Herramientas del BIG DATA y analytics

Empezando a transformar bases de datos con R y dplyr

Autor:
Julio César Alonso

Editorial Universidad Icesi
UNIVERSIDAD ICESI

Empezando a transformar bases de datos con R y dplyr

Julio César Alonso[1]

4 de agosto de 2022

[1]CIENFI - Universidad Icesi, jcalonso@icesi.edu.co

Julio César Alonso C.

Colección «Herramientas del Big Data y Analytics», vol. 2

Cali. Universidad Icesi, 2022.

60 páginas.

Incluye referencias bibliográficas.

ISBN: 978-628-7538-79-5 (eBook).

DOI: https://doi.org/10.18046/EUI/bda.h.2

Palabras Clave: 1. R | 2. Analítica | 3. dplyr | 5. Manipulación base de datos | 5. Big Data Analytics

Clasificación Dewey: 545 ddc 21

CIENFI - Centro de Investigación en Economía y Finanzas

www.icesi.edu.co/centros-academicos/cienfi

Rector: Esteban Piedrahita Uribe

Secretaria General: María Cristina Navia Klemperer

Director Académico: José Hernando Bahamón

Coordinador editorial: Adolfo A. Abadía

Corrección de estilo: Claudia L. González G.

Diseño de portada: Sandra Moreno

Fotos tomadas por: Julio César Alonso

Editorial Universidad Icesi

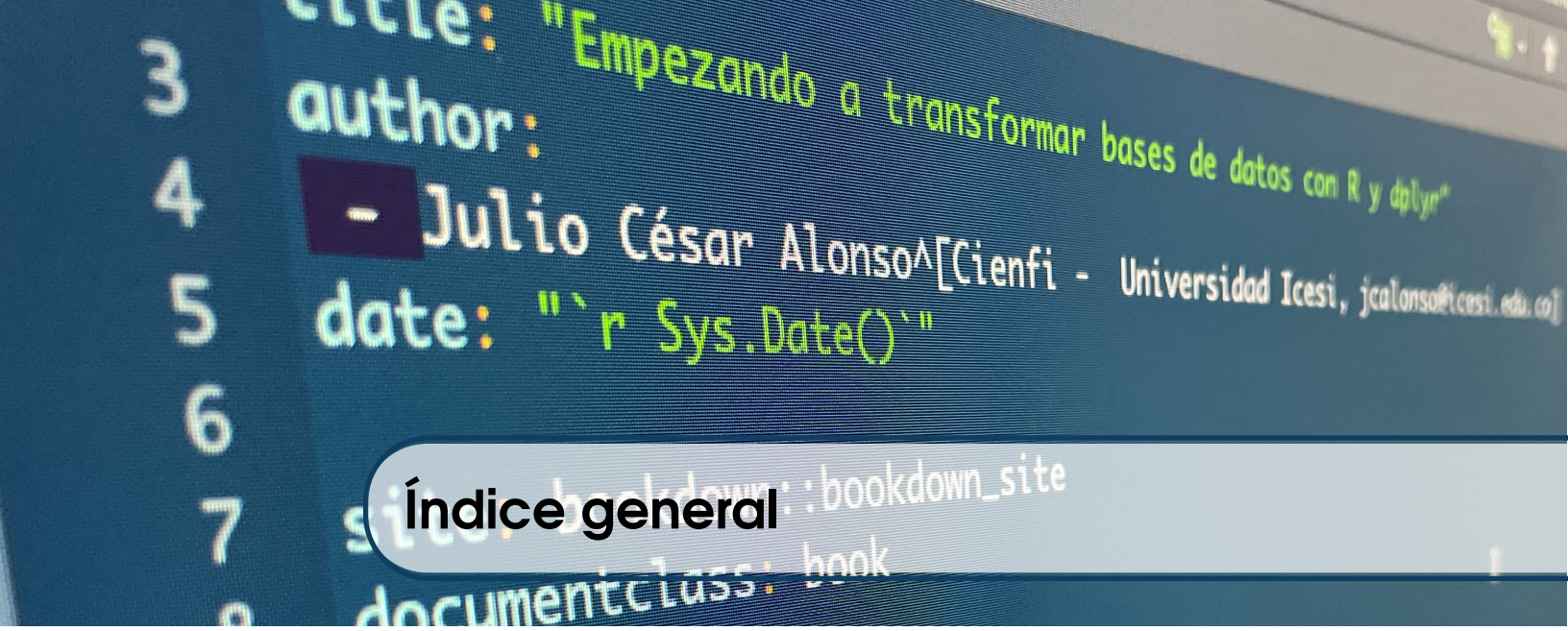
Calle 18 No. 122-135 (Pance), Cali – Colombia

Teléfono: +57 (2) 555 2334 | E-mail: editorial@icesi.edu.co

http://www.icesi.edu.co/editorial

Publicado en Colombia – *Published in Colombia*

La publicación de este libro se aprobó luego de superar un proceso de evaluación doble ciego.



Índice general

Prefacio

Si estás leyendo este libro, ya haces parte de la comunidad que emplea R para analizar datos. Esta obra tiene como objetivo presentar una primera aproximación al paquete *dplyr* (Wickham et al., 2021) que permite manipular rápidamente bases de datos en R. Si eres nuevo en el universo de R o en el uso del paquete *dplyr*, este libro será un buen punto de arranque. Si ya eres usuario de *dplyr*, tal vez este no te aportará nuevos conocimientos, pero será una herramienta de consulta de algunos conceptos básicos.

Esta obra recoge nuestra experiencia trabajando con R y *dplyr* para resolver problemas con datos desde el CIENFI (Centro de Investigación en Economía y Finanzas) de la Universidad Icesi. En el CIENFI, empleamos R para la transformación de datos en conocimiento que facilite la toma de decisiones de organizaciones privadas y públicas. Esta experiencia se plasmará en esta obra para asegurar que nuevas generaciones de profesionales continúen fortaleciendo a la comunidad de R alrededor del mundo.

Este texto presenta las funciones (llamadas "verbos" en este contexto) más comunes para filtrar observaciones, crear variables y unir objetos de datos. El libro presenta ejercicios prácticos que podrás seguir en todo momento. Los archivos con datos empleados aquí los podrás encontrar en la página web del libro en el siguiente enlace: Página del libro.

Este texto es la segundo de una serie de libros introductorios para el uso de R. El primer libro titulado "Empezando a UsaR: Una Guía Paso a Paso" (Alonso Cifuentes y Ocampo, 2022) presenta una breve introducción para iniciar a usar R. Ahí se discute desde cómo instalar R y RStudio y paquetes, hasta cómo cargar diferentes bases de datos. Adicionalmente, el libro discute operaciones aritméticas y lógicas con objetos y las clases esenciales de objetos sencillos y compuestos. No dudes en

consultar ese primer texto si aún no has iniciado tu comino en el universo de R. El tercer libro de esta serie es "Empezando a Visualizar datos con R y Ggplot2" (Alonso y Largo, 2022), en este se presenta una introducción a la construcción de visualizaciones con el paquete *ggplot2* (Wickham, 2016).

¡Espero encuentres esta obra útil y la compartas con otros futuros usuarios interesados! Si tienes alguna sugerencia del libro o corrección, no dudes en escribirme, esta es una obra en constante construcción.

1 . El paquete dplyr

R (R Core Team, 2020) es un lenguaje de programación de uso libre con una gran comunidad a nivel mundial. La base (o también conocido como el *core*) de R es muy potente, permite trabajar con datos grandes y pequeños. El *core* de R permite separar filas (filtrar observaciones), transformar columnas (transformar variables) y unir objetos con datos. Pero estas tareas de filtrar y transformar empleando la base de R, puede ser al principio un poco tedioso. El paquete *dplyr* (Wickham et al., 2021) permite simplificar el proceso de filtrado de observaciones y transformación de variables. Así mismo, este paquete puede ser mas intuitivo para el usuario que apenas está empezando en el universo de R[1].

Antes de entrar en los detalles de cómo filtrar observaciones (ver Capítulo 2), transformar columnas (ver Capítulo 3) o unir objetos con datos (ver Capítulo 4), es importante entender la lógica que sigue el código que emplean las funciones del paquete *dplyr*. En este capítulo nos concentraremos en las generalidades de la filosofía, gramática y estructura detrás del paquete.

[1] Algunas de las funciones de las primeras versiones del paquete *dplyr* presentaban conflicto con las funciones de otros paquetes y de la misma base de R. En las siguientes versiones estos problemas se han solucionado y el nombre de las funciones han sido modificadas para evitar estos inconvenientes. En la actualidad el paquete *dplyr* es muy estable y tiene muy pocos problemas documentados.

1.1 El Universo *tidyverse*

El paquete *dplyr* hace parte de un conjunto de paquetes que se conocen como *tidyverse*[2] . Los cuales fueron diseñados para facilitar operaciones comunes de la ciencia de datos y permitir, de una manera ordenada[3], un flujo de trabajo entre las diferentes tareas.

Los paquetes que hacen parte de *tidyverse* tienen funciones que permiten la gestión, el ordenamiento, la lectura (y escritura), el análisis sintáctico y la visualización de datos, entre muchas otras tareas[4]. Los paquetes son:

- *dplyr*: proporciona una gramática de manipulación de datos. Como lo discutiremos en este libro, el paquete emplea un conjunto de "verbos" que facilitan la manipulación de datos.
- *ggplot2* (Wickham, 2016): crea gráficos siguiendo un gramática de capas. En Alonso y Largo (2022) encontrarás una breve introducción a este paquete.
- *tidyr* (Wickham, 2021): proporciona un conjunto de funciones para transformar la organización de un objeto con datos de clase **data.frame** o **tibble**.
- *readr* (Wickham et al., 2018): lee datos de archivos planos de numerosos formatos de manera mas amigable y rápida que la base de R. No solo lee archivos en formato `.csv` (valores separado por comas, en inglés *comma-separated values*), sino qué también archivos en formato `.tsv` (valores separados por tabuladores, en inglés *tab-separated values*) o `.fwf` (de ancho fijo, en inglés *fixed-width files*). Estos formatos se emplean típicamente para almacenar grandes volúmenes de información.
- *purrr* (Henry y Wickham, 2020): brinda herramientas que facilitan la programación con funciones y vectores, estas permiten eliminar los bucles (también conocidos como *loops*[5]).
- *tibble* (Müller y Wickham, 2021): brinda una alternativa para guardar las bases de datos que complementa la clase **data.frame**, creando lo que se conoce como *tibbles* (lo discutiremos mas adelante en la sección 1.2.2).
- *stringr* (Wickham, 2019): proporciona funciones que facilitan el trabajo con cadenas de carácteres (texto).
- *forcats* (Wickham, 2020): proporciona funciones para resolver problemas que se presentan con la manipulación de variables de cla-

[2]De hecho, *tidyverse* (Wickham et al., 2019) es un paquete que a su vez contiene 8 paquetes.

[3]Nota que *tidy* significa en español ordenado y *verse* es la parte final de *universe* (universo) de ahí que también se emplee la expresión universo *Tidyverse*.

[4]Si deseas conocer con mayor detalle el cómo fue diseñado este universo puedes consultar Wickham y Grolemund (2016) o la versión en linea del libro en el siguiente enlace https://r4ds.had.co.nz.

[5]Si quieres conocer sobre los *loops* en R, puedes consultar Alonso Cifuentes (2021) .

se **factor**.

En la siguiente sección exploraremos la sintaxis básica para trabajar con estos paquetes.

1.2 Consideraciones básicas de la sintaxis en el *tidyverse*

El paquete *dplyr*, como parte del universo *tidyverse*, tiene dos características importantes que estaremos empleando. Por un lado, emplea el operador `pipe` (en español sería tubería) y objetos con datos de la clase **tibble**. Veamos cada uno de estos elementos.

1.2.1 El operador `pipe`

Los paquetes del universo *tidyverse* tienen la peculiaridad de que permiten poner en una "tubería" un objeto con datos y ejecutar diferentes operaciones sobre las observaciones (filas) o variables (columnas), sin tener que guardar los resultados intermedios. El operador `pipe` (`%>%`) permite encadenar resultados rápidamente, de manera que se pueden ejecutar varios comandos de manera fácil e intuitiva.

Veamos un ejemplo: supongamos que se desea sacar la raíz cuadrada de un número, luego al resultado calcularle el logaritmo natural, y finalmente, redondear el resultado para obtener una solución con dos decimales. Esto implica emplear tres funciones: **sqrt()** , **log()** y **round()**. Usando la base de R podemos hacer esto de numerosas maneras, una sería la siguiente:

```
r1 <- sqrt(65)
r2 <- log(r1)
round(r2, digits = 2)
```

```
## [1] 2.09
```

Otra manera de hacer lo mismo, sin guardar un objeto intermedio, sería la que se muestra en la siguiente línea:

```
round(log(sqrt(65)), digits = 2)
```

```
## [1] 2.09
```

Observa que el segundo caso es ideal para no guardar los resultados intermedios en memoria (en especial si hablamos de cálculos en el

mundo del *Big Data*). Pero si quisiéramos encadenar muchos comandos, esto se volvería engorroso y el código sería difícil de seguir.

Aquí es cuando el operador `pipe` (%>%) de los paquetes del *tidyverse* pueden facilitar las tareas. El operador %>% pasa el resultado del último cálculo al primer argumento de la siguiente función, así no es necesario reescribirlo o guardarlo. Es como si los datos entrasen a una tubería (`pipe`) de un proceso de producción y al final solo recibiéramos el producto terminado.

Para el anterior ejemplo, el código empleando el operador %>% de estas librerías sería el siguiente:

```
# cargando la librería
# (recuerda instalarla si no lo has hecho antes)
# installpackages("dplyr")
library(dplyr)

# operación usando el operador pipe
# entran los datos al pipe
65 %>%
  # se calcula la raíz cuadrada
  sqrt() %>%
  # se calcula el log
  log() %>%
  # se redondea
  round( digits = 2)
```

```
## [1] 2.09
```

Nota que es necesario cargar el paquete *dplyr* o cualquiera del conjunto que hace parte de *tidyverse* para usar el operador %>%. Este ejemplo nos permite ver además que el resultado de la operación anterior se reemplaza en el primer argumento de la siguiente función, y los otros argumentos sí debemos especificarlos. También es importante anotar que no todas las funciones son "amigables" al uso del operador %>%, pero afortunadamente todas las funciones del *core* en R sí lo son, y un buen número de funciones en diferentes paquetes son escritas de manera que se puede emplear este operador.

Este pequeño ejemplo muestra que encadenar comandos en R, solo usando el paquete base, puede ser bastante complicado. El operador %>% del *tidyverse* facilita la vida para realizar estos encadenamientos.

