Introducción al Análisis Exploratorio de Datos

Aplicaciones con R y datos reales

Vicente Coll-Serrano

Introducción al Análisis Exploratorio de Datos.

Aplicaciones con R y datos reales.

Vicente Coll Serrano

Profesor Titular de Universidad
Métodos Cuantitativos para la Economía y la Empresa

Facultad de Economía. Universidad de Valencia (España)

25-julio-2024

Información de la obra

Cómo citar de esta publicación:

Coll-Serrano, V. (2023). *Introducción al Análisis Exploratorio de Datos. Aplicaciones con R y datos reales.* Valencia: Leanpub.

Primera edición, 2023 (versión electrónica corregida)

Esta versión ha sido publicada el 25 de julio de 2024 y corrige las **erratas detectadas en la versión publicada el 1 de diciembre de 2023 (consultar: www.uv.es/vcoll/erratas.pdf)**

Diseño de la portada: Michael Brown (https://michaeldbrowndesign.com/)

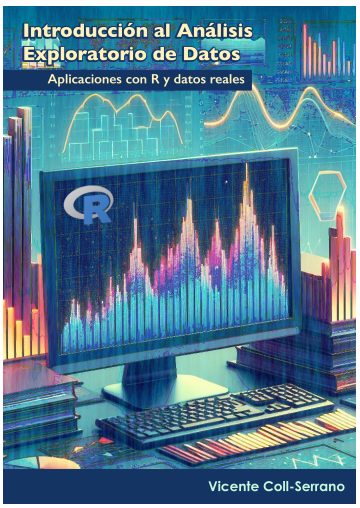
ISBN: 978-84-09-56823-9

Lo único que hacen los ruiseñores es música para que
la disfrutemos. No se comen nada de los jardines, no
hacen nidos en los graneros de maíz, lo único que hacen
es cantar con todo su corazón para nosotros.
(Matar a un ruiseñor, Harper Lee)

A mis estudiantes, los futuros ruiseñores
de la ciencia de datos.

Índice general

Acerca de...



Portada diseñada por: Michael Brown

El Análisis Exploratorio de Datos (Exploratory Data Analysis, EDA), desarrollado por John W. Tukey, es la primera etapa del proceso de análisis de datos y consiste en un conjunto de métodos estadísticos y gráficos que permiten:

- organizar y estructurar los datos.
- explorar la distribución de las variables consideradas con la finalidad, por ejemplo, de detectar la presencia de valores missing, valores anómalos, comprender la dispersión y forma de los datos, etc.
- entender las relaciones entre las variables.
- comprobar el cumplimiento de los supuestos en los que se basan gran parte de métodos multivariantes.
- reorganizar y reestructurar los datos para posteriores procedimientos de análisis.

Este manual cubre los conceptos básicos de un curso introductorio al Análisis Exploratorio de Datos. En la primera parte del libro (capítulos 1 a 5) se introducen las bases para analizar y representar datos con R; en la segunda parte (capítulo 6) se explica como acceder a bases de datos de distintas fuentes (Instituto Nacional de Estadística, Centro de Investigaciones Sociológicas, Eurostat, etc.); y la tercera parte (capítulos 7 a 9) está dedicada a explicar, haciendo uso de R y datos reales, las principales medidas estadísticas utilizadas para describir una variable y para analizar la relación/asociación entre dos variables.

A medida que se adquieran conocimientos de probabilidad y de inferencia estadística, el Análisis Exploratorio de Datos puede extenderse para contrastar las hipótesis de partida requeridas en determinadas técnicas multivariantes, aplicar métodos de imputación de datos faltantes (missing data imputation), etc.

Capítulo 4

Procesando los datos con tidyverse.

4.1 Introducción.

En el capítulo anterior practicamos cómo leer y guardar datos (o resultados) en nuestra sesión de trabajo de R/RStudio. En este capítulo vamos a abordar cómo transformar estos datos para que se adecúen a nuestros análisis. El flujo de trabajo que normalmente seguimos se resumen en la Figura 4.1.

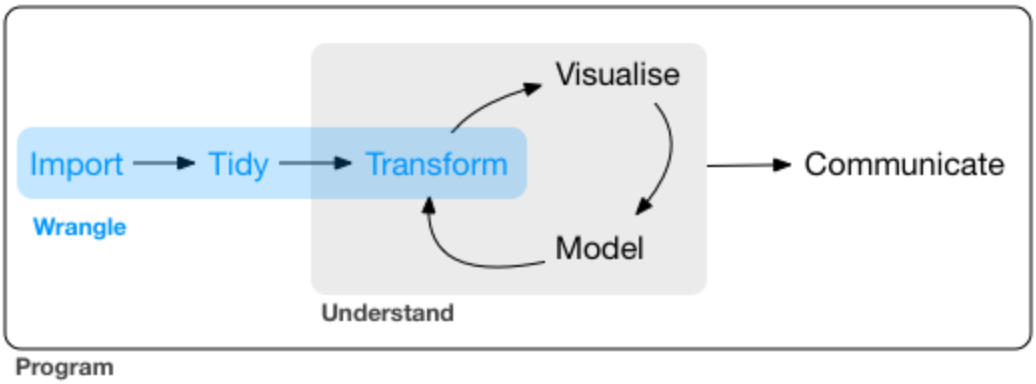


Figura 4.1: Flujo de trabajo en el análisis de datos. Fuente: http://r4ds.had.co.nz/wrangle-intro.html

Comenzamos importando los datos y ordenándolos, es lo que en la terminología se conoce como *data tidying* (Wickham (2014)). Disponer los datos correctamente, con una buena estructura, facilita el proceso de transformación/manipulación, esto es, la selección de casos y de variables de interés, el cálculo de descriptivos básicos, etc. Para responder a las preguntas que nos planteamos efectuamos tanto análisis gráficos (visualización) como estadísticos (modelización). Finalmente, tenemos que saber comunicar nuestros resultados de una forma efectiva.

Hoy en día, cuando analizamos datos con R solemos referirnos con relativa frecuencia al término **tidyverse**. Lo hacemos en un doble sentido. Por un lado, para hacer referencia a una nueva forma de afrontar el análisis de datos en R, más fluído que como lo haríamos con R-base. Por otro lado, para referirnos a todo un conjunto de paquetes (o librerias) interrelacionados que permiten afrontar todo el flujo de trabajo del analista de datos (Figura 4.1): desde la importación de datos hasta la comunicación de los resultados.

Figura 4.2: Paquetes que integran Tidyverse. Fuente: http://www.tidyverse.org

Como observamos en la Figura 4.2, los principales paquetes que integran tidyverse son:

- **`readr`**: para importar datos.
- **`tidyr`**: para convertir los datos a "tidy data", datos ordenados.
- **`dplyr`**: para transformar/manipular datos.
- **`ggplot2`**: para hacer gráficos.
- **`tibble`**: dataframes "actualizados".
- **`forcast`**: para manipular factores.
- **`stringr`**: para manipular strings (cadenas de texto).
- **`purrr`**: para programación funcional (functional programming).

