Importar, limpiar, unir

PID_00265509

Jordi Mas Elias

Tiempo mínimo de dedicación recomendado: 3 horas

Universitat Oberta de Catalunya

Jordi Mas Elias

El encargo y la creación de este recurso de aprendizaje UOC han sido coordinados por el profesor: Jordi Mas Elias (2019)

Primera edición: septiembre 2019

Avda. Tibidabo, 39-43, 08035 Barcelona
Realización editorial: FUOC

Índice

Introducción

No siempre un analista de datos encuentra la información a punto para ser explorada. El trabajo principal del analista es a menudo mucho más pesado de lo que parece, puesto que los datos pueden estar desordenados, con nombres mal escritos difíciles de interpretar o separados en diferentes archivos de internet. Por esta razón, en algunos ámbitos, los científicos de datos tienen que dedicar la mayor parte de su tiempo a estas tareas (Lohr, 2014). El objetivo de este módulo es ayudaros a convertir datos dispersos y desconectados entre sí en un solo marco de datos preparado para ser analizado.

En estudios internacionales, los datos provienen en la mayoría de ocasiones de las principales organizaciones y centros de estudios internacionales, de modo que ya han sido sistematizados previamente. Estas entidades se encargan de recoger, limpiar y ordenar los datos para presentarlos al público. Incluso algunos datos tienen disponible una librería específica en R para facilitarnos el tratamiento. Esto no resta importancia al hecho de aprender a adquirir y preparar los datos para poderlos estudiar a fondo.

En este módulo os enseñaremos tres pasos básicos para la preparación de los datos.

1) El primero consiste en importar los datos a RStudio. En este primer apartado os enseñaremos diferentes maneras de incorporar los datos en el programa.

2) Una vez incorporados, un segundo paso es limpiar los datos. Dedicaremos el segundo apartado del módulo a esta tarea, que consiste en modificar cuando sea necesario la estructura del marco de datos y el tipo de variables que tenemos y a tomar decisiones sobre los valores perdidos y los casos extremos.

3) Finalmente, el tercer paso consiste en unir diferentes marcos de datos. Este proceso es sumamente útil para nuestro trabajo, puesto que nos permite analizar información que proviene de fuentes diferentes. Por lo tanto, el proceso de unir bases de datos será un paso esencial previo al tratamiento.

1. Importar datos

Importar contenido es el primer paso que tenemos que hacer para trabajar con datos en RStudio. En internet tenemos multitud de bases de datos relacionadas con estudios internacionales, como World Bank, Eurostat, Correlates of War, etc.[1] En este apartado aprenderemos a importar a R algunas de ellas. Tenemos dos vías principales para importarlas: mediante un paquete de R o bien importando un archivo localizado en nuestro ordenador o en la red.

(1)Una de las listas más extensas de bases de datos en estudios internacionales que encontramos en la red la tenemos en este GitHub (https://github.com/erikgahner/PolData).

1.1. Importar un paquete

Dentro de los paquetes que ya tenemos actualmente cargados en R ya hay una gran cantidad de bases de datos que nos pueden servir para practicar con el programa. Uno de los paquetes de base de R denominado *datasets* contiene cerca de un centenar de marcos de datos que podéis consultar tecleando `help = "datasets"`. Otros paquetes que ya conocemos, como *dplyr* o *ggplot2*, incorporan también, entre muchos objetos y funciones, algunos marcos de datos en el interior. Hay otros paquetes que están alojados en la biblioteca central de R, el CRAN, como es el caso de *gapminder*, que son fáciles de utilizar porque cuando cargamos el paquete en R se nos cargan también los marcos de datos de su interior. Para instalar un paquete utilizaremos la función `install.packages()` con el nombre del paquete entre comillas, y para cargarlo en R utilizaremos `library()` con el nombre del paquete sin las comillas. Una vez cargado, podemos ver las funciones y los objetos del paquete si introducimos su nombre seguido de dos puntos (por ej., `gapminder::`). Nos aparecerá un desplegable con la información.

Marcos de datos en R

En los paquetes de base de R veréis que hay marcos de datos curiosos como el de `Titanic` (lista de pasajeros supervivientes en el Titánic), `discoveries` (años de descubrimientos importantes) o `sleep` (datos de un estudio que medía el efecto del consumo de drogas en las horas de sueño). También podéis instalar y descargar paquetes que tienen una gran cantidad de marcos de datos para practicar, como `openintro`.

Sin embargo, la mayoría de paquetes que tienen relación con los estudios internacionales son algo más sofisticados, puesto que incorporan varias funciones que tenemos que utilizar para poder trabajar con los marcos de datos. Por ejemplo, *psData* contiene varios marcos de datos de ciencia política, como *Polity IV* o *Democracy-Dictatorship (DD) dataset*. Si queremos utilizarlos tenemos que usar la funciones específicas del paquete. Por ejemplo, si queremos descargar *Polity IV* tenemos que introducir la función `PolityGet()`.

Marcos de datos en *dplyr* y *ggplot2*.

El paquete *dplyr*, por ejemplo, tiene el marco de datos `starwars`. El paquete *ggplot2* incorpora `presidential` (mandatos de los presidentes de los Estados Unidos desde Eisenhower). Si queréis saber más detalles, poned el nombre en la consola con un interrogante delante (ejemplo: `?Titanic`).

```
poliv <- PolityGet()
poliv <- PolityGet(vars = "polity2")
```

Si solo queremos descargar una variable, tendremos que utilizar el argumento *vars* e introducir el nombre de la variable entre comillas. En el caso de la base de datos *DD*, podemos hacer el mismo procedimiento con `DDGet()`. El paquete *psData* también trae incorporado un marco de datos denominado `countrycode_data`, muy útil para unir marcos de datos, puesto que tiene grabados varios nombres y códigos que toman los países en diferentes marcos

de datos. Podéis activarlo introduciendo el nombre y consultar la estructura. En las filas encontraréis los diferentes nombres que puede adoptar un país y en las columnas la nomenclatura que utiliza para cada país una base de datos internacional diferente. Así, podemos transformar fácilmente los nombres de países para poder comparar datos que se encuentran en diferentes marcos de datos.

Un paquete fácil de utilizar es *unvotes*, que consta de tres marcos de datos y tiene registradas todas las votaciones históricas en la Asamblea General de Naciones Unidas. El marco de datos principal del paquete es `un_votes`, que en la variable *rcid* contiene un registro del número de votación y en la variable *vote* el sentido del voto de cada país: sí, no o abstención. En `un_roll_calls` tenéis la información más ampliada de cada votación mientras que `un_roll_call_issues` nos ayuda a buscar por temática de cada votación.

Otros marcos de datos de *psData*

También se puede descargar la *Database of Political Institutions* con `DpiGet()`, la base de datos de crisis económicas de Carmen M. Reinhart con `RRCrisisGet()`, una extensa base de datos del Fondo Monetario Internacional y el Banco Mundial con `IMF_WBGet()` y el `WinsetCreator()`.

El paquete de *World Development Indicators* (WDI) contiene centenares de indicadores que se encuentran alojados en la base de datos del Banco Mundial. Una vez que hayamos cargado el paquete podremos barajar los indicadores con el motor de búsqueda `WDIsearch()`, donde introduciremos la palabra clave que nos interesa. Si queremos buscar por más de una palabra clave las separaremos con `.*`. En el ejemplo que veremos a continuación, la búsqueda por *military* nos devuelve cinco entradas. Hemos hecho una segunda búsqueda con *military* y *gdp*, que nos ha devuelto solo una entrada. En la consola veremos cómo R nos devuelve un marco de datos de dos columnas, el primero con el código de la variable y el segundo con su descripción. En la búsqueda que acabamos de hacer el código de la variable es *MS.MIL.XPND.GD.ZS* y la descripción *Military expenditure (% of GDP)*. Para importar esta información utilizaremos la función `WDI()`, donde indicaremos el código de la variable y, opcionalmente, un vector con los países y los años que queremos que nos filtre.

Paquete *countrycode*

Para una versión más ampliada de códigos de países en varias bases de datos podéis instalar y cargar el paquete *countrycode*. Una vez cargado, imprimid el `codelist_panel` para comprobar la gran cantidad de códigos diferentes que almacena este paquete.

```
WDIsearch("military")
WDIsearch("military.*gdp")
mil_exp <- WDI(indicator = "MS.MIL.XPND.GD.ZS", country = c("GB"), start = 1970, end = 2005)
str(mil_exp)
names(mil_exp)[3] <- "mil_exp_uk"
plot(mil_exp$mil_exp_uk)
```

En el ejemplo, hemos creado el objeto `mil_exp` con los datos del gasto militar del Reino Unido entre los años 1970 y 2005. Después de comprobar que la tercera columna se refiere a los valores con una visualización de la estructura, hemos cambiado el nombre de la columna por otro más fácil de recordar y a continuación hemos visualizado los datos con un diagrama de dispersión.