Recomendación de Estilo

El operador %>% debe tener siempre un espacio por delante y la siguiente operación deberá ir en una línea nueva. Después del primer paso, cada línea debe tener sangría de dos espacios. Esta estructura facilita la lectura del código, la adición de nuevos pasos, la reorganización de los ya existentes y deja en claro cada uno de los pasos al interior de la tubería.

```
# Buena práctica
65 %>%
  sqrt() %>%
  log() %>%
  round(,digits = 2)

# Mala práctica
65 %>% sqrt() %>%   log() %>% round(,digits = 2)
```

Recomendación de Estilo

No tiene sentido hacer tuberías de un solo paso.

```
# Buena práctica
  sqrt(65)

# Mala práctica
65 %>%
  sqrt()
```

Antes de continuar, es importante conocer un **truco de RStudio**[6] **para digitar el operador `pipe` (%>%)**. Este operador se puede generar en la consola o en un *script* presionando las teclas `command` + `shift` + `M` al mismo tiempo. ¡Inténtalo!, esto te ahorrará mucho tiempo.

1.2.2 La clase tibble

La clase **tibble** se puede entender como una clase de objeto similar al **data.frame** para almacenar bases de datos. En ambos casos, las columnas representan variables y las filas observaciones. Las diferencias

[6] RStudio es una interfaz gráfica de usuario que permite trabajar con mayor comodidad en R. Para una discusión de cómo instalar y emplear RStudio puedes consultar Alonso Cifuentes y Ocampo (2022).

principales entre estas dos clases de objetos son:

- Un objeto **tibble** pueden tener números y símbolos en los nombres de las variables (columnas), mientras que el **data.frame** típicamente no lo permite.
- Cuando se llama un objeto **tibble** en la consola, el resultado que se obtiene solo son las 10 primeras filas y todas las columnas que caben en la consola. En el caso de un objeto **data.frame**, se presentan en la consola todos los datos, o si usas RStudio, se presentan hasta 1000 líneas en la consola. Si esta es muy grande, serán difícil de visualizar los datos. Adicionalmente, sólo en el caso del objeto **tible**, además de su nombre, cada columna informa su clase (tal como lo haría la función **str()**).
- Al extraer un subconjunto de variables de un objeto **tibble**, siempre se obtendrá otro **tibble**. Para un objeto de clase **data.frame** al extraer un subconjunto de variables, se podrá obtener un **data.frame** o a veces un vector.
- Un objeto de clase **tibble** puede tener una columna (variable) de clase **list**, en un **data.frame** no es posible.

Trabajemos con un objeto **tible**. Recuerda que la mayoría de los paquetes traen bases de datos (ver Capítulo 6 de Alonso Cifuentes y Ocampo (2022)) y algunas de esa bases de datos ya están en formato **tibble**. Por ejemplo, el paquete *gapminder* (Bryan, 2017) tiene un objeto con datos con el mismo nombre. El objeto `gapminder` tiene datos para la mayoría de los países del mundo de población, PIB per cápita y esperanza de vida. Veamos rápidamente las características que enunciamos arriba con esta base.

Carga el paquete y los datos y asegúrate que el objeto sea de clase **tibble**.

```
# cargar paquete
# install.packages("gapminder")
library(gapminder)
# cargar datos
data("gapminder")
# clase
class(gapminder)
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

Nota que el objeto **tibble** se reporta en la consola como **tbl_df** y también como **data.frame**. Veamos qué ocurre cuando escribimos el nombre del objeto en la consola.

```
gapminder
```

```
## # A tibble: 1,704 x 6
##    country     continent  year lifeExp      pop gdpPercap
##    <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
##  1 Afghanistan Asia       1952    28.8  8425333      779.
##  2 Afghanistan Asia       1957    30.3  9240934      821.
##  3 Afghanistan Asia       1962    32.0 10267083      853.
##  4 Afghanistan Asia       1967    34.0 11537966      836.
##  5 Afghanistan Asia       1972    36.1 13079460      740.
##  6 Afghanistan Asia       1977    38.4 14880372      786.
##  7 Afghanistan Asia       1982    39.9 12881816      978.
##  8 Afghanistan Asia       1987    40.8 13867957      852.
##  9 Afghanistan Asia       1992    41.7 16317921      649.
## 10 Afghanistan Asia       1997    41.8 22227415      635.
## # ... with 1,694 more rows
```

De hecho, es muy fácil convertir un objeto de clase **tibble** a **data.frame** y viceversa. Por ejemplo, podemos emplear a función **as.data.frame()** del *core* de R para convertir el objeto `gapminder` a **data.frame**.

```
# convertir a data.frame
gapminder.df2 <- as.data.frame(gapminder)
# clase del objeto
class(gapminder.df2)
```

```
## [1] "data.frame"
```

Ahora invoca en la consola el objeto `gapminder.df2` y verás la diferencia.

Si quieres convertir un objeto de clase **data.frame** a **tibble** puedes emplear la a función **as_tibble()** del paquete **tibble**.

Todas las funciones de *dplyr* (y también de *tidyverse*) que generan bases de datos, lo harán en formato **tibble**. Así mismo, las funciones que veremos en los siguientes capítulos para manipular bases de datos pueden tomar como argumento, tanto objetos de clase **tibble** como de clase **data.frame**.

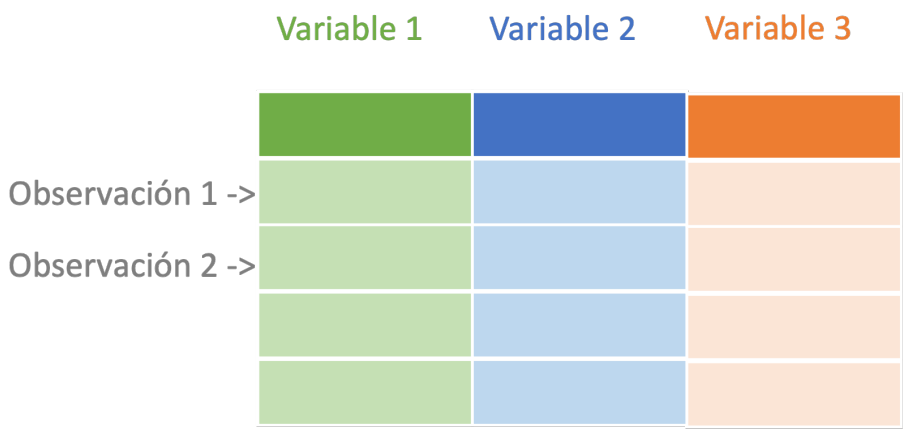
1.3 Comentarios finales

En esta obra veremos una breve introducción a la gramática que emplea el paquete *dplyr* del *tidyverse*. Este paquete emplea diferentes

funciones (que en el universo *tidyverse* se conocen como verbos) para trabajar con observaciones (filtrar) (Capítulo 2), transformar variables (Capítulo 3) y fusionar bases de datos (Capítulo 4).

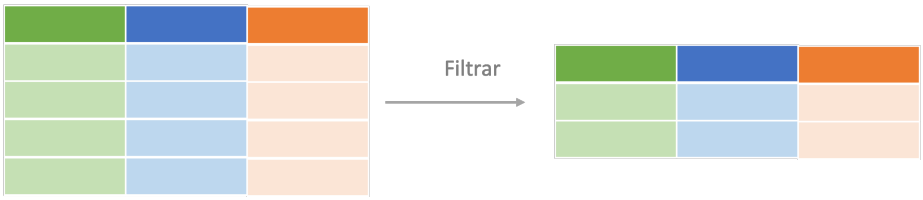
Pero, antes de pasar al siguiente capítulo miremos la representación gráfica de un objeto con datos (ya sea de la clase **tibble** o **data.frame**) que emplearemos a lo largo del libro y que es común en el universo *tidyverse*. En la Figura 1.1 se representa un objeto **tibble** o **data.frame**, donde cada columna corresponde a una variable, y cada fila a una observación (individuo).

Figura 1.1. Representación de un objeto con datos de clase tibble o data.frame

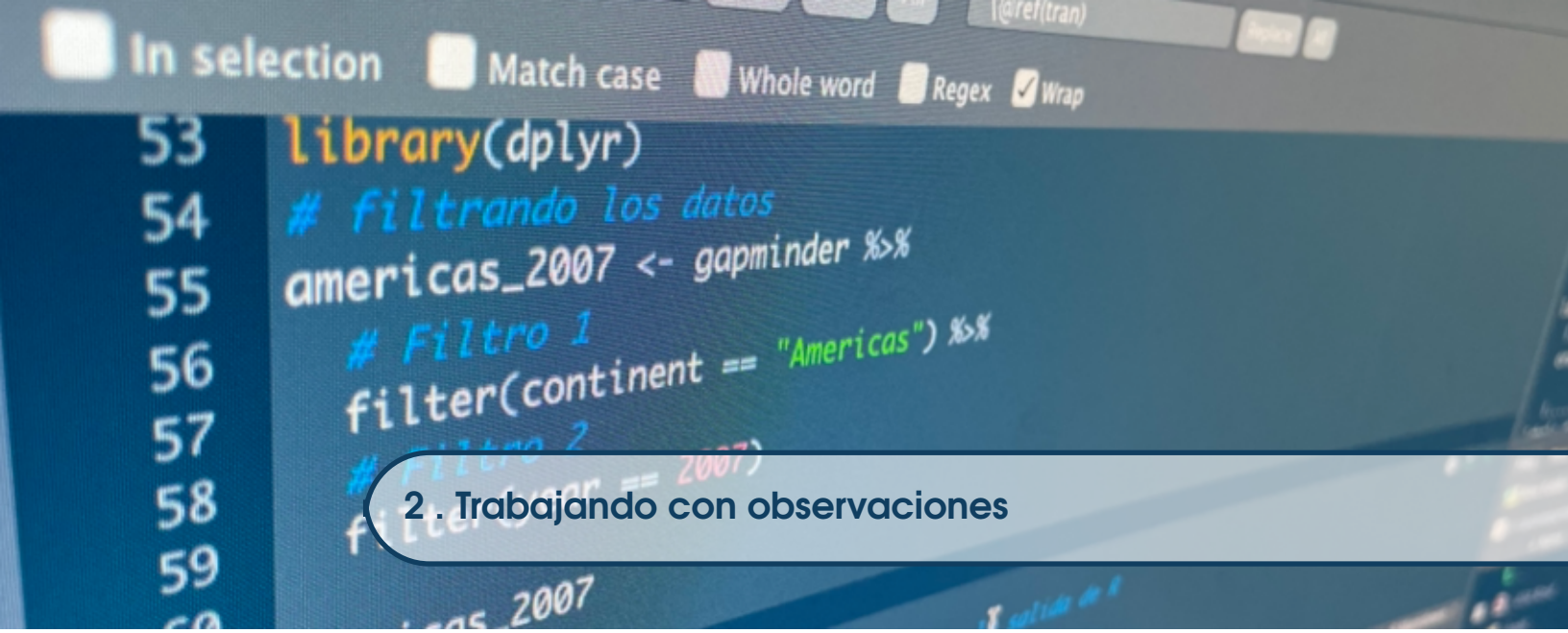


Con este tipo de figura podemos representar qué ocurre cuando filtramos un objeto con datos (ver Figura 1.2). En ese caso estamos seleccionando unas observaciones de la base original.

Figura 1.2. Representación del proceso de filtrado de un objeto de clase tibble o data.frame



En el siguiente capítulo veremos como filtrar objetos con datos de clase **data.frame** y **tibble** empleando el paquete *dplyr*.



2 . Trabajando con observaciones

En el flujo de trabajo de cualquier científico de datos, o persona que quiera analizar datos, se encuentra la tarea de limpiar[1] y manipular las bases de datos. Como vimos en el Capítulo 1 el paquete *dplyr*, y los otros que hacen parte del paquete **tidyverse** facilitan ese flujo de trabajo. También presentamos el operador `pipe` (%>%) del paquete *dplyr* que permite encadenar diferentes comandos. En este capítulo nos concentraremos en operaciones relacionadas a las filas de un objeto con datos que se encuentre en formato **data.frame** o **tibble** (Ver sección 1.2.2 para una breve introducción a esta clase de objetos).

Antes de entrar en detalle, debemos aclarar dos términos comunes en la comunidad de usuarios de R y *dplyr*: los casos y los verbos. El primer ser refiere a las filas de los datos. Ya sabes que las filas de un objeto de clase **data.frame** o **tibble** corresponden a las observaciones de la base y las columnas a las variables. Otro nombre común para las observaciones es *casos* (*cases* en inglés).

El segundo término común cuando trabajamos en *dplyr* es "verbos". Los *verbos* son las funciones que se emplean para realizar operaciones sobre los objetos que contienen datos. Por ejemplo, emplearemos el verbo **filter()** (filtrar en español) para filtrar los datos.

Ahora sí, entremos en detalle de los tres **verbos** que emplearemos para operar sobre los casos de un objeto con datos.

[1] Si quieres ver una breve introducción a la actividad de limpieza de datos puedes ver el siguiente video: https://youtu.be/V2uXaKTlWv4 .

2.1 Filtrar

En este capítulo nos concentraremos en el cómo ejecutar con *dplyr* tres tareas que tienen que ver con las observaciones. La primera operación es escoger observaciones (filas) que cumplan una condición deseada de un objeto de clase **data.frame** o **tibble**. Es decir, extraeremos un subconjunto de observaciones (casos) que cumplan una condición deseada. A esta acción de escoger observaciones que cumplen un criterio se le conoce en este contexto como filtrar. En la Figura 2.1 puedes observar una representación de esta tarea. En este caso el resultado será un un objeto de clase **tibble** con las mismas variables de la base original, pero con menos observaciones (solo aquellos casos que cumplen la condición deseada). Y siguiendo la lógica de los verbos de este paquete, para hacer esta operación emplearemos la función **filter()**[2].

Figura 2.1. Representación del proceso de filtrar

Para emplear esta función se requieren dos argumentos. El primero es el objeto con los datos, que típicamente se pasa empleando el operador %>%. Y el segundo argumento es la condición que deben cumplir las observaciones que pasarán el filtro.

Veamos un ejemplo utilizando los datos del objeto `gpaminder` del paquete *gapminder* (Bryan, 2017). Este objeto tiene datos para la mayoría de países del mundo. Veamos rápidamente las características de este objeto.

```
# cargar paquete
# install.packages("gapminder")
library(gapminder)
# cargar datos
```

[2]Hay que tener mucho cuidado con esta función, pues tiene el mismo nombre de otra función del paquete base de R. Si en alguna ocasión te das cuenta que no está corriendo la función del paquete *dplyr*, puedes especificar en el código que deseas emplear la función del paquete *dplyr* colocando la siguiente expresión **dplyr::filter()**. Los `::` le informan a R que se empleará la función que está a la derecha de dicho símbolo y el paquete a la izquierda de este. En general, este truco se puede usar para solucionar problemas de compatibilidad de *dplyr* con otros paquetes o la base de R. Esto era más común en las primeras versiones del paquete. Esos problemas se han solucionado cada vez más; por lo tanto es poco probable que surjan inconvenientes.