Cuando cargamos el paquete **`tidyverse`** realmente lo que estamos haciendo es cargar los 8 paquetes anteriores (ggplot2, readr, etc.). También podemos cargar cada paquete por separado.

Si quieres saber más sobre **tidyverse**, puedes consultar **www.tidyverse.org**

En este capítulo vamos a introducir los conceptos básicos que nos permitan adquirir las competencias esenciales para manejar o procesar nuestros datos con los paquete **`tidyr`** y **`dplyr`**. El siguiente capítulo lo dedicaremos a realizar una introducción a la visualización de datos con **`ggplot2`**.

4.2 Antes de comenzar...

Abrimos nuestro proyecto de trabajo en RStudio. Para ello, entramos en la carpeta del proyecto **estadistica_basica** y hacemos doble clic sobre el fichero **estadistica_basica.Rproj**.

Creamos un nuevo script de R y lo guardamos con el nombre: **tema_4**. En este script escribiremos todas las instrucciones que trabajaremos en el resto de este cuarto capítulo.

Para ejemplificar el uso de las principales funciones de `tidyr` y `dplyr` utilizaremos un conjunto de datos (dataset) que se encuentra disponible en el paquete `AER` (Kleiber and Zeileis, 2008). Concretamente, haremos uso del dataset *CPS1985* (Determinants of Wages Data, CPS 1985). Más información del dataset en la ayuda del paquete y en el siguiente enlace: **http://lib.stat.cmu.edu/datasets/CPS_85_Wages**

Cargamos en memoria la libreria **AER**. Si no tenemos instalado el paquete, lo instalamos y después lo comentamos. Cargamos el conjunto de datos que vamos a trabajar. Los datos quedan automáticamente asignados al objeto **CPS1985**; compruébalo en el *Environment*.

```
> # install.packages("AER")
> library(AER)
> data("CPS1985")
```

¿Cuántas observaciones hay en el dataset? ¿Qué variables contiene?

Para responder a la primera pregunta utilizaremos la función **dim()**; para la segunda podemos usar la función **names()**.

```
> dim(CPS1985)
[1] 534  11
> names(CPS1985)
 [1] "wage"       "education"  "experience" "age"        "ethnicity"
 [6] "region"     "gender"     "occupation" "sector"     "union"
[11] "married"
```

Así pues, tenemos un total de 534 observaciones y las siguientes 11 variables:

- **Wage**: Wage (dollars per hour).
- **Education**: Number of years of education.
- **Experience**: Number of years of work experience.
- **Age**: Age (years).
- **Ethnicity**: Race (1=Other, 2=Hispanic, 3=White).
- **Region**: Indicator variable for Southern Region (1=Person lives in South, 0=Person lives elsewhere).
- **Gender**: Indicator variable for sex (1=Female, 0=Male).
- **Occupation**: Occupational category (1=Management, 2=Sales, 3=technical,4 =Service, 5=Professional, 6=Other).
- **Sector**: Sector (0=Other, 1=Manufacturing, 2=Construction).
- **Union**: Indicator variable for union membership (1=Union member, 0=Not union member).
- **Married**: Marital Status (0=Unmarried, 1=Married)

Por defecto, cuando R lee un conjunto de datos, ordena los niveles un factor por orden alfabético. Sin embargo, por su diseño, esto no ocurre con el dataframe **CPS1985**.

Por tanto, **siempre tenemos que ser conscientes de cómo están ordenados los niveles de un factor**.

Vamos a obtener un resumen de las variables. Hacemos uso de la función **summary()**.

```
> summary(CPS1985)
      wage          education       experience         age
 Min.   : 1.000   Min.   : 2.00   Min.   : 0.00   Min.   :18.00
 1st Qu.: 5.250   1st Qu.:12.00   1st Qu.: 8.00   1st Qu.:28.00
 Median : 7.780   Median :12.00   Median :15.00   Median :35.00
 Mean   : 9.024   Mean   :13.02   Mean   :17.82   Mean   :36.83
 3rd Qu.:11.250   3rd Qu.:15.00   3rd Qu.:26.00   3rd Qu.:44.00
 Max.   :44.500   Max.   :18.00   Max.   :55.00   Max.   :64.00
    ethnicity     region       gender         occupation            sector
 cauc    :440   south:156   male  :289   worker    :156   manufacturing: 99
 hispanic: 27   other:378   female:245   technical :105   construction : 24
 other   : 67                           services  : 83   other        :411
                                         office    : 97
                                         sales     : 38
                                         management: 55

 union     married
 no :438   no :184
 yes: 96   yes:350
```

En el caso de variables cuantitativas, la función **summary()** proporciona como resumen de la variable los **"cinco números"** (mínimo, cuartil 1, cuartil 2, cuartil 3, máximo) y la media. Si la variable es categórica y se ha identificado como factor, proporciona el recuento de observaciones en cada categoría. En el caso de variables identifacadas como carácter (character) no puede proporcionar ningún resultado, sólo indica la longitud del vector relativo a la variable.

Bien. Ya tenemos una idea de los datos. Ahora…¡¡¡a trabajar con ellos!!!

4.3 El operador %>%

El operador **%>%** se denomina **pipe** (Bache and Wickham, 2022). Es un operador que nos facilita la secuencialización de pasos de un determinado proceso para alcanzar un resultado, haciendo mucho más fluído el flujo de trabajo.

Lo que hace el operador **%>%** es pasar el elemento que está a su izquierda como un argumento de la función que tiene a la derecha.

El operador %>% podemos leerlo como: **y luego** o **entonces** (o algo similar). A partir de la version 4.1 de R puede usarse como operador nativo |> en lugar de %>%

Con el **pipe** la escritura y la lectura de código es mucho más limpia, clara y fluída. En consecuencia, y aunque resulte un tanto redundante decirlo, el código que escribimos es mucho más fácil de leer.

Pongo el ejemplo que utilizamos mi compañero Pedro y yo en uno de nuestros cursos de formación. Se trata de comparar cómo escribiríamos en R-base y en el estilo de **tidyverse** -las funciones y argumentos utilizados son ficticios- la secuencialiación del siguiente proceso:

> Me despierto a las 8:00 de la manaña y salgo de la cama por el lado correcto; me visto con unos pantalones y camiseta y salgo de casa en coche, no tomo la bicicleta.

En R-base escribiríamos algo parecido a:

leave_house(get_dressed(get_out_of_bed(wake_up(me, time = "8:00"), side = "correct"),pants = TRUE, shirt = TRUE), car = TRUE, bike = FALSE)

Observemos que en R-base la secuencia de funciones se leería de dentro (me despierto, wake_up) hacia afuera y de derecha a izquierda. Esto hace que los argumentos de una función se encuentren cada vez más alejados de su función origen. Fijaros los argumentos *car* y *bike* de la función **leave_house** donde se encuentran.

