Clase 2: Tidyverse

Lucía Babino

Tidyverse

Basado en la clase de Juan Barriola, Fernando González y Franco Mastelli

Cuestiones administrativas

- Mail para consultas sobre esta clase y las de causalidad:

lbabinog@gmail.com (Lucía Babino)

Curso on line de Análisis y visualización de datos con R (de Lucía Babino)

- Clase 3: Manipulación, análisis y exportación de datos con tidyverse
- Clase 4: Visualización de datos con ggplot2

(links a todas las clases en el github)

¿Qué es tidyverse?

`tidyverse`: paquete *umbrella* que engloba muchos paquetes con funciones útiles para manipular y organizar datasets y realizar gráficos.

Paquetes en R

- **Paquete:** conjunto de funciones, datos y documentación organizada que extiende las capacidades básicas del lenguaje R.

- R tiene un paquete Base que está disponible al iniciar sesión en R, para usar funciones de otros paquetes, primero debemos *instalarlos* y *cargarlos*.

Cómo usar un paquete

1. **Instalarlo**
   - `install.packages("nombre_paquete")`
   - descarga el paquete de internet y lo guarda en la computadora
   - se hace una sola vez

2. **Cargarlo**
   - `library(nombre_paquete)`
   - carga el paquete en la sesión de R, lo que nos permite usar sus funciones y herramientas.
   - se hace en cada sesión de trabajo

Instalación y carga de paquetes en R

Ejemplo de instalación y carga del paquete “tidyverse”

```r
# Instalar (una sola vez)
install.packages("tidyverse")

# Cargar (cada vez que abrís R)
library(tidyverse)
```

Tidyverse

Tidyverse es un paquete "umbrella" que agrupa una serie de paquetes que tienen una misma lógica. Todos ellos quedan disponibles al instalar y cargar `tydiverse`.

Hoy veremos:

- **magrittr**: permite escribir código más limpio y legible
- **dplyr**: manipular y transformar datos
- **tidyr**: organizar y reestructurar datos en formato ordenado (tidy).
- **lubridate**: para trabajar con datos tipo fecha
- **stringr**: para trabajar con datos tipo string
- **ggplot2**: para realizar gráficos

Operador pipe (de `magrittr`)

Indice de salarios del INDEC

A lo largo de esta clase usaremos parte de un conjunto de datos real de `Indice de salarios del INDEC` que contine las variables:

- `FECHA`: mes y año
- `GRUPO`: sector (Privado registrado, Privado no registrado o Público)
- `INDICE`: estimación del salario medio

Más información en la página del indec

```
FECHA  <-  c("Mar-20", "Mar-20", "Mar-20",
             "Abr-20", "Abr-20", "Abr-20",
             "May-20", "May-20", "May-20")

GRUPO  <-  c("Privado_Registrado","Público","Privado_No_Registrado",
             "Privado_Registrado","Público","Privado_No_Registrado",
             "Privado_Registrado","Público","Privado_No_Registrado")
INDICE  <- c( 286.4,    262.1,  248.5,
              285.7,    263.5,  250.2,
              285.1,    264.9,  249.0 )

Datos <- data.frame(FECHA, GRUPO, INDICE)
```

Indice de salarios del INDEC

```r
FECHA  <-  c("Mar-20", "Mar-20", "Mar-20",
             "Abr-20", "Abr-20", "Abr-20",
             "May-20", "May-20", "May-20")

GRUPO  <-  c("Privado_Registrado","Público","Privado_No_Registrado",
             "Privado_Registrado","Público","Privado_No_Registrado",
             "Privado_Registrado","Público","Privado_No_Registrado")
INDICE  <- c( 286.4,    262.1,  248.5,
              285.7,    263.5,  250.2,
              285.1,    264.9,  249.0 )

Datos <- data.frame(FECHA, GRUPO, INDICE)
Datos
```

```
   FECHA                  GRUPO INDICE
1 Mar-20    Privado_Registrado  286.4
2 Mar-20               Público  262.1
3 Mar-20 Privado_No_Registrado  248.5
4 Abr-20    Privado_Registrado  285.7
5 Abr-20               Público  263.5
6 Abr-20 Privado_No_Registrado  250.2
7 May-20    Privado_Registrado  285.1
```

El operador pipe %>%

- Sirve para **encadenar funciones** de forma legible.
- Atajo de teclado:
  - Ctrl + Shift + M (Windows)
  - Cmd + Shift + M (Mac)

El operador pipe %>%

- Toma el objeto a la izquierda y lo pasa como primer argumento de la función a la derecha.
- Ejemplo: si $f(x, y)$ es una función con dos argumentos

```
x %>% f(.,y)
```

es equivalente a

```
f(x,y)
```

- Particularmente útil cuando queremos encadenar varias funciones.
- Veremos cómo usarlo a lo largo de la clase.

Paquete Dplyr

Útil para manipular y transformar datos

`glimpse()`

Permite ver la estructura de la tabla. Nos muestra:

- número de filas
- número de columnas
- nombre de las columnas
- tipo de dato de cada columna
- las primeras observaciones de cada columna

glimpse(): ejemplo

```
1 library(tidyverse)
2 glimpse(Datos)
```

```
Rows: 9
Columns: 3
$ FECHA  <chr> "Mar-20", "Mar-20", "Mar-20", "Abr-20", "Abr-
20", "Abr-20", "Ma…
$ GRUPO  <chr> "Privado_Registrado", "Público",
"Privado_No_Registrado", "Priv…
$ INDICE <dbl> 286.4, 262.1, 248.5, 285.7, 263.5, 250.2,
285.1, 264.9, 249.0
```

glimpse() con pipe

```
1 Datos %>%
2   glimpse()
```

```
Rows: 9
Columns: 3
$ FECHA  <chr> "Mar-20", "Mar-20", "Mar-20", "Abr-20", "Abr-
20", "Abr-20", "Ma…
$ GRUPO  <chr> "Privado_Registrado", "Público",
"Privado_No_Registrado", "Priv…
$ INDICE <dbl> 286.4, 262.1, 248.5, 285.7, 263.5, 250.2,
285.1, 264.9, 249.0
```

filter()

Queremos filtrar las observaciones (filas) de `Datos` correspondientes a **abril de 2020**.

```
1 Datos
```

```
   FECHA                    GRUPO INDICE
1 Mar-20    Privado_Registrado  286.4
2 Mar-20               Público  262.1
3 Mar-20 Privado_No_Registrado  248.5
4 Abr-20    Privado_Registrado  285.7
5 Abr-20               Público  263.5
6 Abr-20 Privado_No_Registrado  250.2
7 May-20    Privado_Registrado  285.1
8 May-20               Público  264.9
9 May-20 Privado_No_Registrado  249.0
```

filter()

Con paquete base

```
1  Datos[Datos$FECHA == "Abr-20", ]
```

Con tidyverse

```
1  filter(Datos, FECHA == "Abr-20")
```

```
   FECHA                      GRUPO INDICE
1 Abr-20    Privado_Registrado  285.7
2 Abr-20               Público  263.5
3 Abr-20 Privado_No_Registrado  250.2
```

`filter()` con pipe

Sin **pipe**

```
1 filter(Datos, FECHA == "Abr-20")
```

Con **pipe**

```
1 Datos %>%
2   filter(FECHA == "Abr-20")
```

```
  FECHA                      GRUPO INDICE
1 Abr-20    Privado_Registrado  285.7
2 Abr-20              Público  263.5
3 Abr-20 Privado_No_Registrado  250.2
```

`filter()` con varias condiciones

Queremos filtramos las filas con

`INDICE > 250` y `GRUPO = "Privado_Registrado"`

```
1 Datos %>%
2   filter(INDICE > 250, GRUPO == "Privado_Registrado")
```

```
   FECHA                GRUPO INDICE
1 Mar-20 Privado_Registrado  286.4
2 Abr-20 Privado_Registrado  285.7
3 May-20 Privado_Registrado  285.1
```

`filter()` con varias condiciones

Queremos filtramos las filas con

`INDICE > 250` **ó** `GRUPO = "Privado_Registrado"`

```
1 Datos %>%
2   filter(INDICE > 250 | GRUPO == "Privado_Registrado")
```

```
   FECHA                    GRUPO INDICE
1 Mar-20     Privado_Registrado  286.4
2 Mar-20               Público  262.1
3 Abr-20     Privado_Registrado  285.7
4 Abr-20               Público  263.5
5 Abr-20 Privado_No_Registrado  250.2
6 May-20     Privado_Registrado  285.1
7 May-20               Público  264.9
```

rename()

- Renombra una columna de la tabla.
- Sintaxis:

```
Data %>% rename(nuevo_nombre = viejo_nombre)
```

- Ejemplo:

```
1 Datos %>%
2   rename(Periodo = FECHA)
```

```
  Periodo                    GRUPO INDICE
1  Mar-20    Privado_Registrado  286.4
2  Mar-20              Público  262.1
```

3	Mar-20	Privado_No_Registrado	248.5
4	Abr-20	Privado_Registrado	285.7
5	Abr-20	Público	263.5
6	Abr-20	Privado_No_Registrado	250.2
7	May-20	Privado_Registrado	285.1

rename()

Si quiero que el cambio quede guardado en `Datos`

```
1 Datos <- Datos %>%
2   rename(Periodo = FECHA)
3 Datos
```

```
   Periodo                      GRUPO INDICE
1   Mar-20    Privado_Registrado  286.4
2   Mar-20               Público  262.1
3   Mar-20 Privado_No_Registrado  248.5
4   Abr-20    Privado_Registrado  285.7
5   Abr-20               Público  263.5
6   Abr-20 Privado_No_Registrado  250.2
7   May-20    Privado_Registrado  285.1
8   May-20               Público  264.9
```

mutate()

- Crea **nuevas variables** a partir de las existentes.
- Ejemplo

```
1  Datos2 <- Datos %>%
2    mutate(Doble = INDICE * 2)
3  Datos2
```

```
   Periodo                          GRUPO INDICE
Doble
1   Mar-20     Privado_Registrado   286.4
572.8
2   Mar-20                Público   262.1
524.2
```

```
3   Mar-20 Privado_No_Registrado   248.5
497.0
```

`select()`

- Así como `filter()` permite seleccionar determinadas filas, `select()` permite seleccionar determinadas columnas.

- Ejemplo: queremos quedarnos con las columnas `Periodo` e `INDICE`.

```
1 Datos
```

```
  Periodo                    GRUPO INDICE
1  Mar-20     Privado_Registrado  286.4
2  Mar-20                Público  262.1
3  Mar-20 Privado_No_Registrado  248.5
```

```
4   Abr-20    Privado_Registrado  285.7
5   Abr-20                Público  263.5
6   Abr-20 Privado_No_Registrado  250.2
7   May-20    Privado_Registrado  285.1
```

`select()`: ejemplo

Queremos quedarnos con las columnas `Periodo` e `INDICE`.

```
1 Datos %>%
2   select(Periodo, INDICE)
```

```
  Periodo INDICE
1  Mar-20  286.4
2  Mar-20  262.1
3  Mar-20  248.5
4  Abr-20  285.7
5  Abr-20  263.5
6  Abr-20  250.2
7  May-20  285.1
8  May-20  264.9
```

`select()`: ejemplo

Otra forma de quedarnos con las columnas `Periodo` e `INDICE` es "tirar" la columna GRUPO.

```
1 Datos %>%
2   select(-GRUPO)
```

```
  Periodo INDICE
1  Mar-20  286.4
2  Mar-20  262.1
3  Mar-20  248.5
4  Abr-20  285.7
5  Abr-20  263.5
6  Abr-20  250.2
7  May-20  285.1
```

`case_when()`

- Sirve para recodificar una variable según ciertas **condiciones lógicas**.
- Por ejemplo, es útil para categorizar una variable numérica.
- Se suele combinar con `mutate()`.
- veamos cuál es la lógica de su **sintaxis**.

`case_when()`: sintaxis

```
case_when(
  condición_1 ~ valor_1,
  condición_2 ~ valor_2,
  ...
  TRUE        ~ valor_por_defecto
)
```

Idea: crea una nueva variable que toma los valores que estan a la derecha del `~`, según cuál de las condiciones de la izquierda se cumpla.