```
data("gapminder")
# clase
class(gapminder)
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

```
# primeras observaciones
gapminder
```

```
## # A tibble: 1,704 x 6
##    country     continent  year lifeExp      pop gdpPercap
##    <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
##  1 Afghanistan Asia       1952    28.8  8425333      779.
##  2 Afghanistan Asia       1957    30.3  9240934      821.
##  3 Afghanistan Asia       1962    32.0 10267083      853.
##  4 Afghanistan Asia       1967    34.0 11537966      836.
##  5 Afghanistan Asia       1972    36.1 13079460      740.
##  6 Afghanistan Asia       1977    38.4 14880372      786.
##  7 Afghanistan Asia       1982    39.9 12881816      978.
##  8 Afghanistan Asia       1987    40.8 13867957      852.
##  9 Afghanistan Asia       1992    41.7 16317921      649.
## 10 Afghanistan Asia       1997    41.8 22227415      635.
## # ... with 1,694 more rows
```

El objeto `gapminder` contiene datos del país (`country`), continente (`continent`), año (`year`), esperanza de vida (`lifeExp`), población (`pop`) y PIB per cápita (`gdpPercap`). Supongamos que solo queremos trabajar con las observaciones que están en el continente americano y solo nos interesa el año 2007. Para obtener la nueva base deseada (la llamaremos `americas_2007`), debemos aplicar dos filtros:

- **Filtro 1**: el continente que es de clase **factor** debe ser igual a "Americas".
- **Filtro 2**: el año (`year`) que es de clase **integer** debe ser igual a 2007.

Esto lo podemos traducir en el siguiente código:

```
# cargando paquete
library(dplyr)
# filtrando los datos
americas_2007 <- gapminder %>%
  # Filtro 1
  filter(continent == "Americas") %>%
  # Filtro 2
  filter(year == 2007)
```

```
americas_2007
```

```
## # A tibble: 25 x 6
##    country            continent  year lifeExp     pop gdpPercap
##    <fct>              <fct>     <int>   <dbl>   <int>     <dbl>
##  1 Argentina          Americas   2007    75.3  4.03e7    12779.
##  2 Bolivia            Americas   2007    65.6  9.12e6     3822.
##  3 Brazil             Americas   2007    72.4  1.90e8     9066.
##  4 Canada             Americas   2007    80.7  3.34e7    36319.
##  5 Chile              Americas   2007    78.6  1.63e7    13172.
##  6 Colombia           Americas   2007    72.9  4.42e7     7007.
##  7 Costa Rica         Americas   2007    78.8  4.13e6     9645.
##  8 Cuba               Americas   2007    78.3  1.14e7     8948.
##  9 Dominican Republic Americas   2007    72.2  9.32e6     6025.
## 10 Ecuador            Americas   2007    75.0  1.38e7     6873.
## # ... with 15 more rows
```

Es decir, tenemos ahora un conjunto de datos para 25 países que pertenecen al continente americano y con datos solo para el año 2007. Nota que el mismo resultado lo podemos obtener empleando el verbo **filter()** una sola vez. Esto se puede lograr empleando el operador lógico "y" (`&`) para definir la condición en una sola aplicación del verbo. Es decir, la condición sería: continent == "Americas" & year == 2007. Así, siguiendo la segunda Recomendación de Estilo de la sección 1.2.1[3], la siguiente línea de código genera exactamente el mismo resultado.

```
americas_2007 <- filter(gapminder,
                        continent == "Americas" &
                           year == 2007 )

americas_2007
```

```
## # A tibble: 25 x 6
##    country            continent  year lifeExp     pop gdpPercap
##    <fct>              <fct>     <int>   <dbl>   <int>     <dbl>
##  1 Argentina          Americas   2007    75.3  4.03e7    12779.
##  2 Bolivia            Americas   2007    65.6  9.12e6     3822.
##  3 Brazil             Americas   2007    72.4  1.90e8     9066.
##  4 Canada             Americas   2007    80.7  3.34e7    36319.
##  5 Chile              Americas   2007    78.6  1.63e7    13172.
```

[3]Según esa recomendación no tiene sentido hacer tuberías de un solo paso. Por eso no se incluye el operador `%>%` y se especifica el argumento de los datos directamente en la función.

```
##  6 Colombia           Americas   2007    72.9  4.42e7     7007.
##  7 Costa Rica         Americas   2007    78.8  4.13e6     9645.
##  8 Cuba               Americas   2007    78.3  1.14e7     8948.
##  9 Dominican Republic Americas   2007    72.2  9.32e6     6025.
## 10 Ecuador            Americas   2007    75.0  1.38e7     6873.
## # ... with 15 more rows
```

Ahora, supongamos que queremos extraer no solamente los datos del año 2007 para los países del continente americano, sino también los de Europa. En este caso podemos emplear el operador `%in%` y la lista de continentes. Es decir:

```
Euro_ame_2007 <- gapminder %>%
  filter(continent %in% c("Americas", "Europe")) %>%
  filter(year == 2007)

Euro_ame_2007
```

```
## # A tibble: 55 x 6
##    country             continent  year lifeExp    pop gdpPercap
##    <fct>               <fct>     <int>   <dbl>  <int>     <dbl>
##  1 Albania             Europe     2007    76.4 3.60e6     5937.
##  2 Argentina           Americas   2007    75.3 4.03e7    12779.
##  3 Austria             Europe     2007    79.8 8.20e6    36126.
##  4 Belgium             Europe     2007    79.4 1.04e7    33693.
##  5 Bolivia             Americas   2007    65.6 9.12e6     3822.
##  6 Bosnia and Herzego~ Europe     2007    74.9 4.55e6     7446.
##  7 Brazil              Americas   2007    72.4 1.90e8     9066.
##  8 Bulgaria            Europe     2007    73.0 7.32e6    10681.
##  9 Canada              Americas   2007    80.7 3.34e7    36319.
## 10 Chile               Americas   2007    78.6 1.63e7    13172.
## # ... with 45 more rows
```

En el Cuadro 2.1 se presentan algunas de las funciones y operadores lógicos y de relación que se pueden emplear con la función **filter()**. Utilizando dichos operadores y el verbo **filter()** tenemos una poderosa herramienta para manipular cualquier base de datos sin importar su tamaño.

Tabla 2.1. Principales funciones y operadores lógicos y de relación que se pueden emplear con el verbo filter()

Operador	Descripción	Operador	Descripción
<	Menor que	<=	Menor o igual que
>	Mayor que	>=	Mayor o igual que
==	Igual a	!=	No es igual a
!	NO lógico	&	Y lógico
|	ó lógico	%in%	pertenece al vector
is.na()	es un valor perdido	**between(a, b)**	está entre a y b

2.2 Ordenar observaciones

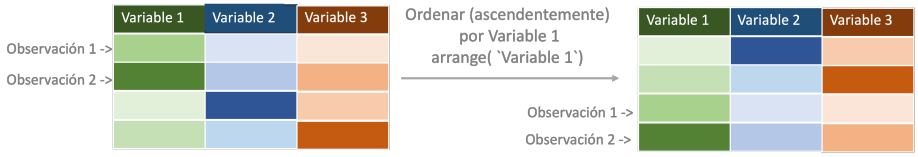
La segunda tarea que estudiaremos es ordenar las observaciones. En algunas ocasiones es deseable ordenar los datos de manera ascendente[4] o descendente con respecto a una o varias variables. Para realizar esto podemos emplear el verbo **arrange()** (organizar en español). Similar al verbo **filter()**, el primer argumento de **arrange()** es el objeto con los datos (clase **data.frame** o **tibble**) que puede pasarse por medio del operador %>%. Y el segundo elemento es la variable que se empleará para organizar de manera ascendente los datos. Si se desea un ordenamiento descendente de los datos, entonces se emplea el verbo **desc()** aplicado a la variable de interés. En la Figura 2.2 se presenta de manera esquemática el procedimiento efectuado cuando empleamos el verbo **arrange** con un objeto con datos.

Veamos un ejemplo. Supongamos que deseamos extraer del objeto `gapminder` un nuevo conjunto de datos que solamente tenga los países de Oceanía y datos para los años disponibles de este siglo. Además queremos tener los datos organizados en orden alfabético por país.

Esto lo podemos hacer con el siguiente código:

[4] Esto implica ordenar de menor a mayor si se trata de una variable de clase **numeric** o **integer**. Si se trata de una variable de clase **character** el orden ascendente implica seguir el orden alfabético (de la A a la Z). Lo mismo ocurre para variables de clase **factor** a las que no se les ha establecido un orden con anterioridad. Si a la variable de clase **factor** se le ha establecido un orden, entonces se empleará dicho orden para organizarla.

Figura 2.2. Representación del proceso de ordenar.




```
oceania_s21 <- gapminder %>%
  filter(continent == "Oceania") %>%
  filter(year > 2000 ) %>%
  arrange(country)

oceania_s21
```

```
## # A tibble: 4 x 6
##   country     continent  year lifeExp      pop gdpPercap
##   <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
## 1 Australia   Oceania    2002    80.4 19546792    30688.
## 2 Australia   Oceania    2007    81.2 20434176    34435.
## 3 New Zealand Oceania    2002    79.1  3908037    23190.
## 4 New Zealand Oceania    2007    80.2  4115771    25185.
```

Ahora supongamos que queremos organizar el año de manera descendente, de tal manera que el último año disponible (para cada país) se presente al principio. En este caso el código sería:

```
oceania_s21 <- gapminder %>%
  filter(continent == "Oceania") %>%
  filter(year > 2000 ) %>%
  arrange(desc(year))

oceania_s21
```

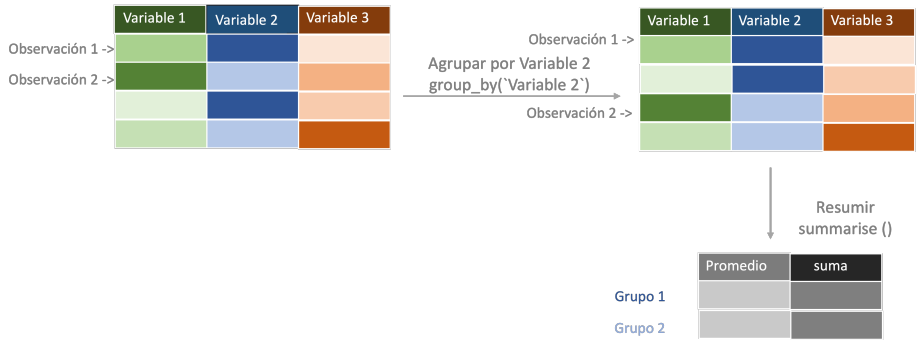
```
## # A tibble: 4 x 6
##   country     continent  year lifeExp      pop gdpPercap
##   <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
## 1 Australia   Oceania    2007    81.2 20434176    34435.
## 2 New Zealand Oceania    2007    80.2  4115771    25185.
## 3 Australia   Oceania    2002    80.4 19546792    30688.
## 4 New Zealand Oceania    2002    79.1  3908037    23190.
```

2.3 Resumir casos

La tercera tarea tiene que ver con resumir las observaciones con una característica común al interior de los datos. Por ejemplo, supongamos que queremos conocer el promedio de la esperanza de vida para un año determinado por continente. Es decir, promediar todas las esperanzas de vida de los países de un mismo continente. O que queremos conocer la población total por continente. En estos dos casos estaríamos creando un nuevo conjunto de datos cuyas filas serían los continentes. En el primer caso como variable tendríamos el promedio calculado por continente, y en el segundo caso una suma calculada por continente.

Para realizar este tipo de operaciones, tenemos que usar dos verbos. Primero agrupamos por (**group_by()**) un criterio (en este caso el continente) y después de tener los grupos resumimos (**summarise()**), ya sea empleando una suma (**sum()**) o un promedio[5] (**mean()**). En la Figura 2.3 puedes observar una representación de este proceso.

Figura 2.3. Representación del proceso de agrupar por y resumir



Ya debe ser clara la gramática que emplean los verbos del paquete *dplyr*. El primer argumento de ambos verbos es el objeto con los datos al que se le quiere realizar la operación, este se puede pasar por medio del operador %>%. Para el verbo **group_by()** el segundo argumento es la variable que se desea emplear para hacer la agrupación. En el caso del verbo **summarise()**[6], el segundo argumento corresponderá a la operación que queremos realizar por grupo.

Por ejemplo, encontremos el promedio de la esperanza de vida al nacer por continente para el año 2007. Esto implicará que primero tenemos que filtrar (**filter()**) los datos para solo tener los datos de 2007, después

[5] La media que es un sinónimo de promedio en inglés es *mean*.
[6] Un verbo sinónimo (hace lo mismo) de **summarise()** es **summarize()**.)

debemos agruparlos (**group_by()**) por continente y finalmente, resumir **summarise()** los datos con la media **mean()** del grupo. Esto en términos de código, se puede traducir de la siguiente manera:

```
EV_continetes <- gapminder %>%
  filter(year ==2007) %>%
  group_by(continent) %>%
  summarise(Exp_Promedio = mean(lifeExp))

EV_continetes
```

```
## # A tibble: 5 x 2
##   continent Exp_Promedio
##   <fct>            <dbl>
## 1 Africa            54.8
## 2 Americas          73.6
## 3 Asia              70.7
## 4 Europe            77.6
## 5 Oceania           80.7
```

Nota que el nuevo objeto `EV_continetes` tiene 5 observaciones (una por cada grupo armado) y una variable, la que decidimos crear. Así mismo, en el verbo **summarise()** hemos especificado el nombre de la nueva variable en la que guardará el resultado de nuestro cálculo.

Naturalmente, podemos emplear otros verbos para organizar los resultados. Por ejemplo, podríamos organizar de manera descendente los resultados:

```
EV_continetes <- gapminder %>%
  filter(year ==2007) %>%
  group_by(continent) %>%
  summarise(Exp_Promedio = mean(lifeExp)) %>%
  arrange(desc(Exp_Promedio))

EV_continetes
```

```
## # A tibble: 5 x 2
##   continent Exp_Promedio
##   <fct>            <dbl>
## 1 Oceania           80.7
## 2 Europe            77.6
## 3 Americas          73.6
## 4 Asia              70.7
## 5 Africa            54.8
```

También podemos crear varias variables en el resumen. Por ejemplo, supongamos que queremos incluir en nuestro nuevo conjunto de datos la población total del continente y el número de países en cada continente (el verbo **n()** cuenta cuántos casos hay por grupo[7]). Esto lo podemos hacer separando con comas las nuevas variables que deseamos crear con el verbo **summarise()**.

```
BD_continetes <- gapminder %>%
  filter(year ==2007) %>%
  group_by(continent) %>%
  summarise(Exp_Promedio = mean(lifeExp),
            Pob_total = sum(pop),
            Paises = n()) %>%
  arrange(desc(Exp_Promedio))

BD_continetes
```

```
## # A tibble: 5 x 4
##   continent Exp_Promedio  Pob_total Paises
##   <fct>            <dbl>      <dbl>  <int>
## 1 Oceania           80.7   24549947      2
## 2 Europe            77.6  586098529     30
## 3 Americas          73.6  898871184     25
## 4 Asia              70.7 3811953827     33
## 5 Africa            54.8  929539692     52
```

[7] Otra forma de contar cuántos casos existen para un posible valor de una variable, y no necesariamente para generar una agrupación, es el verbo **count()** . El primer argumento es, como ya lo intuiste, el objeto con los datos. El segundo argumento es la variable o variables que quieres contar. Adicionalmente, este verbo tiene el argumento lógico **sort** que permite organizar los resultados. Intenta contar los países por continente pasando el objeto `gapminder`, filtrando para un año (`filter(year ==2007)`) y pasando el resultado al verbo contar (`count(continent, sort = TRUE)`).