Moraleja: cuando anidamos varias funciones en R-base el código resulta complicado de entender/leer.

En cambio, esta misma secuencialización en el entorno de tidyverse sería algo similar a lo siguiente:

me %>%
wake_up(time = "8:00") %>%
get_out_of_bed(side = "correct") %>%
get_dressed(pants = TRUE, shirt = TRUE) %>%
leave_house(car = TRUE, bike = FALSE)

No cabe duda que este último código resulta mucho más fluído y comprensible.

El atajo de teclado para el operador %>% o |> es:

en Windows: **ctlr + shift + M**
en Mac: **Cmd + shift + M**

Veamos como utilizar el pipe (**%>%**) con la función **filter()** del paquete **dplyr**, que se utiliza para seleccionar observaciones (filas):

filter(dataframe,criterio)

Nuestro objetivo es seleccionar todas las observaciones del objeto **CPS1985** que cumplan con el criterio: tener una experiencia (**experience**) mayor o igual a 10 años.

Utilizamos la función filter junto con sus argumentos para resolver el supuesto. El resultado lo asignamos al objeto **ej1**

```
> ej1 <- filter(CPS1985, experience >= 10)
```

Ahora, introducimos el uso de %>% en el siguiente código.

```
> ej2 <- CPS1985 %>% filter(. , experience >= 10)
```

El código anterior lo leemos así:

> Tomas el dataframe **CPS1985** y **luego** (es el pipe) lo pasas al primer argumento de la función filter (es lo que representa el .) y seleccionas las observaciones en las que la variable experience sea mayor o igual a 10 (es el criterio). El resultado lo asignas al objeto **ej2**.

Sin embargo, aún podemos simplificar el lenguaje utilizado. De hecho, la **forma más habitual de aplicar el pipe (%>%) es la que se muestra en el código de abajo**:

```
> ej3 <- CPS1985 %>%
+   filter(experience >= 10)
```

Este código lo leemos literalmente como sigue:

> Tomas el dataframe **CPS1985** y **luego** (es el pipe) filtras las observaciones en las que la variable *experience* sea mayor o igual a 10. El resultado lo asignas al objeto **ej3**.

Recordad, es crítico, el pipe (`%>%`) puede leerse como **luego** o **entonces**, y hace que el flujo de trabajo sea más lineal y fluído.

4.4 Proceso de ordenación de datos: `tidyr`.

Una vez hemos cargado los datos con los que vamos a trabajar comienza su proceso de limpieza y ordenación. El proceso de limpieza (**data cleaning process**) de un conjunto de datos consiste en hacer que los datos con el que vamos a trabajar sean consistenten para que resulte más fácil trabajar con ellos a la hora de realizar nuestras visualizaciones y análisis estadísticos. En este proceso las acciones que se llevan a cabo son, entre otras: la detección, corrección y/o eliminación de errores en los datos; la eliminación de datos redundantes; la detección y tratamiento de valores perdidos; la detección y tratamiento de valores anómalos; la unión de bases de datos para consolidar los datos requeridos en el análisis en una única fuente y ordenar los datos finales con los que se va a trabajar.

En este apartado suponemos que tenemos un conjunto de datos limpio y nos centramos en el **data tidying**, en hacer que los datos con los que vamos a trabajar sean lo más **tidy** (odenados) posible. Para ello, utilizaremos funciones del paquete `tidyr` de **tidyverse**.

Lo primero que tenemos que tener absolutamente claro es: ¿**qué son datos tidy (ordenados)**?

La mayoría de datos en Ciencias Sociales se ajustan a la categoría de datos tabulares; es decir, **están organizados en filas y columnas**. En R, como ya sabemos, este tipo de datos se almacenan en dataframes (o tibbles). En esencia, un dataframe/tibble será tidy cuando:

(1) cada columna sea una variable,

(2) cada fila sea una observación y

(3) cada valor tenga su propia celda.

El comentario anterior se refleja en la Figura 4.3.

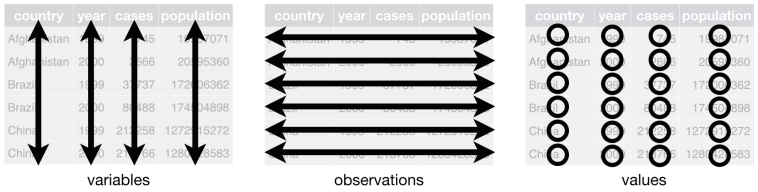


Figura 4.3: Datos tidy, datos ordenados. Fuente: https://r4ds.had.co.nz/tidy-data.html

Veamos un ejemplo. Supongamos que la variable (o atributo) a medir es el salario y la unidad de análisis las personas (son las observaciones o individuos). Hemos recogido datos del salario de 3 personas en el periodo 2021 a 2023.

```
> df1 <- data.frame(Periodo = c("2021", "2022", "2023"),
+                   Nicolas = c(100, 500, 200),
+                   Carla = c(400, 600, 250),
+                   Angela = c(200, 700, 900) )
>
> # nota: omito los acentos en el dataframe
>
> df1
  Periodo Nicolas Carla Angela
1    2021     100   400    200
2    2022     500   600    700
3    2023     200   250    900
```

Entendemos perfectamente los datos, visualmente son resultan cómodos de leer; pero, ¿son datos **tidy**?

No, no es una estructura de datos tidy. En este ejemplo, las observaciones (Nicolás, Carla, Ángela) se encuentran en columnas.

Y la siguiente estructura de datos, ¿es tidy?

```
> df2 <- data.frame(Nombre = c("Nicolas", "Carla", "Angela"),
+                   Salario_2021 = c(100, 400, 200),
+                   Salario_2022 = c(500, 600, 700),
+                   Salario_2023 = c(200, 250, 900))
>
> df2
   Nombre Salario_2021 Salario_2022 Salario_2023
1 Nicolas          100          500          200
2   Carla          400          600          250
3  Angela          200          700          900
```

Para nosotros también es un formato fácil de entender, pero no son tidy, aunque quizá sea el formato al que estamos más acostumbrados (individuos o registros en filas y "variables" en columnas). En la terminología de tidyverse este formato de datos es lo que llamamos un formato **"wide"** (o ancho). Podemos trabajar tranquilamente con esta estructura de datos. ¿Realmente Salario_2021, Salario_2022 y Salario_2023 son tres variables distintas?

