Introducción al análisis de datos con R, *2ª versión*

Curso dirigido a personas de las Ciencias Sociales

Bastián Olea Herrera

2026-03-25

Inicio

Bastián Olea Herrera

bastianolea.rbind.io

Encuentra todo sobre el curso en este post de mi blog.

Curso | Código

Grabaciones | Diapositivas

Contacto

Puedes encontrar más contenidos para aprender R en este sitio, y más cosas de R en mi sitio web.

¿Por qué programar para analizar datos?

Existen muchas diferencias entre *hacer* algo y **programarlo:**

- Aprender a programar es un **cambio de perspectiva**: pasamos de *hacer* cosas a *planificar* cosas 🤌
- Desde la mera **Acción** 🔧 a la **Instrucción** 🧠 y luego al **Proceso** ⚙️
- Trabajamos **desde arriba**, viendo todos sus pasos y componentes 👩🏻‍💻
- Pasos independientes, reutilizables, mejorables, intercambiables ⚙️
- El trabajo que hagas puede **servirte** a futuro, a ti y a otrxs ♻️
- Resultados **replicables** 🔄 que dan paso a la **automatización** 🏭

¿Qué es R?

R es un lenguaje de programación enfocado al **análisis de datos**, las **estadísticas** y la **visualización de datos**.

Descargar R

¿Para qué sirve?

- Para procesar, limpiar, y analizar datos
- Crear gráficos, tablas, mapas, y otros resultados visuales
- Repetir, mejorar, adaptar y reutilizar lo que ya has hecho ♻️
- Automatizar tareas repetitivas 🤖
- Expandir tus resultados con aplicaciones, sitios web, APIs, y más 🚀

¿Por qué usar R?

Gratuito

R es un lenguaje de programación abierto y gratuito, y es parte de la comunidad del *software libre*, así que nunca tendrás que pagar nada!

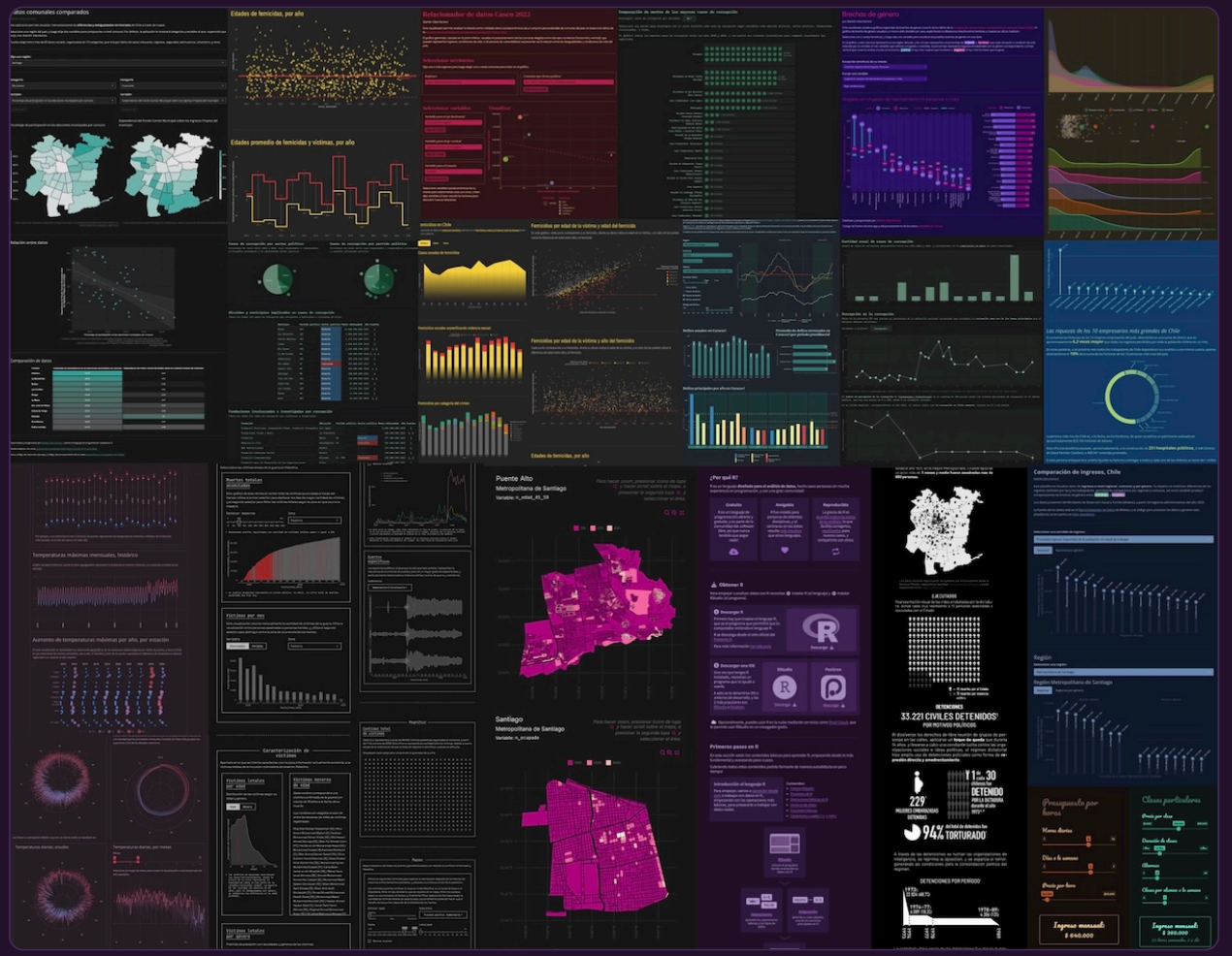
Amigable

R fue creado para personas de distintas disciplinas, y al centrarse en los datos resulta más intuitivo que otros lenguajes.

Reproducible

La gracia de R es guardar todos los pasos de tu análisis, lo que facilita corregirlos, reutilizarlos para nuevos casos, y compartirlo con otros.

¿Qué se puede hacer con R?



Muestra de algunas aplicaciones, gráficos, tablas, y mapas hechos con R. También algunos trabajos que he hecho.

Conceptos clave

Script

Archivo de texto .R en el que escribimos nuestro código, en pasos, y siguiendo un orden lógico.

Consola

Es la forma directa de interactuar con R, un comando a la vez, con resultados efímeros.

Proyecto

Archivo .Rproj que marca nuestro espacio de trabajo: una carpeta específica que reúne todas las piezas de nuestro análisis.

RStudio

RStudio es una IDE (entorno de desarrollo integrado) que nos permite escribir código en R de forma más cómoda, organizada, y visual.

- Entorno de desarrollo integrado (IDE) enfocado en R
- Lanzado en 2011
- Software libre (licencia de código abierto AGPL)

