníamos trabajando, imaginemos que queremos calcular la altura promedio según el sexo:

```
imcenfant %>%
  group_by(SEXE) %>%
  summarize(across(taille, mean))
## # A tibble: 2 × 2
##   SEXE  taille
##   <chr>  <dbl>
## 1 F      99.8
## 2 G     102.
```

O también, la media y desvío de las alturas agrupadas por sexo:

```
imcenfant %>%
  group_by(SEXE) %>%
  summarize(across(taille, list(media = mean, desvio = sd)))
## # A tibble: 2 × 3
##   SEXE  taille_media taille_desvio
##   <chr>      <dbl>      <dbl>
## 1 F      99.8      4.03
## 2 G     102.      4.26
```

Otros ejemplos:

Calculemos por ejemplo la media de las alturas agrupados por sexo y zep (escuela en zona prioritaria)

```
imcenfant %>%
  group_by(SEXE,zep) %>%
  summarize(across(taille, mean, .names = "mean_{.col}"))
## # A tibble: 4 × 3
## # Groups:   SEXE [2]
##   SEXE zep mean_taille
##   <chr> <chr>      <dbl>
## 1 F     N        101.
## 2 F     O         99.5
## 3 G     N        101.
## 4 G     O        102.
```

Calculemos ahora la media de todas las variables numéricas pero agrupadas por sexo:

```
imcenfant %>%
  group_by(SEXE) %>%
  summarize(across(where(is.numeric), mean, .names = "mean_{.col}"))
## # A tibble: 2 × 5
##   SEXE mean_poids mean_an mean_mois mean_taille
##   <chr>      <dbl>   <dbl>   <dbl>      <dbl>
## 1 F          16.0     3.31     5.72       99.8
## 2 G          16.6     3.30     5.53      102.
```

Si queremos calcular más de una estadística a todas las variables numéricas agrupadas por sexo:

```
imcenfant %>%
  group_by(SEXE) %>%
  summarize(
    across(
      where(is.numeric),
      list(media = mean, desvio = sd),
      .names = "{.col}_{.fn}"
    )
  )
## # A tibble: 2 × 9
##   SEXE poids_media poids_desvio an_media an_desvio mois_media
## mois_des...1 taill...2 taill...3
##   <chr>      <dbl>      <dbl>   <dbl>   <dbl>      <dbl>
## <dbl> <dbl> <dbl>
## 1 F          16.0        1.90     3.31     0.466     5.72
## 3.62    99.8     4.03
## 2 G          16.6        1.94     3.30     0.459     5.53
## 3.51   102.     4.26
```

Finalmente, un ejemplo en el que utilizamos una función anónima:

```
imcenfant %>%
```

```
group_by(SEXE,zep) %>%
  summarize(across(taille, function(x){mean(x > 102)}))
## # A tibble: 4 × 3
## # Groups:   SEXE [2]
##   SEXE zep   taille
##   <chr> <chr> <dbl>
## 1 F     N     0.389
## 2 F     O     0.226
## 3 G     N     0.435
## 4 G     O     0.448
```

5.6. Referencias bibliográficas

Wickham, H. (2014). Tidy Data. *Journal of Statistical Software*, 59(10), 1-23.

<https://doi.org/10.18637/jss.v059.i10>

Wickham, H. and Grolemund, G. (2019). R para Ciencia de Datos.

<https://es.r4ds.hadley.nz/>

5.7. Cuaderno de ejercicios

Ejercicio 1

¿Cuál de los siguientes sets de datos integrados en R base (es decir, contenidos en el paquete {datasets}) es *tidy*? Puede elegir más de uno.

C02

HairEyeColor

DNase

euro

Orange

UCBAdmissions

Ejercicio 2

Tres variables contenidas en la base de datos `storms` del paquete `{dplyr}` son `wind` (velocidad máxima sostenida del viento de la tormenta (en nudos)), `hurricane_force_diameter` (diámetro (en millas náuticas) del área que experimenta vientos huracanados (64 nudos o más)) y `status` (clasificación de la tormenta: depresión tropical - tropical depression -, tormenta tropical - tropical storm - o huracán - hurricane -).

Calcular la velocidad promedio y diámetro promedio de las tormentas que han sido declaradas huracanes para cada año.

Ejercicio 3

La base de datos `mpg` del paquete `{ggplot2}` tiene datos de eficiencia vehicular en millas por galón en ciudad (`cty`) en varios vehículos. Obtener los datos de vehículos del año 2004 en adelante que sean compactos y transformar la eficiencia a Km/litro (1 milla = 1.609 km; 1 galón = 3.78541 litros)

Ejercicio 4

El conjunto de datos `lobsters.txt` está formado por los pesos de langostas capturadas en dos zonas; estos pesos están expresados en kg, con una precisión de 0.01 kg. Las separaciones entre columnas son espacios en blanco y tiene una primera fila con los nombres de las columnas.

1. Importa los datos y asígnalos a un *data frame* que se llame `langostas`.
2. Calcula el peso medio y el desvío estándar de los pesos de las langostas.
3. Obtén el peso medio, desvío estándar y la cantidad de observaciones de los pesos de las langostas en cada zona.

4. Entre las langostas que pesan más de 500 g, calcula el peso medio, desvío estándar y la cantidad de observaciones de los pesos de las langostas en cada zona.

Ejercicio 5

Para este ejercicio, usaremos el conjunto de datos `airquality` (información meteorológica de ciertos meses (mayo a septiembre) del año 1973 en Nueva York, disponible en el paquete `{datasets}` de R base). A partir de los datos, responde las preguntas

1. ¿Cuál es la temperatura media de esos días?
2. ¿Cuál es la temperatura media en mayo?
3. ¿Cuál fue el día más ventoso?
4. ¿En cuántos días la velocidad del viento superó a la velocidad promedio de los días observados?
5. Ordena las filas del *dataset* según valores crecientes de temperatura y velocidad del viento (la función `arrange()` podría serte de utilidad)

En el siguiente vídeo podrás acceder a la resolución del ejercicio 4 del cuaderno de ejercicios:



Accede al vídeo