Recomendación de Estilo

Cuando los argumentos de una función no quepan todos en una línea, coloca cada argumento en una linea y usa la sangría de dos espacios.

```
# Buena práctica
BD_continetes <- gapminder %>%
  filter(year ==2007) %>%
  group_by(continent) %>%
  summarise(Exp_Promedio = mean(lifeExp),
            P_tot = sum(pop),
            Paises = n()) %>%

# Mala práctica
BD_continetes <- gapminder %>%
  filter(year ==2007) %>%
  group_by(continent) %>%
  summarise(Exp_Promedio = mean(lifeExp), P_tot = sum(pop),
Paises = n()) %>%
```

En el Cuadro 2.2 se presentan algunas de las funciones que se pueden emplear con la función **summarise()** para resumir una variable. Empleando estas funciones tenemos una poderosa herramienta para resumir cualquier base de datos sin importar su tamaño.

Tabla 2.2. Principales funciones que se pueden emplear para hacer operaciones sobre una variable con el verbo summarise()

Función	Descripción	Función	Descripción
min()	valor mínimo	**max()**	valor máximo
quantile()	el cuantil deseado	**IQR()**	rango intercuartílico
last()	último valor	**first()**	mayor o igual que
mean()	media	**median()**	mediana
var()	varianza	**sd()**	desviación estándar
n()	número de casos	**n_distinct**	número de caso que no se repiten

En algunas ocasiones no queremos agregar los datos de un grupo con una suma o un promedio, pero queremos encontrar el caso con el valor más grande de una variable o las primeras 10 observaciones con los valores mas altos (top 10) en su respectivo grupo. En este caso empleamos el verbo **top_n()**. El primer argumento son los datos, que se pasan con el operador %>%, el segundo es **n**, el número de valores top que queremos obtener; el último es la variable para la cual vamos a buscar los **n** casos.

Veamos un ejemplo. Supongamos que queremos encontrar los países que en cada continente tienen la esperanza de vida al nacer más grande en 2007. Esto lo podemos hacer de la siguiente manera:

```
gapminder %>%
  filter(year ==2007) %>%
  group_by(continent) %>%
  top_n(1, lifeExp)
```

```
## # A tibble: 5 x 6
## # Groups:   continent [5]
##   country   continent  year lifeExp       pop gdpPercap
##   <fct>     <fct>     <int>   <dbl>     <int>     <dbl>
## 1 Australia Oceania    2007    81.2  20434176    34435.
## 2 Canada    Americas   2007    80.7  33390141    36319.
## 3 Iceland   Europe     2007    81.8    301931    36181.
## 4 Japan     Asia       2007    82.6 127467972    31656.
## 5 Reunion   Africa     2007    76.4    798094     7670.
```

Noten que el verbo **top_n()** extrae la o las observaciones con el mayor valor para la variable deseada (en este caso `lifeExp`), pero el resultado trae todas las variables correspondientes a ese caso (no solo la que se empleó para hacer el filtrado de los datos).

Veamos otro ejemplo. Ahora, busquemos los 5 países con el PIB per cápita mas grande en su respectivo continente en 2007. Y ordenemos el resultado en orden alfabético por continente y al interior del continente por el orden decente del PIB per cápita.

```
gapminder %>%
  filter(year ==2007) %>%
  group_by(continent) %>%
  top_n(5, gdpPercap) %>%
  arrange(continent, desc(gdpPercap))
```

```
## # A tibble: 22 x 6
```

```
## # Groups:   continent [5]
##    country             continent  year lifeExp    pop gdpPercap
##    <fct>               <fct>     <int>   <dbl>  <int>     <dbl>
##  1 Gabon               Africa     2007    56.7 1.45e6    13206.
##  2 Botswana            Africa     2007    50.7 1.64e6    12570.
##  3 Equatorial Guinea   Africa     2007    51.6 5.51e5    12154.
##  4 Libya               Africa     2007    74.0 6.04e6    12057.
##  5 Mauritius           Africa     2007    72.8 1.25e6    10957.
##  6 United States       Americas   2007    78.2 3.01e8    42952.
##  7 Canada              Americas   2007    80.7 3.34e7    36319.
##  8 Puerto Rico         Americas   2007    78.7 3.94e6    19329.
##  9 Trinidad and Tobago Americas   2007    69.8 1.06e6    18009.
## 10 Chile               Americas   2007    78.6 1.63e7    13172.
## # ... with 12 more rows
```

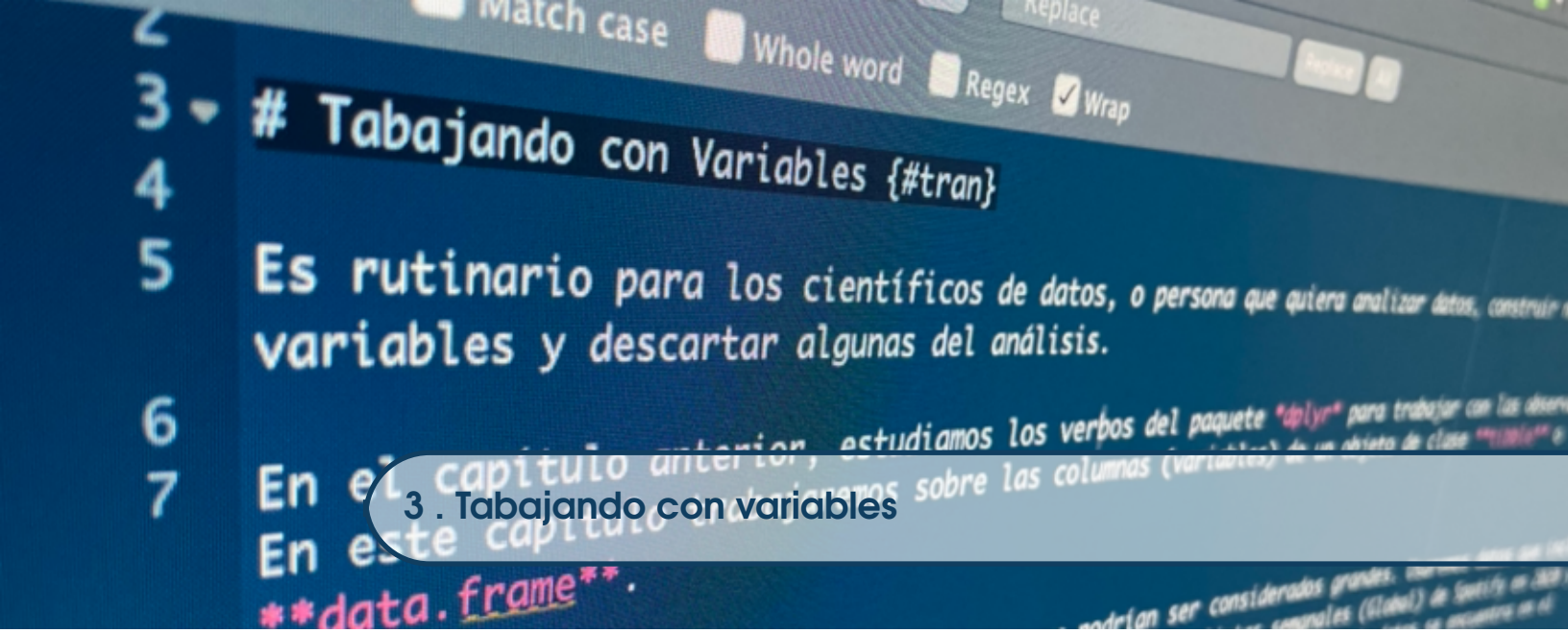
Observa que para Oceanía solo se presentan dos países, pues son los dos únicos disponibles en los datos.

Finalmente, también está disponible el verbo **ungroup()** (desagrupar) que permite quitar las agrupaciones realizadas y retornar en el flujo de trabajo a la base original. Esta función no requiere de argumentos diferentes al objeto con los datos.

2.4 Comentarios finales

En este capítulo estudiamos diferentes verbos del paquete *dplyr* que nos permiten extraer diferentes observaciones, ordenarlas y agruparlas para calcular algunas cantidades que resuman los grupos. Estas nuevas herramientas son muy útiles para hacerle preguntas a los datos. En el Capítulo 3 veremos verbos que nos permitirán manipular variables.

Antes de terminar con la discusión de los verbos asociados a los casos, es importante mencionar que existen muchos más verbos que serán útiles en diferentes ocasiones. Puedes encontrar ayuda rápidamente en la comunidad por medio de los diferentes buscadores. A medida que empieces a usar este paquete y seas más fluido en sus funciones, probablemente te encontrarás en la necesidad de encontrar más verbos para expresarte en tus análisis.



3 . Tabajando con variables

Es rutinario para los científicos de datos, o persona que quiera analizar datos, construir nuevas variables y descartar algunas del análisis.

En el capítulo anterior, estudiamos los verbos del paquete *dplyr* para trabajar con las observaciones. En este capítulo trabajaremos sobre las columnas (variables) de un objeto de clase **tibble** o **data.frame**.

En este capítulo emplearemos unos datos que podrían ser considerados grandes. Usaremos datos que incluyen todas las canciones que han estado en el Top 200 de las listas semanales (Global) de Spotify en 2020 y 2021, y algunas características de las canciones (Pillai, 2021). La base de datos se encuentra en el archivo `spotify_dataset.csv`[1].

La base cuenta con 1556 observaciones con las siguientes 23 variables:

- `Highest.Charting.Position`: la posición más alta en la que ha estado la canción en las listas semanales de Spotify Top 200 Global Charts en 2020 y 2021.
- `Number.of.Times.Charted`: el número de veces que la canción ha estado en la lista de las 200 mejores canciones semanales de Spotify en 2020 y 2021.
- `Week.of.Highest.Charting`: la semana en la cual la canción tuvo la posición más alta en las listas globales semanales de las 200 mejores canciones de Spotify en 2020 y 2021.
- `Song.Name`: el nombre de la canción que ha estado en las Top 200 Weekly Global Charts de Spotify en 2020 & 2021.
- `Song.ID`: el ID de la canción proporcionado por Spotify (único para cada canción).

[1] El archivo lo puedes descargar de la página web del libro (http://www.icesi.edu.co/editorial/empezando-transformar).

- `Streams`: el número aproximado de streams que tiene la canción.
- `Artist`: el artista o artistas principales que han participado en la elaboración de la canción.
- `Artist.Followers`: el número de seguidores que tiene el artista principal en Spotify.
- `Genre`: los géneros a los que pertenece la canción.
- `Release.Date`: la fecha inicial de lanzamiento de la canción.
- `Weeks.Charted`: las semanas que la canción ha estado en el Top 200 de las listas semanales de Spotify en 2020 y 2021.
- `Popularity`: la popularidad de la canción. El valor estará entre 0 y 100, siendo 100 la más popular.
- `Danceability`: la bailabilidad describe lo adecuado que es un tema para bailar basándose en una combinación de elementos musicales que incluyen el tempo, la estabilidad del ritmo, la fuerza del compás y la regularidad general. Un valor de 0,0 es el menos bailable y 1,0 es el más bailable.
- `Acousticness`: una medida de 0,0 a 1,0 de si la pista es acústica.
- `Energy`: la energía es una medida de 0,0 a 1,0 y representa una medida perceptiva de intensidad y actividad. Normalmente, las pistas energéticas se sienten rápidas, fuertes y ruidosas.
- `Liveness`: detecta la presencia de público en la grabación. Los valores más altos de `liveness` representan una mayor probabilidad de que la pista haya sido interpretada en directo.
- `Loudness`: la sonoridad general de una pista en decibelios (dB). Los valores de sonoridad se promedian en toda la pista. Los valores suelen oscilar entre -60 y 0 db.
- `Speechiness`: la expresividad detecta la presencia de palabras habladas en una pista. Cuanto más exclusivamente hablada sea la grabación (por ejemplo, un programa de entrevistas, un audiolibro o una poesía), más se acercará a 1,0 el valor del atributo.
- `Tempo`: el tempo global estimado de una pista en pulsaciones por minuto (BPM). En la terminología musical, el tempo es la velocidad o el ritmo de una pieza determinada y se deriva directamente de la duración media de los tiempos.
- `Duration..ms.`: duración de la canción en milisegundos.
- `Valence`: una medida de 0,0 a 1,0 que describe la positividad musical que transmite una pista. Las pistas con valencia alta suenan más positivas (por ejemplo, felices, alegres, eufóricas), mientras que las pistas con valencia baja suenan más negativas (por ejemplo, tristes, deprimidas, enfadadas).
- `Chord`: el acorde principal de la canción instrumental.

Antes de entrar en la discusión de los verbos carguemos la base de datos en un objeto de clase **tibble**, al cual llamaremos `spotify`.

```
# Carga de bases de datos
spotify <- read.csv("spotify_dataset.csv",
                    header = TRUE, sep = ",")
# mirar la clase del objeto
class(spotify)
```

```
## [1] "data.frame"
```

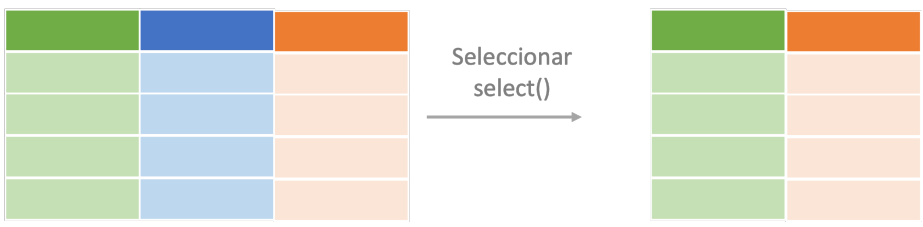
```
# cambiando a clase tibble
library(dplyr)

spotify <- as_tibble(spotify)
```

Asegúrate de que el objeto `spotify` tenga 1556 observaciones y 23 variables. Esto lo puedes hacer con el verbo **glimpse()** del paquete *dplyr*. Este verbo, que en español sería vislumbrar o dar una mirada rápida, se puede utilizar para ver las columnas (variables) del objeto, al tiempo que se muestran los datos que puedan caber en una sola línea. Así mismo, podemos ver la clase de cada variable. ¡Inténtalo!