Descargar RStudio

Paneles de RStudio

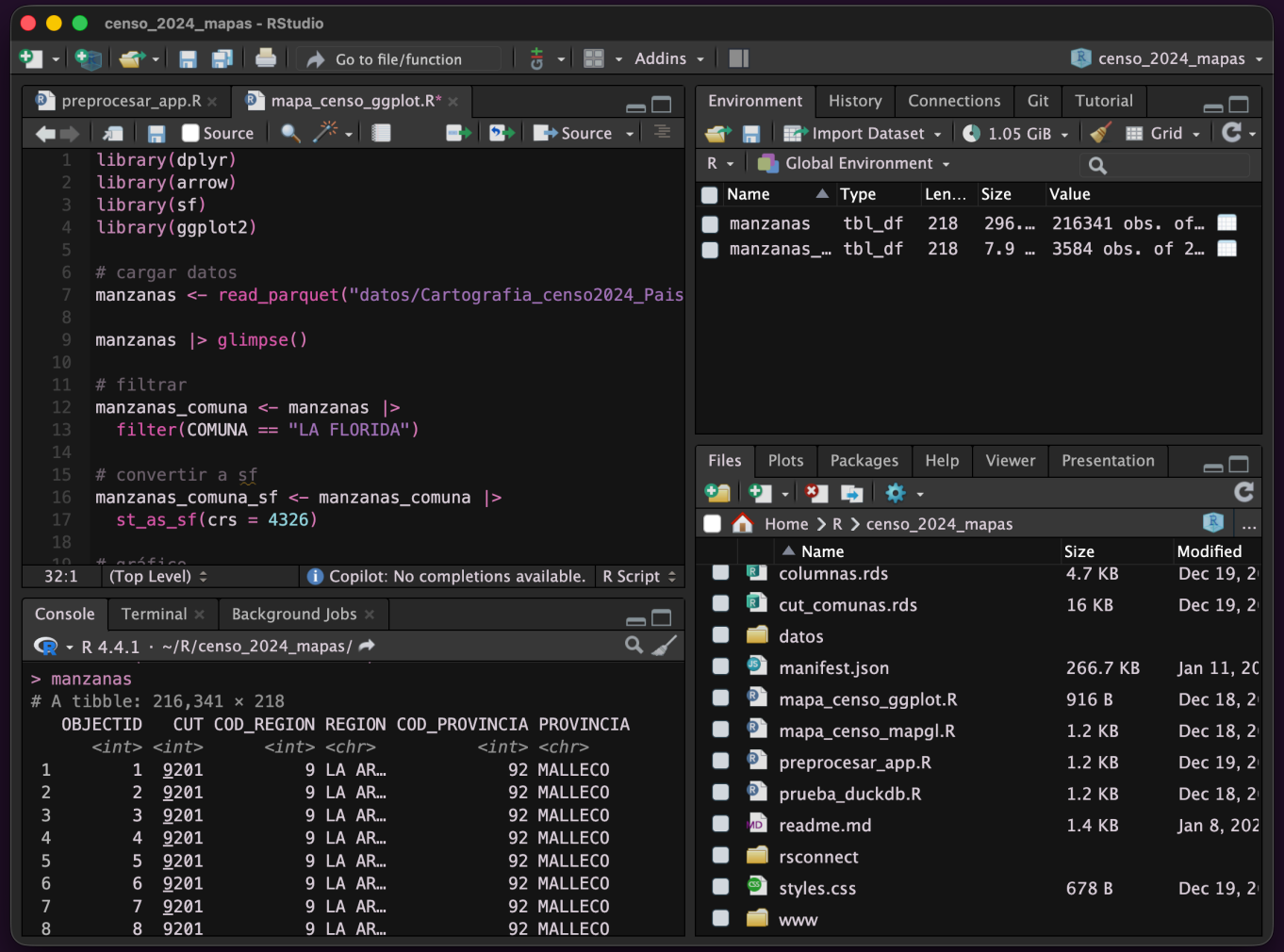
Descargar RStudio

1 Scripts

2 Consola

3 Entorno

4 Archivos



Conceptos

❶ **Panel de scripts:** los archivos de texto con nuestro código. Podemos tener varias pestañas. Ejecutamos el código poniendo el cursor en la línea y presionando `control + enter`, o el botón *Run*.

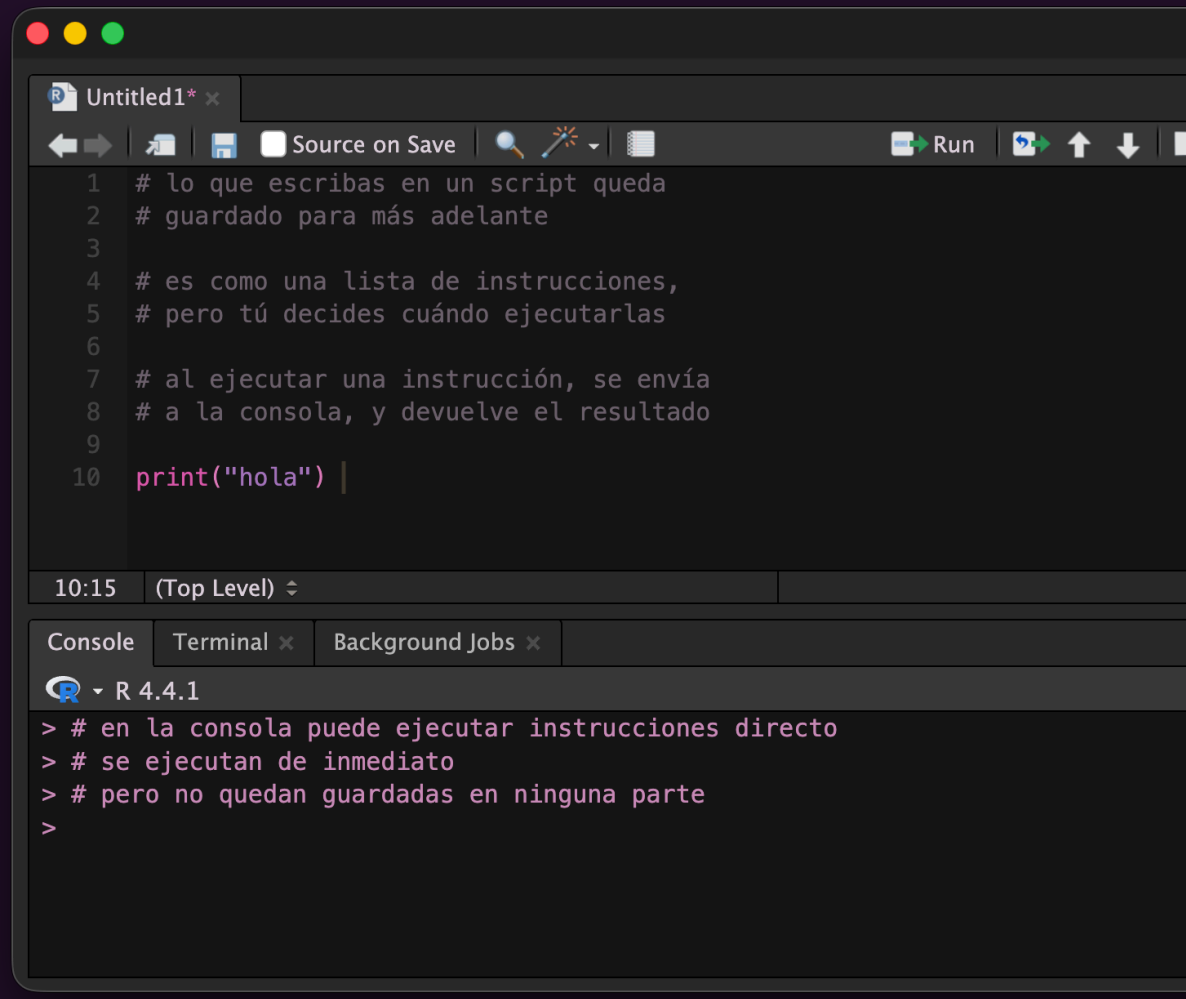
❷ **Panel de consola:** en la consola se imprimen los **resultados** del código que ejecutamos. También podemos ejecutar código directamente en ella escribiendo y presionando `enter`.

❸ **Panel de entorno:** acá veremos los **objetos** que vayamos creando o cargando, que pueden ser números, texto, tablas de datos, funciones, gráficos y otros.

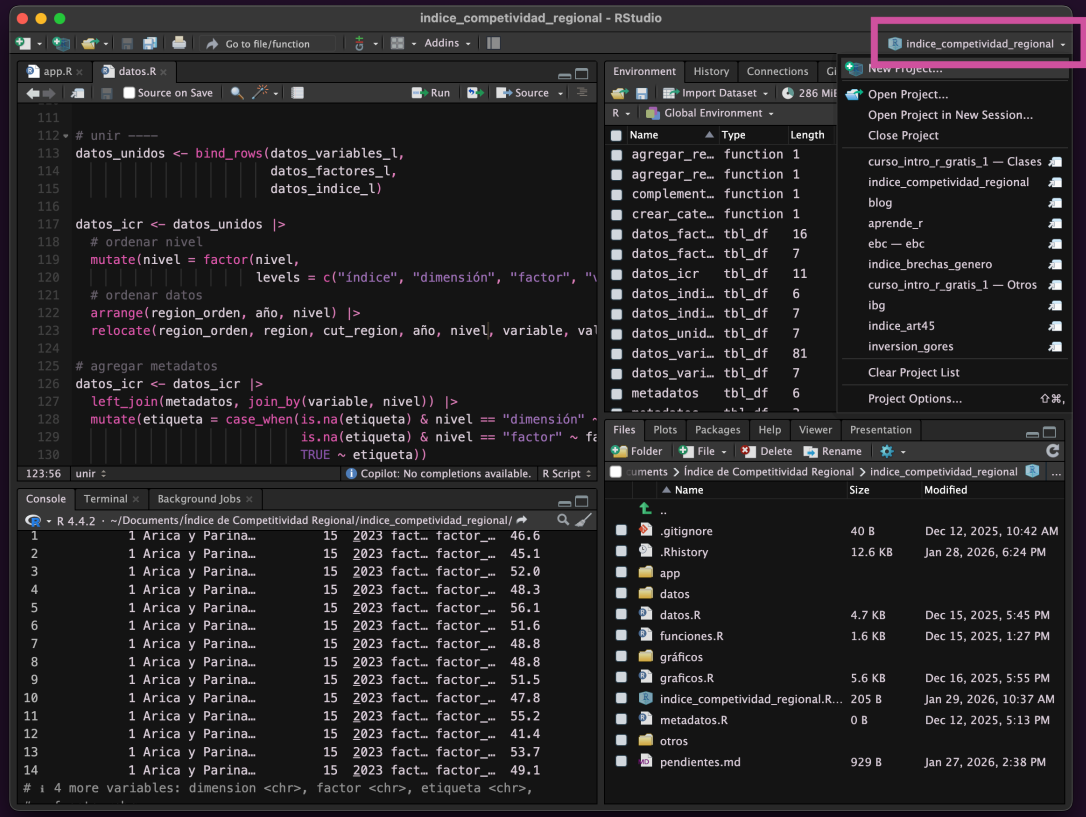
❹ **Panel de archivos:** en este panel podemos navegar los archivos y carpetas de nuestro proyecto y/o computador. La idea es que *todo* esté dentro del proyecto!