El paquete *eurostat* funciona de una manera parecida al *WDI*, también con un motor de búsqueda `search_eurostat()`. Hay que estar atento a mayúsculas y minúsculas porque este motor es sensible a ellas. La búsqueda es algo menos

intuitiva, por lo cual si lo preferís podéis buscar las variables que os interesen en el web de Eurostat. En el ejemplo siguiente hemos buscado la palabra *pollution* y hemos obtenido dos entradas. A continuación hemos seleccionado solo la primera columna de los datos para ver más claramente la descripción. Hemos comprobado que la segunda entrada *Environmental protection expenditure...* era la que más nos interesaba y hemos pedido la segunda fila de la segunda columna, que nos ha devuelto el código *env_ac_exp3*. A continuación, hemos importado la variable de interés con la función `get_eurostat()` y hemos denominado `pollution` al marco de datos resultante. Finalmente hemos pedido un sumario.

```
search_eurostat("pollution")
search_eurostat("pollution")[, 1]
search_eurostat("pollution")[2, 2]
pollution <- get_eurostat("env_ac_exp3", time_format = "num")
summary(pollution$values)
```

Hay más bases de datos que tienen paquete en R y funcionan de manera parecida a los descritos anteriormente: o bien podemos acceder directamente al marco de datos o bien incorporan un motor de búsqueda con la función *search* y una función *get* para importar los datos. En la tabla 1 mostramos una selección de varios paquetes relacionados con los estudios internacionales.

Paquetes fuera del CRAN

Hay otros paquetes que no están alojados en el CRAN, pero para los que algún desarrollador ha creado algún procedimiento especial para poderlos cargar. Ved, por ejemplo, los tutoriales de uso de *rqog* (Quality of Government) o de *wgi* (World Governance Indicators).

Tabla 1. Paquetes de R de bases de datos

Base de datos.	Código	Descripción
Banco Mundial	`wbstats`	Paquete de datos del Banco Mundial.
Ciencia política	`psData`	Recopilación de bases de datos de ciencia política.
Comercio OEC	`oec`	Paquete del Observatory of Trade Complexity (OEC).
Comercio UN	`comtradr`	Datos de comercio de Naciones Unidas.
Eurostat	`Eurostat`	Paquete de la base de datos Eurostat.
Estados	`states`	Datos de estados independientes en el mundo desde 1816.
FMI (1)	`IMFData`	Paquete del Fondo Monetario Internacional (FMI).
FMI (2)	`imfr`	Alternativa para acceder al FMI.
OCDE	`OECD`	Paquete para extraer datos de la OCDE.
Votos en la ONU	`unvotes`	Votos de cada país en la Asamblea General de la ONU.
WDI	`WDI`	Indicadores de desarrollo del Banco Mundial.

1.2. Importar un archivo

Para importar un archivo, lo podemos hacer manualmente por medio de File -> Import Dataset o mediante el código. En este apartado explicamos cómo crear el código para importar los archivos. Lo primero que tenemos que saber es que las bases de datos pueden estar almacenadas en diferentes tipos de archivo. Los más comunes los describimos en la tabla 2:

Tabla 2. Tipos de archivo para almacenar bases de datos[2]

Tipo	Extensión	Paquetes y funciones
CSV	`.csv`	`utils::read.csv(), utils::read.csv2(), utils::read.table()`
Excel	`.xls / .xlsx`	`gdata::read.xls(),readxl::read_excel()`
R	`.Rdata`	`base::readRSD()`
SPSS	`.sav`	`haven::read_spss, foreign_read.spss()`
STATA	`.dta`	`haven::read_stata(), haven::read_data(), foreign::read.dta()`
TXT	`.txt, .tab, .tsv`	`utils::read.delim(), utils::read.delim2(), utils::read.table()`

[2]En la columna de paquetes y funciones hemos puesto el nombre del paquete que corresponde a cada función.

Lo segundo que tenemos que saber es que el primer argumento que introduciremos en todas estas funciones es la localización del archivo que contiene la base de datos. El archivo puede estar localizado en nuestro ordenador o en la red. Si está localizado en nuestro ordenador, recomendamos tenerlo ubicado en la carpeta que hemos indicado a R como nuestro directorio de trabajo y aplicar una de las funciones de la tabla anterior. Si, por el contrario, el archivo está localizado en la red, podemos actuar de varias maneras:

Descarga desde el directorio de trabajo

Podemos saber donde tenemos ubicado el directorio de trabajo con la función `getwd()` mientras que con `dir()` veremos qué archivos tenemos guardados. Con `setwd()` o manualmente en la pestaña *Files* podremos cambiar la ubicación del directorio de trabajo. En este manual no explicaremos cómo cargar archivos ubicados en nuestro ordenador pero fuera del directorio de trabajo.

1) La primera es descargar el archivo manualmente, ponerlo en nuestro directorio de trabajo y abrirlo con una de las funciones de la tabla anterior como si estuviera localizado en nuestro ordenador.

2) La segunda es por medio de la función `download.file()`. Con esta función indicamos la dirección web en el primer argumento y el nombre del archivo que crearemos en el segundo argumento. Esta operación nos guardará el archivo en el directorio de trabajo y después lo podremos importar como en los procedimientos que acabamos de explicar.

3) La tercera opción es importarlo directamente de internet con una de las funciones indicadas en la tabla 2 usando como primer argumento la dirección web. La opción elegida dependerá de vuestras preferencias y, a veces, de cómo esté ubicado el archivo en la red.[3]

[3]A veces encontraremos bancos de datos en los que solo podemos descargar una base de datos si estamos subscritos a ellos, por lo que no podremos descargar el archivo directamente.

1.2.1. Archivos planos (.csv, .txt, .tab, .tsv)

Entendemos por archivos planos los archivos de texto que contienen los valores separados por algún carácter no alfabético, como un espacio, un guion, un punto y coma o un punto. En este apartado distinguiremos dos tipos:

- los archivos Comma Separated Values (CSV)
- los Tab-Separated Values (utilizaremos TXT como abreviación)

Estos archivos ocupan muy poco espacio de memoria de manera que el proceso para importarlos o guardarlos suele ser muy rápido. Normalmente los CSV están separados por comas, mientras que los TXT acostumbran a estar separados por tabulaciones o espacios. Si abrimos un CSV tendría más o menos el aspecto siguiente:

```
pais,pibcap,habitantes,deuda,exportaciones,importaciones
Alemania,44469,82.79,64.1,1.33,1.08
Austria,47290,8.77,78.4,2.45,3.44
Bélgica,43323,11.35,103.1,4.66,5.32
```

Podemos ver cómo, en cada fila, la separación entre valores está marcada por la coma. El punto sirve para marcar los decimales mientras que, en el caso del ejemplo, podemos observar que la primera fila representa los títulos de las columnas. Para importar estos tipos de archivo a R tenemos diferentes paquetes. En esta asignatura enseñamos el más básico, que se encuentra en el paquete *utils*, que viene de serie con R[4]. La función para importar archivos CSV `read.csv()` lleva asociados por defecto los argumentos siguientes:

(4)En el CRAN podemos encontrar otros paquetes que sirven para importar archivos planos como *readr* o *data.table*.

```
read.csv("archivo.csv", header = TRUE, sep = ",", dec = ".", stringsAsFactors = TRUE)
```

En primer lugar, pondremos el nombre del archivo que habremos situado antes en el directorio de trabajo o la dirección web donde se encuentra. Es posible que con el primer argumento ya baste para importar la tabla y no tengamos que especificar ninguno más. Pero suele pasar que la primera fila no lleve el título de las columnas, de manera que tendremos que especificar `header = FALSE`. Si es así, dentro de la función podemos especificar un nuevo argumento `col.names`, donde indicaremos el título de las columnas (por ej., `col.names = c("pais", "pibcap", "deuda", "exportaciones", "importaciones")`). El vector tendrá que tener tantos valores como columnas tiene el marco de datos.[5] En los archivos CSV, por defecto el carácter que separará los archivos es una coma, pero si es otro carácter lo tendremos que especificar. Por ejemplo, `sep = " "` indicará que el separador es un espacio. Los separadores de los archivos planos tienen mucho que ver con los sistemas de almacenar bases de datos. El sistema americano utiliza `read.csv()` y separa los decimales con el punto y los valores con la coma, mientras que el sistema europeo utiliza `read.csv2()` y separa los decimales con una coma y los

(5)Del mismo modo, R nos creará el tipo de vector asociado a cada variable por defecto, pero lo podemos especificar manualmente con el argumento `colClasses` (por ej., `colClasses = c("character", "factor", "NULL", "numeric", "logical")`). R no nos importará la columna indicada como NULL.

valores con un punto y coma. Finalmente, todos los valores que nos detecte como vectores de carácter se convertirán en factores. Si queremos cambiarlo, podemos poner el argumento en `stringsAsFactors = FALSE`.

Para importar archivos TXT utilizaremos `read.delim()`, que hace exactamente lo mismo que la función que acabamos de ver, pero tiene por defecto el separador `sep = "\t"` (que significa tabulación). La función `read.delim2()` nos importará los archivos en sistema europeo. La función `read.table()` puede leer tanto CSV como TXT, pero normalmente le tendremos que especificar más argumentos. Su separador por defecto es `sep = "/"`[6].

[6]De hecho, tenemos que pensar que en la práctica todas estas funciones que hemos visto son la misma función pero se distinguen porque tienen por defecto algunos argumentos diferentes.