Formalmente...

- Se leen las condiciones **de arriba hacia abajo**.
- La **primera condición verdadera** determina el valor asignado.
- Se puede agregar una última condición `TRUE` para asignar un **valor por defecto** a la variable en el caso en que no se cumpla ninguna de las condiciones anteriores.

`case_when()`: ejemplo

Queremos crear una variable `indice_cat` que valga:

- **Bajo** si INDICE < 250
- **Medio** si 250<= INDICE < 270
- **Alto** si INDICE >= 270.

```
1 Datos
```

	Periodo	GRUPO	INDICE
1	Mar-20	Privado_Registrado	286.4
2	Mar-20	Público	262.1
3	Mar-20	Privado_No_Registrado	248.5
4	Abr-20	Privado_Registrado	285.7

```
5   Abr-20                    Público  263.5
6   Abr-20 Privado_No_Registrado  250.2
7   May-20    Privado Registrado  285.1
```

case_when(): ejemplo

```r
Datos <- Datos %>%
  mutate(indice_cat = case_when(
    INDICE < 250 ~ "Bajo",
    INDICE < 270 ~ "Medio",
    TRUE         ~ "Alto"
    ))
Datos %>% select(INDICE, indice_cat)
```

```
  INDICE indice_cat
1  286.4       Alto
2  262.1      Medio
3  248.5       Bajo
4  285.7       Alto
```

```
5   263.5       Medio
6   250.2       Medio
7   285.1        Alto
```

Se evalúan las condiciones en orden y se asigna el valor correspondiente a la primera condición verdadera.

case_when(): ejemplo sin TRUE

```
1 Datos <- Datos %>%
2   mutate(indice_cat = case_when(
3     INDICE < 250 ~ "Bajo",
4     INDICE < 270 ~ "Medio"
5     ))
6 Datos %>% select(INDICE, indice_cat)
```

```
  INDICE indice_cat
1  286.4       <NA>
2  262.1      Medio
3  248.5       Bajo
4  285.7       <NA>
5  263.5      Medio
```

```
6   250.2        Medio
7   285.1         <NA>
```

Eliminamos `indice_cat`

Antes de seguir, eliminemos `indice_cat` del data set

```
1  Datos <- Datos %>%
2    select(-indice_cat)
```

arrange()

Ordena una tabla según los valores de una o más variables.

```
1 Datos <- Datos %>%
2   arrange(GRUPO, INDICE)
3 Datos
```

```
   Periodo                      GRUPO INDICE
1   Mar-20 Privado_No_Registrado  248.5
2   May-20 Privado_No_Registrado  249.0
3   Abr-20 Privado_No_Registrado  250.2
4   May-20    Privado_Registrado  285.1
5   Abr-20    Privado_Registrado  285.7
6   Mar-20    Privado_Registrado  286.4
7   Mar-20               Público  262.1
8   Abr-20               Público  263.5
```

arrange()

Por defecto, `arrange()` ordena en forma ascendente.

Si queremos ordenar alguna variable en forma descendente, usamos `desc()`.

```
1 Datos <- Datos %>%
2   arrange(GRUPO, desc(INDICE))
3 Datos
```

```
   Periodo                        GRUPO INDICE
1   Abr-20 Privado_No_Registrado  250.2
2   May-20 Privado_No_Registrado  249.0
3   Mar-20 Privado_No_Registrado  248.5
4   Mar-20    Privado_Registrado  286.4
```

5	Abr-20	Privado_Registrado	285.7
6	May-20	Privado_Registrado	285.1
7	May-20	Público	264.9

`group_by()` + `summarise()`

- La combinación de `group_by()` y `summarise()` nos permite **calcular medidas resumen por grupo**.

(por ejemplo, podemos obtener el promedio del `INDICE` para cada sector).

- `group_by()` agrupa la tabla según una o más variables.
- `summarise()` calcula valores resumen (como promedio, suma, mínimo, etc.) dentro de cada grupo.

Promedio del INDICE por GRUPO

```
1 Datos %>%
2   group_by(GRUPO) %>%
3   summarise(promedio_indice = mean(INDIC
```

```
# A tibble: 3 × 2
  GRUPO                  promedio_indice
  <chr>                            <dbl>
1 Privado_No_Registrado             249.
2 Privado_Registrado                286.
3 Público                           264.
```

Promedio del INDICE por GRUPO y Periodo

```
1 Datos %>%
2   group_by(Periodo, GRUPO) %>%
3   summarise(promedio_indice = mean(INDICE))
```

```
# A tibble: 9 × 3
# Groups:   Periodo [3]
  Periodo GRUPO                  promedio_indice
  <chr>   <chr>                            <dbl>
1 Abr-20  Privado_No_Registrado            250.
2 Abr-20  Privado_Registrado                286.
3 Abr-20  Público                          264.
4 Mar-20  Privado_No_Registrado            248.
5 Mar-20  Privado_Registrado               286.
6 Mar-20  Público                          262.
7 May-20  Privado_No_Registrado            249
8 May-20  Privado_Registrado               285.
```

Múltiples estadísticas por grupo

```r
Datos %>%
  group_by(GRUPO) %>%
  summarise(
    promedio = mean(INDICE),
    minimo   = min(INDICE),
    maximo   = max(INDICE),
    desv     = sd(INDICE)
  )
```

```
# A tibble: 3 × 5
  GRUPO                promedio minimo maximo  desv
  <chr>                   <dbl>  <dbl>  <dbl> <dbl>
1 Privado_No_Registrado    249.   248.   250. 0.874
2 Privado_Registrado       286.   285.   286. 0.651
3 Público                  264.   262.   265. 1.40
```

Supongamos que queremos ver qué grupo tiene mayor salario medio.

Ordenar con arrange()

```r
Datos %>%
  group_by(GRUPO) %>%
  summarise(
    promedio = mean(INDICE),
    minimo   = min(INDICE),
    maximo   = max(INDICE),
    desv     = sd(INDICE)) %>%
  arrange(desc(promedio))
```

```
# A tibble: 3 × 5
  GRUPO                promedio minimo maximo  desv
  <chr>                   <dbl>  <dbl>  <dbl> <dbl>
1 Privado_Registrado       286.   285.   286. 0.651
2 Público                  264.   262.   265. 1.40
3 Privado_No_Registrado    249.   248.   250. 0.874
```

Joins

- Otra implementación muy importante del paquete dplyr son las funciones para unir tablas (**joins**).
- Veamos un ejemplo de la función **left_join** (una de las más utilizadas en la práctica).
- Más tipos de joins en el apunte del github.

`left_join()`

- Une dos tablas tomando como referencia una o más columnas comunes.

- Para ejemplificar su uso crearemos un data frame que contenga un **Ponderador** para cada uno de los **Grupos** del data frame `Datos`.

Data frame Ponderadores

```r
Ponderadores <- data.frame(
  GRUPO = c("Privado_Registrado",
            "Público",
            "Privado_No_Registrado"),
  PONDERADOR = c(50.16, 29.91, 19.93))
Ponderadores
```

```
                    GRUPO PONDERADOR
1     Privado_Registrado       50.16
2                Público       29.91
3 Privado_No_Registrado       19.93
```

left_join(): ejemplo

```
1 Ponderadores
```

```
                  GRUPO PONDERADOR
1    Privado_Registrado      50.16
2               Público      29.91
3 Privado_No_Registrado      19.93
```

```
1 Datos %>%
2   left_join(Ponderadores, by = "GRUPO") %>%
3   head(5)
```

```
  Periodo                  GRUPO INDICE PONDERADOR
1  Abr-20 Privado_No_Registrado  250.2      19.93
2  May-20 Privado_No_Registrado  249.0      19.93
3  Mar-20 Privado_No_Registrado  248.5      19.93
4  Mar-20    Privado_Registrado  286.4      50.16
5  Abr-20    Privado_Registrado  285.7      50.16
```

Unimos las tablas `Datos` y `Ponderadores` según la **columna** `GRUPO`, conservando todas las filas de la tabla de la izquierda `Datos` y agregando el valor de la columna `PONDERADOR` en aquellas filas donde los valores de `GRUPO` coinciden.

`left_join()`: generalización

`left_join()` se utiliza para **unir dos tablas** según una o más **columnas en común** (especificadas en `by`), conservando todas las filas de la tabla de la izquierda y agregando las columnas de la tabla de la derecha en aquellas filas donde los valores de las columnas especificadas en `by` coinciden.

```
1 Ponderadores
```

```
                 GRUPO PONDERADOR
1    Privado_Registrado      50.16
2              Público      29.91
3 Privado_No_Registrado      19.93
```

```
1 Datos %>%
2   left_join(Ponderadores, by = "GRUPO")
```

```
  Periodo                 GRUPO INDICE PONDERADOR
1  Abr-20 Privado_No_Registrado  250.2      19.93
2  May-20 Privado_No_Registrado  249.0      19.93
3  Mar-20 Privado_No_Registrado  248.5      19.93
4  Mar-20    Privado_Registrado  286.4      50.16
```

5	Abr-20	Privado_Registrado	285.7	50.16
6	May-20	Privado_Registrado	285.1	50.16
7	May-20	Público	264.9	29.91
8	Abr-20	Público	263.5	29.91
9	Mar-20	Público	262.1	29.91

Más Joins

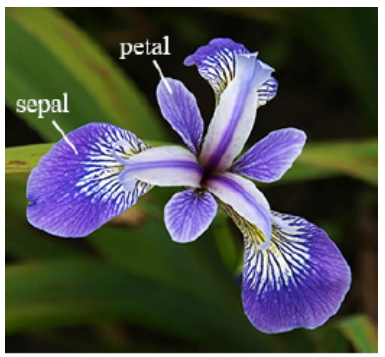
Para más detalle sobre joins, ver notebook opcional de distintos tipos de joins.

Tidyr

- El paquete **tidyr** esta pensado para “emprolijar los datos”.
- Veremos dos funciones:
  - `pivot_longer`
  - `pivot_wider`
- Para ejemplificar su uso utilizaremos un conjunto de datos (`iris`) que viene con el paquete base.

Iris

Cada observación es una flor de *Iris* con la medición del largo y ancho de su sépalo y pétalo y el tipo de especie a la cual pertenece.




```
1 iris
```

```
    Sepal.Length Sepal.Width Petal.Length Petal.Width
Species
1            5.1         3.5          1.4         0.2
setosa
2            4.9         3.0          1.4         0.2
setosa
3            4.7         3.2          1.3         0.2
setosa
4            4.6         3.1          1.5         0.2
setosa
```

5	5.0	3.6	1.4	0.2

Iris

Agregamos una columna que tenga el ID de cada flor y la acomodamos para que quede al principio.

```
1  iris <- iris %>%
2    mutate(id = 1:nrow(.)) %>% # se agrega un ID
3    select(id, everything()) # se acomoda para que el id este primero.
4  iris
```

	id	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	1	5.1	3.5	1.4	0.2	setosa
2	2	4.9	3.0	1.4	0.2	setosa
3	3	4.7	3.2	1.3	0.2	setosa
4	4	4.6	3.1	1.5	0.2	setosa
5	5	5.0	3.6	1.4	0.2	setosa
6	6	5.4	3.9	1.7	0.4	setosa
7	7	4.6	3.4	1.4	0.3	setosa
8	8	5.0	3.4	1.5	0.2	setosa
9	9	4.4	2.9	1.4	0.2	setosa
10	10	4.9	3.1	1.5	0.1	setosa
11	11	5.4	3.7	1.5	0.2	setosa
12	12	4.8	3.4	1.6	0.2	setosa
13	13	4.8	3.0	1.4	0.1	setosa
14	14	4.3	3.0	1.1	0.1	setosa

Iris

Queremos un dataset en el cual cada observación sea la medición de cada uno de los 4 atributos de cada flor.