Consola y scripts

- Todos los **comandos** se ejecutan por medio de la consola
- Pero al ejecutar código en la consola, el código no se guarda!
- Para **guardar** el código, escribimos en **scripts**
- Escribimos las **instrucciones** en un script, y RStudio se encarga de pasarlos a la consola y ejecutarlos cuando se lo pidamos 💡



Botones



Cambiar de **proyecto** o crear uno nuevo

Introducción a R

Veamos ahora los **elementos más básicos** del lenguaje R, para luego avanzar a su aplicación al **análisis de datos**.

Guía de aprendizaje

Operaciones básicas

Ver tutorial

- Podemos realizar cualquier operación matemática en R.
- Para **ejecutar** un comando, pon el cursor de texto en la línea o expresión que desees ejecutar, o selecciónala, y presiona el botón *Run*, o las teclas `control + enter`.
- Lo importante es que el cursor de texto | esté en cualquier lugar de la línea.
- El **resultado** de todas las operaciones aparece en la **consola**.

```
1  2 + 2 #suma
```

```
[1] 4
```

```
1  50 * 100 #multiplicación
```

```
[1] 5000
```

```
1  4556 - 1000 #resta
```

```
[1] 3556
```

```
1  6565 / 89 #división
```

```
[1] 73.76404
```

```
1  10^4 #potencias
```

Comentarios

- Los comentarios nos permiten poner texto en cualquier parte del script sin que afecte los cálculos.
- También podemos poner un comentario al final de una línea sin que afecte el código

```
1 1 + 1 + 1 + 1
2 # comentario: quizás esto debería
3 # ser de otra forma, porque
4 # la verdad quedó bien mal...
5
6 1 * 4 # así queda mucho mejor
```

Tipos de datos

Lo que podemos hacer siempre va a depender del **tipo** de cada objeto.

En R existen varios tipos:

Numéricos

```
1  2  3  4  5.1  5.2  5.333
```

Pueden ser decimales (*doubles*) o enteros (*integers*)

Caracter (texto)

```
"ésta es una cadena de texto"
```

Lógicos (verdadero o falso)

```
TRUE FALSE TRUE
```

Objetos

Se le llama **objeto** a cualquier dato, variable, o elemento que tengas en R.

Podemos guardar *todo* como un objeto.

Para **crear** un objeto, simplemente le damos un nombre y le *asignamos* un contenido.

nombre ⟵ *contenido*

Para asignar algo a un objeto, usamos el operador `<-`

```
cifra <- 4
```

El objeto `cifra` se crea cuando declaramos que va a contener el valor `4`

```
cifra
```

```
[1] 4
```

Asignación

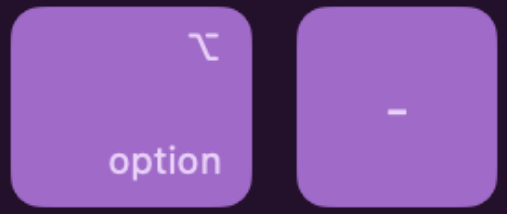
Ver tutorial

Con el *operador de* ***asignación*** creamos objetos nuevos.

Se escribe con:

Windows:

Mac:



Al asignar algo, **creamos** o **modificamos** un objeto con el valor que le estamos asignando.

Al ejecutar un objeto, obtenemos su valor.

```
1  edad <- 32
2  edad
```

```
[1] 32
```

```
1  animal = "gato"
2  animal
```

```
[1] "gato"
```

Podemos usar el objeto creado para lo que queramos.

Crear objetos es como **asignar variables**, y podemos usar estas variables para llevar a cabo operaciones!

```
1  edad <- 32
2  año <- 2026
3  año - edad
```

```
[1] 1994
```

```
1  perros = 2
2  gatos = 3
3  perros + gatos
```

```
[1] 5
```

```
1  presupuesto = 100000
2  pizza = 15000
3  presupuesto - pizza * 10
```

```
[1] -50000
```

Vectores

Los vectores son **secuencias** de elementos. Un objeto que contiene cero o más datos.

- Estos elementos son de un **mismo tipo**: numérico, carácter o lógico.
- Son la forma más básica de registrar e interactuar con varios datos a la vez.

```r
c(1, 2, 3, 4, 5, 6)
```

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

```r
c("a", "b", "c", "d")
```

```
[1] "a" "b" "c" "d"
```

```r
c(TRUE, FALSE, FALSE)
```

```
[1]  TRUE FALSE FALSE
```

```r
c(0.2, 0.1, -0.0, -0.1)
```

```
[1]  0.2  0.1  0.0 -0.1
```

Operaciones sobre vectores

Podemos realizar cualquier **operación** sobre los elementos de un vector:

```r
año <- 2026
edades <- c(54, 34, 65, 21, 32)
año - edades
```

```
[1] 1972 1992 1961 2005 1994
```

Funciones

- Las **funciones** son pequeños programas que permiten realizar distintas operaciones.

`función(argumento = 123)`

Algunas funciones comunes:

```
mean() # calcular promedio
median() # calcular mediana
sum() # sumar elementos
min() # valor mínimo
max() # valor máximo
```

Por ejemplo:

```
1 mean(c(1, 2, 3, 4, 5))
```

```
[1] 3
```

Si no sabemos cómo usar una función, podemos buscar su nombre en el panel **Ayuda** de RStudio, o escribir el nombre de la función precedido de un signo de interrogación: `?funcion`

Paquetes

- **Extensiones** de R que te permiten agregar **nuevas funciones** al lenguaje, **datos**, y más.
- Se instalan desde internet
- Son creados y mantenidos **por la comunidad**
- Se revisan para garantizar su seguridad y estabilidad

```
install.packages()
```

Datos 🗄️

Para cargar datos, tenemos que:

- Tener un **archivo** y conocer su **formato** (Excel, Stata, CSV, etc.)
- ~~Saber **dónde está** el archivo en nuestro computador~~
- **Guardar** el archivo dentro del proyecto de R 📂
- Conocer una **función** y posiblemente un **paquete** que lea ese formato de archivos

Cargar datos

- Las funciones de carga de datos usan como argumento la **ruta** del archivo.

Cargar un archivo en la misma carperta del proyecto:

```
datos <- read.csv("datos.csv")
```

Cargar un archivo en una subcarpeta del proyecto:

```
datos <- read.csv("carpeta/datos.csv")
```

Paquetes para carga de datos

- `{readxl}` es uno de los paquetes de lectura de planillas Excel. Para escribir un archivo Excel está `{writexl}`
- `{readr}` lee y escribe datos de múltiples formatos (csv, rds, rdata), usualmente de forma más veloz y cómoda
- `{haven}` permite cargar archivos SPSS, Stata, y SAS
- `{arrow}` es un formato moderno de datos columnares, optimizado para grandes volúmenes y velocidad de carga, y pensado para usarse en distintos softwares

Carga de datos desde Excel

Probemos cargando unos datos!

Descargar datos de pobreza en Excel

Cargamos los datos usando una función del paquete `{readxl}`:

```r
# cargar paquete
library(readxl)

# cargar datos
datos <- read_excel("datos/pobreza/estimaciones_pobreza.xlsx")

# mirar datos
head(datos)
```

```
# A tibble: 6 × 10
  codigo region   comuna       personas_proy personas porcentaje limite_inferior
  <chr>  <chr>    <chr>                <dbl>    <dbl>      <dbl>           <dbl>
1 1101   Tarapacá Iquique             229674   41967.      0.183           0.162
2 1107   Tarapacá Alto Hospic…        138527   45162.      0.326           0.295
3 1401   Tarapacá Pozo Almonte         18290    4563.      0.250           0.199
4 1402   Tarapacá Camiña                1380     308.      0.223           0.138
5 1403   Tarapacá Colchane              1575     473.      0.300           0.187
6 1404   Tarapacá Huara                 3072    1185.      0.386           0.319
# ℹ 3 more variables: limite_superior <dbl>, casen <chr>, tipo_estimacion <chr>
```

{dplyr}

- Paquete de exploración, manipulación y transformación de datos
- Sus funciones se escriben como verbos
- Las acciones se encadenan con el operador de conexión o *pipe*: `|>`


```r
# install.packages("dplyr")
library(dplyr)
```

Ver tutorial

¿Para qué sirve {dplyr}?