1.2.2. Excel

Excel es un programa muy útil para trabajar con hojas de cálculo y bases de datos de pequeñas dimensiones. A menudo, algunas bases de datos se guardan en este formato por lo cual necesitaremos saber cómo importar archivos de Excel a R. En primer lugar, tenemos que tener en cuenta dos consideraciones importantes: los datos acostumbran a estar repartidos en varias hojas (sabréis que accedemos a ellos desde pestañas visualizables en la parte inferior de las hojas de cálculo) y muchas veces los datos no empiezan en la primera fila y la primera columna de la hoja de cálculo. Es por eso por lo que quizá valga la pena echar un vistazo previo al documento en Excel para hacernos una idea de cómo están distribuidos los datos de interés.[7] Los archivos de Excel suelen tener las extensiones *.xls* o *.xlsx*. Para importarlos a R utilizaremos los paquetes *readlxl* o *gdata*.

[7]Incluso puede ser conveniente preparar el archivo en el propio Excel, quitar las primeras filas y columnas vacías, y guardarlo directamente en formato *.csv*.

El paquete XLConnect

Otra opción para descargar Excel es XLConnect, que permite hacer una importación más selectiva y escoger entre diferentes pestañas dentro de una hoja de cálculo. Por ejemplo, si queremos obtener solo las 10 primeras columnas de la segunda pestaña del documento "documento.xlsx", primero cargaremos el archivo `doc <- loadWorkBook ("documento.xlsx")` y después lo leeremos especificando lo que queremos importar: `readWorkSheet(doc, sheet = 2, startCol = 1, endCol = 10)`. La función `getSheets()` hace una importación general de todo el documento.

Dentro del paquete *readxl*, necesitaremos la función `read_excel()` para importar los datos. Si solo introducimos dentro de la función el nombre del documento, R nos importará la primera hoja. Podemos especificar la hoja que queremos importar con el argumento adicional *sheet* donde indicamos el número de hoja o el nombre de la hoja entre comillas. El ejemplo siguiente indica a R que tiene que buscar el archivo `"docexcel.xls"` en el directorio de trabajo, importar la primera hoja de cálculo del documento, interpretar la primera fila como el título de las columnas y no eliminar ninguna fila.

```
read_excel("docexcel.xls", sheet = 1, col_names = TRUE, skip = 0)
```

En el argumento `col_names` podemos especificar TRUE si la primera fila representa el nombre de las columnas y FALSE si no hay una fila con el nombre de las columnas, o bien podemos crear un vector de carácter indicando los nombres de las columnas. Con `skip` indicamos cuántas filas nos tenemos que saltar antes de empezar a importar datos.

Importar varias hojas

Si queremos bajar varias hojas de cálculo, tendremos que crear varios códigos, cada uno importando una hoja de cálculo diferente. Para saber el nombre de cada hoja podemos utilizar la función `excel_sheets()` y especificar el nombre del archivo del que queremos obtener la lista.

El paquete *gdata* también lee documentos de Excel con la función `read.xls()`. Podemos especificar el número de hoja del mismo modo que hemos indicado anteriormente. La principal diferencia que tiene esta función

es que utiliza la mayoría de argumentos por defecto de `read.csv()`, de manera que tenemos la opción de especificar si queremos que nos convierta los caracteres en factores o no (será TRUE, por defecto).

1.2.3. Stata y SPSS

Unir varias hojas de cálculo de Excel

Una vez que hemos importado en varios marcos de datos diferentes hojas de cálculo de Excel, tenemos la opción de unirlos, siempre que los marcos de datos tengan el mismo número de filas. Con la función `cbind()` podemos especificar el nombre de cada uno de los marcos de datos. Si las primeras columnas (por ejemplo, país y año) se repiten en todas las hojas de cálculo, podemos eliminarlas indicando el número de la columna entre corchetes (por ej., `cbind(hoja1, hoja2[-2], hoja3[-1])`).

También hay paquetes pensados para importar bases de datos de software parecido a R, como Statistics and Data (Stata) o Statistical Package for Social Sciences (SPSS). El programa Stata se utiliza principalmente para econometría y guarda los archivos con *.dta*, mientras que SPSS se utiliza más bien en ciencias sociales y guarda los archivos con *.sav* o *.por*. Para importar estos archivos a R podemos utilizar dos paquetes: *haven* o *foreign*.

1) El paquete *haven* utiliza `read_stata()` y `read_dta()` para importar archivos STATA de las versiones 8 a la 15 y `read_spss()` para importar archivos SPSS. La complicación principal para R con los archivos provenientes de estos programas estadísticos es que los vectores de carácter están almacenados como numéricos y guardan los caracteres como etiquetas. Con *haven* no es posible convertir los datos directamente a factores, de manera que lo tendremos que hacer manualmente después de la importación. Para hacerlo, utilizaremos la función `as_factor()` de *haven* para transformar el vector en cuestión en factor y, si hace falta, después lo transformaremos en carácter con `as.character()`.

2) Como alternativa a descargar STATA y SPSS tenemos el paquete *foreign*, que soporta muchos tipos de formatos diferentes aparte de los mencionados. En el caso de STATA utilizaremos `read.dta()` y en el caso de SPSS utilizaremos `read.spss()`. El principal inconveniente que tiene la función `read.dta()` es que no soporta archivos STATA superiores a la versión 12. Sin embargo, esta función sí que nos permite convertir directamente las etiquetas de las variables en factores con el argumento ya establecido por defecto `convert.factors = TRUE`. La función `read.spss()` también tiene predeterminado que convierta las etiquetas de SPSS en factores mediante el argumento `use.value.labels = TRUE`. Sí que es importante especificar siempre `to.data.frame = TRUE`, puesto que por defecto no nos crea un marco de datos sino otro tipo de objeto.

1.3. Guardar el marco de datos

Finalmente, después de haber hecho las transformaciones pertinentes en un marco de datos, podemos tener la necesidad de guardarlo en un archivo. El programa R tiene un formato propio con la extensión *.RData* y utiliza la función `saveRDS()`. La gran ventaja de guardarlo en este formato es que no perderemos información sobre el tipo de variables, el orden de los factores, etc. A continuación, veremos cómo guardar el marco de datos `md` en formato R,

que denominaremos "docR.xls". En segundo lugar, veremos la operación contraria: cómo importar el archivo hacia el marco de datos md mediante la función readRDS():

```
saveRDS(md, "docR.xls")
md <- readRDS("docR.xls")
```

También podemos guardar el marco de datos en un formato más versátil que se pueda abrir en otros programas como Excel, SPSS o STATA. Una buena opción es hacerlo en CSV por medio de write.table(). En el código siguiente hemos ilustrado un ejemplo en el que guardamos el marco de datos de nombre md en el archivo "docR.csv". En el tercer argumento especificamos que queremos que nos separe los valores por comas, que el punto marque los decimales y que queremos que nos guarde los nombres de las columnas en la primera fila. Por defecto, row.names es TRUE, de manera que nos guardará una primera columna adicional con el número de cada fila. Es conveniente evitarlo, por lo tanto pondremos el argumento en FALSE.

```
write.table(md, file = "docR.csv", sep = ",", dec =  ".", col.names = TRUE,  row.names = FALSE)
```

Los paquetes que hemos visto, como *readxl*, *haven* o *foreign* tienen sus propias funciones para exportar los datos a archivos en formato Excel, SPSS o STATA. También es una buena opción importar y exportar archivos con el paquete *readr*, que no hemos visto en este apartado pero que tendremos que conocer si queremos adquirir un dominio más avanzado en la importación y exportación de marcos de datos.

2. Limpiar bases de datos

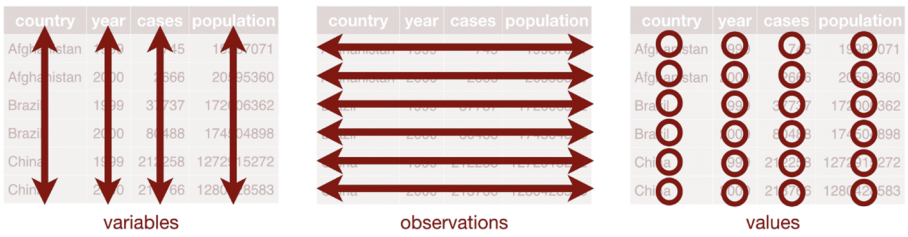
Cuando importamos un marco de datos a R, no necesariamente nos llega limpio, con los datos a punto para ser explorados y analizados. Consideramos limpio un marco de datos si cumple cuatro requisitos (Wickham, 2014):

- cada tipo de unidad observacional tiene que ser un marco de datos diferente,
- las observaciones tienen que estar representadas en las filas,
- las variables tienen que estar representadas en las columnas, y
- cada casilla nos tiene que indicar un valor.

El primer requisito implica que solo podemos tener una sola unidad de análisis en cada marco de datos. ¿Qué observa el marco de datos en cuestión? ¿Personas, países, datos de países, regiones? Lo que no tendría sentido es que los datos en un mismo marco se refirieran a veces a personas y a veces a países. En la mayoría de casos trabajaremos con el país como unidad de análisis.