Es decir, queremos una fila por cada flor y cada tipo de medición y una variable que indique el valor de esa medición.

```
1 iris
```

	id	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	1	5.1	3.5	1.4	0.2	setosa
2	2	4.9	3.0	1.4	0.2	setosa
3	3	4.7	3.2	1.3	0.2	setosa
4	4	4.6	3.1	1.5	0.2	setosa
5	5	5.0	3.6	1.4	0.2	setosa
6	6	5.4	3.9	1.7	0.4	setosa
7	7	4.6	3.4	1.4	0.3	setosa
8	8	5.0	3.4	1.5	0.2	setosa
9	9	4.4	2.9	1.4	0.2	setosa
10	10	4.9	3.1	1.5	0.1	setosa
11	11	5.4	3.7	1.5	0.2	setosa
12	12	4.8	3.4	1.6	0.2	setosa
13	13	4.8	3.0	1.4	0.1	setosa
14	14	4.3	3.0	1.1	0.1	setosa

Iris

Iris orginal...

```
  id Sepal.Length Sepal.Width Petal.Length Petal.Width
Species
1  1          5.1         3.5          1.4         0.2
setosa
2  2          4.9         3.0          1.4         0.2
setosa
3  3          4.7         3.2          1.3         0.2
setosa
4  4          4.6         3.1          1.5         0.2
setosa
5  5          5.0         3.6          1.4         0.2
setosa
```

Iris transformado (que queremos)...

```
  id Species tipo_medicion valor
1  1  setosa  Sepal.Length   5.1
```

pivot_longer()

```r
# Reorganizamos iris al formato largo
iris_vertical <- iris %>%
  pivot_longer(
    cols = c(Sepal.Length, Sepal.Width, Petal.Length, Peta
    names_to = "tipo_medicion",
    values_to = "valor"
  )
```

```
    id    Species tipo_medicion valor
1    1     setosa  Sepal.Length   5.1
2    1     setosa   Sepal.Width   3.5
3    1     setosa  Petal.Length   1.4
4    1     setosa   Petal.Width   0.2
5    2     setosa  Sepal.Length   4.9
6    2     setosa   Sepal.Width   3.0
7    2     setosa  Petal.Length   1.4
8    2     setosa   Petal.Width   0.2
9    3     setosa  Sepal.Length   4.7
10   3     setosa   Sepal.Width   3.2
```

pivot_longer()

- **cols**: columnas que queremos pivotear al formato “largo”.
- **names_to**: nombre de la columna que contendrá los nombres de las columnas que pivoteamos.
- **values_to**: nombre de la columna que contendrá los valores que estaban en las columnas que pivotemos.

```
1 iris_vertical <- iris %>%
2   pivot_longer(
3     cols = c(Sepal.Length, Sepal.Width, Petal.Length, Peta
4     names_to = "tipo_medicion",
5     values_to = "valor"
6   )
```

pivot_wider()

Supongamos que queremos volver al Iris original. Es decir,

Queremos pasar de...

```
  id Species tipo_medicion valor
1  1  setosa  Sepal.Length   5.1
2  1  setosa   Sepal.Width   3.5
3  1  setosa  Petal.Length   1.4
4  1  setosa   Petal.Width   0.2
5  2  setosa  Sepal.Length   4.9
6  2  setosa   Sepal.Width   3.0
```

A...

```
  id Sepal.Length Sepal.Width Petal.Length Petal.Width
Species
1  1          5.1         3.5          1.4         0.2
setosa
2  2          4.9         3.0          1.4         0.2
setosa
```

pivot_wider()

```r
# Volvemos al formato "ancho" a partir del formato largo
iris_horizontal <- iris_vertical %>%
  pivot_wider(
    names_from = tipo_medicion,
    values_from = valor
  )
```

	id	Species	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
1	1	setosa	5.1	3.5	1.4	0.2
2	2	setosa	4.9	3.0	1.4	0.2
3	3	setosa	4.7	3.2	1.3	0.2
4	4	setosa	4.6	3.1	1.5	0.2
5	5	setosa	5.0	3.6	1.4	0.2
6	6	setosa	5.4	3.9	1.7	0.4
7	7	setosa	4.6	3.4	1.4	0.3
8	8	setosa	5.0	3.4	1.5	0.2
9	9	setosa	4.4	2.9	1.4	0.2
10	10	setosa	4.9	3.1	1.5	0.1
11	11	setosa	5.4	3.7	1.5	0.2
12	12	setosa	4.8	3.4	1.6	0.2
13	13	setosa	4.8	3.0	1.4	0.1
14	14	setosa	4.3	3.0	1.1	0.1

pivot_wider()

- En **names_from** indicamos el nombre de la columna que contiene los valores que se usarán como nombres de las nuevas columnas (es decir, las que queremos "expandir" en formato ancho).
- En **values_from** indicamos el nombre de la columna que contiene los valores que se ubicarán dentro de esas nuevas columnas.

```
# Volvemos al formato "ancho" a partir del formato largo
iris_horizontal <- iris_vertical %>%
  pivot_wider(
    names_from = tipo_medicion,
    values_from = valor
  )
```

Lubridate

- Sirve para trabajar datos que contienen **fechas** y **horas**.
- Incluye funciones que permiten convertir datos de tipo `character` en objetos de tipo `date` (para trabajar con fechas) o `datetime` (para trabajar con fechas y horarios).

Funciones para fechas

- Funciones para convertir datos de tipo `character` a `date`:
  - `dmy()`: **día-mes-año**
  - `mdy()`: **mes-día-año**
  - `ymd()`: **año-mes-día**

dmy()

```
1 fecha1 <- "25-07-2025"  # día-mes-año
2 fecha1
```

```
[1] "25-07-2025"
```

```
1 class(fecha1)
```

```
[1] "character"
```

En este caso el orden es día-mes-año, entonces usamos `dmy()`

```
1 fecha1 <- dmy(fecha1)
2 fecha1
```

```
[1] "2025-07-25"
```

```
1 class(fecha1)
```

```
[1] "Date"
```

mdy()

```
1 fecha2 <- "07-25-2025"  # mes-día-año
2 fecha2
```

```
[1] "07-25-2025"
```

```
1 class(fecha2)
```

```
[1] "character"
```

En este caso el orden es mes-día-año, entonces usamos `mdy()`

```
1 fecha2 <- mdy(fecha2)
2 fecha2
```

```
[1] "2025-07-25"
```

```
1 class(fecha2)
```

```
[1] "Date"
```

Funciones para fechas y horarios

- Funciones para convertir texto que incluye **fecha y hora**:
  - `dmy_hms()`: **día-mes-año hora:minuto:segundo**
  - `mdy_hms()`: **mes-día-año hora:minuto:segundo**
  - `ymd_hms()`: **año-mes-día hora:minuto:segundo**

dmy_hms()

```
fecha <- "25-07-2025 14:30:00"  # día-mes-año
fecha
```

```
[1] "25-07-2025 14:30:00"
```

```
class(fecha)
```

```
[1] "character"
```

Como el orden es día-mes-año-horas-minutos-segundos usamos...

```
fecha <- dmy_hms(fecha)
fecha
```

```
[1] "2025-07-25 14:30:00 UTC"
```

```
class(fecha)
```

```
[1] "POSIXct" "POSIXt"
```

parse_date_time()

- `parse_date_time()` permite construir objetos `datetime` a partir de datos más complejos, como por ejemplo cuando aparece el nombre del mes y el año.

- Ejemplo

```
1 fecha  <- "Dec-92"
2 fecha <- parse_date_time(fecha, orders = 'my')
3 fecha
```

```
[1] "1992-12-01 UTC"
```

En el parámetro `orders` especificamos el orden en el cual se encuentra la información de la fecha.

Extracción de información

El paquete `lubridate` también tiene funciones para extraer información de un objeto `datetime`. Algunas son:

- `year()`: **obtener el año**
- `month()`: **obtener el año**
- `day()`: **obtener el día**
- `wday()`: **obtener el nombre del día**
- `hour()`: **obtener la hora**

Ejemplos

```
1 fecha
```

```
[1] "1992-12-01 UTC"
```

```
1 year(fecha) # Obtener el año
```

```
[1] 1992
```

```
1 month(fecha) # Obtener el mes
```

```
[1] 12
```

```
1 day(fecha) # Obtener el día
```

```
[1] 1
```

```
1 hour(fecha) # Obtener la hora
```

```
[1] 0
```

Operaciones

- Podemos sumar o restarle cualquier período de tiempo a un objeto **datetime**.
- Ejemplos

```
# Sumo dos días
fecha + days(2)
```

```
[1] "1992-12-03 UTC"
```

```
# Resto 1 semana y dos horas
fecha - (weeks(1) + hours(2))
```

```
[1] "1992-11-23 22:00:00 UTC"
```

Stringr

El paquete `stringr` sirve para trabajar con texto de forma sencilla. Algunas de sus funciones son:

str_length()

Calcula el largo de un string:

```
1 string1 <- "abcdefghi"
2 str_length(string1)
```

```
[1] 9
```

Cuidado que cuenta los espacios en blanco como un caracter!

```
1 string2 <- "abcd efghi"
2 str_length(string2)
```

```
[1] 10
```

`str_sub()`

- Permite extraer los caracteres que se encuentran entre determinadas posiciones.
- Tiene tres argumentos:
  - el **string**
  - el orden del caracter **a partir** del cual tiene que empezar a extraer
  - el orden del caracter **hasta** el cual tiene que extraer.

`str_sub()`: ejemplos

```
1 string1
```

```
[1] "abcdefghi"
```

```
1 # quiero del segundo al quinto caracter
2 str_sub(string1, 2, 5)
```

```
[1] "bcde"
```

Puedo pasarle la posición de los caracteres con un menos para indicar que quiero que cuente de atrás para adelante.

```
1 # quiero la última y anteúltima posición
2 str_sub(string1, -2, -1)
```

```
[1] "hi"
```

`str_sub()`: ejemplos

Lo podemos usar para reemplazar elementos. Supongamos que

```
1 string1
```

```
[1] "abcdefghi"
```

```
1 # quiero reemplazar la última letra por
2 str_sub(string1,-1,-1) <- "z"
3 string1
```

```
[1] "abcdefghz"
```

Manejo de espacios en blanco

- Es frecuente que aparezcan datos mal cargados o con errores de tipeo que tienen espacios donde no debería haberlos.
- La función `str_trim()` permite borrar los espacios en blanco a izquierda, derecha o ambos lados de nuestro string.

`str_trim()`: sintaxis

`str_trim(x, side = )` tiene dos argumentos:

- `x`: string al que le queremos borrar los espacios en blanco
- `side`: 'left', 'right' o 'both' según si queremos borrar los espacios en blanco a izquierda, derecha o ambos lados.

str_trim(): ejemplo

```
1 string3 <- c("  acelga   ", "brocoli   ",  "   choclo")
2 string3
```

```
[1] "  acelga   " "brocoli   "  "   choclo"
```

```
1 str_trim(string3, side = 'both')
```

```
[1] "acelga"  "brocoli" "choclo"
```

```
1 str_trim(string3, "left")
```

```
[1] "acelga   "  "brocoli   " "choclo"
```

Mayúsculas y minúsculas

Funciones para manipular las mayusculas/minúsculas de los strings:

- `str_to_lower()`: lleva todo a minúsculas
- `str_to_upper()`: lleva todo a mayúsculas
- `str_to_title()`: lleva a mayúscula la primer letra de cada palabra

Mayúsculas y minúsculas: ejemplos

```
1 string4 <- "No me gusta el frio"
2 string4
```

```
[1] "No me gusta el frio"
```

Llevo todo a minúsculas

```
1 str_to_lower(string4)
```

```
[1] "no me gusta el frio"
```

Mayúsculas y minúsculas: ejemplos

```
1 string4 <- "No me gusta el frio"
2 string4
```

```
[1] "No me gusta el frio"
```

Llevo todo a mayúsculas

```
1 str_to_upper(string4)
```

```
[1] "NO ME GUSTA EL FRIO"
```

Mayúsculas y minúsculas: ejemplos

```
1 string4 <- "No me gusta el frio"
2 string4
```

```
[1] "No me gusta el frio"
```

Llevo a mayúscula la primer letra de cada palabra

```
1 str_to_title(string4)
```

```
[1] "No Me Gusta El Frio"
```

str_split()

Permite partir un string de acuerdo a algún separador.

```
1 string5 <- "ab-cd-ef"
2 string5
```

```
[1] "ab-cd-ef"
```

Lo separamos por el guion

```
1 str_split(string5, pattern = "-")
```

```
[[1]]
[1] "ab" "cd" "ef"
```

Reemplazar elementos de strings

- Las funciones `str_replace()` y `str_replace_all()` permiten reemplazar parte de un string por otro string.
- `str_replace()` reemplaza **solo la primera coincidencia** de cada elemento.
- `str_replace_all()` reemplaza **todas las coincidencias** dentro de cada elemento.