Para **explorar** y **modificar** los datos.

Seleccionar columnas:

```
1 datos |> select(region, comuna, porcentaje)
```

```
# A tibble: 4 × 3
  region   comuna        porcentaje
  <chr>    <chr>              <dbl>
1 Tarapacá Iquique            0.183
2 Tarapacá Alto Hospicio      0.326
3 Tarapacá Pozo Almonte       0.250
4 Tarapacá Camiña             0.223
```

Filtrar datos:

```
1 datos |> filter(porcentaje > 0.3)
```

```
# A tibble: 4 × 3
  region   comuna          porcentaje
  <chr>    <chr>                <dbl>
1 Tarapacá Alto Hospicio        0.326
2 Tarapacá Colchane             0.300
3 Tarapacá Huara                0.386
4 Atacama  Tierra Amarilla      0.345
```

Ordenar observaciones:

```
1 datos |> arrange(porcentaje)
```

```
# A tibble: 4 × 3
  region        comuna         porcentaje
  <chr>         <chr>               <dbl>
1 Magallanes    Río Verde         0.00113
2 Magallanes    Laguna Blanca     0.00353
3 Magallanes    Cabo De Hornos    0.0158
4 Metropolitana Vitacura          0.0244
```

Crear variables:

```
1 datos |> mutate(ranking = row_number())
```

```
# A tibble: 4 × 3
  comuna        porcentaje ranking
  <chr>              <dbl>   <int>
1 General Lagos      0.586       1
2 Saavedra           0.445       2
3 Ercilla            0.416       3
4 Galvarino          0.399       4
```

Y más!

Conector

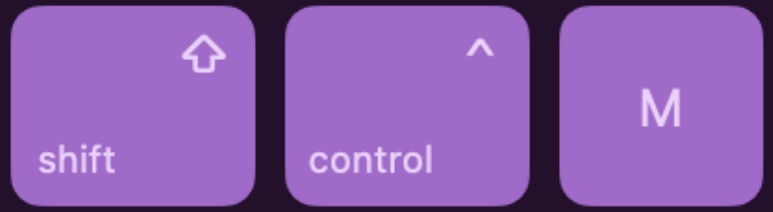
Ver tutorial

El conector o *pipe* `|>` (o también `%>%`) nos permite encadenar varias funciones de forma más legible.

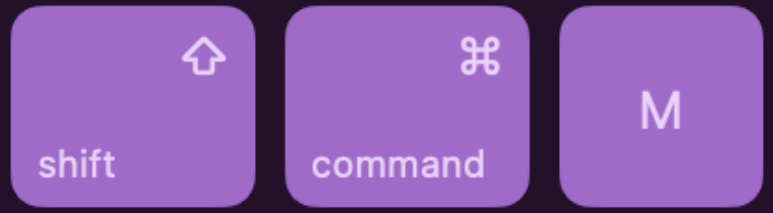
`|>` o `%>%`

Se escribe con:

Windows:



Mac:



Usando el conector

Ver tutorial

Aplicar una función normalmente:

```
funcion(objeto) # a la función le pasamos un objeto
```

Otra forma de hacerlo con `|>` es:

```
objeto |> funcion() # al objeto luego le aplicamos la función
```

Se lee: *al `objeto` le aplicamos la `funcion()`*

Luego podemos encadenar varias funciones:

```
objeto |> funcion() |> funcion() # al objeto le aplicamos dos funciones
```

También podemos ordenar las instrucciones hacia abajo:

```
1 resultado <- objeto |>
2   funcion() |>
3   funcion()
```

Seleccionar y ordenar

Ver tutorial

Usaremos los datos de **pobreza** del Ministerio Desarrollo Social y Familia que vimos antes. Si no los cargaste, descárgalos y los cargas así:

```
1  library(dplyr)
2  library(readxl)
3
4  datos <- read_excel("datos/pobreza/estimaciones_pobreza.xlsx")
```

Seleccionar

```
1  datos |>
2    select(region, comuna, personas) |>
3    head()
```

```
# A tibble: 6 × 3
  region   comuna        personas
  <chr>    <chr>            <dbl>
1 Tarapacá Iquique         41967.
2 Tarapacá Alto Hospicio   45162.
3 Tarapacá Pozo Almonte     4563.
4 Tarapacá Camiña            308.
5 Tarapacá Colchane          473.
6 Tarapacá Huara            1185.
```

Ordenar

```
1  datos |>
2    select(region, comuna, personas) |>
3    arrange(desc(personas)) |>
4    head()
```

```
# A tibble: 6 × 3
  region        comuna       personas
  <chr>         <chr>           <dbl>
1 Metropolitana Puente Alto   125235.
2 Metropolitana Santiago       87355.
3 Metropolitana Maipú          86976.
4 Antofagasta   Antofagasta    73103.
5 Metropolitana San Bernardo   64276.
6 Valparaíso    Valparaíso     60901.
```

¿Qué estamos haciendo?

Al aplicar una función sobre un *dataframe*, por ejemplo, `select()` para seleccionar columnas, **no estamos modificando los datos**, sino que estamos **trabajando con una copia** de los datos para no afectar los datos originales.

Todo lo que hagamos con `{dplyr}` serán **pruebas** y **exploración** de los datos, hasta que **decidamos** que los cambios que hagamos deben guardarse, así que entonces los **asignamos** a un **objeto nuevo**.

```
# prueba sin consecuencias
datos |>
  select(comuna, personas)

# creamos un nuevo objeto a partir de los datos originales
datos_filtrados <- datos |>
  select(comuna, personas)
```

Como asignamos los resultados a un **objeto nuevo**, tampoco estamos modificando ni afectando los datos originales. Así mantenemos un proceso **reproducible!**

Comparaciones

Podemos usar cifras u objetos para compararlas entre sí: la respuesta será `TRUE` (verdadero) o `FALSE` (falso).

```
1  edad >= 18
```

```
[1] TRUE
```

```
1  minimo <- 35
2  edad > minimo
```

```
[1] FALSE
```

```
1  año == 2024
```

```
[1] FALSE
```

```
1  año != 2024
```

```
[1] TRUE
```

Las comparaciones son el principio que luego nos permitirá filtrar datos, crear variables, y más!

Comparaciones sobre vectores

También podemos realizar **comparaciones** sobre los valores de un vector:

```r
edades <- c(54, 34, 65, 21, 32)
edades > 40
```

```
[1]  TRUE FALSE  TRUE FALSE FALSE
```

En el fondo, cuando filtremos datos, las **columnas** de un *dataframe* serán **vectores**, y las comparaciones se harán sobre cada uno de los valores de la columna, entregando un resultado `TRUE` o `FALSE` para cada fila, que luego usaremos para quedarnos sólo con las filas que cumplen la condición que queremos.

Filtrar datos

Ver tutorial

- Para filtrar, realizamos una **comparación** entre los valores de una columna y un valor específico.

```
1  library(dplyr)
2
3  datos |>
4    filter(porcentaje > 0.35) |>
5    arrange(desc(porcentaje)) |>
6    head()
```

```
# A tibble: 6 × 10
  codigo region         comuna personas_proy personas porcentaje limite_inferior
  <chr>  <chr>          <chr>          <dbl>    <dbl>      <dbl>           <dbl>
1 15202  Arica y Parin… Gener…           668     392.      0.586           0.435
2 9116   La Araucanía   Saave…         12717    5654.      0.445           0.361
3 9204   La Araucanía   Ercil…          8422    3503.      0.416           0.361
4 9106   La Araucanía   Galva…         12594    5029.      0.399           0.313
5 8314   Biobío         Alto …          6784    2640.      0.389           0.319
6 10306  Los Lagos      San J…          7521    2923.      0.389           0.314
# ℹ 3 more variables: limite_superior <dbl>, casen <chr>, tipo_estimacion <chr>
```

Funciones comunes

Algunas de las funciones más comunes de usar en `{dplyr}`, que son también las que usaremos para interactuar con los datos.