Los otros tres requisitos para tener un marco de datos limpio se refieren a la estructura de un marco de datos, que resumimos en la figura 1. Un marco de datos es una colección de valores, cada uno de ellos representado en una casilla diferente. Cada valor pertenece a una variable y observación determinada. Cada variable contiene todos los valores que miden el mismo atributo en todas las unidades. Cada observación contiene todos los valores que miden la misma unidad mediante todos sus atributos. Si la tabla cumple estos requisitos, quiere decir que tenemos una tabla limpia.

Figura 1. Requisitos para una tabla limpia



Fuente: Grolemund y Wickham (2017). *R for Data Science.* O'Reilly. CC BY-NC-ND 3.0 OS.

Lo contrario de una tabla limpia es una tabla sucia. Puede ser que los nombres de las columnas sean valores en lugar de variables, que algunas variables no estén representadas por el tipo de vector que queremos en cada caso, que haya más filas o más columnas de las necesarias o que haya una gran cantidad de datos perdidos o valores extremos que pueden ser errores en los datos. Uno de los síntomas de una tabla sucia (aunque no es un requisito necesario) es que tengamos muchas variables y pocas observaciones. Lo primero que haremos

antes de empezar la limpieza es un análisis exploratorio general, para hacernos una idea de la estructura del marco de datos con funciones como `str()`, `glimpse()`, `names()`, `summary()`, `hist()` o `plot()`.

2.1. Limpiar con *tidyr*

Si vemos que un marco de datos no cumple los requisitos de una tabla limpia, lo que necesitaremos es el paquete *tidyr*. Tendremos que asegurarnos que esté instalado y cargado en nuestro ordenador. Este paquete nos ayuda a transformar un marco de datos principalmente de dos maneras:

- convirtiendo nombres de columna en valores de una nueva variable, o
- convirtiendo valores de una variable en nombres de columnas.

Para saber más

Sobre el paquete *tidyr*, el científico de datos de RStudio Garrett Grolemund ha preparado un GitHub con una muy buena guía (http://garrettgman.github.io/tidying/) para limpiar marcos de datos.

En el primer caso alargamos el marco de datos, puesto que reducimos el número de columnas y ampliamos el número de filas. Lo haremos con la función `gather()`. En el segundo caso ensanchamos el marco de datos, puesto que reducimos el número de filas y ampliamos el número de columnas. Lo haremos con la función `spread()`. También aprenderemos dos funciones más de *tidyr* que van muy ligadas a estas dos: `separate()` y `unite()`. Encontramos resumida la utilidad de las cuatro funciones en la tabla 3.

Tabla 3. Funciones básicas con *tidyr*

`gather()`	Alarga la tabla transformando varias columnas en valores de una nueva columna.
`spread()`	Ensancha la tabla transformando los valores de una columna en nombres de nuevas columnas.
`separate()`	Separa los valores de una columna en varias columnas.
`unite()`	Une los valores de varias columnas en una columna.

2.1.1. Las cuatro funciones de *tidyr*.

Empezamos aprendiendo el funcionamiento de `gather()`. Fijaos en el marco de datos siguiente, `tabla_sucia`, que encontraréis disponible en el anexo, en el cual vemos el número de protestas ciudadanas que ha habido en varios países, separadas por el año en el que se han producido en cada columna. ¿Os parece que es una tabla limpia?

```
> tabla_sucia
     pais X2016 X2017 X2018
1  España    64    83    95
2 Francia    43    34    33
3  Italia    56    53    67
4  México    35   101   120
5   Japón    12    19    18
```

Parece que no. En las columnas vemos indicados los años 2016, 2017 y 2018, que realmente tendrían que ser valores de la variable año.[8] Siguiendo los criterios de lo que consideramos una tabla limpia, necesitaríamos eliminar las columnas existentes y crear dos columnas nuevas, la columna *año* y la columna *protesta*. Lo podemos hacer fácilmente con la función `gather()`, donde primero indicaremos el marco de datos que queremos transformar, seguido por los nombres de las dos nuevas columnas: el primero indicará el nombre de la nueva columna que representará los valores que hasta ahora eran títulos de columna (la llamaremos *columna clave*), mientras que el segundo nos indicará el nombre de la nueva columna que representará los datos que estaban en las observaciones (la llamaremos *columna valor*). En último lugar, indicaremos los nombres de las columnas que hay que agrupar. Si son todas las columnas del marco de datos no hace falta que indiquemos nada. Si son varias columnas lo tendremos que especificar con un concatenado. Es muy posible que nos sea más cómodo indicar con el símbolo negativo las columnas que no se tienen que agrupar. Las dos posibilidades, que especificamos a continuación, darán el mismo resultado.

[8] Recordad que los nombres de columnas no pueden empezar con un número, de manera que, en R, ponemos una X delante. Más adelante ya la quitaremos.

Funciones *parse* del paquete *readr*

El paquete *readr* contiene las funciones *parse*, que también son muy útiles para arreglar marcos de datos. Por ejemplo, en este caso, podemos limpiar un vector de carácter y dejar solo los números con `parse_number()`. En nuestro caso, introduciríamos `tabla_limpia$año <- parse_number(tabla_limpia$año)`.

```
gather(marco de datos, columna clave, columna valor, columnas que se tienen que agrupar)
tabla_limpia <- gather(tabla_sucia, ano, protestas, c(X2016, X2017, X2018))
tabla_limpia <- gather(tabla_sucia, ano, protestas, -pais)
```

Ahora tenemos el nuevo marco de datos `tabla_limpia`, que cumple los requisitos de tener las observaciones en las filas y las variables en las columnas. Para acabar de limpiar la tabla correctamente todavía tendremos que dar un paso más, puesto que en la columna de año tenemos los valores con una X al principio: X2016, X2017 y X2018. Una manera de quitar esta X es con la función `separate()`, que separa los valores de una columna por el carácter que indicamos. Dentro de la función tenemos que indicar el nombre del marco de datos, seguido de la columna que contiene los valores que queremos separar. A continuación introducimos un vector con el nombre acompañado de comillas que tomarán las nuevas columnas. Finalmente, con el atributo *sep* indicaremos, también entre comillas, cuál es el carácter o caracteres que hacen de separadores y que, por lo tanto, serán eliminados.

Limpiar vectores con *stringr*

Otra opción que permite eliminar parte de un vector de carácter es con la función `string_replace()` del paquete *stringr*. En el ejemplo, tendríamos que introducir la operación siguiente: `str_replace(tabla_limpia$ano, "X", "")`.

```
separate(marco de datos, columna, "nombre nuevas columnas", sep = " ")
tabla_sep <- tabla_limpia %>%
  separate(ano, c("nada", "ano"), sep = "X") %>%
  select(-nada) %>%
  head(8)
```

Si solo utilizamos la función `separate()`, R nos creará dos columnas: la columna *año* con el año correspondiente y la columna *nada*, que tendrá todos los valores en blanco, puesto que no hay nada a la izquierda de la X que hemos indicado como separador.[9] Para poder eliminar la columna *nada*, en la función anterior hemos abierto una *pipe* para el objeto `tabla_limpia`. Después

[9] Una manera más rápida de quitar la X es con `separate(tabla_limpia, año, into = "ano", extra = "drop", sep = "X")`.

de separar la columna de interés, hemos utilizado la función `select()` de *dplyr* para eliminar la columna *nada*. Finalmente, hemos imprimido las ocho primeras filas de la tabla resultante.

Seleccionar el separador

Utilizar el argumento *sep* es, en muchas ocasiones, opcional. R interpretará de manera automática que el separador es un carácter no alfabético como un espacio, un guion, una barra baja, etc. y lo eliminará. En el supuesto de que el símbolo sea un punto (`.`), lo tendremos que indicar así: `"\\."`.

El paquete *tidyr* también nos permite hacer el procedimiento contrario al que acabamos de ver. La función `spread()` hace lo contrario que `gather()`, es decir, podemos tomar los valores categóricos de una columna y convertirlos en títulos de columna. La función que indicamos en el código siguiente nos ilustra este procedimiento, en el cual introducimos primero el nombre del marco de datos y en segundo lugar la columna clave del marco de datos original que contiene los datos que queremos que se conviertan en los títulos de las nuevas columnas. En tercer lugar, introduciremos el nombre de la columna del marco de datos original que contiene los valores de las columnas que crearemos. Recordad que, al referimos en estos procesos a nombres de variables existentes, no nos hace falta especificar el nombre entre comillas.

```
spread(marco de datos, columna clave, columna valores)
spread(tabla_limpia, ano, protestas)
```

Del mismo modo que `spread()` hace lo contrario que `gather()`, la función `unite()` hará lo contrario que `separate()`. Es decir, esta función nos une los datos de dos columnas en una misma columna. Por defecto, `unite()` unirá los valores indicados con un guion bajo. Si queremos que utilice otro símbolo para unir los datos lo tendremos que indicar expresamente. En el ejemplo que veremos a continuación, hay un marco de datos creado anteriormente, `tabla_sep`, que tiene los valores *año* y *mes* separados en dos variables diferentes.

```
> tabla_sep
      pais  ano mes protestas
1   España 2016   6        42
2   España 2017   1        33
3   España 2017   6        83
4  Francia 2016   6        42
5  Francia 2017   1        44
6  Francia 2017   6        34


unite(marco de datos, columna nueva, columna1, columna2 ..., sep = "_")
unite(tabla_sep, ano_mes, ano, mes, sep = "/")
```

Si queremos juntar los valores de las columnas *año* y *mes* con una barra inclinada, tenemos que indicar dentro de `unite()` el marco de datos, seguido del nombre de la columna que queremos crear. A continuación, introduciremos el nombre de las dos columnas del marco de datos que queremos unir. En es-

te caso, no tenemos que indicar el nombre de las columnas entre comillas. Finalmente, introduciremos *sep* seguido del símbolo que queremos que una los valores.