`str_replace()`: ejemplos

```
frases1 <- c(
  "Nota: Entregar el informe mañana",
  "Nota: Revisar gráficos antes de envia
)
```

Queremos reemplazar “Nota” por “Importante”.

```
str_replace(frases1, "Nota", "Importante
```

```
[1] "Importante: Entregar el informe
mañana"
[2] "Importante: Revisar gráficos antes de
enviar"
```

`str_replace_all()`: ejemplos

```
frases2 <- c(
  "Está muy bueno y muy claro",
  "Es muy útil, muy práctico, muy recome
)
```

Queremos reemplazamos todas las veces que aparece la palabra “muy” por “MUY”

```
str_replace_all(frases2, "muy", "MUY")
```

```
[1] "Está MUY bueno y MUY claro"
[2] "Es MUY útil, MUY práctico, MUY
recomendado"
```

`str_detect()` detectar patrones en strings:

- La funcion `str_detect()` permite encontrar expresiones dentro de nuestros strings.
- Nos reporta VERDADERO o FALSO de acuerdo a si encuentra la expresión que estamos buscando.

```
1 string7 <-c("caño", "baño", "ladrillo")
2 str_detect(pattern = "ñ", string = strin
```

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

ggplot2

- Paquete muy útil para realizar gráficos.
- Los gráficos de `ggplot2` se construyen paso a paso agregando nuevos elementos o **capas**.

ggplot2: sintaxis básica

Necesitamos como mínimo, estas dos capas:

```
ggplot(
  data = mi_df,
  mapping = aes(x = var_x, y = var_y)
  ) +
  geom_point()
```

Capas geométricas

Las funciones de tipo geom definen el tipo de gráfico.

Las más usadas son:

- `geom_point()`: scatter plot
- `geom_boxplot()`: boxplot
- `geom_bar()`: gráfico de barras
- `geom_line()`: líneas o curvas

Capas adicionales

Podemos enriquecer nuestro gráfico agregando más capas:

- `labs()`: títulos, subtítulos y etiquetas de ejes
- `theme()`: estilo visual (fuentes, márgenes, colores)
- `facet_wrap()` / `facet_grid()`: paneles por variable

Capa 1: `ggplot()` - argumentos

- `data`: data frame con tus datos.
- `mapping = aes()`: mapeo de variables a estéticos

(qué variables vamos a graficar y cómo).

Ejemplo:

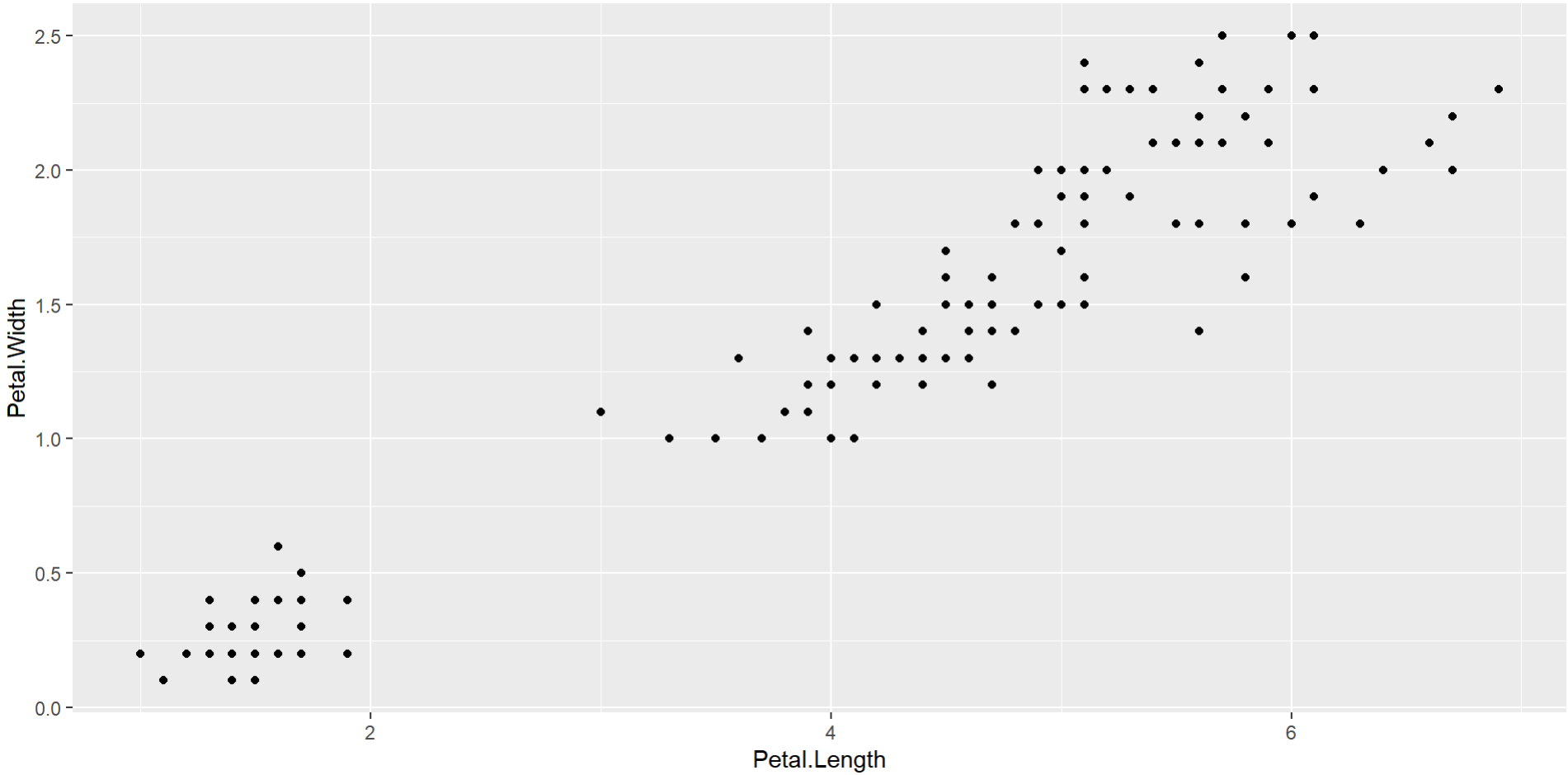
```
1 ggplot(
2   data = iris,
3   mapping = aes(x = Petal.Length, y = Petal.Width)
4   )
```

Ejemplo con iris

```
ggplot(
  data = iris,
  mapping = aes(x = Petal.Length, y = Petal.Width)
  )
```

Ejemplo con iris

```
ggplot(
  data = iris,
  mapping = aes(x = Petal.Length, y = Petal.Width)
  ) +
  geom_point()
```


2.5
2.0
1.5
1.0
0.5
0.0
Petal.Width
2
4
6
Petal.Length

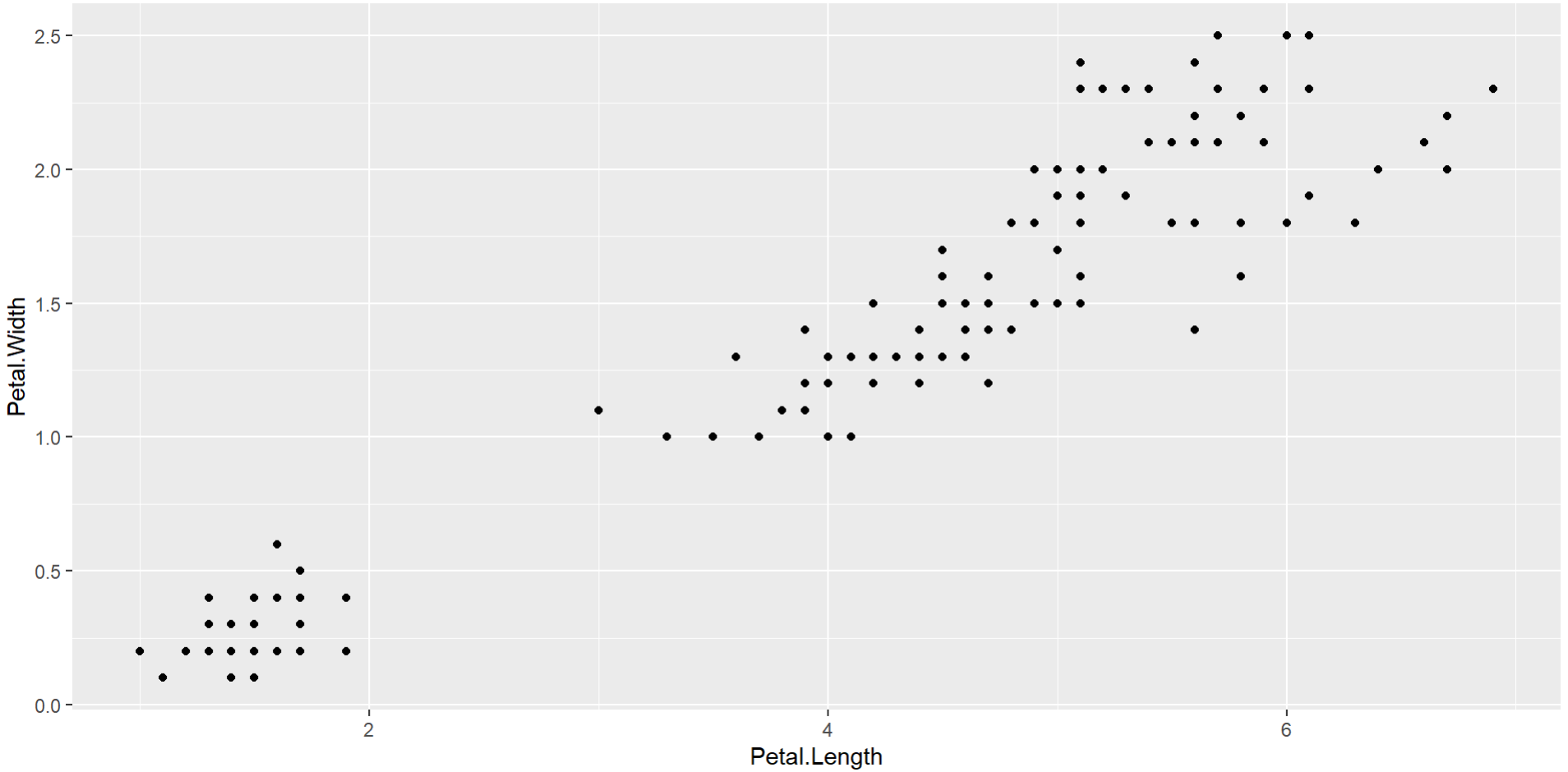
Ejemplo con iris: otro *geom*

```
ggplot(
  data = iris,
  mapping = aes(x = Petal.Length, y = Petal.Width)
  ) +
  geom_line()
```