Si queremos sacar el máximo rendimiento a tidyverse es mejor tener los datos en **long format** (formato largo). Fijaros en la siguiente estructura de datos:

```
> df3 <- data.frame(Nombre = c("Nicolas", "Carla", "Angela", "Nicolas",
+                              "Carla", "Angela","Nicolas", "Carla", "Angela"),
+                   Periodo = c("2021", "2021", "2021",  "2022",
+                               "2022", "2022",  "2023", "2023", "2023"),
+                   Salario = c(100, 400, 200, 500,
+                               600, 700, 200, 250,900) )
> df3
   Nombre Periodo Salario
1 Nicolas    2021     100
2   Carla    2021     400
3  Angela    2021     200
4 Nicolas    2022     500
5   Carla    2022     600
6  Angela    2022     700
7 Nicolas    2023     200
8   Carla    2023     250
9  Angela    2023     900
```

Los datos almacenados en **df3** son **tidy**. Este formato es más difícil de leer para nosotros, pero es más eficiente para los ordenadores; ¡¡y los datos los procesan los ordenadores!! De hecho, en muchas ocasiones, cuando nos descargamos datos de bases de datos, la estrucutura con la que nos los descargamos son en formato **long**.

4.4.1 Datos en formato largo y ancho: `pivot_longer()` y `pivot_wider()`.

Si tenemos un dataframe en formato **wide** tenemos que pasarlo a formato **long** para trabajar más eficientemente. Para ello utilizamos la función **pivot.longer()**.

Generalmente, para mostrar nuestros resultados pasamos de formato **long** a formato **wide**, porque los entendemos mejor. Para ello utilizamos la función **pivot.wider()**

Por si leemos código de otros usuarios, en versiones anteriores del paquete **tidyr** las funciones eran:

- para pasar de formato ancho a largo: **gather()**
- para pasar de formato largo a ancho: **spread()**

La función **pivot_longer()** convierte dataframes/tibbles de formato *wide* a formato *long*. Su estructura se ilustra en la Figura 4.4.

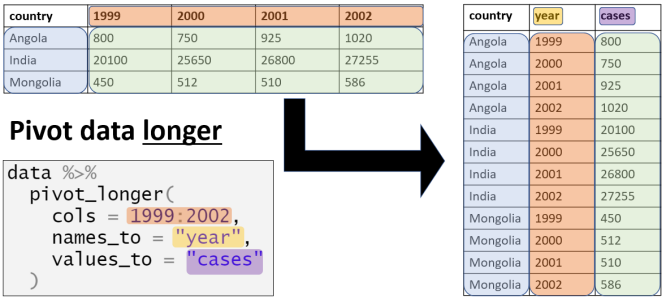


Figura 4.4: pivot_longer: de formato ancho a formato largo. Fuente: https://epirhandbook.com/en/index.html

Si pedimos la ayuda de la función **pivot_longer()** veremos que tiene muchos argumentos (ver Figura 4.5)

```
pivot_longer(
  data,
  cols,
  names_to = "name",
  names_prefix = NULL,
  names_sep = NULL,
  names_pattern = NULL,
  names_ptypes = NULL,
  names_transform = NULL,
  names_repair = "check_unique",
  values_to = "value",
  values_drop_na = FALSE,
  values_ptypes = NULL,
  values_transform = NULL,
  ...
)
```

Figura 4.5: Uso de la función: pivot_longer (ayuda de la función)

pero en la mayor parte de los casos es suficiente con trabajar con los básicos:

- *data*: dataframe/tibble.
- *cols*: columnas que queremos "recoger".
- *names_to*: nombre de la columna (variables) que recoge *cols*.
- *values_to*: nombre de la columnas (variable) que recoge los datos de las celdas.
- *values_drop_na*: si queremos eliminar los valores NA. Si es así, debemos pasar el argumento a TRUE.

Bien, el objeto **df2** está en formato ancho, lo pasamos a formato largo. El resultado lo guardamos en el objeto **data_long**.

```
> data_long <- df2 %>%
+   pivot_longer(cols=2:4,names_to="Periodo",values_to="Salario")
> data_long
# A tibble: 9 x 3
  Nombre  Periodo      Salario
```

```
  <chr>   <chr>          <dbl>
1 Nicolas Salario_2021     100
2 Nicolas Salario_2022     500
3 Nicolas Salario_2023     200
4 Carla   Salario_2021     400
5 Carla   Salario_2022     600
6 Carla   Salario_2023     250
7 Angela  Salario_2021     200
8 Angela  Salario_2022     700
9 Angela  Salario_2023     900
```

Luego volveremos sobre este objeto para arreglar la variable **Periodo**.

Pasar pasar de formato *long* a *wide*, `tidyr` dispone de la función **pivot_wider()**, como se muestra en la Figura 4.6.

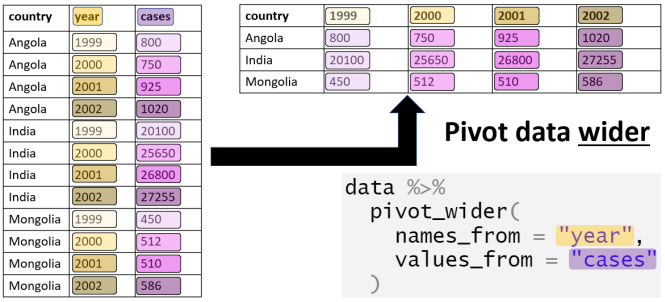


Figura 4.6: pivot_wider: de formato largo a formato ancho. Fuente: https://epirhandbook.com/en/index.html

El uso de la función **pivot_wider()** puede verse en la ayuda y se reproduce en la Figura 4.7.

```
pivot_wider(
  data,
  id_cols = NULL,
  id_expand = FALSE,
  names_from = name,
  names_prefix = "",
  names_sep = "_",
  names_glue = NULL,
  names_sort = FALSE,
  names_vary = "fastest",
  names_expand = FALSE,
  names_repair = "check_unique",
  values_from = value,
  values_fill = NULL,
  values_fn = NULL,
  unused_fn = NULL,
  ...
)
```

Figura 4.7: Uso de la función: pivot_wider (ayuda de la función)

En la mayor parte de las ocasiones solo vamos a utilizar, al menos por ahora, tres argumentos:

- *data*: dataframe/tibble.

- *names_from*: nombre/posición de la columna (variable) que queremos extender.
- *values_from*: nombre/posición de la columna (variable) de donde se recogerán los datos de las celdas.

Por ejemplo, convertimos en formato ancho el objeto **data_long** que se encuentra en formato largo. Guardamos el resultado en el objeto **data_wide**.

```
> data_wide <- data_long %>%
+   pivot_wider(names_from = "Periodo", values_from = "Salario")
>
> data_wide
# A tibble: 3 x 4
  Nombre  Salario_2021 Salario_2022 Salario_2023
  <chr>          <dbl>        <dbl>        <dbl>
1 Nicolas          100          500          200
2 Carla            400          600          250
3 Angela           200          700          900
```

4.4.2 Las funciones: `separate()` y `unite()`.

Como puede deducirse por los nombres, estas funciones se utilizan para separar y unir columnas.