2.2. Convertir variables

El segundo problema que podemos tener con un marco de datos es que las variables no tengan asignado el tipo de vector que queremos. Por ejemplo, podemos encontrar que algunas de las variables categóricas que queremos como factores estén codificadas como vectores de carácter o que algunas numéricas sean dobles en lugar de números enteros. Parte del problema lo podemos solucionar cuando descargamos la base de datos con atributos que tienen algunas funciones de tipo `StringAsFactors = FALSE`, pero es posible que una vez descargadas queramos solucionar algunos casos.

Las funciones que tenemos que utilizar para convertir variables ya las conocemos: `as.numeric()`, `as.character()`, `as.integer()`, `as.logical()`, etc. Sin embargo, tenéis que tener en cuenta algunas consideraciones, sobre todo cuando convertimos factores en otros tipos de variables. Como sabéis, los factores son vectores enteros tratados como variables categóricas. Como tales, cada categoría es realmente un número entero con una etiqueta con información categórica. Cuando pasamos un factor a carácter, convertirá las etiquetas en *strings* y eliminará los números enteros. Por el contrario, cuando pasamos un factor a numérico convertirá los números enteros en numéricos y eliminará las etiquetas. En el supuesto de que las etiquetas de un factor sean números y nos interese guardarlos como tales, tendremos que hacer una conversión doble: primero transformar el factor en carácter para obtener los números y después transformarlo en numérico: `as.numeric(as.character(factor))`.

Estas conversiones no hace falta que las hagamos una por una, sino que también las podemos hacer a gran escala. La función `mutate_at()` del paquete *dplyr* permite tomar muchas variables de un marco de datos y convertirlas de tipos en una sola operación. Supongamos que queremos cambiar a numérico las columnas 6 a 50 del marco de datos `pol`. Introduciremos `mutate_at(pol, vars(6:50), funs(as.numeric))`. Podemos indicar indistintamente el nombre de la columna o el número de la columna.

Cambiar el nombre de las variables

Si queremos cambiar el nombre de las variables, tenemos dos opciones. La primera es cambiar el nombre de todas las variables del marco de datos. Para eso tenemos que crear un vector con los nombres nuevos e incorporarlo a los nombres de las columnas. En este ejemplo, modificamos el nombre de las cinco columnas del hipotético marco de datos `md`:

```
colnames(md) <- c("pais", "ano", "poblacion", "pib_cap", "democracia")
```

Si, al contrario, solo queremos cambiar el nombre de una variable que se encuentra dentro de un marco de datos, podemos indicar el número de columna entre corchetes e insertar el nombre nuevo dentro de un vector de carácter. Por ejemplo, hemos decidido:

```
names(md)[3] <- c("pop")
```

Hay algunas cuestiones estilísticas que hay que tener en cuenta, como intentar poner todos los nombres de columna en minúsculas y utilizar siempre la barra baja (_) como separador dentro del mismo nombre en lugar de otros símbolos. Para poner todos los nombres en minúsculas utilizaremos `tolower(names(md))` y para ponerlos en mayúsculas usaremos `toupper(names(md))`.

Quizás en algún momento durante la limpieza del marco de datos también queréis crear una columna que sea sencillamente el orden de las filas. Para crear la hipotética columna `num` en el marco de datos `md`, tendríais que utilizar el código siguiente: `md$num <- 1:nrow(md)`.

2.3. Eliminar filas y columnas

Una parte importante de la limpieza de los marcos de datos es eliminar las filas y las columnas que sabemos que no usaremos. Eliminarlas lo hace más facil a la hora de trabajar y significa más velocidad para R a la hora de procesar la información. Como ya sabéis, *dplyr* tiene la función `filter()` para eliminar filas y `select()` para eliminar columnas. Alternativamente, también podéis utilizar los corchetes para suprimir las filas y columnas que no nos interesen. Recordad que para hacer una selección de un marco de datos, dentro del corchete tenemos que introducir un vector con la selección de filas y, separado por una coma, el vector con la selección de columnas. El símbolo negativo indica que en lugar de seleccionarlas, las queremos eliminar. En el código siguiente tenéis varios ejemplos de selecciones:

```
md <- md[-c(4, 23:50), ]
md <- md[ ,-c(14:ncol(md))]
md <- md[1000:nrow(md), -c(1, 5:7, 10:20)]
```

En el primer código eliminamos la fila 4 y las filas de la 23 a la 50. En el segundo ejemplo pedimos suprimir desde la columna 14 hasta el final. En el tercero queremos conservar desde la fila 1.000 hasta el final y eliminar las columnas 1, 5, 6, 7 y de la 10 a la 20. Las funciones `ncol()` y `nrow()` nos devuelven el número total de filas y columnas respectivamente.

2.4. Anomalías en los datos

Cuando cargamos una base de datos de la red, también es habitual que tenga un contenido incompleto. No siempre todas las casillas tienen datos y a veces estos datos no son correctos. En el caso de casillas sin datos (datos perdidos), tendremos que detectarlos haciendo una observación rápida al marco de datos una vez lo hayamos descargado. La función más genérica y útil para detectar datos perdidos es la de `summary()`. En caso de que haya datos perdidos, R nos devolverá una fila extra en la descripción de la variable indicando el número de datos perdidos. En cuanto a los errores, no hay ninguna solución mágica para encontrarlos, pero con un sumario podemos detectar, algunas veces, valores que se escapan de toda lógica.

2.4.1. Datos perdidos

En R veremos los datos perdidos codificados como NA (*Not Available*, no disponible).[10] Hay infinidad de razones por las cuales podemos tener datos perdidos en una base de datos. En los estudios internacionales, la presencia de datos está muy asociada a la capacidad de cada estado de recolectarlos (Stone, 2008). Así, si buscamos dentro del banco de datos del Banco Mundial, es muy probable que tengamos mucha información de los Estados Unidos pero en cambio variables vacías de información sobre países pequeños o en desarrollo como Tonga o Angola. Los datos perdidos también pueden ser aleatorios debido a errores al codificar la información.

[10] Hay que distinguir los datos perdidos (NA) de los valores infinitos, señalizados con *Inf*, y de los valores que no son números, señalizados con *NaN* (*Not a Number*).

Para saber si tenemos NA en nuestro marco de datos, lo primero que podemos hacer es especificar el marco de datos en cuestión dentro de la combinación de funciones `any(is.na())`. R nos devolverá TRUE si tenemos algún dato perdido y FALSE si no tenemos ninguno. Si R nos dice que no tenemos ninguno, ya podemos saltar al paso siguiente. Pero si R nos dice que sí que tenemos, podemos pedir que cuente la cantidad de datos perdidos del marco de datos con la combinación de funciones `sum(is.na())`. R sumará la cantidad de TRUE del marco de datos.

La función *is.na*

Si preguntamos simplemente `is.na()`, R devolverá el mismo marco de datos entero pero con valores TRUE o FALSE en cada casilla, de manera que tendremos marcados los datos perdidos como TRUE y los otros como FALSE. Este método no es recomendable con marcos de datos grandes.

Una vez ya sabemos si tenemos o no NA, el segundo paso es localizar donde están. A ello nos puede ayudar la función `summary()`. Si aplicamos esta función al marco de datos obtendremos los estadísticos descriptivos de cada variable y, si alguna variable tiene datos perdidos, veremos el número de NA al final del sumario de la variable. A continuación hemos pedido un sumario hipotético de la variable `md$gdp_cap`. R devuelve lo siguiente:

```
> summary(md$gdp_cap)
     Min.   1st Qu.    Median      Mean   3rd Qu.      Max.    NA's
   -241.2    1413.1    4291.9    7215.3    9325.5  213623.1       4
```

Tal como vemos al final del sumario, esta variable tiene cuatro datos perdidos. Para identificar en qué fila se encuentra exactamente cada uno de estos NA, podemos utilizar la función `which(is.na())`, que devolverá un vector numérico con la posición de las filas que tienen datos perdidos de la variable indicada.[11] Ahora solo tenemos que visualizar las filas en cuestión seleccionándolas del marco de datos. Supongamos que la operación anterior nos ha devuelto las filas 34, 567, 1.234 y 1.520. Podemos crear un vector con estos números y seleccionarlos como filas en el marco de datos.