- `head()`, `tail()`, `glimpse()`: vistazos a los datos
- `select()`: seleccionar columnas o variables de una tabla
- `slice()`: extraer filas de una tabla por su posición o distintos criterios
- `pull()`: extraer una columna como vector
- `filter()`: filtrar observaciones en base a condiciones personalizadas
- `distinct()`: filtrar observaciones repetidas en una o más columnas
- `count()`: conteo de casos únicos en una o más columnas
- `mutate()`: crear columnas entregando valores, funciones sobre columnas, o calculando a partir de otras columnas

Datos de texto

El paquete `{stringr}` se especializa en texto.


```
1 library(stringr)
```

```
1 texto <- "todxs pueden aprender a programar 💕"
2
3 str_detect(texto, "todos")
```

```
[1] FALSE
```

```
1 str_detect(texto, "aprender")
```

```
[1] TRUE
```

```
1 datos |>
2   filter(str_detect(comuna, "Alto")) |>
3   select(region, comuna)
```

```
# A tibble: 4 × 2
  region       comuna
  <chr>        <chr>
1 Tarapacá     Alto Hospicio
2 Atacama      Alto Del Carmen
3 Biobío       Alto Biobío
4 Metropolitana Puente Alto
```

Crear/modificar variables

Ver tutorial

Con `mutate()` para crear una nueva variable o columna

Le aplicamos `mutate()` a un *data frame* conectando los datos a la función con un conector (`|>` o `%>%`)

```
1 datos |>
2   select(codigo, comuna) |>
3   mutate(saludo = "hola") |>
4   head()
```

```
# A tibble: 6 × 3
  codigo comuna        saludo
  <chr>  <chr>         <chr>
1 1101   Iquique       hola
2 1107   Alto Hospicio hola
3 1401   Pozo Almonte  hola
4 1402   Camiña        hola
5 1403   Colchane      hola
6 1404   Huara         hola
```

```
1 datos |>
2   select(comuna, personas) |>
3   mutate(miles = personas/1000) |>
4   head()
```

```
# A tibble: 6 × 3
  comuna        personas  miles
  <chr>            <dbl>  <dbl>
1 Iquique         41967. 42.0
2 Alto Hospicio   45162. 45.2
3 Pozo Almonte     4563.  4.56
4 Camiña            308.  0.308
5 Colchane          473.  0.473
6 Huara            1185.  1.18
```

Aplicar funciones a columnas

Ver tutorial

Las funciones realizan operaciones sobre todos los datos de una columna, **reemplazando** los datos existentes, o **rellenando** datos de una columna nueva.

Dependiendo de lo que queramos hacer, las funciones que apliquemos con `mutate()` pueden hacer referencia a **una o a varias** columnas:

```
datos |>
  select(comuna, personas) |>
  arrange(desc(personas)) |>
  head() |>
  mutate(total_personas = sum(personas)) |> # aplicada a una columna
  mutate(letras = nchar(comuna)) |> # aplicada a otra columna
  mutate(orden = row_number()) |> # hace algo por sí misma
  mutate(porcentaje = round(personas/total_personas, 2)) # función con argumento
```

```
# A tibble: 6 × 6
  comuna       personas total_personas letras orden porcentaje
  <chr>           <dbl>          <dbl>  <int> <int>      <dbl>
1 Puente Alto   125235.        497846.     11     1       0.25
2 Santiago       87355.        497846.      8     2       0.18
3 Maipú          86976.        497846.      5     3       0.17
4 Antofagasta    73103.        497846.     11     4       0.15
5 San Bernardo   64276.        497846.     12     5       0.13
6 Valparaíso     60901.        497846.     10     6       0.12
```

Variables condicionales

Ver tutorial

`ifelse()`

Si ocurre *esto*, **entonces** retorno *lo primero*, y **si no**, *lo segundo*.

```
1 datos |>
2   select(comuna, porcentaje) |>
3   mutate(nivel = ifelse(
4     porcentaje >= 0.3,
5     yes = "alta",
6     no = "baja")) |>
7   head()
```

```
# A tibble: 6 × 3
  comuna        porcentaje nivel
  <chr>              <dbl> <chr>
1 Iquique            0.183 baja
2 Alto Hospicio      0.326 alta
3 Pozo Almonte       0.250 baja
4 Camiña             0.223 baja
5 Colchane           0.300 alta
6 Huara              0.386 alta
```

`case_when()`

Recodificación avanzada con **múltiples condicionales**.

```
1 datos |>
2   select(comuna, porcentaje) |>
3   mutate(miles = case_when(
4     porcentaje >= 0.3 ~ "alta",
5     porcentaje >= 0.2 ~ "media",
6     porcentaje < 0.2 ~ "baja")) |>
7   head()
```

```
# A tibble: 6 × 3
  comuna        porcentaje miles
  <chr>              <dbl> <chr>
1 Iquique            0.183 baja
2 Alto Hospicio      0.326 alta
3 Pozo Almonte       0.250 media
4 Camiña             0.223 media
5 Colchane           0.300 alta
6 Huara              0.386 alta
```

Resúmenes de datos

Ver tutorial

Aplicar una **función** para **resumir** todas las filas de una tabla en un sólo resultado:

```
1 datos |>
2   summarize(total = sum(personas))
```

```
# A tibble: 1 × 1
     total
     <dbl>
1 3368245.
```

Si **agrupamos** los datos, obtendremos una fila de resultado por cada grupo:

```
1 datos |>
2   group_by(region) |>
3   summarize(total = sum(personas)) |>
4   head(n = 4)
```

```
# A tibble: 4 × 2
  region               total
  <chr>                <dbl>
1 Antofagasta        121379.
2 Arica y Parinacota  48086.
3 Atacama             64481.
4 Aysén               15113.
```

Resumiendo datos

Veamos los datos de educación del **Censo 2024**, que contienen la escolaridad promedio de cada comuna, por sexo.

Descargar datos de educación en Excel

```
educacion <- readxl::read_xlsx("datos/censo/educacion.xlsx")

educacion |>
  filter(region != "País") |>
  filter(sexo == "Total Comuna") |>
  group_by(region) |>
  summarize(escolaridad = mean(escolaridad),
            escolaridad_mayores_edad = mean(escolaridad_mayores_edad),
            n_comunas = n()) |>
  head(n = 4)
```

```
# A tibble: 4 × 4
  region                          escolaridad escolaridad_mayores_…¹ n_comunas
  <chr>                                 <dbl>                  <dbl>     <int>
1 Antofagasta                            9.77                   11.6         9
2 Arica y Parinacota                     8.6                     9.85        4
3 Atacama                                9.5                    11.1         9
4 Aysén del General Carlos Ibáñez …      9.45                   11.0        10
# ℹ abbreviated name: ¹escolaridad_mayores_edad
```

Cruce de datos o *left join*

Ver tutorial

Para cruzar dos tablas de datos
usamos `left_join()` de `{dplyr}`

tabla X

número	cifra	id
1	30	A
2	25	B
3	32	C

tabla Y

id	letra	nivel
A	Z	alto
B	X	alto
C	Y	bajo

resultado

número	cifra	id	letra	nivel
1	30	A	Z	alto
2	25	B	X	alto
3	32	C	Y	bajo

Descargar datos de clasificación comunal en Excel

Cruzando datos ⊞

Veamos un ejemplo donde cruzamos dos tablas, la de **pobreza** y una nueva de **clasificación comunal** (urbana, mixta, rural según la PNDR), a partir de la columna que tienen en común: `codigo`

```
clasificacion <- readr::read_csv2("datos/clasificacion/clasificacion.csv")
```

```
clasificacion_b <- clasificacion |>
  select(codigo, clasificacion)
```

```
datos_clasif <- datos |>
  select(codigo, comuna, personas, porcentaje) |>
  mutate(codigo = as.numeric(codigo)) |>
  left_join(clasificacion_b, by = "codigo")

datos_clasif |> head(n = 3)
```

```
# A tibble: 3 × 5
  codigo comuna        personas porcentaje clasificacion
   <dbl> <chr>            <dbl>      <dbl> <chr>
1   1101 Iquique         41967.      0.183 Urbana
2   1107 Alto Hospicio   45162.      0.326 Urbana
3   1401 Pozo Almonte     4563.      0.250 Rural
```

Transformar datos

Ver tutorial

```
1 library(tidyr)
```

- El paquete `{tidyr}` se especializa en **transformar la estructura de los datos**.
- Esto nos permite **manipular** los datos para que sea más cómodo o lógico realizar ciertas operaciones.
- Por ejemplo:
- Si necesitamos sumar varias variables, es más cómodo tenerlas en filas (para poder usar algo como `sum(valores)`) que en columnas (y tener que sumar `a+b+c+d`...)
- Si queremos comparar varias variables, o mostrarlas a un público, es mejor que estén lado a lado, como en las tablas comunes.