Si recordamos el objeto **data_long**, la variable **Periodo** tomaba los valores: Salario_2020, Salario_2021 y Salario_2022. La función **separate()** nos permitiría separar esta variable en dos (o más si queremos) columnas. Observemos el siguiente código:

```
> data_long <- data_long %>%
+   separate(Periodo, c("Año", "Valor"), sep = "_")
>
> data_long
# A tibble: 9 x 4
  Nombre  Año     Valor Salario
  <chr>   <chr>   <chr>   <dbl>
1 Nicolas Salario 2021      100
2 Nicolas Salario 2022      500
3 Nicolas Salario 2023      200
4 Carla   Salario 2021      400
5 Carla   Salario 2022      600
6 Carla   Salario 2023      250
7 Angela  Salario 2021      200
8 Angela  Salario 2022      700
9 Angela  Salario 2023      900
```

Hemos separado la variable **Periodo** en dos columnas, a las que hemos llamado **Año** y **Valor**. En el argumento *sep* indicamos el separador de columnas.

Para unir columnas utilizamos para la función **unite()**. En el siguiente ejemplo unimos en la columna **Periodo_y_Salario** desde la columna **Año** hasta la columna **Valor**, y el contenido se separará con un espacio en blanco.

```
> data_long <- data_long %>%
+   unite(Periodo_y_Salario, Año:Valor, sep = " ")
>
> data_long
# A tibble: 9 x 3
  Nombre  Periodo_y_Salario Salario
  <chr>   <chr>                <dbl>
1 Nicolas Salario 2021           100
2 Nicolas Salario 2022           500
3 Nicolas Salario 2023           200
4 Carla   Salario 2021           400
5 Carla   Salario 2022           600
6 Carla   Salario 2023           250
7 Angela  Salario 2021           200
8 Angela  Salario 2022           700
9 Angela  Salario 2023           900
```

4.5 Transformación de datos: `dplyr`.

El paquete **`dplyr`** lo utilizamos para transformar/manipular los datos. Es decir, para seleccionar observaciones (filas) o columnas (variables), crear nuevas variables, obtener resúmenes (contar observaciones, calcular medidas estadísticas,...), etc.

Las principales funciones (**verbos**) de **`dplyr`** para la transformación de datos son:

- **`filter()`**: filtrar datos (idea similar al filtrado de Excel). Permite seleccionar filas que cumplan con una o varias condiciones.
- **`group_by()`**: agrupar filas según las categórias de una o variables variables.
- **`summarize()`**: resumir (colapsar) datos a un solo valor (según una función: media, desviación típica, etc.).
- **`mutate()`**: crear nuevas variables.
- **`select()`**: seleccionar variables (columnas).
- **`arrange()`**: ordenar filas.
- **`join()`**: unir dataframes.

Con estas funciones se pueden resolver la gran mayoría de problemas asociados a la manipulación de datos. Cada una de estas funciones hace "solo una cosa", así que para realizar transformaciones complejas hay que ir concatenando instrucciones sencillas. Todas las funciones tienen una estructura o comportamiento similar:

- el primer argumento siempre es un dataframe/tibble.
- los siguientes argumentos describen qué hacer con los datos.

El resultado es siempre un nuevo dataframe/tibble.

Antes de continuar, la cheat sheet de **dplyr** puede descargarse del siguiente enlace: **https://www.uv.es/vcoll/LIBRO_ESTADISTICA_CON_R/dplyr.pdf**

En los siguientes subapartados se explica cómo hacer uso de las funciones básicas de **dplyr** con la excepción de la función **join()**.

Para ejemplificar el uso de las funciones de **dplyr** usaremos el dataset **CPS1985** que hemos cargado al inicio del capítulo.

4.5.1 Selección de observaciones: `filter()`.

Esta función se utiliza para seleccionar filas de un dataframe que cumplan determinado criterio.

Por ejemplo, vamos a seleccionar los trabajadores en el dataframe CPS1985 que están casados. Lo primero es saber qué distintas categorías tiene la variable **married** (casado) y después aplicar el criterio de selección para realizar el filtrado (**married == "yes"**).

```
> # Valores únicos que toma la variable married (casados)
> unique(CPS1985$married)
[1] yes no
Levels: no yes
>
> # Guardamos la selección en el objeto aa
> aa <- CPS1985 %>%
+   filter(married == "yes")
>
> nrow(aa)   #alternativamente dim(aa)[1]
[1] 350
```

Por tanto, en el conjunto de datos un total de 350 trabajadores están casados.

Ahora nos preguntamos, ¿qué trabajadores tienen menos de 10 años de educación? En este caso el criterio de selección es **education < 10**. Guardamos el resultado en el objeto **aa**.

En primer lugar, podemos preguntarnos por los valores que toma la variable objeto de estudio. Como vemos en el código más abajo, la variable **education** (educación) toma 17 valores distintos; en el código se ha utlizado la función **sort()** para ordenar los valores de menor a mayor. Así, fácilmente observamos que el menor valor de educación es 2 años y el mayor es 18 años. Utlizamos la función **dim()** para obtener la dimensión del dataframe **aa**; como el resultado de **dim** es un vector que indica filas y columnas, seleccionamos el primer elemento, las filas.

```
> # valores únicos de education:
> unique(CPS1985$education) # con R-base
 [1]  8  9 12 13 10 16  7 11  6 14 17  3 15  5 18  4  2
>
> # ordenación de los valores únicos de education:
> sort(unique(CPS1985$education)) # con R-base
 [1]  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
>
> # Selección de trabajadores de acuerdo con el criterio
```

```
> aa <- CPS1985 %>%
+   filter(education < 10)
>
> dim(aa)[1]
[1] 39
```

Un total de 39 trabajadores tienen menos de 10 años de educación.

¿Cuántos trabajadores tienen entre 10 y 15 años de educación (ambos inclusive)?

Podemos responder a esta preguntas de varias formas. En la tercera opción utilizamos conjuntamente la función **between()** y **filter()** de **dplyr**.

```
> aa <- CPS1985 %>%
+   filter(education >= 10 , education <= 15)
>
> aa2 <- CPS1985 %>%
+   filter(education >= 10 & education <= 15)
>
> # Con la función between.
> # consultar el uso de la función: ?between
> aa3 <- CPS1985 %>%
+   filter(between(education, 10, 15))
```

Para establecer el criterio A Y B (es una intersección, deben cumplirse al mismo tiempo el criterio A y el criterio B), podemos utilizar indistintamente los símbolos: **,** o **&**

Planteamos otro ejemplo. ¿Cuántos trabajadores casados y con años de educación mayor o igual a la media hay en nuestro conjunto de datos?

```
> aa <- CPS1985 %>%
+   filter(married=="yes", education >= mean(education))
```

El número medio de años de educación de los trabajadores en nuestro dataset es de 13.02 años. Un total de 125 trabajadores están casados y tienen una educación igual o superior a 13.02 años (como educación toma valores enteros, se han seleccionado todos los trabajadores con 14 o más años de educación).