[11] Es mejor hacer esta operación con vectores, no con marcos de datos. Si indicamos todo el marco de datos, R devolverá el número que tienen las casillas donde hay NA. Esta información nos será menos útil.

```
> na_md <- c(34, 567, 1234, 1520)
> md[na_md, ]
   pais     continente   ano   gdp_cap
1 Chad      Africa      1993        NA
2 Myanmar   Asia        2005        NA
```

```
3 Senegal  Africa      2001        NA
4 Tonga    Oceania     1999        NA
```

Llegados a este punto, la gran pregunta es: ¿qué hacemos con estos datos? Tenemos dos opciones principales: eliminar la fila entera o sustituir los datos perdidos por otros. No hay una única respuesta a esta pregunta y dependerá en buena parte de la posibilidad de averiguar cuáles son los datos que faltan o de la validez de los métodos para imputar un valor a los NA. El manual *Handbook on Constructing Composite Indicators* de la OCDE (2008) ofrece varias posibilidades a la hora de imputar nuevos valores. Si decidimos eliminar las filas, el paquete *tidyr* nos ofrece una solución rápida con la función `drop_na()`, que devolverá un nuevo marco de datos con las filas que no tengan valores perdidos.[12]

[12]La función `na.omit()` del paquete de base *stats* hace el mismo procedimiento.

Si lo que queremos es sustituir los datos perdidos por unos valores concretos, tenemos dos posibilidades:

- la opción automática que nos ofrece *tidyr*,
- la opción manual con corchetes.

Añadir NA a un marco de datos

Es posible que encontremos variables con filas vacías que queramos cambiar a NA. Para convertirlas en NA aplicaremos `md$var[md$var == ""] <- NA`.

Si aplicamos la función `fill()` de *tidyr* a un vector llenará la fila con el valor más reciente mientras que `replace_na()` hará una estimación a partir de los valores superior e inferior de la casilla donde tengamos el NA.

La opción manual consiste en identificar la columna y el número o números de casilla que tengan datos perdidos y asignar individualmente o conjuntamente un nuevo valor. Como veremos a continuación, en el primer caso hemos asignado el valor 2.045 a la fila 34 de la variable `md$gdp_cap`, mientras que en el segundo caso hemos aprovechado el vector `na_md` creado previamente para asignar el valor 2.045 a las cuatro filas con datos perdidos.

```
md$gap_cap[34] <- 2045
md$gap_cap[na_md] <- 2045
```

2.4.2. Valores extremos (*outliers*)

Otro tipo de anomalía que podemos encontrar en un marco de datos son los valores extremos. Un valor extremo es un valor fuera de lo normal en nuestra distribución. Es posible que este valor sea perfectamente válido, como el caso del PIB per cápita de Kuwait en el marco de datos *gapminder*[13]. Por lo tanto,

[13]Si tenéis cargado el paquete *dplyr* haced la prueba: `arrange(filter(gapminder, year == 1952), desc(gdpPercap))`.

que sea extremo no quiere decir que no sea «real», pero siempre será motivo de sospecha. Los valores extremos son normalmente consecuencia de errores en el proceso de creación de los datos o fruto de codificaciones especiales.

Las codificaciones de Polity IV

La base de datos Polity IV codifica los países en una escala de -10 a +10 según si son más autocráticos o más democráticos. Pero los valores toman codificaciones especiales cuando son regímenes en transición (código -88), interrupción del régimen (código -66) o periodo *interregnum* (código -77). Usar Polity IV comporta tener que tomar decisiones sobre cómo recodificar estos valores.

Como ya hemos explicado, no hay ninguna fórmula mágica para identificar valores extremos. Lo más importante es hacer una lectura atenta de los datos, explorar cada variable y pensar si tiene sentido que los datos tengan el valor que tienen. Algunos errores son fáciles de detectar. Si sabemos, por ejemplo, que la esperanza de vida mediana oscila normalmente entre 50 y 82 años, nos sorprenderá mucho si encontramos un valor de 900. También será difícil que un país tenga un PIB per cápita negativo. Por lo tanto, lo más importante es hacer un esfuerzo inicial para conocer a fondo qué miden las variables que queremos trabajar, qué unidades de medida utilizan y entre qué valores acostumbran a oscilar estas unidades.

Fijémonos bien en el sumario que hemos hecho en la sección anterior. El valor mínimo de la variable `md$gdp_cap` es negativo y, por lo que sabemos, los PIB per cápita no suelen ser negativos. También observamos que el valor máximo supera los 200.000 dólares por habitante y ningún país acostumbra a superar los 100.000. Nuevamente podemos utilizar la función `which()` para identificar la fila o filas que tienen valores poco habituales. En el primer caso hemos seleccionado las filas con valores negativos de la variable `md$gdp_cap`, mientras que en el segundo caso hemos buscado los valores extremos de dos maneras: mirando qué valores son superiores a 100.000 o directamente yendo a buscar el número concreto que habíamos encontrado a partir de la función `summary()`.

```
md[which(md$gdp_cap < 0), ]
  pais       continente    ano   gdp_cap
1 Singapur   Asia        1991    -241.2

md[which(md$gdp_cap > 100000), ]
md[which(md$gdp_cap == 213623.1), ]
  pais      continente   ano    gdp_cap
1 Canada    Africa     2005   213623.1
```

Una manera alternativa de identificar casos extremos es visualizando la distribución de una variable. Normalmente con `hist()` o `plot()` podremos comprobar si hay valores que se salen de la norma. Una vez los tenemos identificados, el procedimiento vuelve a ser el mismo que para el caso de los NA.[14]

[14] Por ejemplo, hemos encontrado en otra fuente que el PIB per cápita de Singapur en 1991 era de 11.463 dólares. Podemos utilizar este valor en el marco de datos: `md$gap_cap[which(md$gdp_cap < 0)] <- 11463`.

3. Unir marcos de datos

Los analistas de datos a menudo quieren analizar la relación entre dos variables, pero resulta que estas variables están separadas en dos bases de datos diferentes. Por ejemplo, podría ser que en una base de datos tuviéramos información sobre democracia y en otra tuviéramos información sobre conflictos. Nos gustaría saber si hay alguna relación entre democracia y conflicto, pero para ello antes que nada tendríamos que unir estas dos bases de datos en una. Unirlas puede ser un proceso complicado, no solamente por el gran volumen de información que puede contener cada una de ellas, sino por varios obstáculos que pueden haber, como que la unidad de análisis no coincide o que el nombre de los países que queremos unir están registrados en idiomas diferentes. Normalmente, ante este tipo de situación, la solución que se acaba adoptando es unirlas manualmente, fila a fila, en un proceso que puede tardar horas y horas. Por suerte, programas como R tienen recursos para unir tablas con cierta facilidad. En este apartado descubriremos la segunda gran virtud del paquete *dplyr*: aparte de manipular datos, esta librería también tiene como segunda especialidad unir marcos de datos.

Métodos para unir marcos de datos

Las funciones del paquete *dplyr* tienen una sintaxis muy intuitiva y que puede ser aplicada a otros objetos aparte de un marco de datos. Podéis ver algunos ejemplos en este enlace (https://dplyr.tidyverse.org/reference/join.html). Aparte de *dplyr*, hay una función de base de R, `merge()`, que también puede unir tablas.

3.1. Juntar marcos de datos

Para juntar dos marcos de datos con *dplyr*, el escenario ideal que nos gustaría encontrar es que hubiera una columna común de referencia a las dos tablas. Es decir, como veremos en el ejemplo siguiente, el marco de datos `md_dem` (que indica si han tenido un régimen democrático en los últimos treinta años) y el marco de datos `md_war` (que indica hipotéticamente el número de disputas militares en los últimos treinta años) tienen una columna común, *country*, que tiene las categorías con el mismo nombre. Francia es *France* en los dos marcos de datos, España es *Spain*, y así sucesivamente. En esta situación ideal, juntar marcos de datos es sencillo con *dplyr*.

Disponibles en el anexo

Los códigos de los marcos de datos `md_dem` y `md_war` están disponibles en el anexo.

```
> md_dem                > md_war
  country   dem           country war
1  France  TRUE         1  France  21
2   Spain  TRUE         2   Spain  13
3 Germany  TRUE         3 Germany   9
4   China FALSE         4   China  24
5   Yemen FALSE         5   Yemen  14
6   Haiti FALSE         6   Congo  42
```

A la columna que ambos marcos de datos tienen en común la denominaremos *columna clave* y será la columna que nos servirá para juntar las bases de datos. Para unirlas, siempre entenderemos que hay un marco de datos de referencia (que recibirá los datos del otro marco de datos) y un marco de datos secunda-

rio. Supongamos, en el caso que nos ocupa, que queremos traer los datos de `md_war` hacia `md_dem` y como referencia tenemos la columna *country*. A la columna *country* de `md_dem` la llamaremos *columna clave primaria*, mientras que a la columna *country* de `md_war` la denominaremos *columna clave secundaria*. La distinción es importante, puesto que en algunas operaciones R tendrá que decidir si conserva algunos valores, y en estos casos siempre conservará los valores de la clave primaria.