Ejemplo de transformación de datos

- **Pivotar** una tabla sirve para cambiar su forma:
- De variables en columnas a filas: `pivot_longer()`
- De variables en filas a columnas: `pivot_wider()`

formato ancho

país	2023	2024	2025
A	**100**	**110**	**120**
B	80	90	100
C	60	70	80

formato largo

país	nombre	valor
A	2023	**100**
A	2024	**110**
A	2025	**120**

formato largo

país	nombre	valor
A	2023	**100**
A	2024	**110**
A	2025	**120**

formato ancho

país	2023	2024	2025
A	**100**	**110**	**120**
B	80	90	100
C	60	70	80

Transformando datos

Veamos un ejemplo donde transformamos los datos de **estimaciones de población** del INE, desde un formato **ancho** (años en columnas) a uno **largo** (años en filas).

Descargar datos de estimaciones de población en Excel

```
library(readxl)

poblacion <- read_xlsx("datos/estimaciones/

poblacion |>
  select(1:6) |>
  head()
```

```
# A tibble: 6 × 6
  edad  sexo    `1992` `1993` `1994` `1995`
  <chr> <chr>    <dbl>  <dbl>  <dbl>  <dbl>
1 0     Hombres 151857 148533 146443 144016
2 1     Hombres 153970 151413 148142 146101
3 2     Hombres 153669 153837 151313 148047
4 3     Hombres 147481 153581 153800 151266
5 4     Hombres 140829 147469 153571 153785
6 5     Hombres 134445 140838 147461 153551
```

```
library(tidyr)

poblacion |>
  pivot_longer(cols = `1992`:`2070`,
               names_to = "año",
               values_to = "poblacion") |>
  head()
```

```
# A tibble: 6 × 4
  edad  sexo    año   poblacion
  <chr> <chr>   <chr>     <dbl>
1 0     Hombres 1992     151857
2 0     Hombres 1993     148533
3 0     Hombres 1994     146443
4 0     Hombres 1995     144016
5 0     Hombres 1996     139539
6 0     Hombres 1997     138013
```

Tutoriales de `{dplyr}`

Para que puedan repasar o guiarse con **casos básicos de uso** de `{dplyr}` con **datos reales**, dejo un par de tutoriales:

- Tutorial con datos censales
- Tutorial con datos sociales

Datos para practicar

Algunos conjuntos de datos limpios para practicar

- Clasificaciones comunales
- Estimaciones de población
- Escolaridad por comuna
- Población según género

{ggplot2}

- Paquete de **visualización de datos**
- Una de las soluciones más populares a nivel mundial
- Prodice gráficos por medio de **capas**
- Entre sus beneficios están su flexibilidad, reusabilidad y extensibilidad


```r
# install.packages("ggplot2")
library(ggplot2)
```

Ver tutorial

Gramática de gráficos

- **Datos:** variables disponibles para construir el gráfico
- **Mapeos:** conexión de variables a distintos aspectos de la visualización
- **Capas:** elementos geométricos agregados unos sobre otros
- **Escalas:** especificación de la forma en que se mapean las variables a los ejes
- **Facetas:** dividir la visualización en paneles distintos según una variable
- **Coordenadas:** proyección de las coordenadas en el plano
- **Temas:** definición de la apariencia general del gráfico, y específica para cada elemento

Gramática aplicada al código

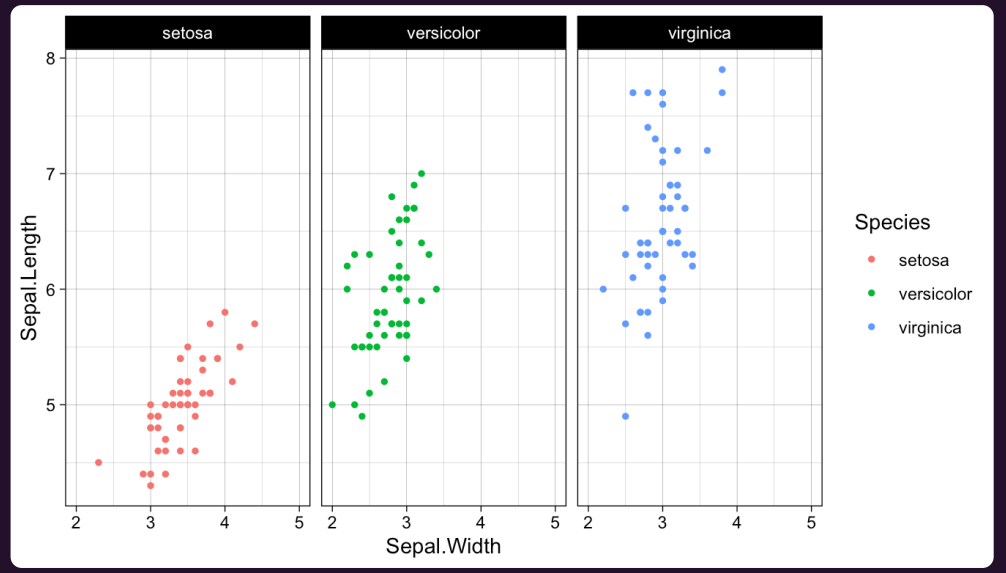
Ver tutorial

Conceptos que constituyen la **gramática** de los gráficos:

- Datos `línea 1`
- Mapeos `línea 2`
- Capas `línea 3`
- Escalas `línea 4`
- Facetas `línea 5`
- Coordenadas `línea 6`
- Temas `línea 7`

Conceptos de la gramática de gráficos aplicados al código:

```
1 ggplot(iris) + # datos
2   aes(Sepal.Width, Sepal.Length, color = Species) + #
3   geom_point(size = 1) + # capas
4   scale_color_discrete() + # escalas
5   facet_wrap(~Species) + # facetas
6   coord_cartesian(xlim = c(2, 5)) + # coordenadas
7   theme_linedraw() # temas
```

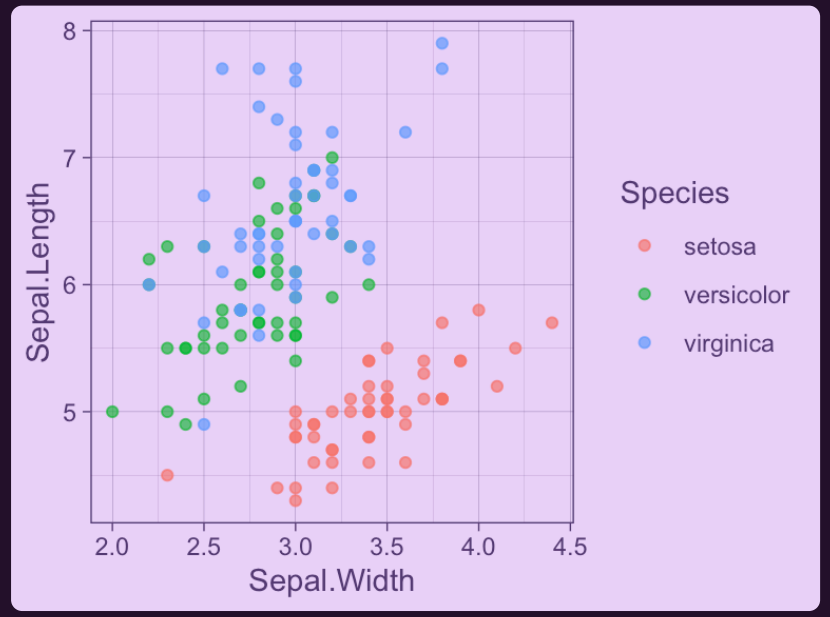


Colores del gráfico

```
grafico <- ggplot(iris) +
  aes(Sepal.Width, Sepal.Length, color = Species) +
  geom_point(alpha = 0.7)
```