Por último, también podemos establecer como criterio de selección que se cumpla la condición A o B. Este criterio correspondería al concepto de unión en teoría de conjuntos: que se cumpla sólo A, o solo B, o ambos (A y B). Para realizar esta selección se utiliza el símbolo |

Por ejemplo, ¿qué trabajadores están empleados en el sector construction o están afiliados a un sindicato? En caso de estar interesados en trabajar sobre el resultado del filtrado, asignaríamos el subconjunto de datos obtenidos a un objeto.

```
> unique(CPS1985$sector) # para conocer los niveles de la variable
[1] manufacturing other         construction
Levels: manufacturing construction other
>
> aa <- CPS1985 %>%
+   filter(sector == "construction" | union == "yes")
```

R ordena alfabéticamente -por defecto- los niveles de los factores. Sin embargo, observad que esto no ocurre con el conjunto de datos con el que estamos trabajando.

A veces interesa seleccionar determinadas observaciones atendiendo a la posición que ocupan. En estos casos en lugar de **filter()** puede utilizarse la función **slice()**. Por ejemplo, supongamos que estamos interesados en seleccionar las observaciones que se encuentran en las posiciones (filas): 2, 5, 15 y 27.

```
> aa <- CPS1985 %>%
+   slice(c(2,5,15,27))
```

Como podemos ver en la seccion *extraer casos* de la cheat sheet de dplyr, la función **slide** tiene distintas variantes: **slide_sample**, para seleccionar aleatoriamente filas; **slide_min** y **slide_max**, que seleccionan filas con los menores y los mayores valores; y **slide_head** y **slide_tail**, que seleccionan primeras o últimas filas (son equivalentes a **head** y **tail** que vimos en el apartado 2.4.2.1).

Veamos una aplicación. Para seleccionar el 25% de las observaciones que tienen un menor salario escribiríamos el siguiente código. El 25% de las observaciones seleccionadas serán ordenadas en orden ascedente según el salario.

```
> aa <- CPS1985 %>%
+   slice_min(wage,prop=0.25)
```

Para terminar con la selección de observaciones (casos), si queremos eliminar filas con casos duplicados, utilizamos la función **distinct()**. Para más detalles consultar la ayuda de la función.

4.5.2 Agrupar por categoría de una variable: `group_by()`.

Con esta función ya empezaremos a ver la potencia de **dplyr**. En el análisis de datos es frecuente que determinadas medidas (media, mediana, moda, cuantiles, etc.) queramos calcularlas para distintos grupos (hombre, mujer, etc.). Por ejemplo, podemos estar interesados en conocer el salario medio de todos los trabajadores (hombres y mujeres), pero también es bien interesante conocer el salario medio de los hombres y el de las mujeres. La función **group_by()** es la que nos va a permitir hacer este tipo de operación. Como podemos ver en la Figura 4.8, lo que hace **group_by()** es agrupar las observaciones según los valores/categorías de una o más variables.

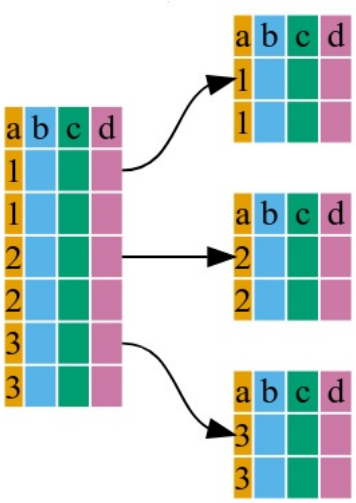


Figura 4.8: Lógica de la función group by. Fuente: R for Reproducible Scientific Analysis (https://swcarpentry.github.io/r-novice-gapminder/13-dplyr/index.html)

Es decir, **group_by()** considera un dataframe y lo convierte en un **"dataframe agrupado"**. En ese nuevo "dataframe agrupado" las operaciones que hagamos con **summarize()** (resumir) se harán por separado para cada uno de los grupos que hayamos definido. Ahora lo vemos.

Si, por ejemplo, agrupamos nuestro dataframe de **CPS1985** por la varaiable **gender**, la función **summarize()** nos dará el resumen para cada categoría de **gender**.

De la lectura de esta función se desprende una idea importante: **group_by()** y **summarize()** se utilizan (normalmente) juntas.

En **CPS1095**, ¿cuántas observaciones (rows) tenemos de cada categoría de **gender**? Para responder a esta pregunta primero vamos a agrupar el conjunto de datos por **gender** y luego realizaremos el conteo de observaciones en cada grupo. Como queremos resumir la información utilizamos la función **summarize** y dentro de ella el resumen que queremos hacer: el conteo de observaciones, Para esto último puede utilizarse la función **n()**.

```
> CPS1985 %>%
+   group_by(gender) %>%
+   summarize(total = n()) # la función n() devuelve el tamaño (número de casos) de cada grupo
# A tibble: 2 x 2
  gender total
  <fct>  <int>
1 male     289
2 female   245
>
> CPS1985 %>%
+   summarize(total = n_distinct(gender))
  total
1     2
```

Para contar los valores únicos de una o más variables también podemos utilizar la función **count()**. El mismo resultado que en el código anterior podíamos haberlo obtenido de esta otra forma:

```
> CPS1985 %>%
+   count(gender)
  gender   n
1   male 289
2 female 245
```

Consultad la ayuda de la función **count()**.

dplyr tiene otra función que permite contar el número de valores únicos (o distintas combinaciones) en uno o más vectores, esta función es **n_distinct()**, sería equivalente a escribir esta instrucción en R-base: nrow(unique(NUESTRO_DATA_FRAME)))

¿Y cómo es la distribución de la raza (**ethnicity**) por sexo (**gender**)? Dependiendo de lo que queramos mostrar, primero podemos agrupar por *etnicity* y luego cada categía de esta variable la agrupamos por *gender*, o al contrario. Observad los resultados del siguiente código

```
> # primero agrupamos por género y luego por raza
> CPS1985 %>%
+   group_by(gender, ethnicity) %>%
+   summarize(total = n())
# A tibble: 6 x 3
# Groups:   gender [2]
  gender ethnicity total
  <fct>  <fct>     <int>
1 male   cauc        236
2 male   hispanic     14
3 male   other        39
4 female cauc        204
5 female hispanic     13
6 female other        28
>
> # primero agrupamos por raza y luego por género
> CPS1985 %>%
+   group_by(ethnicity, gender) %>%
+   summarize(total = n())
# A tibble: 6 x 3
# Groups:   ethnicity [3]
  ethnicity gender total
  <fct>     <fct>  <int>
1 cauc      male     236
2 cauc      female   204
3 hispanic  male      14
4 hispanic  female    13
5 other     male      39
6 other     female    28
>
> # en group_by estamos utilizando la , para separar las variables de agrupación
> # el orden en que se escriben es el orden de la agrupación
```

Alternativamente,

```
> CPS1985 %>%
+   count(ethnicity,gender)
  ethnicity gender   n
1      cauc   male 236
2      cauc female 204
3  hispanic   male  14
4  hispanic female  13
5     other   male  39
6     other female  28
```

¿Cuál es el salario medio de los trabajadores según el sexo y la raza?