En la tabla 4 vemos las seis funciones que estudiaremos de *dplyr*. Todas ellas tienen la misma sintaxis, formada por tres argumentos dentro de la función: el nombre del primer marco de datos, el nombre del segundo marco de datos y el argumento *by* seguido del nombre de la columna clave entre paréntesis.

Tabla 4. Unir marcos de datos con *dplyr*

`left_join()`	Devuelve todas las filas del primer marco de datos y las filas del segundo marco de datos que coincidan con la columna o columnas clave.
`right_join()`	Devuelve todas las filas del segundo marco de datos y las filas del primer marco de datos que coincidan con la columna o columnas clave.
`inner_join()`	Devuelve las filas del primero y el segundo marco de datos que coincidan con la columna o columnas clave.
`full_join()`	Devuelve todas las filas y columnas del primero y el segundo marco de datos.
`semi_join()`	Devuelve las filas del primer marco de datos que coincidan con la columna o columnas clave del segundo marco de datos..
`anti_join()`	Devuelve las filas del primer marco de datos que no coincidan con la columna o columnas clave del segundo marco de datos.

Más de una columna clave

En la unión de dos marcos de datos podemos identificar más de una columna clave. Aunque en los ejemplos hemos trabajado con una sola columna clave, muchas veces utilizaremos dos. Trabajar con dos columnas clave es habitual, por ejemplo, cuando tenemos los datos agrupados por país y año. En este caso, la sintaxis será: `xxxx_join(x, y, by = c("pais", "ano"))`.

La sintaxis de las funciones *join*

La sintaxis en código sería `xxxx_join(x, y, by = "columna")`. Si el nombre de la columna clave no coincide en los dos marcos de datos lo especificaremos de la manera siguiente: `xxxx_join(x, y, by = c("nombre_col1" = "nombre_col2"))`. R mantendrá el nombre del primer marco de datos. Si no indicamos el argumento *by* en la sintaxis, R mirará los dos marcos de datos y combinará las columnas que encuentre con el mismo nombre. En la consola, R explicará qué procedimiento ha seguido.

La única diferencia entre `left_join()` y `right_join()` es que la primera utiliza el primer marco de datos como primario mientras que la segunda utiliza el segundo. Con `left_join()`, R devolverá todas las filas del primer marco de datos y añadirá cualquier fila del segundo marco de datos que encaje con la columna clave. Con `right_join()` R hará exactamente lo contrario: devolverá todas las filas del segundo marco de datos y añadirá cualquier fila del primer marco de datos que encaje con la columna clave.

Fijaos cómo, a continuación, unimos los marcos de datos `md_dem` y `md_war` de dos maneras diferentes. Con `left_join()`, hemos conservado Haití porque estaba en el primer marco de datos, pero, al no tener datos para Haití de la columna *war*, nos ha devuelto un NA. Con `right_join()` conservamos los datos del segundo marco de datos y por lo tanto tenemos el Congo en lugar de Haití. R ha buscado los datos de democracia del Congo, pero al no encontrarlos en la clave secundaria, nos devuelve un NA.

```
> left_join(md_dem, md_war)
  country   dem war
1  France  TRUE  21
2   Spain  TRUE  13
```

```
3 Germany  TRUE   9
4   China FALSE  24
5   Yemen FALSE  14
6   Haiti FALSE  NA

> right_join(md_dem, md_war)
  country   dem war
1  France  TRUE  21
2   Spain  TRUE  13
3 Germany  TRUE   9
4   China FALSE  24
5   Yemen FALSE  14
6   Congo    NA  42
```

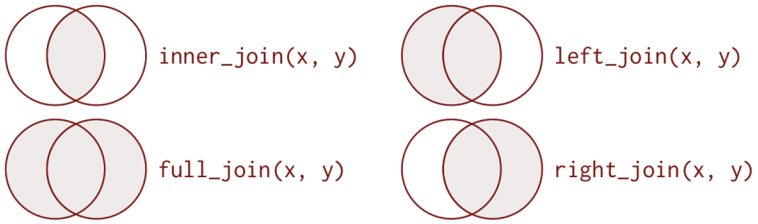
Las dos funciones que acabamos de ver, `inner_join()` y `full_join()`, tienen una lógica muy parecida. En el caso de `inner_join()`, R solo devolverá las filas que aparezcan en los dos marcos de datos, mientras que el en caso de `full_join()` obtendremos los valores que aparezcan tanto en un marco de datos como en el otro. Lógicamente, `full_join()` producirá muchos NA. Como podemos ver a continuación, el código de la izquierda nos une los marcos de datos de manera exclusiva, puesto que eliminará todo lo que no tenga información en los dos marcos de datos, como es el caso de Haití y Congo. La segunda función es inclusiva, puesto que conserva los valores aunque solo aparezcan en un marco de datos.

```
> inner_join(md_dem, md_war)
  country   dem war
1  France  TRUE  21
2   Spain  TRUE  13
3 Germany  TRUE   9
4   China FALSE  24
5   Yemen FALSE  14

> full_join(md_dem, md_war)
  country   dem war
1  France  TRUE  21
2   Spain  TRUE  13
3 Germany  TRUE   9
4   China FALSE  24
5   Yemen FALSE  14
6   Haiti FALSE  NA
7   Congo    NA  42
```

Estas cuatro funciones de *dplyr* que acabamos de ver están resumidas visualmente en la figura 2. En esta imagen vemos la lógica de cada función y observamos claramente que `inner_join()` es más exclusiva que `full_join()`.

Figura 2. Lógica de unión de las funciones de *dplyr*



Fuente: Grolemund y Wickham (2017). *R for Data Science.* O'Reilly. CC BY-NC-ND 3.0 OS.

Lo que hemos visto hasta ahora con estas cuatro funciones de *dplyr* ha sido como unir valores de dos marcos de datos diferentes en una o varias columnas clave. A continuación, veremos dos funciones que hacen una tarea parecida, pero que no incorporan los valores del segundo marco de datos. Es decir, lo que hacen es filtrar el primer marco de datos con criterios del segundo, pero no incorporan los valores del segundo marco de datos en el primero. Estas dos funciones son `semi_join()` y `anti_join()`. La función `semi_join()` filtra el primer marco de datos según los valores que coinciden con el segundo marco de datos. La función `anti_join()` hace todo lo contrario que `semi_join()`: filtra el primer marco de datos según los valores que no coinciden con el segundo marco de datos. En el primer caso, el único valor que no coincidía con el segundo marco de datos es Haití, por lo cual lo ha filtrado del marco de datos original. En el segundo caso, ha hecho a la inversa. Como Haití era el caso que no coincidía con el segundo marco de datos, lo ha mantenido.

```
> semi_join(md_dem, md_war)
  country   dem
1  France  TRUE
2   Spain  TRUE
3 Germany  TRUE
4   China FALSE
5   Yemen FALSE

> anti_join(md_dem, md_war)
  country   dem
1   Haiti FALSE
```

A simple vista, parece que `semi_join()` y `anti_join()` no tengan demasiada utilidad práctica, pero la realidad es más bien lo contrario. La función `semi_join()` va bien para conservar unos datos que cumplan unos requisitos determinados. La función `anti_join()` suele ser muy útil para diagnosticar problemas. Supongamos que tenemos dos marcos de datos que queremos unir, pero que hay filas que no nos une correctamente. Seguramente esto se debe a que estos marcos de datos no son exactamente iguales. Las filas en cuestión serán fáciles de identificar con un *anti join*.

Ejemplo con *semi_join*

Paquetes de datos como el de Eurostat dan la posibilidad, por un lado, de crear marcos de datos con los datos que queremos y, por el otro también permiten crear marcos de datos con una relación de países según la organización a la que pertenecen. Si, por ejemplo,

solo queremos datos de la EFTA, podemos primero generar unos datos determinados y a continuación hacer un *semi_join* con el marco de datos de la EFTA. Así nos preservará solo los datos de estos países.

3.2. Combinar marcos de datos

La agrupación de marcos de datos puede seguir una lógica diferente a la que hemos explicado hasta ahora. Pensemos en este supuesto: un grupo de dos investigadores ha pasado una encuesta a diferentes estudiantes universitarios. Los dos han hecho las mismas preguntas, pero como no se han coordinado a la hora de pasar la encuesta, saben que puede haber estudiantes que hayan respondido tanto en una encuesta como en la otra. Cada investigador se ha encargado de construir el marco de datos desde su ordenador y ahora quieren combinar los marcos. En este caso tenemos marcos de datos con las mismas variables, pero con filas que pueden estar repetidas. Veamos los marcos de datos `encuesta1` y `encuesta2`:

```
> encuesta1
  nombre              edad  P1  P2  P3  P4
1 Josep Claramunt       21   8   3   9   5
2 Josefina Curtiella    24   4   5   4   4
3 Andreu López          23   9   1   3   2
4 Matilde Llauradó      21  10   6   8   7
5 Rafa Sopena           19   1   5   3   4

> encuesta2
  nombre              edad  P1  P2  P3  P4
1 Anna Abelló           23   4   5   6   3
2 Lluís Busquets        22   5   6   7   1
3 Josep Claramunt       21   8   3   9   5
4 Albert Cusidó         23   9   1   3   2
5 Jofre Monés           20   7   6   8   4
```