3.1 Seleccionar

En algunas ocasiones es deseable trabajar con un conjunto de datos que tenga reducido número de variables. Para seleccionar un grupo de variables se puede emplear el verbo **select()**. Siguiendo la gramática del paquete *dplyr*, el primer argumento de la función **select()** es el objeto de clase **tibble** o **data.frame** que se puede pasar empleando el operador %>%. Los siguientes argumentos son las variables que se quieren seleccionar. En la Figura 3.1 se presenta de manera esquemática el procedimiento efectuado cuando se emplea el verbo **select** en un objeto con datos.

Figura 3.1. Representación del proceso de seleccionar (select()) aplicado a un objeto de clase data.frame o tibble



Supongamos que queremos crear un nuevo objeto con datos que solo tenga las variables `Song.Name`, `Genre` y `Streams`. Esto lo podemos hacer con el siguiente código.

```
spotify %>%
  select(Song.Name, Genre, Streams) %>%
  glimpse()
```

```
## Rows: 1,556
## Columns: 3
## $ Song.Name <chr> "Beggin'", "STAY (with Justin Bieber)", "go~
## $ Genre     <chr> "['indie rock italiano', 'italian pop']", "~
## $ Streams   <int> 48633449, 47248719, 40162559, 37799456, 339~
```

Regresando a este ejemplo, es importante anotar que podemos combinar verbos que actúen sobre variables y observaciones en un mismo `pipe`. Por ejemplo, supongamos que queremos construir una base que tenga solo las variables `Song.Name` y `Streams`, pero sólo de aquellas canciones que pertenecen al género `pop`. Esto implicará, primero filtrar el objeto `spotify` por género (`Genre == "['pop']"`), y luego seleccionar las variables deseadas. El siguiente código efectuará estas operaciones:

```
spotify_pop <- spotify %>%
  filter(Genre == "['pop']") %>%
  select(Song.Name, Streams)

# mirando los resultados
glimpse(spotify_pop)
```

```
## Rows: 18
## Columns: 2
## $ Song.Name <chr> "good 4 u", "traitor", "deja vu", "drivers ~
## $ Streams   <int> 40162559, 19480679, 18571755, 15684978, 148~
```

El verbo **select** lo podemos complementar con otros que agilizarán nuestro flujo de trabajo. Por ejemplo, está el verbo **contains()** que permite seleccionar todas las variables que contengan uno o varios carácteres deseados. En particular, supongamos que queremos seleccionar todas las variables que tienen un punto (`.`) en su nombre[2]. Esto lo podemos hacer de la siguiente manera:

[2]Esto no tiene mucho sentido en este contexto, pero el verbo puede ser útil en otros contextos.

```
spotify_punto <- select(spotify, contains("."))
# mirando los resultados
glimpse(spotify_punto)
```

```
## Rows: 1,556
## Columns: 9
## $ Highest.Charting.Position <int> 1, 2, 1, 3, 5, 1, 3, 2, 3, ~
## $ Number.of.Times.Charted   <int> 8, 3, 11, 5, 1, 18, 16, 10,~
## $ Week.of.Highest.Charting  <chr> "2021-07-23--2021-07-30", "~
## $ Song.Name                 <chr> "Beggin'", "STAY (with Just~
## $ Artist.Followers          <int> 3377762, 2230022, 6266514, ~
## $ Song.ID                   <chr> "3Wrjm47oTz2sjIgck11l5e", "~
## $ Release.Date              <chr> "2017-12-08", "2021-07-09",~
## $ Weeks.Charted             <chr> "2021-07-23--2021-07-30\n20~
## $ Duration..ms.             <int> 211560, 141806, 178147, 231~
```

Puedes observar que se obtiene un objeto con las 9 variables cuyos nombres contienen un punto (`.`). En el Cuadro 3.1 se presentan algunas de las funciones que se pueden emplear con el verbo **delect()**.

Tabla 3.1. Principales funciones que se pueden emplear con el verbo select()

Función	Variables seleccionadas
contains()	Con nombre que contiene una cadena de caracteres.
ends_with()	Con nombre que termina con una cadena de caracteres.
starts_with()	Con nombre que inicia con una cadena de caracteres.
last_col()	La última.
everything()	Todas.
num_range()	En un rango de variables que crea la misma función. Por ejemplo, `select(num_range("x", 1:5))` selecciona las variables con nombres `x1`, `x2`, `x3`, `x4` y `x5`.
one_of()	Con nombres en un grupo de nombres. recuerda poner el grupo como una lista empleando **c()**

También podemos emplear operadores como los dos puntos (`:`) con el verbo **select()** para escoger un rango de variables que estén adyacentes. Por ejemplo, para seleccionar las variables que están entre `Acousticness` y `Valence` puede hacerse con el siguiente código que evita digitar el nombre de todas las variables deseadas:

```
spotify %>%
  select(Acousticness:Valence) %>%
  glimpse()
```

```
## Rows: 1,556
## Columns: 5
## $ Acousticness  <dbl> 0.12700, 0.03830, 0.33500, 0.04690, 0.0~
## $ Liveness      <dbl> 0.3590, 0.1030, 0.0849, 0.3640, 0.0501,~
## $ Tempo         <dbl> 134.002, 169.928, 166.928, 126.026, 149~
## $ Duration..ms. <int> 211560, 141806, 178147, 231041, 212000,~
## $ Valence       <dbl> 0.5890, 0.4780, 0.6880, 0.5910, 0.8940,~
```

También es posible emplear el operador signo de exclamación (`!`) para escoger las variables que no correspondan a un conjunto de variables. Por ejemplo, supongamos que queremos excluir las variables que están entre `Song.Name` y `Chord` en el objeto original. Esto lo podemos hacer de la siguiente manera:

```
spotify_red <- select(spotify, !Song.Name:Chord)

# mirando los resultados
glimpse(spotify_red)
```

```
## Rows: 1,556
## Columns: 4
## $ Index                      <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, ~
## $ Highest.Charting.Position <int> 1, 2, 1, 3, 5, 1, 3, 2, 3, ~
## $ Number.of.Times.Charted   <int> 8, 3, 11, 5, 1, 18, 16, 10,~
## $ Week.of.Highest.Charting  <chr> "2021-07-23--2021-07-30", "~
```

El mismo resultado se logra con el siguiente código:

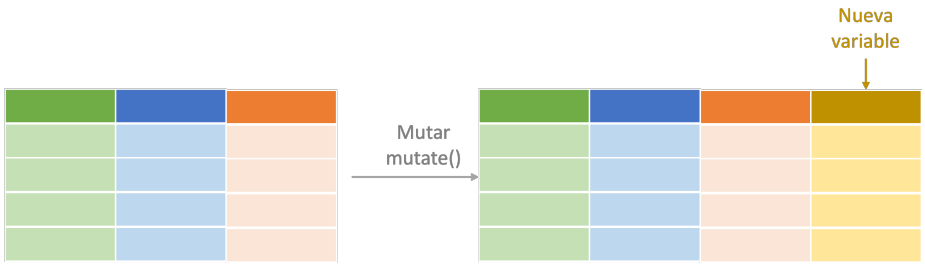
```
spotify_red <- select(spotify, -c(Song.Name:Chord))
```

Esta ultima líneas de código es una forma de expresarnos más parecida a lo que habitualmente se acostumbra en la base de R para no seleccionar columnas de una matriz o **data.frame**.

3.2 Crear variables nuevas (Mutar)

En otras ocasiones, podemos querer crear nuevas variables a partir de operaciones de otras columnas. Para usar la jerga de *dplyr*, necesitamos mutar variables ya existentes en otras. Esto puede hacerse con el verbo **mutate()** . El primer argumento de esta función es el objeto con los datos y los restantes corresponden a las nueva variables que se quieran crear. Las nuevas variables se especifican con su nuevo nombre y la operación que se desea realizar. En la Figura 3.2 se presenta de manera esquemática el procedimiento efectuado cuando empleamos el verbo **mutate**.

Figura 3.2. Representación del proceso de mutar (mutate())



Por ejemplo, supongamos que queremos conocer cuántas reproducciones tiene cada canción por seguidor del artista principal en Spotify. Es decir, la división de la variable `Streams` por `Artist.Followers`. Esto lo podemos hacer rápidamente de la siguiente manera:

```
spotify %>%
  mutate(Rep_follow = Streams / Artist.Followers) %>%
  glimpse()
```

```
## Rows: 1,556
## Columns: 24
## $ Index                     <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, ~
## $ Highest.Charting.Position <int> 1, 2, 1, 3, 5, 1, 3, 2, 3, ~
## $ Number.of.Times.Charted   <int> 8, 3, 11, 5, 1, 18, 16, 10,~
## $ Week.of.Highest.Charting  <chr> "2021-07-23--2021-07-30", "~
## $ Song.Name                 <chr> "Beggin'", "STAY (with Just~
## $ Streams                   <int> 48633449, 47248719, 4016255~
## $ Artist                    <chr> "Måneskin", "The Kid LAROI"~
## $ Artist.Followers          <int> 3377762, 2230022, 6266514, ~
## $ Song.ID                   <chr> "3Wrjm47oTz2sjIgck11l5e", "~
## $ Genre                     <chr> "['indie rock italiano', 'i~
```

```
## $ Release.Date          <chr> "2017-12-08", "2021-07-09",~
## $ Weeks.Charted         <chr> "2021-07-23--2021-07-30\n20~
## $ Popularity            <int> 100, 99, 99, 98, 96, 97, 94~
## $ Danceability          <dbl> 0.714, 0.591, 0.563, 0.808,~
## $ Energy                <dbl> 0.800, 0.764, 0.664, 0.897,~
## $ Loudness              <dbl> -4.808, -5.484, -5.044, -3.~
## $ Speechiness           <dbl> 0.0504, 0.0483, 0.1540, 0.0~
## $ Acousticness          <dbl> 0.12700, 0.03830, 0.33500, ~
## $ Liveness              <dbl> 0.3590, 0.1030, 0.0849, 0.3~
## $ Tempo                 <dbl> 134.002, 169.928, 166.928, ~
## $ Duration..ms.         <int> 211560, 141806, 178147, 231~
## $ Valence               <dbl> 0.5890, 0.4780, 0.6880, 0.5~
## $ Chord                 <chr> "B", "C#/Db", "A", "B", "D#~
## $ Rep_follow            <dbl> 14.3981278, 21.1875573, 6.4~
```

Esto crea un objeto de datos igual al original e incluye la variable nueva `Rep_follow` como última columna.

Con el verbo **mutate** podemos emplear los operadores aritméticos, de relación y lógicos del paquete base de R[3], así como las funciones aritméticas y matemáticas como **log()** y **sqrt()**. Además, se pueden emplear funciones especiales que trae este verbo. En el Cuadro 3.2 se presentan algunas de dichas funciones.

Tabla 3.2. Principales funciones (de dplyr) que se pueden emplear con el verbo mutate()

Función	Operación
lag(n=1)	Trae la observación anterior de la variable. Si se desean dos observaciones atrás, entonces n=2.
lead(n=1)	Trae la siguiente observación de la variable. Si se desean dos observaciones adelante, entonces n=2.
cume_dist()	Calcula la proporción de observaciones menores o iguales a cada uno de los valores de la variable. La distribución acumulada.

[3]Puedes ver el Capítulo 4 de Alonso Cifuentes y Ocampo (2022) para una discusión de estos operadores.

Función	Operación
cumsum()	Calcula la suma acumulada hasta esta observación de todas las observaciones anteriores de la variable.

3.3 Otras operaciones con variables

En algunas oportunidades veremos la necesidad de renombrar variables. El verbo **rename()** permite cambiar el nombre de una o más variables. Después de especificar el objeto que tiene los datos en el primer argumento de la función, solo se necesita especificar el nuevo nombre de la variable y el nombre original.

Veamos un ejemplo: supongamos que queremos cambiarle el nombre a la variable `Artist.Followers` a `Artist_Followers`, dado que es una mala práctica emplear el punto (`.`) en el nombre de una variable. Esto lo podemos hacer con la siguiente línea de código:

```
spotify_rename <- rename(spotify,
                  Artist_Followers = Artist.Followers)
# mirando solo los nombres de las variables
names(spotify_rename)
```

```
##  [1] "Index"                    "Highest.Charting.Position"
##  [3] "Number.of.Times.Charted"  "Week.of.Highest.Charting"
##  [5] "Song.Name"                "Streams"
##  [7] "Artist"                   "Artist_Followers"
##  [9] "Song.ID"                  "Genre"
## [11] "Release.Date"             "Weeks.Charted"
## [13] "Popularity"               "Danceability"
## [15] "Energy"                   "Loudness"
## [17] "Speechiness"              "Acousticness"
## [19] "Liveness"                 "Tempo"
## [21] "Duration..ms."            "Valence"
## [23] "Chord"
```

Otra función relacionada con esta que puede ser útil para automatizar cambios de muchas variables a la vez es **rename_with()** . Esta función permite renombrar las variables de un objeto empleando funciones. Por ejemplo, podemos utilizar la función **toupper()** para pasar todos los nombres de las variables a solo mayúsculas (en inglés *upper case*).

```
spotify_upper1 <- rename_with(spotify, toupper)
# mirando solo los nombres de las variables
names(spotify_upper1)
```

```
##  [1] "INDEX"                      "HIGHEST.CHARTING.POSITION"
##  [3] "NUMBER.OF.TIMES.CHARTED"    "WEEK.OF.HIGHEST.CHARTING"
##  [5] "SONG.NAME"                  "STREAMS"
##  [7] "ARTIST"                     "ARTIST.FOLLOWERS"
##  [9] "SONG.ID"                    "GENRE"
## [11] "RELEASE.DATE"               "WEEKS.CHARTED"
## [13] "POPULARITY"                 "DANCEABILITY"
## [15] "ENERGY"                     "LOUDNESS"
## [17] "SPEECHINESS"                "ACOUSTICNESS"
## [19] "LIVENESS"                   "TEMPO"
## [21] "DURATION..MS."              "VALENCE"
## [23] "CHORD"
```

También podemos jugar un poco más con esta función y las otras que ya conocemos, por ejemplo,

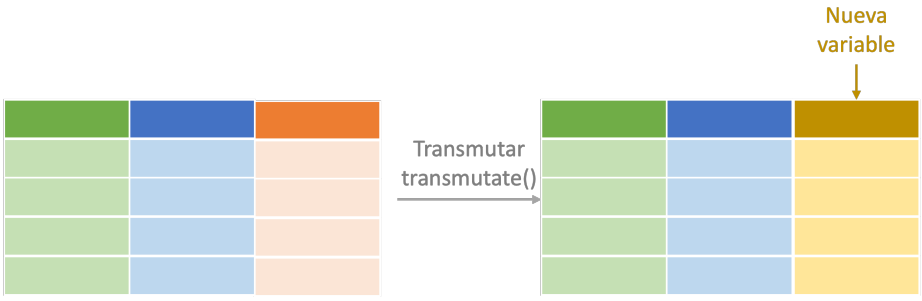
```
spotify_upper2 <- rename_with(spotify, toupper, contains("."))
# mirando solo los nombres de las variables
names(spotify_upper2)
```

```
##  [1] "Index"                      "HIGHEST.CHARTING.POSITION"
##  [3] "NUMBER.OF.TIMES.CHARTED"    "WEEK.OF.HIGHEST.CHARTING"
##  [5] "SONG.NAME"                  "Streams"
##  [7] "Artist"                     "ARTIST.FOLLOWERS"
##  [9] "SONG.ID"                    "Genre"
## [11] "RELEASE.DATE"               "WEEKS.CHARTED"
## [13] "Popularity"                 "Danceability"
## [15] "Energy"                     "Loudness"
## [17] "Speechiness"                "Acousticness"
## [19] "Liveness"                   "Tempo"
## [21] "DURATION..MS."              "Valence"
## [23] "Chord"
```

Nota que solo pasamos a mayúsculas las variables cuyos nombres contienen un punto (`.`). Ahora puedes intentar con diferentes opciones.