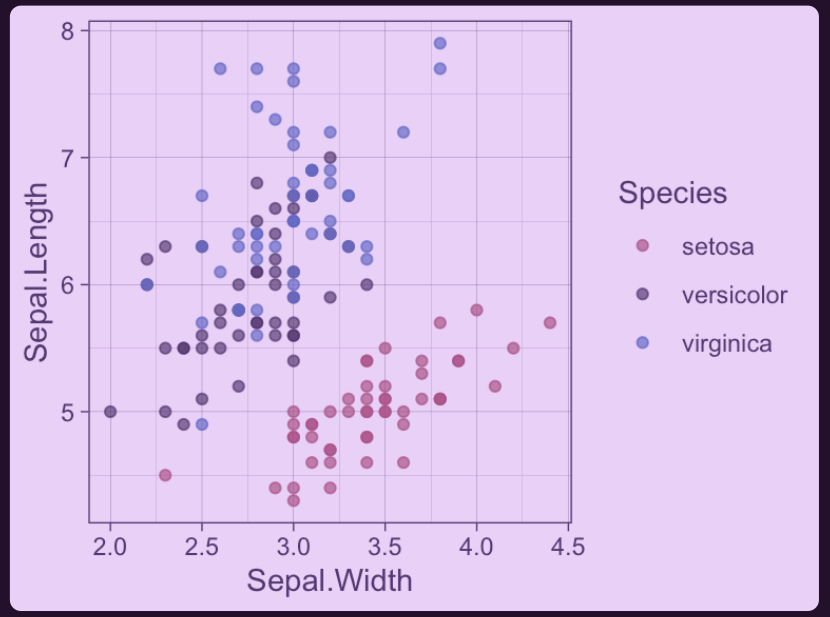
Colores generales del gráfico

```
grafico_tema <- grafico +
  theme_linedraw(paper = "#EAD2FA",
                 ink = "#553A74",
                 accent = "#9069C0")
grafico_tema
```



Colores de las categorías

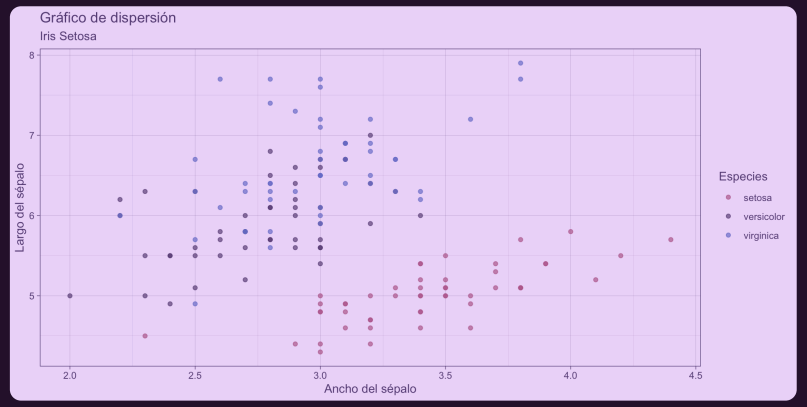
```
grafico_color <- grafico_tema +
  scale_color_discrete(
    palette = c("#AC558A", "#553A74",
                "#666BC7"))
grafico_color
```



Textos del gráfico

Podemos configurar y agregar textos del gráfico con la capa `labs()`:

```
1 grafico_textos <- grafico_color +
2   labs(x = "Ancho del sépalo",
3        y = "Largo del sépalo",
4        color = "Especies",
5        title = "Gráfico de dispersión",
6        subtitle = "Iris Setosa")
7
8 grafico_textos
```



Temas

Configuración completa de cualquier aspecto o elemento del gráfico.

Funciones de temas

Agregar un tema prediseñado que cambia la apariencia general del gráfico:

- `theme_minimal()`
- `theme_classic()`
- `theme_void()`
- `theme_linedraw()`

Componentes del gráfico

En `theme()` podemos cambiar aspectos específicos. Los principales empiezan con:

- `panel`
- `grid`
- `scales`
- `axis`
- `strip`
- `legend`

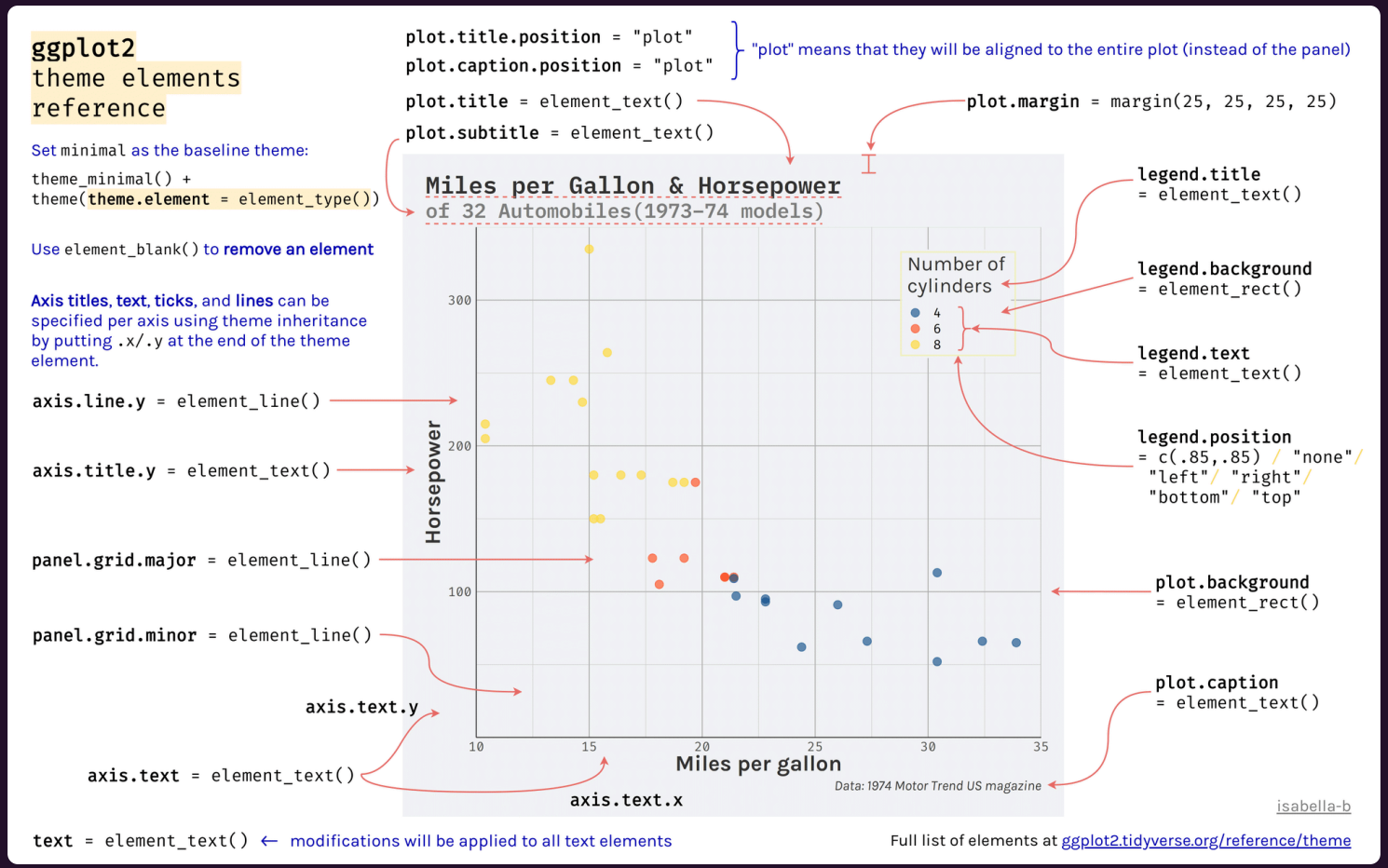
Tipos de componente

Cada aspecto del gráfico es de un **tipo** distinto, y cada tipo se configura mediante una función:

- `element_text()`, txtos
- `element_line()`, líneas
- `element_rect()`, rectángulos
- `element_blank()`, ocultar

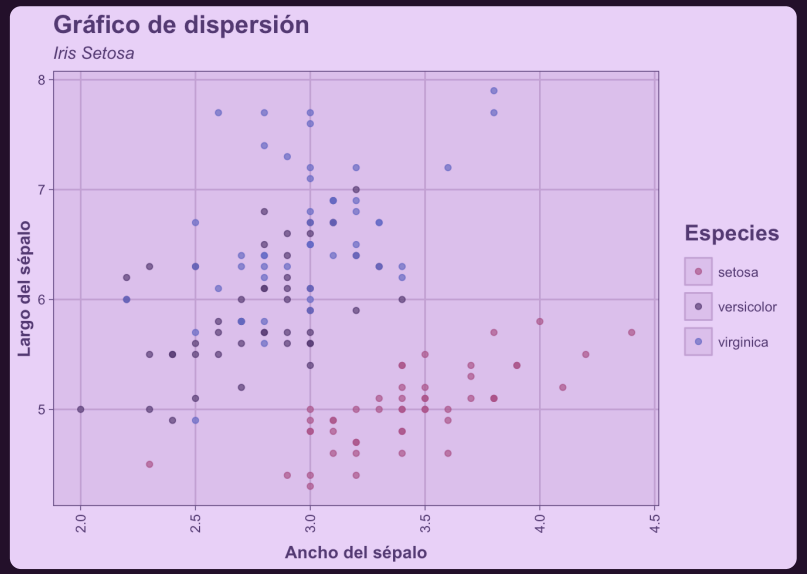
Referencia de temas

Hoja de referencia de los elementos de temas en `{ggplot2}`, fuente: Isabella Benabaye