2.5
2.0
1.5
1.0
0.5
0.0
Petal.Width
2
4
6
Petal.Length

Ejemplo con iris

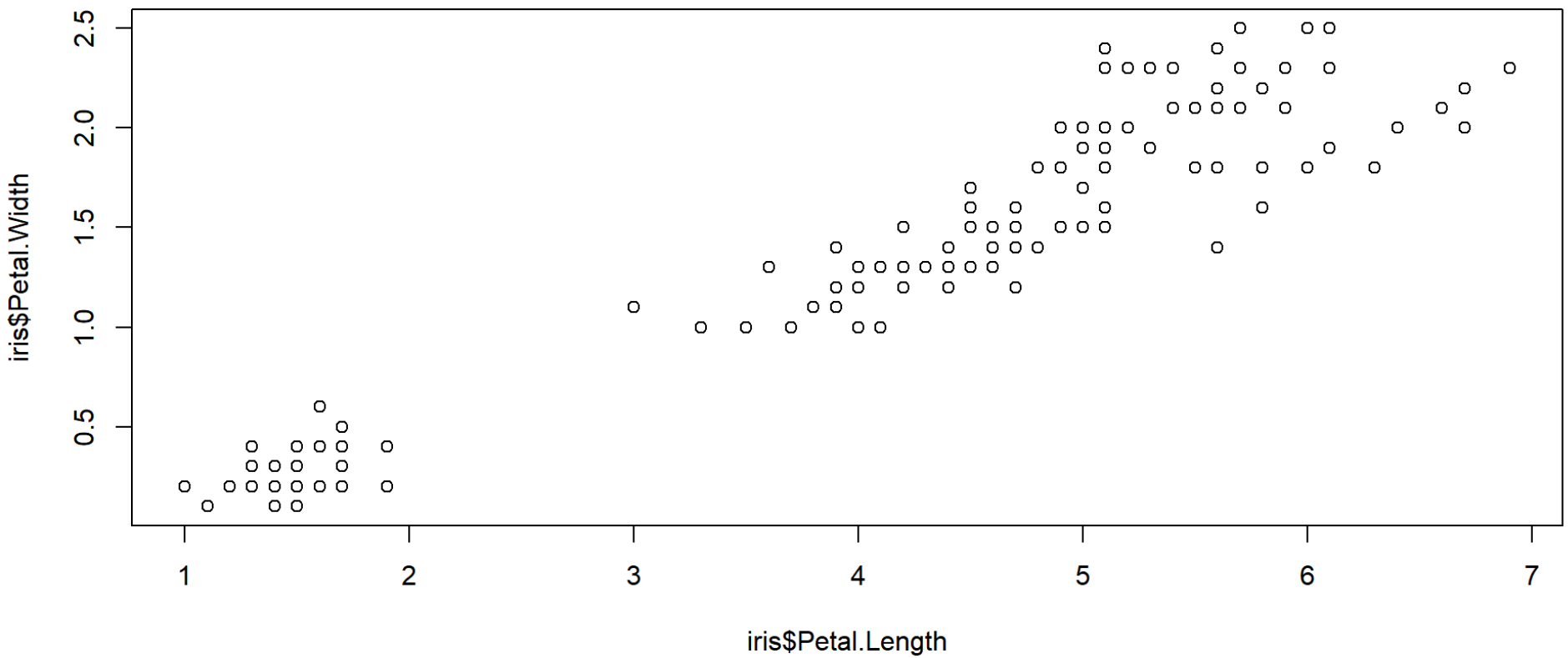
```
ggplot(
  data = iris,
  mapping = aes(x = Petal.Length, y = Petal.Width)
  ) +
  geom_point()
```



También podemos hacer un scatter plot con la función `plot()` del paquete base

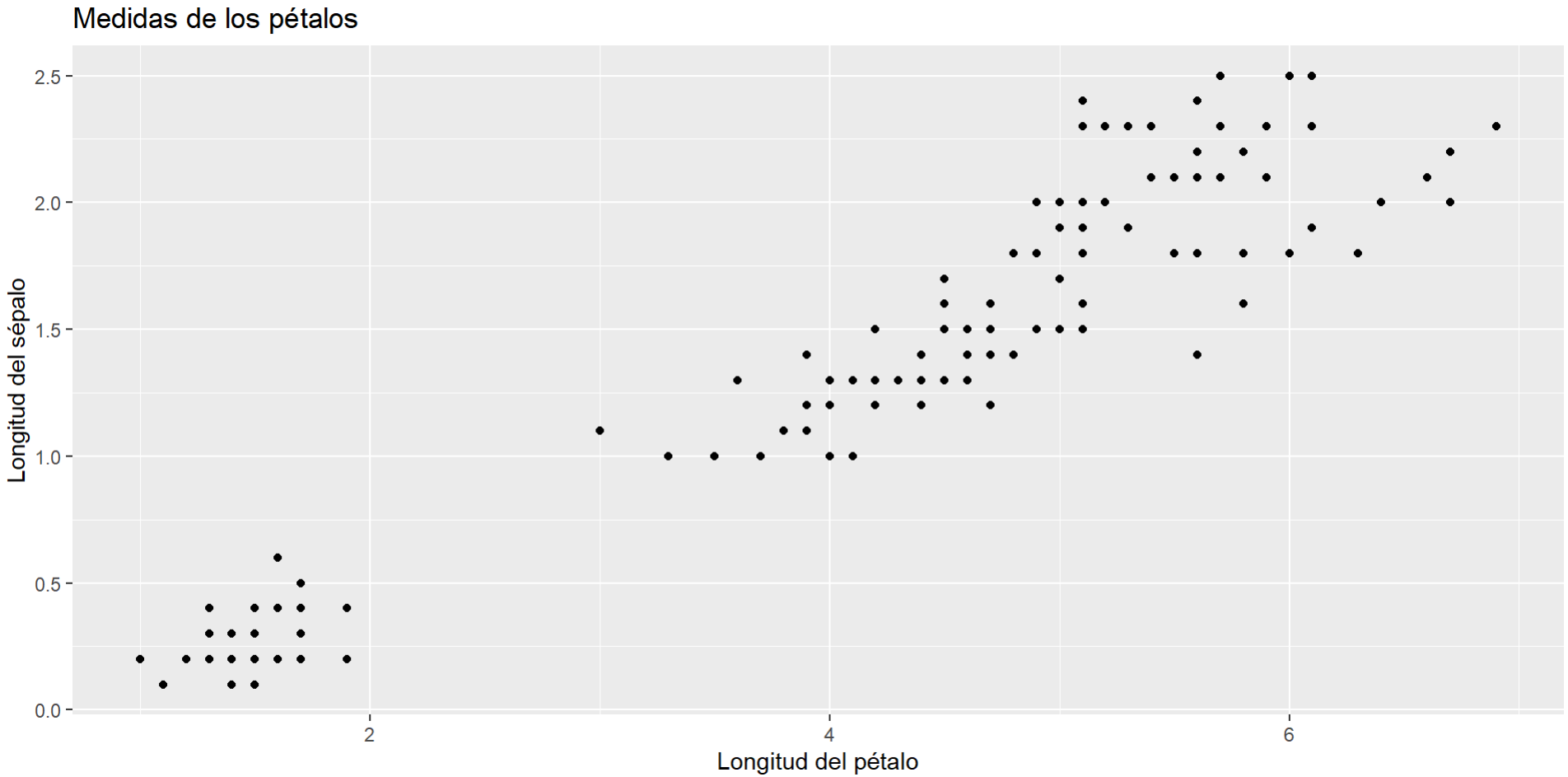
Scatter plot con paquete Base

```
1 plot(iris$Petal.Length, iris$Petal.Width
```



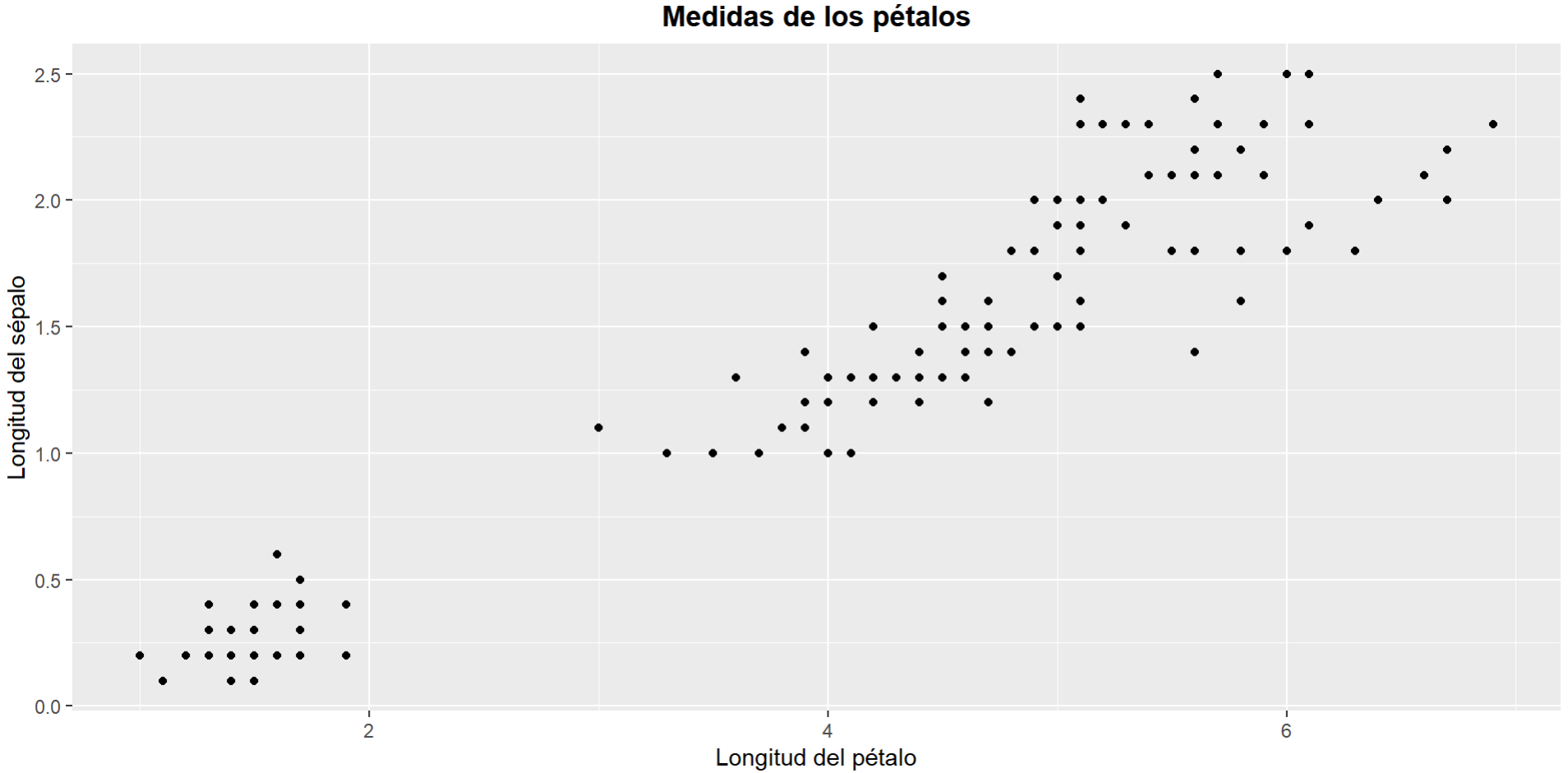
ggplot2: agregamos etiquetas y título

```
ggplot(iris, aes(x = Petal.Length, y = Petal.Width)) +
  geom_point() +
  labs(title = "Medidas de los pétalos",
       x = "Longitud del pétalo",
       y = "Longitud del sépalo")
```


Medidas de los pétalos
Longitud del sépalo
2.5
2.0
1.5
1.0
0.5
0.0
2
4
6
Longitud del pétalo

ggplot2: título centrado y en negrita

```
ggplot(iris, aes(x = Petal.Length, y = Petal.Width)) +
  geom_point() +
  labs(title = "Medidas de los pétalos",
       x = "Longitud del pétalo",
       y = "Longitud del sépalo") +
  theme(plot.title = element_text(hjust = 0.5, face = "bold"))
```