Finalmente, consideremos el verbo **transmute()** , este es una combinación de los verbos **select()** y **mutate()**. Es decir, nos permite seleccionar un conjunto de variables al mismo tiempo que se crean otras nuevas. En la Figura 3.3 se presenta de manera esquemática el procedimiento efectuado cuando se empleamos el verbo transmutar.

Figura 3.3. Representación del proceso de transmutar (transmute()).



Supongamos que deseamos crear la variable de reproducciones de cada canción por seguidor del artista principal en Spotify (antes la definimos como `Rep_follow`). Además, queremos una base que solo tenga el nombre de la canción (`Song.Name`), el género (`Genre`), las reproducciones (`Streams`) y la nueva variable creada (`Rep_follow`). Esto lo logramos con el siguiente código:

```
spotify_t <- transmute(spotify, Song.Name,
                       Genre, Streams,
              Rep_follow = Streams / Artist.Followers)

# mirando solo los nombres de las variables
names(spotify_t)
```

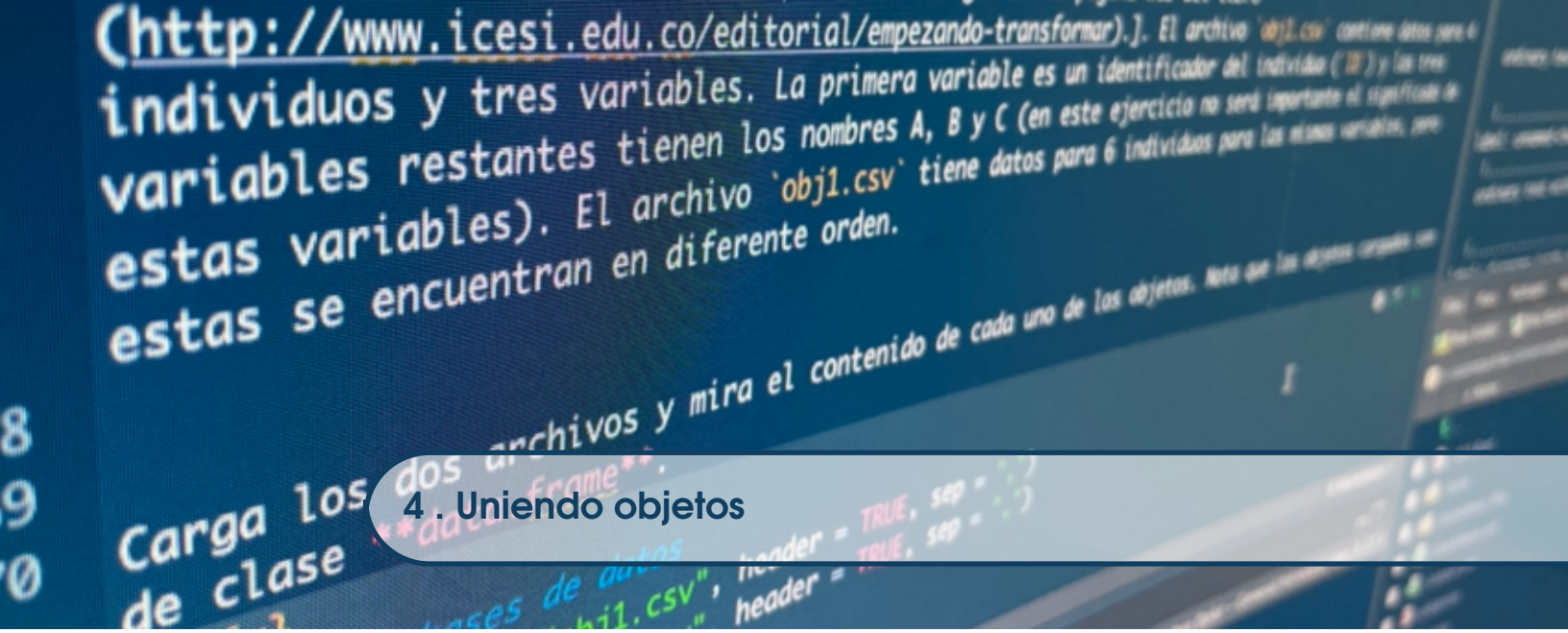
```
## [1] "Song.Name"  "Genre"      "Streams"    "Rep_follow"
```

3.4 Comentarios finales

En este capítulo discutimos cuatro verbos que nos permiten trabajar con variables: **select()**, **mutate()**, **rename()** y **transmute()**. Estas funciones posibilitan seleccionar un subconjunto de variables, crear nuevas variables y renombrarlas. De hecho, estos verbos están relacionados entre sí. Lo que tienen en común **select()** y **transmute()**, es que generan un objeto solo con las variables especificadas. Por otro lado, los verbos **rename()** y **mutate()** tienen en común que el resultante conjunto de datos tiene todas las columnas que ya estaban en el objeto original.

Si combinamos estos verbos con aquellos que permiten filtrar observaciones, tenemos una herramienta muy potente para transformar cualquier objeto de datos, sin importar su tamaño, de acuerdo con nuestras

necesidades. En el próximo capítulo discutiremos cómo unir dos objetos con datos empleando los verbos de *dplyr*.



4 . Uniendo objetos

Ya discutimos cómo, a partir de unos datos, podemos obtener un conjunto de datos con las observaciones de condiciones deseables (ver Capítulo 2) y las variables tal como queramos (ver Capítulo **??**). En el flujo de trabajo de un científico de datos, o persona que trabaja con datos, encontraremos la situación en la que tenemos que unir diferentes objetos con datos. En este capítulo nos concentraremos en algunas de las principales situaciones con las cuales nos toparemos al unir objetos con datos.

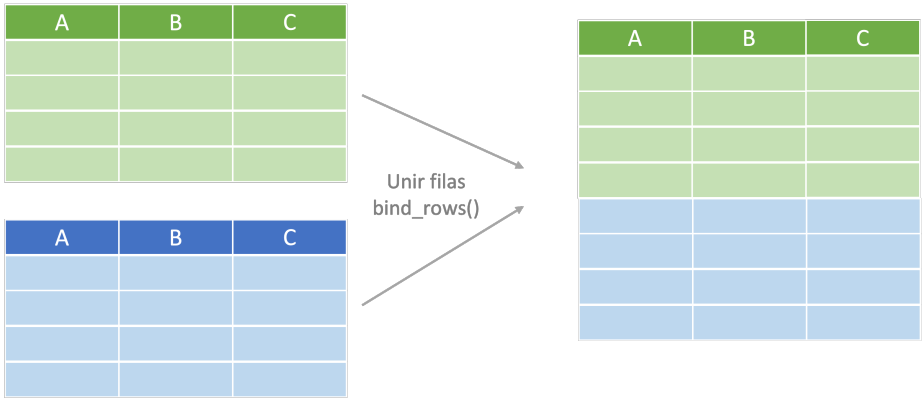
4.1 Combinar observaciones

En algunas ocasiones nos enfrentaremos al caso en el cual dos objetos con las mismas variables (no necesariamente en el mismo orden) y cada objeto tiene observaciones para diferentes individuos o casos. En este sencillo caso, lo único necesario es colocar las filas de un objeto debajo de las filas del otro. Esto lo podemos realizar con el verbo **bind_rows()** (unir filas en español). En la Figura 4.1 se presenta esquemáticamente este caso.

Para este ejemplo emplearemos dos bases de datos que se encuentran en los archivos `obj1.csv` y `obj2.csv`[1]. El archivo `obj1.csv` contiene datos para 4 individuos y tres variables. La primera variable es un identificador del individuo (`ID`) y las tres variables restantes tienen los nombres A, B y C (en este ejercicio no será importante el significado de estas variables). El archivo `obj1.csv` tiene datos para 6 individuos para las mismas variables, pero estas se encuentran en diferente orden.

[1] Los archivos los puedes descargar de la página web del libro (http://www.icesi.edu.co/editorial/empezando-transformar).

Figura 4.1. Representación del proceso de unir filas



Carga los dos archivos y mira el contenido de cada uno de los objetos. Nota que los objetos cargados son de clase **data.frame**.

```
# Carga de bases de datos
obj1 <- read.csv("obj1.csv", header = TRUE, sep = ",")
obj2 <- read.csv("obj2.csv", header = TRUE, sep = ",")
# Mirando los objetos
obj1
```

```
##     ID A B  C
## 1 2290 6 6  6
## 2 2278 5 9 10
## 3 2321 6 1  2
## 4 2308 8 8  1
```

```
obj2
```

```
##     ID  B  A C
## 1 2305  7  5 1
## 2 2246  6 10 8
## 3 2262 10  8 7
## 4 2227  5 10 8
## 5 2321  5  9 2
## 6 2237  0  7 5
```

Ahora, para combinar los dos objetos en uno solo, usaremos el verbo **bind_rows()** empleando los dos objetos como argumentos.

```
obj3 <- bind_rows(obj1, obj2)

obj3
```

```
##       ID  A  B  C
## 1   2290  6  6  6
## 2   2278  5  9 10
## 3   2321  6  1  2
## 4   2308  8  8  1
## 5   2305  5  7  1
## 6   2246 10  6  8
## 7   2262  8 10  7
## 8   2227 10  5  8
## 9   2321  9  5  2
## 10  2237  7  0  5
```

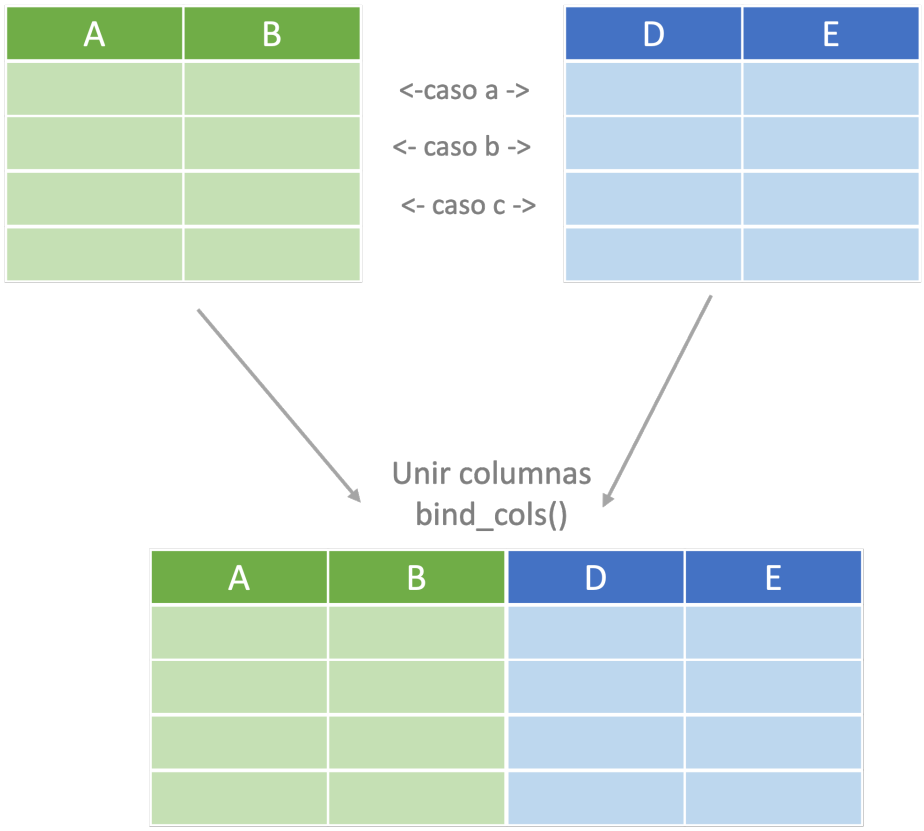
En la base de R se encuentra la función **rbind()** que realiza una tarea similar. No obstante, existen diferentes ventajas al emplear este verbo del paquete *dplyr*, dentro que las que se destacan dos; la función **bind_rows()** al requerir menos memoria RAM, es mas rápida. Esto permite agilizar el análisis, en especial si tenemos muchos datos o estamos empleando *Big Data*. Y por otro lado, la función **bind_rows()** permite combinar dos objetos de clase *data.frame* que tengan diferente número de columnas. **rbind()** presentará un error, mientras que **bind_rows()** asigna "NA" a aquellas filas de las columnas que faltan.

4.2 Combinar variables

En otras ocasiones tendremos datos para los mismos individuos en dos objetos[2] y necesitamos fusionarlos en uno solo. Esto puede realizarse con el verbo **bind_cols()** (unir columnas en español). En la Figura 4.2 se presenta de manera esquemática este caso.

[2]Es importante que las observaciones se encuentren en el mismo orden en los dos archivos.

Figura 4.2. Representación del proceso de unir columnas



En el archivo `obj1b.csv`[3] tenemos datos para las variables `D` y `E` de los mismos individuos que están en el objeto `obj1`. Carguemos y miremos los datos.

```
# Carga de bases de datos
obj1b <- read.csv("obj1b.csv", header = TRUE, sep = ",")

# Mirando los objetos
obj1b
```

```
##     ID  D E
## 1 2290 10 8
## 2 2278  7 9
## 3 2321 10 8
## 4 2308  2 4
```

```
obj1
```

```
##     ID A B  C
## 1 2290 6 6  6
## 2 2278 5 9 10
## 3 2321 6 1  2
## 4 2308 8 8  1
```

Unamos los dos objetos[4] empleando el verbo **bind_cols** y como argumentos los dos objetos.

```
obj4 <- bind_cols(obj1, obj1b)
```

```
## New names:
## * ID -> ID...1
## * ID -> ID...5
```

```
obj4
```

```
##   ID...1 A B  C ID...5  D E
## 1   2290 6 6  6   2290 10 8
## 2   2278 5 9 10   2278  7 9
## 3   2321 6 1  2   2321 10 8
## 4   2308 8 8  1   2308  2 4
```

[3] El archivo lo puedes descargar de la página web del libro (http://www.icesi.edu.co/editorial/empezando-transformar).