La sintaxis para combinar estos dos marcos de datos en uno solo es muy sencilla, puesto que únicamente tenemos que introducir la función y el nombre de los dos marcos de datos como argumentos. La función `union()` nos agrupa las filas de los dos marcos de datos y, en caso de que haya filas idénticas, dejará solo una. La función `intersect()` nos ayuda a encontrar las filas que aparecen en los dos marcos de datos, mientras que la función `setdiff()` nos devuelve los valores que aparecen en el primer marco de datos pero no en el segundo.[15] A continuación veremos el marco de datos que devuelve R en cada una de las operaciones:

[15] Siguiendo el álgebra booleana, la primera función equivaldría a OR, la segunda a AND y la tercera a OR menos NOT: `(encuesta1 | encuesta2) - (encuesta1 & encuesta2)`.

```
> union(encuesta1, encuesta2)
  nombre              edad  P1  P2  P3  P4
1 Anna Abelló           23   4   5   6   3
2 Lluís Busquets        22   5   6   7   1
3 Josep Claramunt       21   8   3   9   5
```

```
4 Josefina Curtiella    24   4   5   4   4
5 Andreu López          23   9   1   3   2
6 Matilde Llauradó      21  10   6   8   7
7 Rafa Sopena           19   1   5   3   4

> intersect(encuesta1, encuesta2)
  nombre              edad  P1  P2  P3  P4
1 Josep Claramunt       21   8   3   9   5

> setdiff(encuesta1, encuesta2)
  nombre              edad  P1  P2  P3  P4
1 Josefina Curtiella    24   4   5   4   4
2 Andreu López          23   9   1   3   2
3 Matilde Llauradó      21  10   6   8   7
4 Rafa Sopena           19   1   5   3   4
```

Una segunda opción para combinar marcos de datos es con las funciones del paquete *dplyr* `bind_rows()` y `bind_cols()`, que permiten ligar marcos de datos que contienen la misma cantidad de filas o la misma cantidad de columnas. La sintaxis es simple y solo se trata de poner el primer marco de datos como primer argumento y el segundo marco de datos como segundo argumento. La función `bind_rows()` tiene un tercer argumento muy útil, `.id`, que crea una columna nueva que permite identificar el marco de donde provienen los datos. En el ejemplo siguiente, hemos pedido que cree una nueva variable que se llame encuesta. En esta variable, los valores tomarán el nombre de Encuesta 1 si provienen del marco de datos *encuesta1* y el nombre de Encuesta 2 si provienen del marco de datos *encuesta2*.

```
> bind_rows(Encuesta 1 = encuesta1, Encuesta 2 = encuesta2, .id = "encuesta")
   encuesta    nombre              edad  P1  P2  P3  P4
1  Encuesta 1  Josep Claramunt       21   8   3   9   5
2  Encuesta 1  Josefina Curtiella    24   4   5   4   4
3  Encuesta 1  Andreu López          23   9   1   3   2
4  Encuesta 1  Matilde Llauradó      21  10   6   8   7
5  Encuesta 1  Rafa Sopena           19   1   5   3   4
6  Encuesta 2  Anna Abelló           23   4   5   6   3
7  Encuesta 2  Lluís Busquets        22   5   6   7   1
8  Encuesta 2  Josep Claramunt       21   8   3   9   5
9  Encuesta 2  Albert Cusidó         23   9   1   3   2
10 Encuesta 2  Jofre Monés           20   7   6   8   4
```

En el paquete de base de R, `rbind()` y `cbind()` hacen una tarea muy parecida a las funciones que acabamos de ver. Pero estas funciones son más lentas y no devuelven un *tibble*.

Resumen

Este es un módulo más bien complementario para el aprendizaje de RStudio, puesto que no muestra directamente cómo analizar datos en el sentido de visualizar variables o transformar información para generar estadísticos descriptivos útiles. Sin embargo, los tres apartados de estas páginas son imprescindibles para dominar el análisis de datos de manera autónoma. Principalmente, el contenido de este módulo nos permite descargar cualquier base de datos disponible en organizaciones y centros de estudios internacionales y preparar los datos para generar el contenido que queremos.

Es por eso por lo que, si queremos analizar datos de manera autónoma, tendremos que dominar con agilidad los tres pasos que hemos aprendido en este módulo:

1) La importación de marcos de datos nos permite incorporar a R cualquier base de datos, esté en el formato que esté, para poder trabajar.

2) La limpieza de marcos de datos nos permite preparar los datos para su análisis, un paso muy importante cuando no obtenemos los datos de fuentes oficiales y que, por lo tanto, no han sido preparados previamente. En este módulo hemos aprendido a tomar decisiones importantes sobre qué estructura tienen que tener los datos, cómo tenemos que tipificar las variables para poderlas trabajar con más comodidad y qué tenemos que hacer con los valores perdidos y los casos extremos.

3) Finalmente, uno de los grandes valores añadidos de programas como R es la facilidad con la que pueden unir diferentes marcos de datos que proceden de fuentes diferentes. Este proceso es sumamente laborioso con otros programas y, en cambio, con R es posible con una sencilla línea de código.

Ejercicios de autoevaluación

Para un mejor aprendizaje, intenta hacer mentalmente el máximo de ejercicios posible sin utilizar R.

1. Crea el marco de datos `const_pol` pidiendo la variable siguiente de Polity IV.

```
exconst
```

2. Busca la palabra siguiente en la base de datos de WDI y pide los siete primeros resultados.

```
women
```

3. Elige la mejor función para descargar el archivo siguiente en formato europeo.

```
preguntasexamen.csv
```

4. Descarga la segunda hoja de este documento. Ten en cuenta que no hay una fila con los nombres de columna.

```
solucionesPEC.xls
```

5. Transforma en columnas los valores de la primera variable con parámetros de la segunda variable.

```
wbtrade$regions, wbtrade$gdp
```

6. El marco de datos siguiente contiene la variable *date* con los meses y los años separados por un guion bajo. Separa las variables.

```
wbtrade
```

7. Convierte los nombres de columna del marco de datos siguiente en mayúscula.

```
un_votes
```

8. Detecta qué valor de la variable *age* puede ser un error.

```
35, 40, NA, 42, 15, 17, 49, 65, 37, -2, 47, 21
```

9. Une de manera exclusiva los marcos de datos siguientes.

```
un_votes / un_roll_call_issues
```

10. Queremos juntar los marcos de datos siguientes por la columna *country* de manera que nos filtre las filas del primer marco de datos según los valores del segundo.

```
europe, efta
```

Solucionario

1. `const_pol <- PolityGet(vars = "exconst")`

2. `WDIsearch("women")[1:7,]`

3. `read.csv2("preguntasexamen.csv")`

4. `read_excel("solucionesPEC.xls", sheet = 2, col_names = FALSE)`

5. `spread(wbtrade, regions, gdp)`

6. `separate(wbtrade, date, c("month", "year"), sep = "_")`

7. `toupper(names(un_votes))`

8. `-2`

9. `inner_join(un_votes, un_roll_call_issues)`

10. `semi_join(europe, efta)`

Bibliografía

Lohr, S. (2014). «For Big-Data Scientists, 'Janitor Work' Is Key Hurdle to Insights». *The New York Times*. Disponible en línea .

Stone, R. W. (2008). *The Scope of IMF Conditionality*. *International Organization* (vol. 62, n. 4, págs. 589–620).

Wickham, H. (2014). *Tidy Data*. *Journal of Statistical Software* (vol. 50, n. 10, págs. 1-23).

Anexo del módulo

Código del apartado 2.1.1

```
tabla_sucia <- data.frame(pais = c("España", "Francia", "Italia", "México", "Japón"),
'2016' = c(64, 43, 56, 35, 12), '2017' = c(83, 34, 53, 101, 19), '2018' = c(95, 33, 67, 120, 18))
```

Código del apartado 3.1

```
md_dem <- data.frame(country = c("France", "Spain", "Germany",
                                 "China", "Yemen", "Haiti"),
                     dem = c(TRUE, TRUE, TRUE, FALSE, FALSE, FALSE))

md_war <- data.frame(country = c("France", "Spain", "Germany",
                                 "China", "Yemen", "Congo"),
                     war = c(21, 13, 9, 24, 14, 42))
```

Código del apartado 3.2

```
encuesta1 <- tribble(~nombre, ~edad, ~P1, ~P2, ~P3, ~P4,
                     "Josep Claramunt", 21, 8, 3, 9, 5,
                     "Josefina Curtiella", 24, 4, 5,  4, 4,
                     "Andreu López", 23, 9, 1, 3, 2,
                     "Matilde Llauradó", 21, 10, 6, 8, 7,
                     "Rafa Sopena", 19, 1, 5, 3, 4)

encuesta2 <- tribble(~nombre, ~edad, ~P1, ~P2, ~P3, ~P4,
                     "Anna Abelló", 23, 4, 5, 6, 3,
                     "Lluís Busquets", 22, 5, 6, 7, 1,
                     "Josep Claramunt", 21, 8, 3, 9, 5,
                     "Albert Cusidó", 23, 9, 1, 3, 2,
                     "Jofre Monés", 20, 7, 6, 8, 4)
```