Medidas de los pétalos
Longitud del sépalo
2.5
2.0
1.5
1.0
0.5
0.0
2
4
6
Longitud del pétalo

ggplot2: incluimos a la especie

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

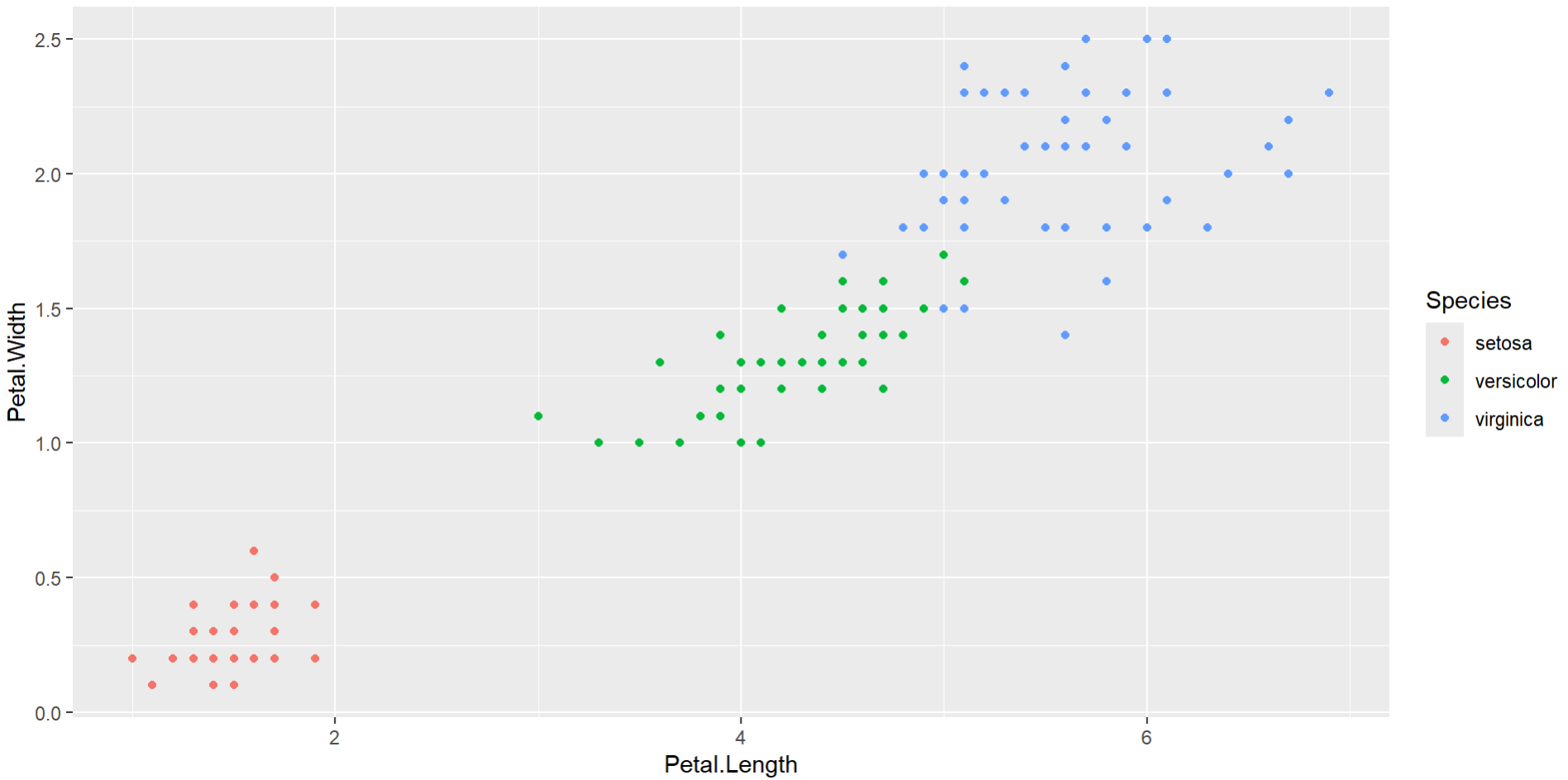
Queremos visualizar la variable `Species`.

`Species` es una variable categórica con 3 niveles: **setosa**, **versicolor** y **virginica**.

Una forma de visualizar una variable categórica es a través del **color**.

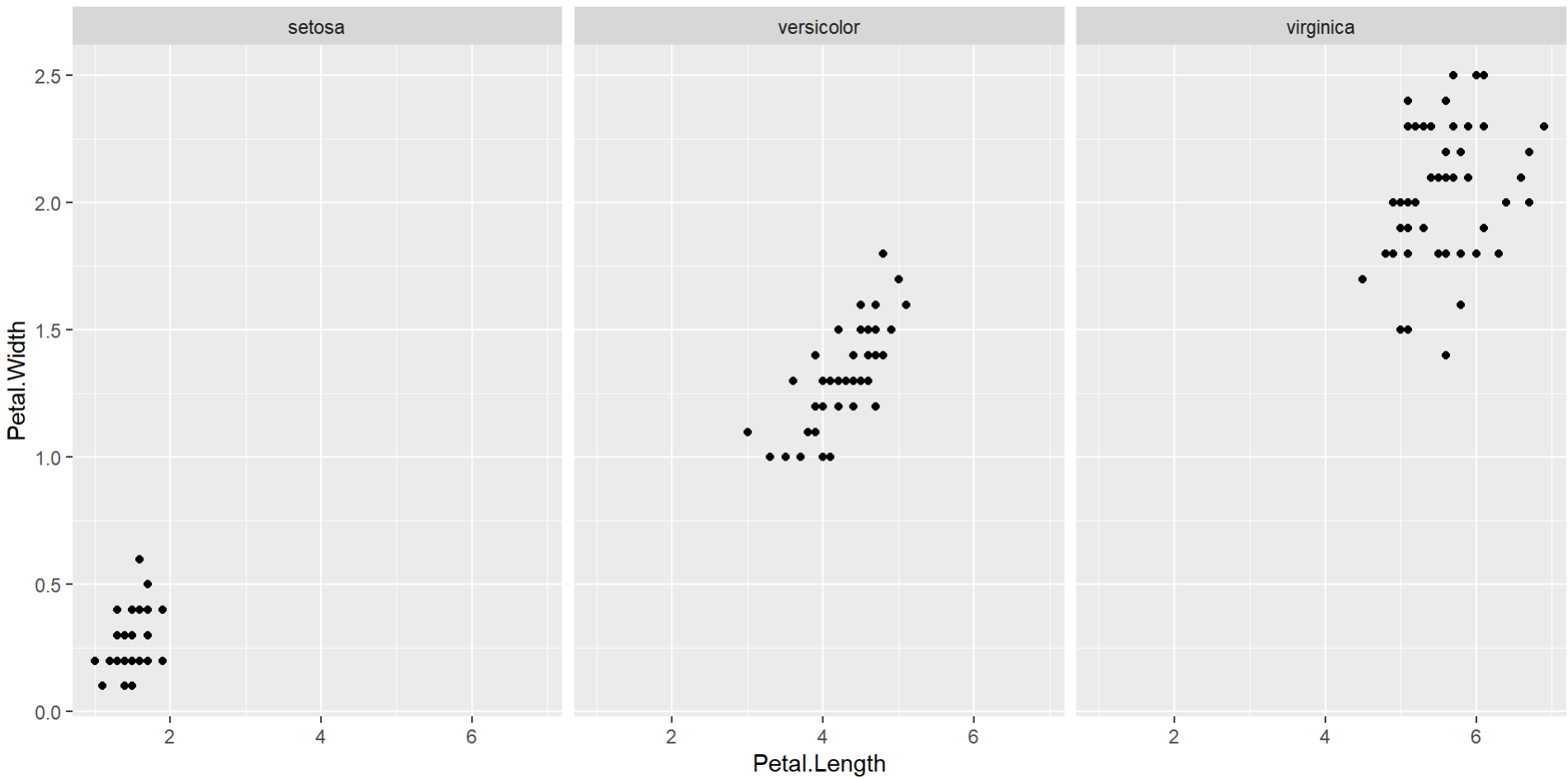
ggplot2: especie con color

```
1 ggplot(iris, aes(x = Petal.Length, y = Petal.Width, color = Species)) +
2   geom_point()
```



ggplot2: especie en facetas

```
1  ggplot(iris, aes(x = Petal.Length, y = Petal.Width)) +
2    geom_point() +
3    facet_wrap(~ Species)
```



ggplot2: títulos más grandes

```
ggplot(iris, aes(x = Petal.Length, y = Petal.Width)) +
  geom_point() +
  facet_wrap(~ Species) +
  theme(strip.text = element_text(size = 14))
```

Variable numérica vs. categórica

Visualizar la relación entre una variable numérica y una categórica, ¿qué tipo de gráfico puedo usar?

ggplot2: boxplot

- Cuando queremos explorar la relación entre una variable numérica y una variable categórica, los **gráficos de dispersión (scatter plots)** no son la mejor opción.
- Una alternativa adecuada es el uso de un **boxplot**, que resume la distribución de la variable numérica dentro de cada categoría (nivel) de la variable categórica.
- Veamos cómo crear un boxplot con `ggplot2`.

- Vamos a usar una base de datos reales de la Encuesta Permanente de Hogares (EPH) que publica el INDEC correspondiente al 1er trimestre de 2022.

Datos EPH

```r
datos <- read.table(
  "./Datos/usu_individual_T122.txt",
  sep=";",
  dec=",",
  header = TRUE)
glimpse(head(datos, 4))
```

```
Rows: 4
Columns: 177
$ CODUSU     <chr> "TQRMNOQXQHLOKQCDEGKDB00777573",
"TQRMNOQXQHLOKQCDEGKDB0077…
$ ANO4       <int> 2022, 2022, 2022, 2022
$ TRIMESTRE  <int> 1, 1, 1, 1
$ NRO_HOGAR  <int> 1, 1, 1, 1
$ COMPONENTE <int> 2, 3, 4, 1
$ H15        <int> 1, 1, 1, 1
$ REGION     <int> 43, 43, 43, 1
```

Datos EPH

Trabajaremos con las variables:

- `CH04`: sexo (*categórica* con 2 niveles)
- `NIVEL_ED`: nivel educativo (*categórica* con 6 niveles)
- `P21`: ingreso (*numérica*)

Veamos cómo leyó R cada una de esas variables:

```
1 datos %>%
2   select(CH04, NIVEL_ED, P21) %>%
3   head(n = 10) %>%
4   glimpse()
```

```
Rows: 10
Columns: 3
$ CH04     <int> 2, 2, 1, 2, 1, 2, 1, 2, 2, 1
$ NIVEL_ED <int> 6, 5, 3, 6, 4, 6, 6, 4, 4, 3
$ P21      <int> 150000, 10000, 0, -9, 0, -9, 80000, 0, 0, 0
```

Datos EPH

Queremos avisarle a R que `CH04` y `NIVEL_ED` son variables categóricas para que las analice correctamente. Para lo cual debemos pasarlas a tipo **factor**.

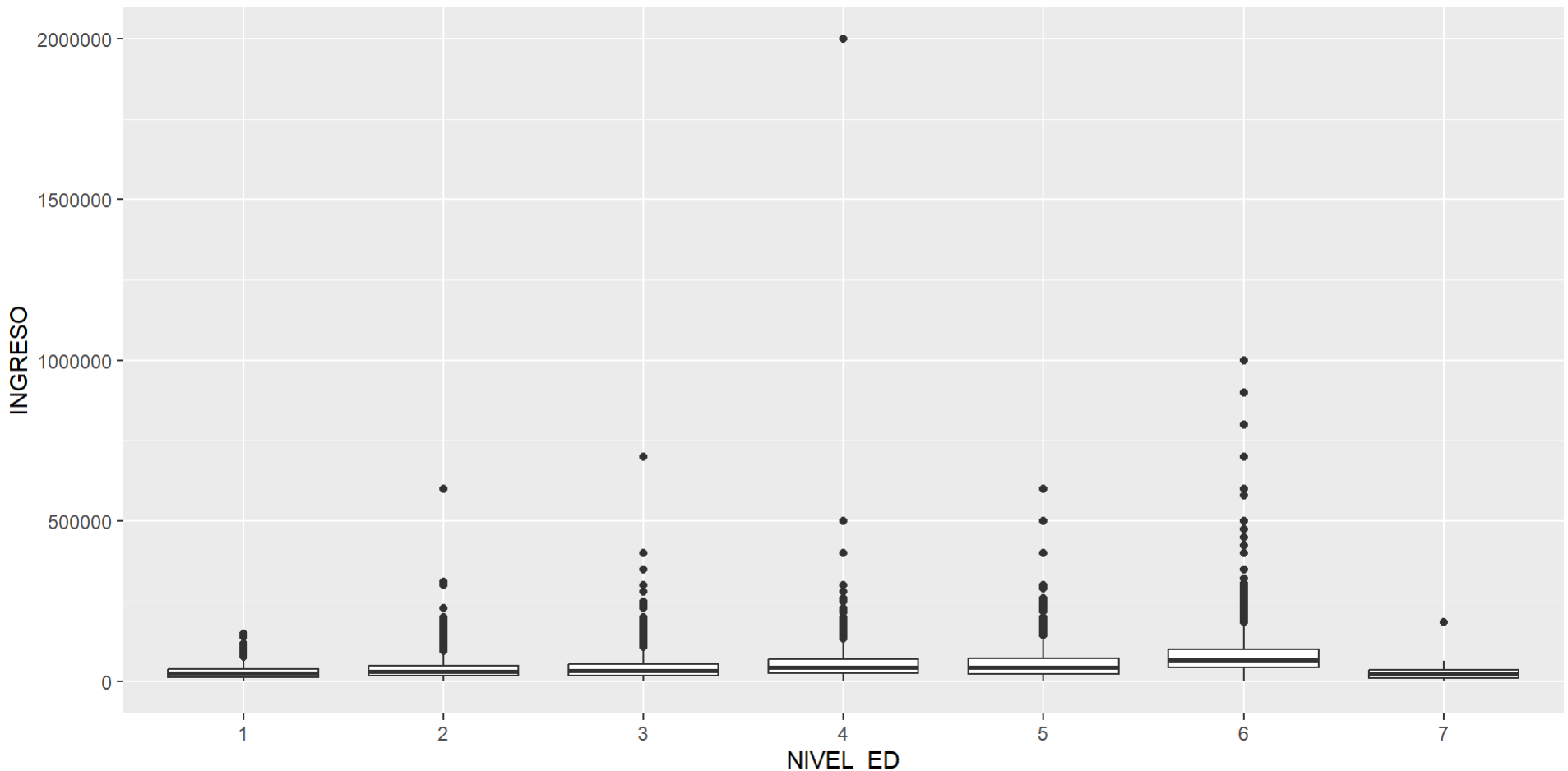
Objetos de tipo **factor**: objetos de R adecuados para trabajar con variables categóricas.