[4] Nota que los datos están organizados de la misma manera en ambos objetos. Es decir, la primera fila en ambos objetos corresponde al mismo individuo (`ID`) y así para todos los otros casos.

La variable `ID` estaba en ambos objetos y por tanto aparece dos veces en el nuevo objeto creado. Para evitar esto podemos usar el verbo **select()**, estudiado en el Capítulo **??**.

```
obj4 <- obj1b %>%
              select(-c(ID)) %>%
              bind_cols(obj1)
obj4
```

```
##    D E   ID A B  C
## 1 10 8 2290 6 6  6
## 2  7 9 2278 5 9 10
## 3 10 8 2321 6 1  2
## 4  2 4 2308 8 8  1
```

4.3 Combinar Observaciones y variables

En la sección anterior discutimos cómo unir dos objetos en condiciones ideales. Los casos están en los dos archivos y se encuentran en el mismo orden. Pero, a veces esas condiciones ideales no se presentan. Podemos encontrarnos con situaciones en las que no todos los casos se encuentren en los dos objetos, o estén organizados de forma diferente.

Por ejemplo, veamos el archivo `obj5.csv`[5] que contiene datos para las variables `F`, `G` y `H`. Carguemos los datos y veámoslo.

```
# Carga de bases de datos
obj5 <- read.csv("obj5.csv", header = TRUE, sep = ",")

# Mirando los objetos
obj5
```

```
##     ID F G H
## 1 2305 6 3 4
## 2 2227 4 3 5
## 3 2230 2 3 7
## 4 2403 3 3 8
```

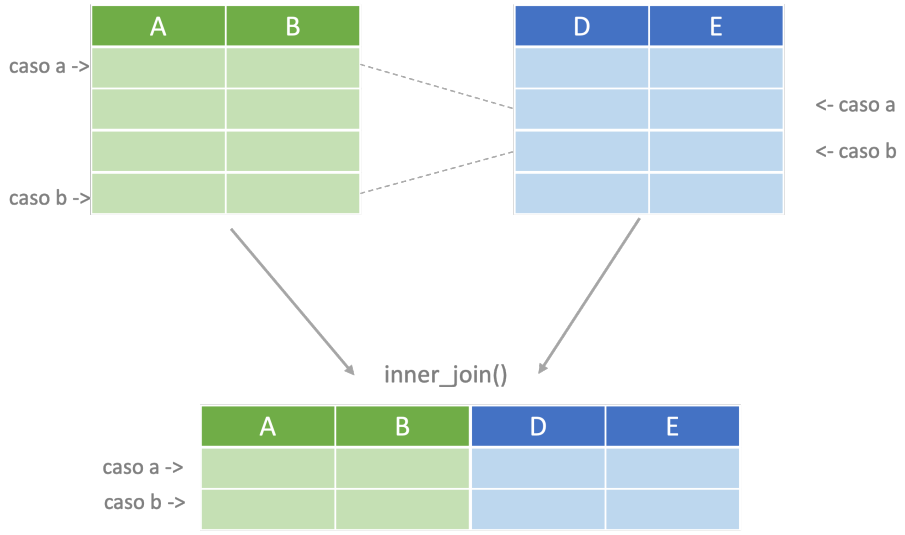
```
obj2
```

[5] El archivo lo puedes descargar de la página web del libro (http://www.icesi.edu.co/editorial/empezando-transformar).

```
##      ID  B  A C
## 1  2305  7  5 1
## 2  2246  6 10 8
## 3  2262 10  8 7
## 4  2227  5 10 8
## 5  2321  5  9 2
## 6  2237  0  7 5
```

Observa que de los 4 casos del `obj5`, dos están en el `obj2` (`ID` 2305 y 2227) y 2 no lo están (`ID` 2230 y 2403). Si queremos unir los datos de los dos archivos hay varias posibilidades. La primera es que al unir el `obj2` con el `obj5`, creemos un nuevo objeto con solo los casos que se encuentran en los dos objetos. Es decir, con solo los casos para los cuales tenemos datos de todas las variables. Esto corresponde al verbo unir internamente o **inner_join()** en el lenguaje de *dplyr*. Este verbo implica retener solo los casos que están en ambos objetos. En la Figura 4.3 se presenta de manera esquemática esta operación.

Figura 4.3. Representación del proceso de unir internamente



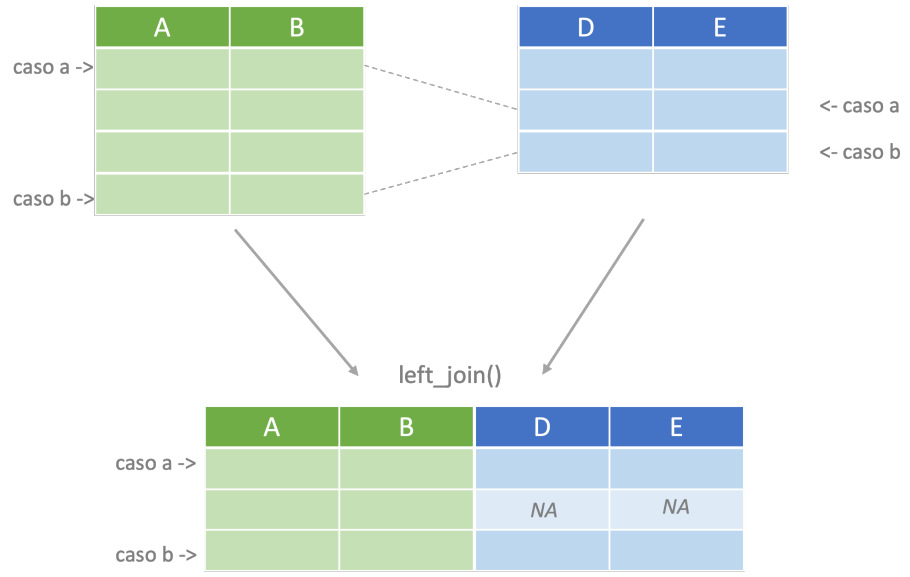
Este verbo tiene como argumentos a los dos objetos y al argumento **by** que permite especificar qué se empleará para identificar los casos en ambos objetos. Sino se especifica el argumento **by**, esta función automáticamente encontrará cuál variable se repite en ambos objetos.

```
# inner join
obj6 <- inner_join(obj5, obj2, by = "ID")
# mirando el objeto
obj6
```

```
##     ID F G H B  A C
## 1 2305 6 3 4 7  5 1
## 2 2227 4 3 5 5 10 8
```

Otra posibilidad para unir los dos objetos, es crear uno nuevo que tenga todos los casos del primero (objeto de la izquierda). Sino se encuentran datos para una de las variables del primer objeto en el segundo objeto para un determinado caso, entonces se asignará el valor "NA" (no disponible o valor perdido) para esa variable y caso. Esto se puede realizar con el verbo unir a la izquierda (**left_join()**) . Este verbo retiene solo los casos que están en el primer objeto (ver Figura 4.4).

Figura 4.4. Representación del proceso de unir a la izquierda



Este verbo funciona de manera similar al **inner_join**. Continuemos con nuestro ejemplo.

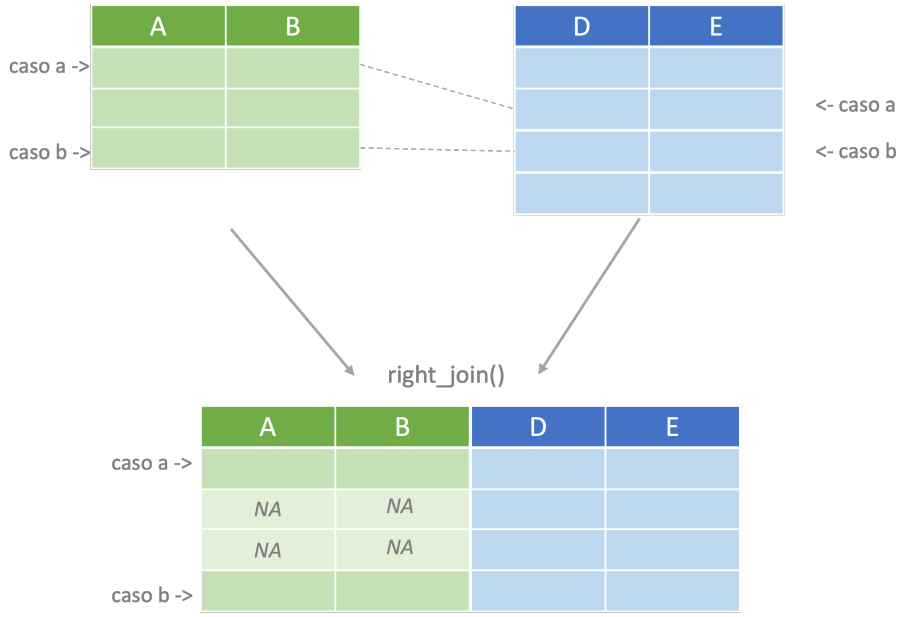
```
# left join
obj7 <- left_join(obj5, obj2, by = "ID")
```

```
# mirando el objeto
obj7
```

```
##     ID F G H  B  A  C
## 1 2305 6 3 4  7  5  1
## 2 2227 4 3 5  5 10  8
## 3 2230 2 3 7 NA NA NA
## 4 2403 3 3 8 NA NA NA
```

Una tercera posibilidad para unir los dos objetos es mantener todos los casos que están en el segundo objeto. Es decir, el verbo **right_join()** o unir a la derecha en español. En la Figura 4.5) se representa esta operación.

Figura 4.5. Representación del proceso de unir a la derecha



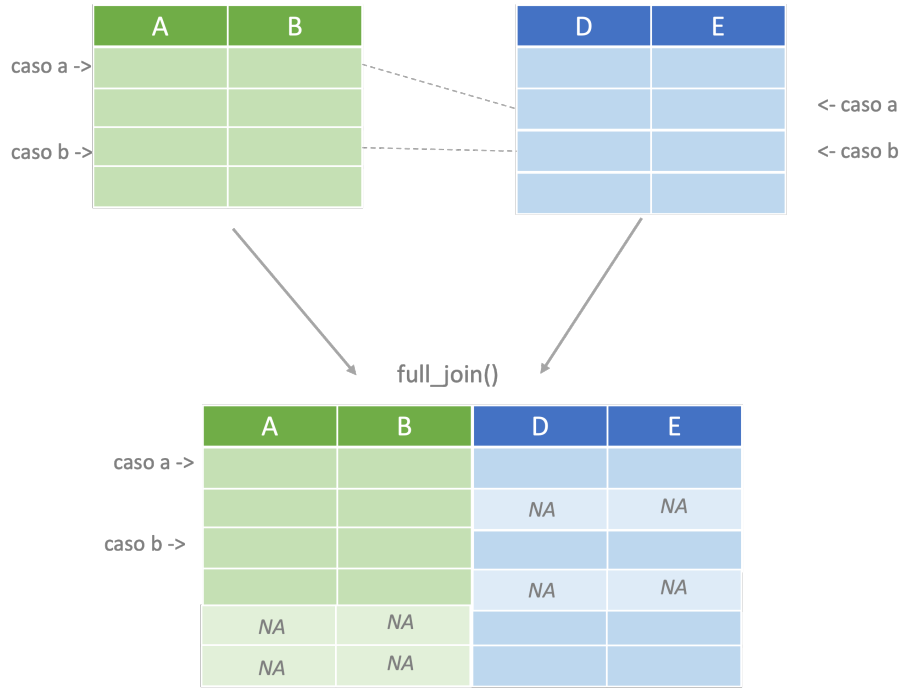
A estas alturas, la gramática de estos verbos debe ser evidente para ti. Sigamos con nuestro ejemplo.

```
# right join
obj8 <- right_join(obj5, obj2, by = "ID")
# mirando el objeto
obj8
```

```
##     ID  F  G  H  B  A C
## 1 2305  6  3  4  7  5 1
## 2 2227  4  3  5  5 10 8
## 3 2246 NA NA NA  6 10 8
## 4 2262 NA NA NA 10  8 7
## 5 2321 NA NA NA  5  9 2
## 6 2237 NA NA NA  0  7 5
```

La última posibilidad que consideraremos para unir objetos es cuando necesitamos construir un objeto que tenga todos los casos presentes en los dos objetos. Esto se realiza con el verbo **full_join()** o unir completamente en español (Ver Figura 4.6).

Figura 4.6. Representación del proceso de unir completamente.



El siguiente código presenta un ejemplo del verbo **full_join()**.

```
# full join
obj9 <- full_join(obj5, obj2, by = "ID")
# mirando el objeto
obj9
```

```
##     ID  F  G  H  B  A  C
## 1 2305  6  3  4  7  5  1
## 2 2227  4  3  5  5 10  8
## 3 2230  2  3  7 NA NA NA
## 4 2403  3  3  8 NA NA NA
## 5 2246 NA NA NA  6 10  8
## 6 2262 NA NA NA 10  8  7
## 7 2321 NA NA NA  5  9  2
## 8 2237 NA NA NA  0  7  5
```

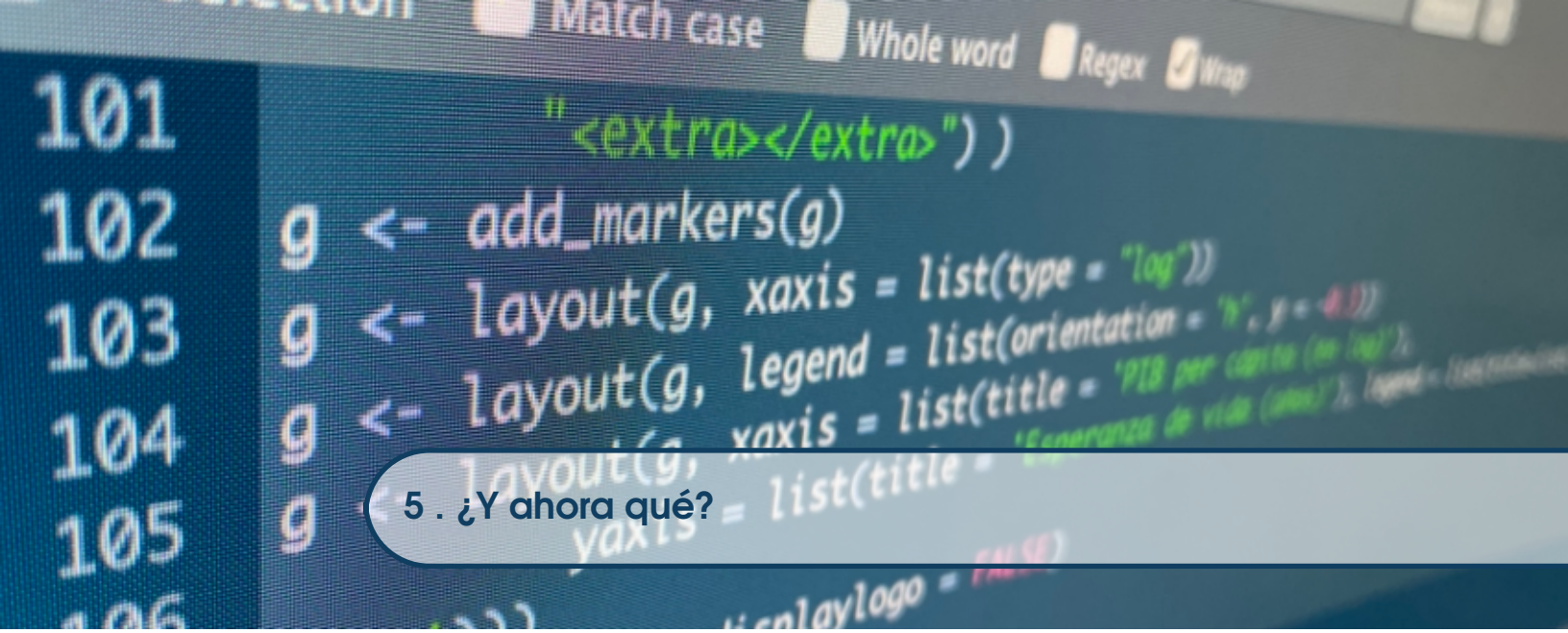
En la base de R se encuentra la función **merge()**, esta realiza una tarea similar a los verbos **join()** y los otros terminados en **_join()** . No obstante, al igual que con el caso **bind_rows()** y **rbind()**, existen diferentes ventajas al emplear los verbos del paquete *dplyr*. Se destacan dos razones. Las funciones **join()** de *dplyr* tienden a ser más rápidas que **merge()** cuando se emplean objetos de datos grandes. Esto sucede porque el paquete *dplyr* usa de manera mas eficiente la RAM de los equipos.