En esta ocasión agrupamos los datos por género y raza y luego para cada subconjunto de datos resultado de la agrupación calculamos la media.

```
> CPS1985 %>%
+   group_by(gender,ethnicity) %>%
+   summarize(Salario_medio = mean(wage,na.rm=TRUE))
# A tibble: 6 x 3
# Groups:   gender [2]
  gender ethnicity Salario_medio
  <fct>  <fct>             <dbl>
1 male   cauc              10.3
2 male   hispanic           8.66
3 male   other              8.46
4 female cauc               8.06
5 female hispanic           5.80
6 female other              7.49
>
> # es conveniente introducir en la función de la media el argumento na.rm=TRUE
> # El argumento lo que hace es que si hay NAs los quita para caluclar la media.
```

4.5.3 Resumen con `summarize()`.

Como hemos podido comprobar en los ejemplos anteriores, la función **summarize()** se utiliza para **resumir** (o "colapsar filas"). Toma un grupo de valores como input y devuelve un solo valor; por ejemplo, calcula la media aritmética (o el mínimo, o el máximo …) de un grupo de valores.

Vamos a calcular algunos estadísticos de una variable. Realmente, para hacer esto no nos hace falta **dplyr**, pero conviene que nos vayamos habituando a su sintaxis. Tratamos de leer el código que se muestra más abajo y pensamos en el resultado que esperamos nos devuelva.

```
> #- devuelve un único valor: la media de la variable "wage".
> CPS1985 %>%
+   summarize(media = mean(wage))
```

```
>
> #- devuelve un único valor: el número de filas
> CPS1985 %>%
+   summarize(total_observaciones = n())
>
> #- devuelve un único valor: la cuasi desviación típica de "wage"
> CPS1985 %>%
+   summarize(desviacion_tipica = sd(wage))
>
> #- devuelve un único valor: el máximo de la variable "wage"
> CPS1985 %>%
+   summarize(max(wage))
>
> #- devuelve 2 valores: la media y la cuasi desviación típica de "wage""
> CPS1985 %>%
+   summarize(media_mpg = mean(wage),
+             sd_mpg = sd(wage))
>
> #- devuelve 2 valores: las medias de "wage" y "education"
> CPS1985 %>%
+   summarize(media_wage = mean(wage),
+             media_education = mean(education))
>
> #- devuelve 4 valores: la medida y cuasi desviación típica de "wage" y "education"
> CPS1985 %>%
+   summarize_at(vars(wage,education), funs(mean, sd))    # funs() está obsoleta, se recomienda list()
>
> CPS1985 %>%
+   summarize_at(vars(wage,education), list(mean, sd))
>
> CPS1985 %>%
+   summarize_at(vars(wage,education), list(media=mean, desviacion=sd))
>
```

En el capítulo 7, dedicado al estudio de las medidas estadísticas descriptivas básicas, estudiaremos en los apartados 7.6.3 y 7.6.4 la diferencia entre **desviación típica (muestral)** y **cuasi desviación típica (muestral)**.

La función **summarize_at()**, que ya hemos utilizado en un ejemplo anterior, permite seleccionar las columnas sobre las que se pasará las funciones de resumen.

```
> CPS1985 %>%
+   filter(occupation %in% c("worker", "technical","sales")) %>%
+   group_by(ethnicity) %>%
+   summarize_at(vars(wage, education), list(media = mean, mediana = median))
# A tibble: 3 x 5
  ethnicity wage_media education_media wage_mediana education_mediana
  <fct>          <dbl>           <dbl>        <dbl>             <dbl>
```

```
1 cauc           9.86          13.4          8.9           12
2 hispanic       7.12          11.2          6             12
3 other          7.97          12.2          7             12
```

También podemos calcular estadísticos de **todas las variables** del dataframe con la función **summarize_all()**. Si la variable no es numérica y pasamos soló una función, se devolverá NA; si pasamos más de una función entonces devolverá un error. Observad los siguientes dos ejemplos.

```
> #- media de cada una de las 11 variables.
> CPS1985 %>%
+   summarize_all(mean)
>
> #- media y cuasi desviación típica de las 11 variables.
> CPS1985 %>%
+   summarize_all(funs(mean, sd) )     # funs() está obsoleta, se recomienda cambiar a list()
>
```

Por último, la función **summarize_if()** aplicará una (o varias) funciones de resumen sobre las columnas que devuelvan el valor lógico *TRUE* al cumplirse la condición.

```
> CPS1985 %>%
+   filter(occupation %in% c("worker", "technical","sales")) %>%
+   group_by(ethnicity) %>%
+   summarize_if(is.numeric, funs(media=mean, mediana=median))  # si la variable
# A tibble: 3 x 9
  ethnicity wage_media education_media experience_media age_media wage_mediana
  <fct>          <dbl>           <dbl>            <dbl>     <dbl>        <dbl>
1 cauc            9.86            13.4             16.8      36.1          8.9
2 hispanic        7.12            11.2             17.5      34.7          6
3 other           7.97            12.2             20.5      38.8          7
# i 3 more variables: education_mediana <dbl>, experience_mediana <dbl>,
#   age_mediana <dbl>
> # es numérica, calcula la media y la mediana.
```

En ejemplos anteriores hemos utilizado el operador **%in%**. Este operador se utiliza para comprobar si los valores del primer argumento se encuentran en el segundo. El resultado de aplicar este operador es un vector lógico. En los ejemplos se ha utilizado **%in%** como criterio para seleccionar filas con **filter**.

4.5.4 Crear variables: `mutate()`.

Con esta función **creamos nuevas variables** (columnas). Es muy útil en análisis de datos.

Por ejemplo, supongamos que queremos crear una columna que refleje el salario semanal. Tenemos la variable salario por hora (**wage**). Suponiendo que se trabaja 8 hora al día de lunes a viernes, tendríamos que el salario semanal se obtendría a partir de la expresión:

$$salario.semanal = 5 * 8 * wage$$

Creamos la variable: `salario.semanal = 40 * wage`

```
>  CPS1985 <- CPS1985 %>%
+   mutate(salario.semanal = 40 * wage)
```

La estructura de **CPS1985** ha cambiado, ahora tiene 12 variables en lugar de las 11 iniciales. Por defecto, la nueva variable creada se sitúa en la última columna. El número de observaciones no ha cambiado. Si queremos crear la nueva variable en una determinada posición podemos hacer uso de los argumentos *.after*, para situarla depués de una variable dada, o *.before*, para situarla antes.

```
>  CPS1985 <- CPS1985 %>%
+   mutate(salario.semanal = 40 * wage, .after=wage)    # sitúa salario.semanal después de wage
>
>
>  CPS1985 <- CPS1985 %>%
+   mutate(salario.semanal = 40 * wage, .before=wage)    # sitúa salario.semanal antes de wage
```

Si queremos manipular varias variables al mismo tiempo podemos recurrir a la función **across()**. La estructura de la función **across** es: **across(.cols, .funs, ..., .names = NULL)**. Lo que hace **across** es pasar una o varias funciones (argumento *.funs*) a múltiples columnas. La función **across()** se puede utilizar combinada con **summarize()**, **mutate()** o **select()**.