Limpieza del data set

```
datos <- datos %>%
  rename(SEXO = CH04,
         INGRESO = P21) %>%
  mutate(NIVEL_ED = as.factor(NIVEL_ED),
         SEXO     = as.factor(SEXO),
         SEXO = recode(SEXO,
                       `1` = "Hombre",
                       `2` = "Mujer")) %
  filter(INGRESO > 0)
```

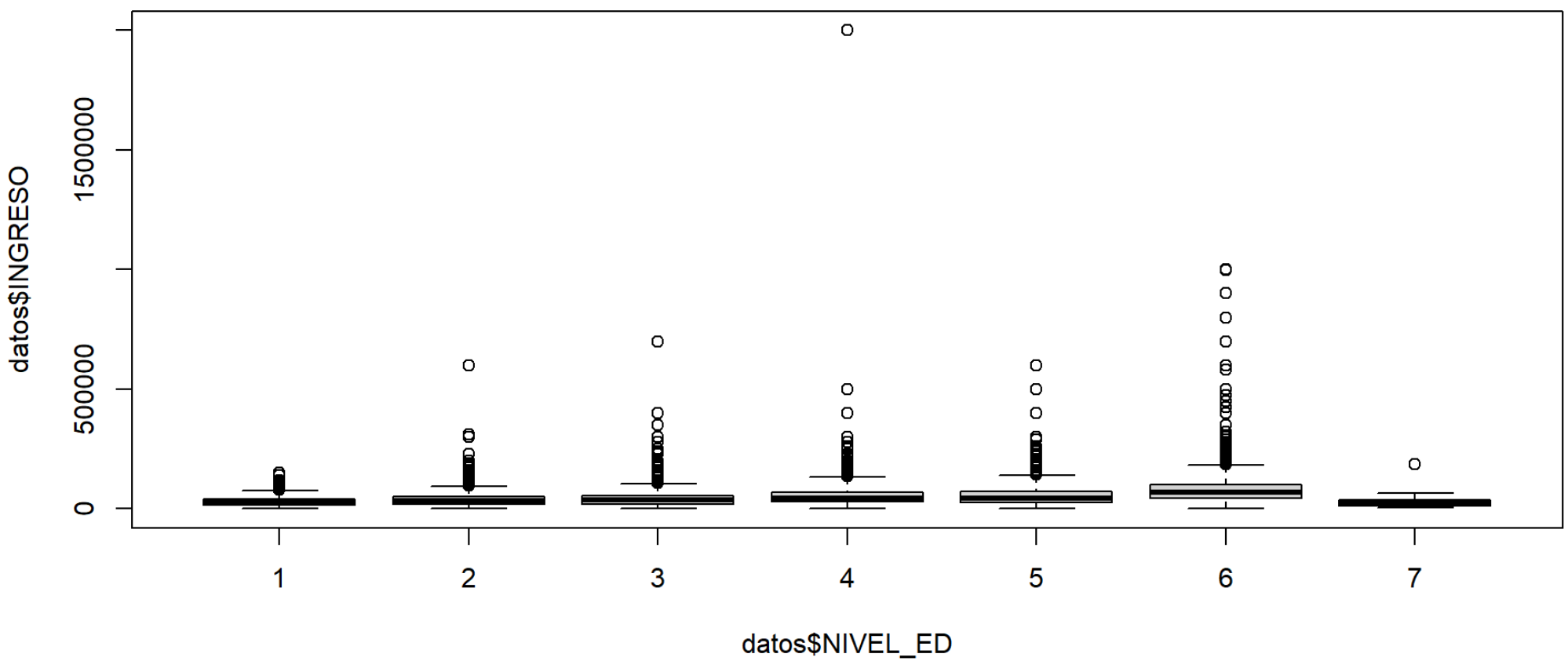
Boxplot de Ingreso por Nivel Educativo

```
1 ggplot(datos, aes(x = NIVEL_ED, y = INGRESO)) +
2   geom_boxplot()
```



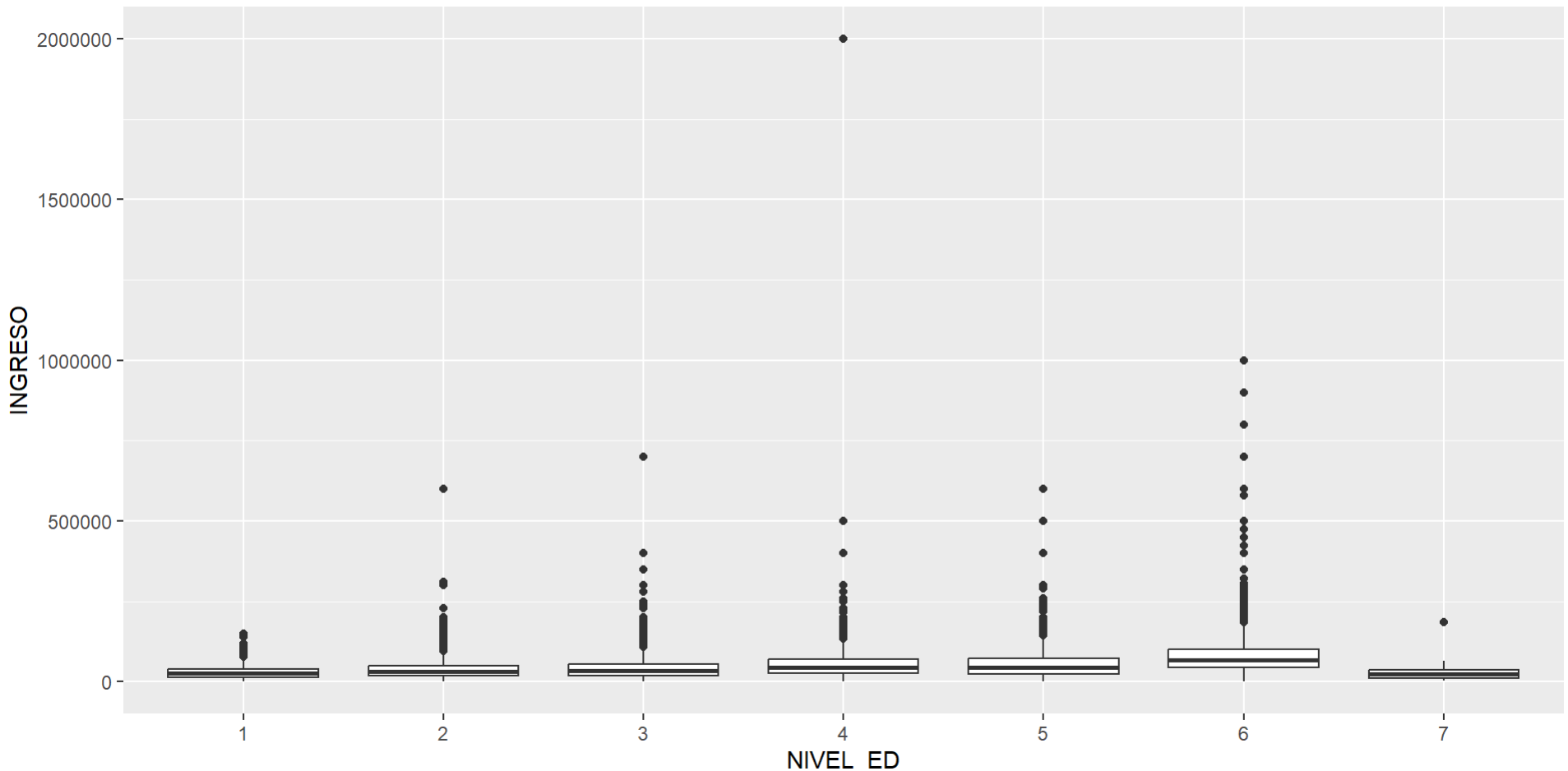
Boxplot con Paquete Base

```
1 boxplot(datos$INGRESO ~ datos$NIVEL_ED)
```



Boxplot de Ingreso por Nivel Educativo

```
1 ggplot(datos, aes(x = NIVEL_ED, y = INGRESO)) +
2   geom_boxplot()
```



Trunquemos el eje y en 200.000 para visualizar mejor las cajas.

Boxplot truncados (sin outliers)

```
1 ggplot(datos, aes(x = NIVEL_ED, y = INGRESO)) +
2   geom_boxplot() +
3   coord_cartesian(ylim = c(0, 200000))
```

Incluimos al sexo

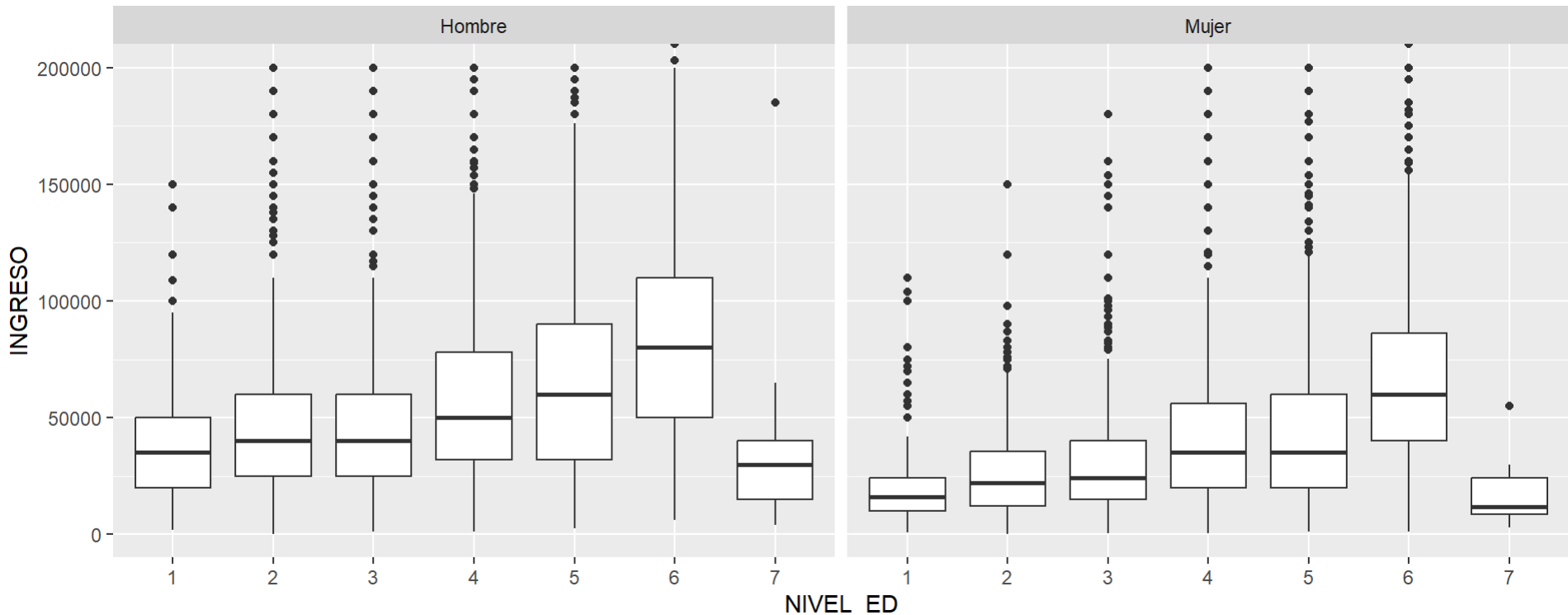
Si queremos analizar la relación entre el **ingreso** y el **nivel educativo** para cada **sexo**, podemos separar el gráfico en `facetas` según el sexo.