Por otro lado, las funciones **join()** de *dplyr* conservan el orden original de las filas en los *data.frame*, mientras que la función **merge()** ordena automáticamente las filas en orden alfabético según la columna utilizada para realizar la unión.

4.4 Comentarios finales.

En este capítulo revisamos los verbos que nos permiten unir dos objetos con datos. Si los unimos con los que hemos estudiado en los capítulos anteriores, tendremos una caja de herramientas completa para trabajar con datos.

Antes de continuar, es importante mencionar que existen otros verbos para unir objetos que podrían ser útiles en alguna oportunidad. Por ejemplo: **intersect()**, **union()**, **semi_join()** y **anti_join()**. Ya puedes ver la ayuda de estas funciones y emplearlas sin ningún problema.



5 . ¿Y ahora qué?

A lo largo del libro discutimos cómo emplear el paquete *dplyr* para manipular datos. También estudiamos cómo filtrar casos que nos interesan, seleccionar variables, crear nuevas variables y unir objetos con datos. Todas estas son tareas rutinarias cuando empleamos datos.

R es una herramienta versátil y adicionar *dplyr* la hace una herramienta aún mas versátil. Con estos elementos es posible optimizar el flujo de trabajo cuando empleamos datos para sacar conclusiones.

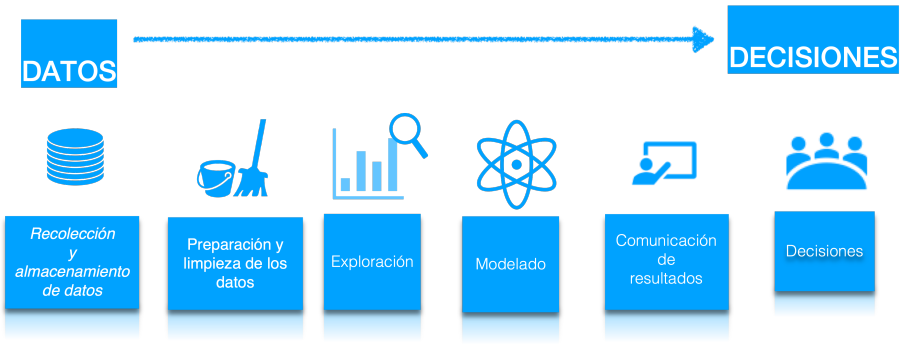
El flujo de trabajo[1] inicia desde la preparación y limpieza de los datos que previamente han sido recolectados y almacenados (ver Figura 5.1). Después se exploran los datos para entender las relaciones entre las variables. Esta parte del proceso puede implicar la visualización de los datos. Posteriormente, pasamos al modelamiento necesario para lograr el objetivo de nuestro análisis. Finalmente, procedemos a la comunicación de los resultados, esto implica generar visualizaciones de nuestros hallazgos. En esta etapa los datos ya se han transformado en una conclusión que permite la toma de decisiones.

Las herramientas estudiadas en este libro nos facilitan la tarea de preparación y limpieza de los datos para su posterior exploración y modelado. Si te interesan las herramientas que estudiamos en este libro, con seguridad estarás interesado en las herramientas que brinda R para el modelado de los datos y su visualización.

En la comunidad R encontrarás numerosos paquetes para el modelado de los datos. En el siguiente enlace encontrará información de las diferentes temáticas para las cuáles existen paquetes: https://cran.r-project.org/web/views/ . Hay numerosos paquetes, que van desde la

[1] Para una explicación breve de las actividades en el proceso de análisis de datos puedes ver el video en el siguiente enlace: https://youtu.be/rhLWa-vOxyU .

Figura 5.1. Flujo de trabajo para pasar de datos a tomar decisiones



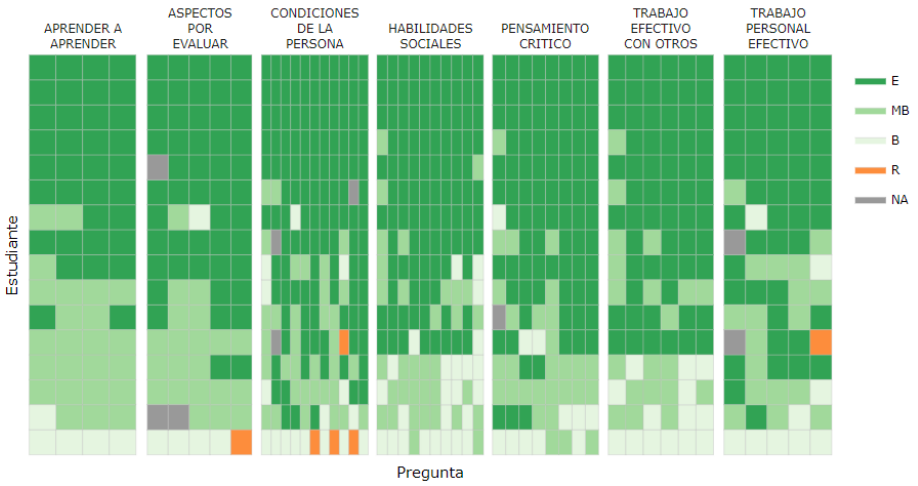
estadística, hasta otros relacionados con genética y métricas del medio ambiente, pasando por algoritmos de inteligencia artificial. En el universo de R encontrarás paquetes que te permitirán realizar cualquier actividad que te permita sacarle provecho a datos cuantitativos y cualitativos (Alonso Cifuentes y Ocampo, 2022).

La **Visualización** facilita la comprensión y el descubrimiento de los datos por medio de gráficos. Las visualizaciones nos permiten tanto entender nuestros datos antes de analizarlos, como comunicar los resultados de un análisis (Alonso Cifuentes y Ocampo, 2022).

El paquete **ggplot2** (Wickham, 2016) permite realizar visualizaciones de alta calidad siguiendo una gramática sencilla[2]. La Figura 5.2 presenta el resumen de los resultados de valoración del desempeño en práctica a un número relativamente grande de estudiantes de un programa de la Universidad Icesi. Las filas representan a cada uno de los estudiantes, y las columnas una pregunta del instrumento utilizado. Los colores representan la valoración en una escala cualitativa. (Alonso Cifuentes y Ocampo, 2022)

[2]En el siguiente enlace encontrarás un video con una breve introducción al paquete **ggplot2**: https://youtu.be/IVkn7spjZ1Q. Si deseas una introducción algo mas profunda a este paquete puedes consultar Alonso Cifuentes y González (2012), Alonso Cifuentes y Montenegro (2012) o Alonso y Largo (2022).

Figura 5.2. Visualización de todas las preguntas del intrumento de valoración de estudiantes en práctica de un programa de la Universidad Icesi



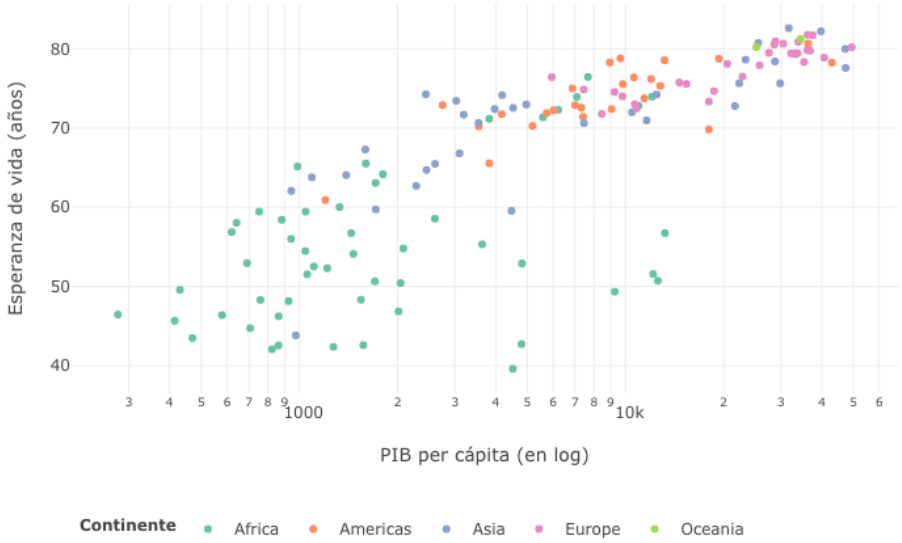
Fuente: Cienfi.

Con R también podemos construir visualizaciones interactivas (ver Figuras 5.3). La Figura 5.3 presenta los datos del paquete **gapminder** del PIB per cápita (en escala logarítmica) y la esperanza de vida alrededor del mundo para 2007[3]. Este tipo de visualizaciones le permiten al usuario interactuar con los datos. Se pueden apagar los continentes y hacer *zoom*. Pasa el cursor por encima de un punto para ver la información. (La interacción solo funciona en la versión Web del libro).

Las posibilidades de visualizar los datos con R son numerosas. Un buen inicio para aprender a visualizar datos con R es el paquete *ggplot2* (Wickham et al., 2021) . Te invito a leer el siguiente libro de esta serie: "Empezando a Visualizar Datos con R y Ggplot2" (Alonso y Largo, 2022). Espero que esta obra te motive a continuar tu camino de aprendizaje y unirte a la gran comunidad de R. En este universo de R, ¡la imaginación es el límite!

[3]Puedes encontrar una breve introducción a la construcción de gráficos interactivo con el paquete **plotly** (Sievert, 2020) en el siguiente enlace: https://youtu.be/EWjxic2ce9g .

Figura 5.3. Gráfico interactivo de la relación del PIB per cápita y la esperanza de vida alrededor del mundo (2007)



Fuente: Datos de Gapminder y cálculos propios empleando el paquete plotly.

Bibliografía

Alonso, J. C. y Largo, M. F. (2022). Empezando a visualizar datos con r y ggplot2.

Alonso Cifuentes, J. C. (2021). Una introducción a los Loops en R (y algunas alternativas). Icesi Economics Lecture Notes 019408, Universidad Icesi.

Alonso Cifuentes, J. C. y González, A. (2012). Ggplot: gráficos de alta calidad. *Apuntes de Economía*, (33):29.

Alonso Cifuentes, J. C. y Montenegro, S. (2012). Visualización de información georeferenciada en ggplot2. *Apuntes de Economía*, (35):16.

Alonso Cifuentes, J. C. y Ocampo, M. P. (2022). Empezando a usar: Una guía paso a paso.

Bryan, J. (2017). *gapminder: Data from Gapminder*. R package version 0.3.0.

Henry, L. y Wickham, H. (2020). *purrr: Functional Programming Tools*. R package version 0.3.4.

Müller, K. y Wickham, H. (2021). *tibble: Simple Data Frames*. R package version 3.1.6.

Pillai, S. (2021). Spotify top 200 charts (2020-2021).

R Core Team (2020). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Sievert, C. (2020). *Interactive Web-Based Data Visualization with R, plotly, and shiny*. Chapman and Hall/CRC.

Wickham, H. (2016). *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York.

Wickham, H. (2019). *stringr: Simple, Consistent Wrappers for Common String Operations*. R package version 1.4.0.

Wickham, H. (2020). *forcats: Tools for Working with Categorical Variables (Factors)*. R package version 0.5.0.

Wickham, H. (2021). *tidyr: Tidy Messy Data*. R package version 1.1.3.

Wickham, H., Averick, M., Bryan, J., Chang, W., McGowan, L. D., François, R., Grolemund, G., Hayes, A., Henry, L., Hester, J., Kuhn, M., Pedersen, T. L., Miller, E., Bache, S. M., Müller, K., Ooms, J., Robinson, D., Seidel, D. P., Spinu, V., Takahashi, K., Vaughan, D., Wilke, C., Woo, K., y Yutani, H. (2019). Welcome to the tidyverse. *Journal of Open Source Software*, 4(43):1686.

Wickham, H., François, R., Henry, L., y Müller, K. (2021). *dplyr: A Grammar of Data Manipulation*. R package version 1.0.7.

Wickham, H. y Grolemund, G. (2016). *R for data science: import, tidy, transform, visualize, and model data*. .º'Reilly Media, Inc.".

Wickham, H., Hester, J., y Francois, R. (2018). *readr: Read Rectangular Text Data*. R package version 1.3.1.

Índice alfabético

Editorial Universidad Icesi

UNIVERSIDAD ICESI

Si estás leyendo este libro, ya haces parte de la comunidad que emplea R para analizar datos. Esta obra tiene como objetivo presentar una primera aproximación al paquete *dplyr* que permite manipular rápidamente bases de datos en R. Si eres nuevo en el universo de R o en el uso del paquete *dplyr*, este libro puede ser un buen punto de arranque. Si ya eres usuario de *dplyr*, seguramente este libro no te aportará nuevos conocimientos, pero puede ser una herramienta de consulta de algunos conceptos básicos.

Herramientas del BIG DATA y analytics