Calculamos la media de todas las variables de CPS1985.

```
> CPS1985 %>%
+   summarize(across(everything(),mean))
      wage education experience      age ethnicity region gender occupation
1 9.024064  13.01873    17.8221 36.83333        NA     NA     NA         NA
  sector union married
1     NA    NA      NA
```

everything(), *starts_with()*, *ends_with()*, *contains()*, *matches()*,etc. se utilizan con las funciones **select()** y **across()** para ayudar en la selección.

Al ejercutar la instrucción obtenemos las medias de las variables numéricas, pero también un *warning* porque hemos especificado el cálculo de la media para todas las variables y en **CPS1985** tenemos variables cualitativas como **ethnicity** o **region**. En esta situación, podemos especificar las posiciones de las variables numéricas o, alternativamente, hacer uso de la función **where()**, que permite seleccionar variables con una función. Consideremos los siguientes ejemplos:

```
> # aplica sobre las primeras cuatro columnas la media
> CPS1985 %>%
+   summarize(across(1:4,mean))
      wage education experience      age
1 9.024064  13.01873    17.8221 36.83333
>
> # aplica sobre las primeras cuatro columnas la media y la mediana
> CPS1985 %>%
+   summarize(across(1:4,list(media=mean,mediana=median)))
  wage_media wage_mediana education_media education_mediana experience_media
1   9.024064         7.78        13.01873                12          17.8221
  experience_mediana age_media age_mediana
1                 15  36.83333          35
> # selecciona las columnas numéricas y aplica la media
> CPS1985 %>%
+   summarize(across(where(is.numeric),mean))
      wage education experience      age
1 9.024064  13.01873    17.8221 36.83333
```

Antes de pasar al siguiente verbo/función de **dplyr**, un ejemplo de uso combinado de **mutate()** y **across()**. Observad que en el siguiente código **where(is.factor)** selecciona las columnas que satisfacen la condición, luego se aplica para esas columnas las función **as.character**.

```
> # convertimos todas las variables factor a carácter
> CPS1985 <- CPS1985 %>%
+   mutate(across(where(is.factor), as.character))
```

4.5.5 Seleccionar variables: `select()`.

Esta función sirve para **seleccionar columnas** (o variables si el fichero es tidy) por nombre o posición.

Podemos seleccionar las variables por nombre. Por ejemplo, vamos a seleccionar de **CPS1985** las varibles **wage** y **gender**, el dataframe resultante lo guardamos en el objeto **aa**.

```
> #- se leerá: toma el dataframe CPS1985 y entonces selecciona wage y gender
> aa <- CPS1985 %>%
+   select(wage, gender)
>
> aa <- CPS1985 %>%
+   select(c(wage, gender))  # equivalente al anterior
```

Si queremos seleccionar de **CPS1985** todas las variables excepto **union** escribiríamos:

```
> aa <- CPS1985 %>%
+   select(-union)
```

En el caso de querer eliminar más de una variable lo que hacemos es nombrar esas variables dentro de un vector:

```
> aa <- CPS1985 %>%
+ select(-c(union, married))
```

Estamos realizando la selección por el nombre de la variable, pero también podemos realizar la selección por la posición (número de columna) que ocupa en el dataframe.

Seleccionamos las siguientes variables de nuestro dataframe: de la primera a la tercera y también la quinta.

```
> aa <- CPS1985 %>%
+   select(1:3,5)
>
> # particularmente, prefiero seleccionar por nombre
```

De forma análoga, podemos estar interesados en seleccionar todas las variables del dataframe **excepto** las siguientes: de la primera a la tercer y la quinta.

```
> aa <- CPS1985 %>%
+   select(- c(1:3, 5))
```

La función `select()` también puede utilizarse para renombrar variables (columnas) o para reordenarlas.

De **CPS1985** queremos seleccionar, por este orden, **age** y **wage**.

```
> aa <- CPS1985 %>%
+   select(age, wage)
```

Pero no solo eso, quiero seleccionar **age** y **wage** y cambiarles el nombre. Para ello, escribimos:

```
> aa <- CPS1985 %>%
+   select(Edad = age, Salario.hora = wage)
```

Supongamos que queremos que la variable **experience** de **CPS1985** a pase a ocupar la primera columna en el dataframe. Esto lo podemos hacer con la función **select()** y `everything()`, que es una *función auxiliar de select*:

```
> #- "experience" que es la tercera columna pasa a ser la primera
> aa <- CPS1985 %>%
+   select(experience, everything())
```

`dplyr` cuenta con otras dos funciones relacionadas con la extracción de variables. Por un lado, tenemos la función **pull()**, que extrae los valores de una columna como un vector. Por otro lado, está la función **relocate()**, que nos permite mover columnas. Vemos los siguientes ejemplos:

```
> # extraemos el salario (utilizando el nombre de la variable)
> CPS1985 %>%
+   pull(wage)
>
> # extraemos el salario (utilizando la posición de la variable)
> CPS1985 %>%
+   pull(1)
>
> # movemos las columnas wage y education delante de married
> CPS1985 <- CPS1985 %>%
+   relocate(c(wage,education), .before=married)
>
> # movemos las columnas wage y education detrás de married
> CPS1985 <- CPS1985  %>%
+   relocate(c(wage,education), .after=married)
```

4.5.6 Ordenar los datos: `arrange()`.

Esta función se utiliza para **ordenar las filas** de una o más variables de un dataframe. Por defecto los valores se ordenan en onrden ascendente, de menor a mayor. Si queremos ordenar de mayor a menor, orden descendente, hay que cambiar usar la función auxiliar **desc()**. Aquí hay algunos ejemplos.

```
> #- ordena las filas de MENOR a mayor según los valores de "wage""
> aa <- CPS1985 %>%
+   arrange(wage)
>
> #- ordena las filas de MAYOR a menor según los valores de "wage""
> aa2 <- CPS1985 %>%
+   arrange(desc(wage))
>
> #- ordenada las filas de MENOR a mayor según los valores "wage" (primero) y si hay empates
> # se resuelve con la variable "experience"
> aa3 <- CPS1985 %>%
+   arrange(wage, experience) # ver el dataframe
```