Para ello debemos incluir una capa con

```
1  facet_wrap(~ SEXO)
```

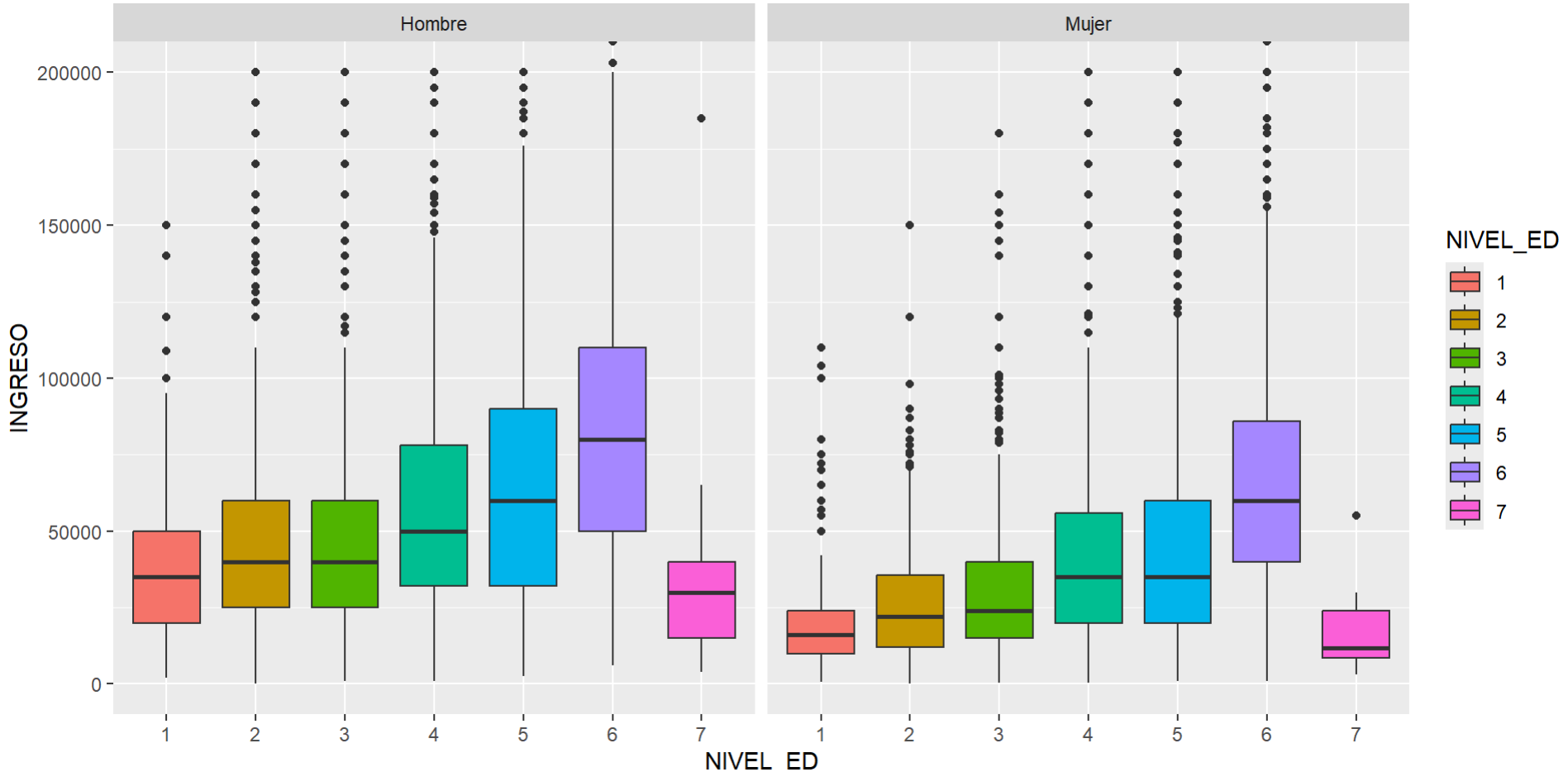
Incluimos al sexo

```
1  ggplot(datos, aes(x = NIVEL_ED, y = INGRESO)) +
2    geom_boxplot() +
3    coord_cartesian(ylim = c(0, 200000)) +
4    facet_wrap(~ SEXO)
```



Boxplots coloreados por nivel educativo

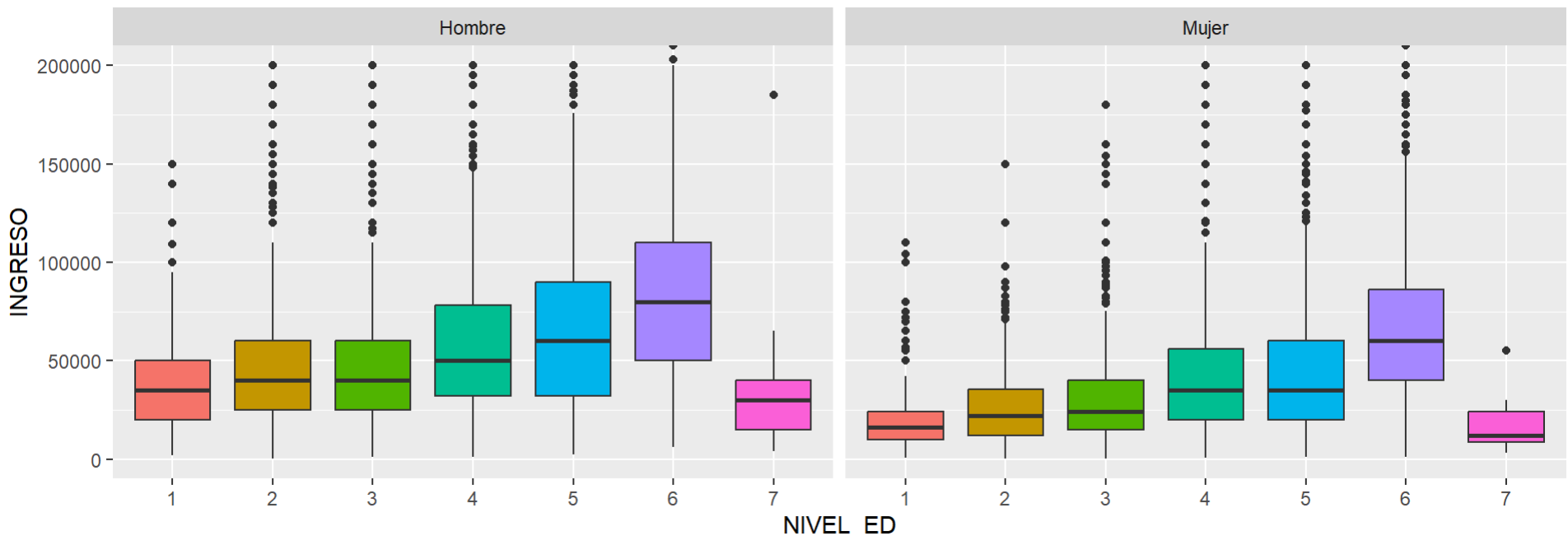
```
ggplot(datos, aes(x = NIVEL_ED, y = INGRESO, fill = NIVEL_ED)) +
  geom_boxplot() +
  coord_cartesian(ylim = c(0, 200000)) +
  facet_wrap(~ SEXO)
```



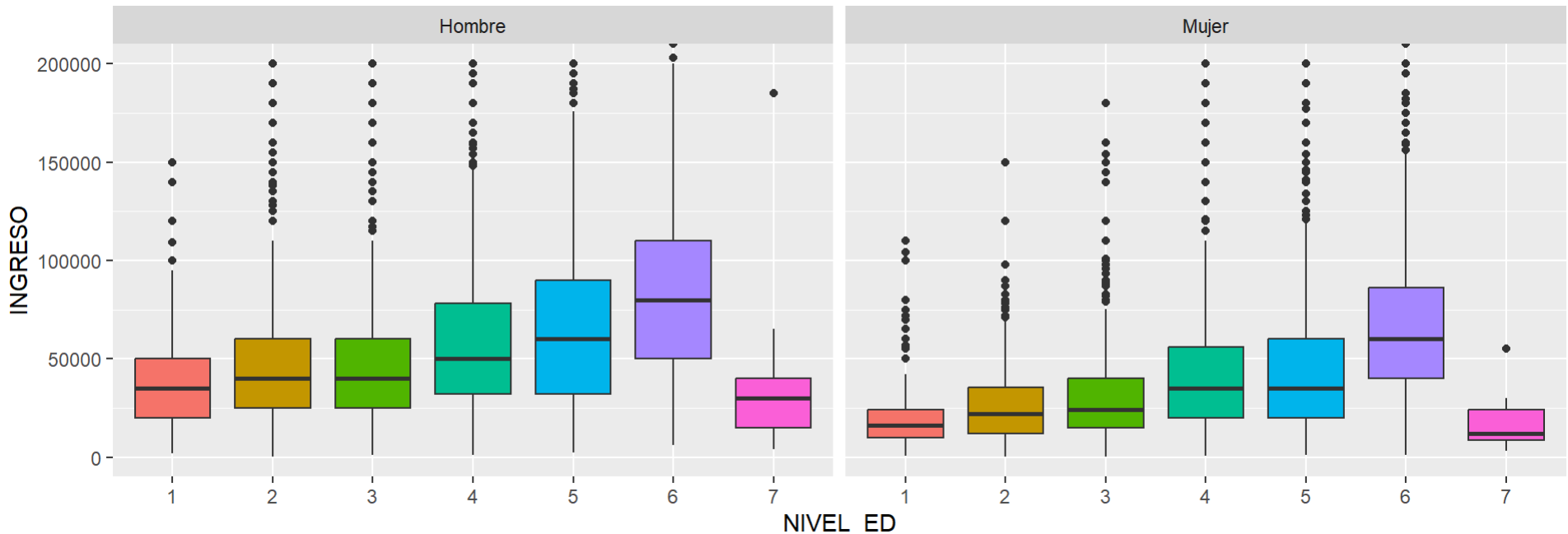
La leyenda del color es redundante, podemos eliminarla con `theme(legend.position = "none")`.

Eliminamos leyenda

```
ggplot(datos, aes(x = NIVEL_ED, y = INGRESO, fill = NIVEL_ED)) +
  geom_boxplot() +
  coord_cartesian(ylim = c(0, 200000)) +
  facet_wrap(~ SEXO) +
  theme(legend.position = "none")
```



Ingreso vs. Nivel Educativo por Sexo



Este gráfico permite comparar el **ingreso** entre los distintos **niveles educativos** para cada **sexo**.

¿Qué hacemos si queremos comparar el **ingreso** entre ambos **sexos** para cada **nivel educativo**?

Ingreso vs. Sexo por Nivel Educativo

```
ggplot(datos, aes(x = SEXO, y = INGRESO, fill = SEXO)) +
  geom_boxplot() +
  coord_cartesian(ylim = c(0, 200000)) +
  facet_wrap(~ NIVEL_ED) +
  theme(legend.position = "none")
```