Personalización de temas

```
grafico_textos +
  theme(plot.title = element_text(face = "
        plot.subtitle = element_text(famil
        panel.grid.major = element_line(li
        panel.grid.minor = element_blank()
        panel.background = element_rect(fi
        axis.title = element_text(face = "
        axis.title.x = element_text(margin
        axis.text.x = element_text(angle =
        legend.title = element_text(face =
        legend.text = element_text(margin
        legend.key = element_rect(fill = "
        legend.key.spacing.y = unit(1.8, "
```



Extensiones de `{ggplot2}`

Existen muchos paquetes que extienden las capacidades de `{ggplot2}`:

- `{ggforce}`: nuevas geometrías
- `{patchwork}`: combinación de gráficos
- `{ggrepel}`: separación de textos sobrepuestos
- `{gganimate}`: gráficos animados
- `{marquee}` y `{ggtext}`: textos con formato, color, y más
- `{ggiraph}` y `{plotly}`: gráficos interactivos

Acá hay una lista de extensiones de `ggplot2`

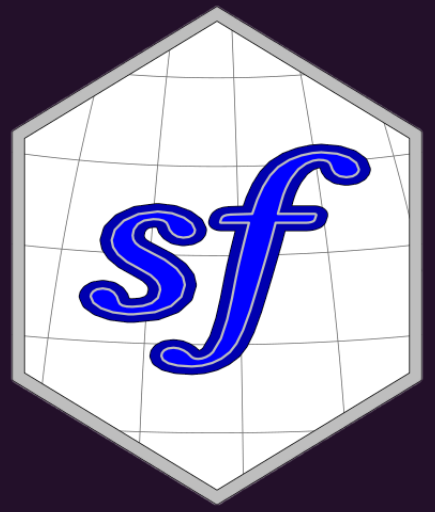
Recursos de aprendizaje

Tutorial de visualización de datos con `{ggplot2}`

- R Graph Gallery
- R Charts
- A ggplot2 Tutorial for Beautiful Plotting in R, tutorial de `{ggplot2}` en inglés

Mapas

- El paquete `{sf}` entrega herramientas para trabajar con datos geoespaciales en R
- Permite representar características espaciales en *data frames*
- Se complementa con `{ggplot2}`




```r
# install.packages("sf")
library(sf)
```

Ver tutorial

Mapas de Chile

Ver tutorial

El paquete `{chilemapas}` ofrece funciones que simplifican la obtención de mapas de Chile en R.

```r
# install.packages("chilemapas")
library(chilemapas)

mapa_comunas |>
  filter(codigo_region == "14")
```

```
# A tibble: 12 × 4
   codigo_comuna codigo_provincia codigo_region                    geometry
   <chr>         <chr>            <chr>                  <MULTIPOLYGON [°]>
 1 14203         142              14          (((-71.65597 -40.35386, -71.657…
 2 14204         142              14          (((-71.88915 -40.55144, -71.891…
 3 14201         142              14          (((-73.15077 -40.06596, -73.147…
 4 14202         142              14          (((-71.81459 -40.08101, -71.820…
 5 14104         141              14          (((-72.02965 -39.93203, -72.034…
 6 14105         141              14          (((-72.61245 -39.62475, -72.607…
 7 14103         141              14          (((-72.32959 -39.51024, -72.327…
 8 14106         141              14          (((-72.83588 -39.41328, -72.841…
 9 14107         141              14          (((-72.92395 -39.94016, -72.916…
10 14108         141              14          (((-71.81459 -40.08101, -71.815…
11 14102         141              14          (((-73.39203 -39.88653, -73.389…
12 14101         141              14          (((-73.2668 -39.88605, -73.2621…
```

Visualizando datos en un mapa

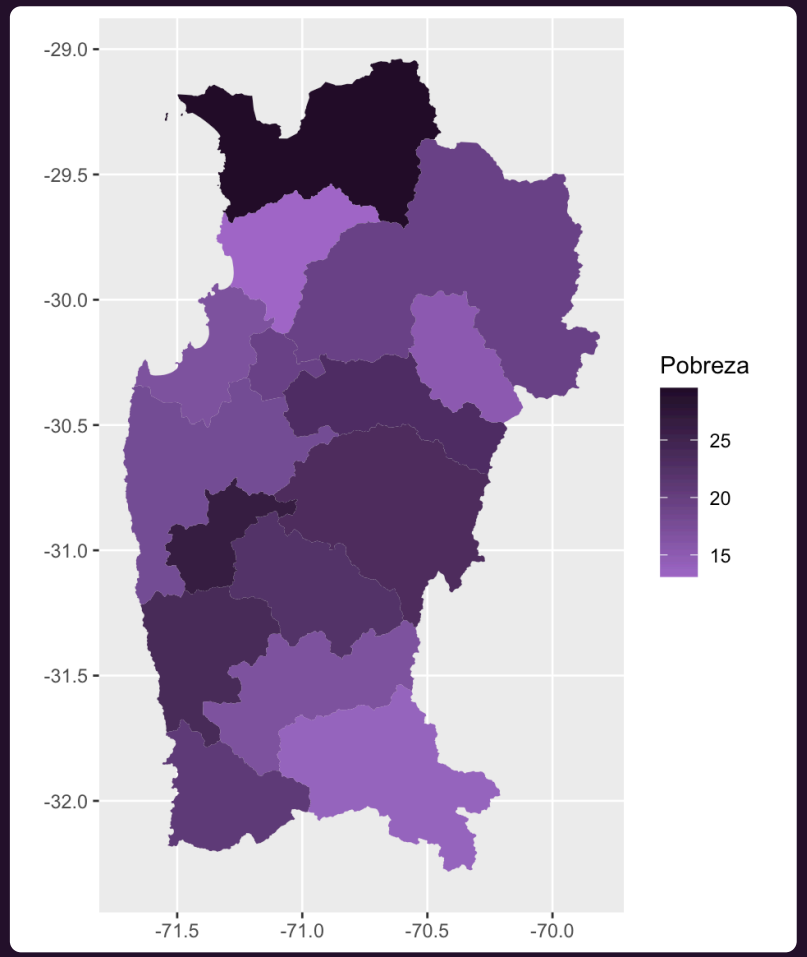
Sólo basta con cruzar una tabla con las geometrías

```r
# cargar datos de pobreza por comuna
pobreza <- read_xlsx("datos/pobreza/estima
  mutate(codigo = as.numeric(codigo))

# filtrar mapa en una región específica
mapa_region <- mapa_comunas |>
  filter(codigo_region == "04") |>
  mutate(codigo = as.numeric(codigo_comuna

# unir el mapa con los datos
mapa_datos <- mapa_region |>
  left_join(pobreza, by = "codigo")

# visualizar
mapa_datos |>
  ggplot() +
  aes(geometry = geometry,
      fill = porcentaje * 100) +
  geom_sf(linewidth = 0) +
  scale_fill_gradient(low = "#9F69C7", hig
  labs(fill = "Pobreza")
```



Datos para practicar

- Estimaciones de población Censo
- Variables educacionales Censo 2024
- Estimación de pobreza multidimensional 2022
- Clasificaciones comunales 2024

Puedes encontrar más datos en mi repositorio de datos sociales, en el banco integrado de datos del Ministerio de Desarrollo, en los datos abiertos del Estado Chile, y muchos más.

Ahora que ya sabes lo básico, todo lo demás se trata de aprender piezas y herramientas que puedes ir agregando a tus análisis! 🧩🛠️

Consejos

- Mantengan su propio **libro de aprendizajes** 📝
- Lean bien los **errores** y busquen su significado
- Si algo sale mal, prueben **paso a paso**
- También pueden devolverse y **empezar de nuevo**
- Dense una vuelta y a veces la respuesta llega solita 😌
- No se vuelvan dependientes de la IA 😵
- Si quieren usar IA:
- **Autocompletado** de GitHub Copilot
- **Mensaje de sistema** para su proveedor de IA
- Aprendendan a **buscar** bien en internet
- Compartan lo que hacen y socialicen sus aprendizajes! ✨

Gracias ♥

Puedes escribirme cualquier duda o comentario por aquí

Aprende R | Chat grupal

Curso | Código | Grabaciones