La Gramática de la Manipulación de Datos

Unidad 3: Descripción y Visualización de Datos

Gabriel Sotomayor

2025-09-22

Objetivos de la Sesión de Hoy

- **Introducir** la “gramática de la manipulación de datos” del paquete `dplyr`.
- **Aprender y aplicar** los cinco “verbos” principales de `dplyr`: `select()`, `filter()`, `arrange()`, `mutate()` y `summarise()`.
- **Entender y utilizar** el operador *pipe* (`%>%`) para encadenar operaciones de forma clara y legible.
- **Aplicar** la combinación `group_by()` + `summarise()` para calcular estadísticas descriptivas por subgrupos.

1. Una Forma Intuitiva de Trabajar: El Tidyverse y `dplyr`

Repaso y Conexión

En la clase anterior, aprendimos el **protocolo de exploración inicial**:

1. Cargamos una base de datos (`haven::read_sav`).
2. Inspeccionamos su estructura (`dim`, `str`, `summary`).
3. Creamos una submuestra para hacerla manejable (`casen_sub <- casen[...]`).
4. Calculamos nuestras primeras estadísticas descriptivas (`table`, `mean`).

El Desafío: A medida que nuestras preguntas se vuelven más complejas, como "calcular el ingreso promedio *solo de las mujeres de la Región Metropolitana*", el código en R base puede volverse anidado y difícil de leer. Hoy aprenderemos un enfoque más intuitivo.

El Ecosistema Tidyverse: Un Lenguaje para la Ciencia de Datos

El **Tidyverse** no es solo un paquete, es una **colección de paquetes** diseñados para trabajar juntos de forma coherente en todo el ciclo de análisis de datos. Comparte una filosofía común que hace el código más **legible, consistente e intuitivo**.

- **Origen del Nombre:** Viene del concepto de **"tidy data" (datos ordenados)**, que es el formato de datos ideal para el análisis.
- **Paquetes Principales:** Incluye herramientas esenciales como:
  - `dplyr`: Para manipulación de datos.
  - `ggplot2`: Para visualización de datos.
  - `readr`: Para importar datos de archivos de texto.
  - `tidyr`: Para ordenar y reestructurar datos.
  - Y muchos más (`purrr`, `stringr`, `forcats`...).

El Ecosistema Tidyverse: Un Lenguaje para la Ciencia de Datos

Al cargar el paquete `tidyverse` con `library(tidyverse)`, se cargan automáticamente los paquetes más importantes de este ecosistema.

```r
# Al cargar 'tidyverse', nos informa qué paquetes se adjuntan
library(tidyverse)
```

```
── Attaching core tidyverse packages ──────────────────────── tidyverse 2.0.0 ──
✓ dplyr     1.1.4     ✓ readr     2.1.5
✓ forcats   1.0.0     ✓ stringr   1.5.1
✓ ggplot2   3.5.1     ✓ tibble    3.2.1
✓ lubridate 1.9.4     ✓ tidyr     1.3.1
✓ purrr     1.0.4
── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
✖ dplyr::filter() masks stats::filter()
✖ dplyr::lag()    masks stats::lag()
ℹ Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become errors
```

`dplyr`: La Gramática de la Manipulación de Datos

El paquete `dplyr` es el corazón del Tidyverse para la manipulación de datos. Su diseño se basa en una "gramática" simple y consistente, definida por tres reglas clave:

1. **El primer argumento es siempre un `data.frame`.**
    - Todas las funciones principales de `dplyr` están diseñadas para recibir una tabla de datos como su primer insumo.
2. **Los argumentos siguientes describen qué hacer, usando los nombres de las variables sin comillas.**
    - Esto hace que el código sea mucho más legible y menos propenso a errores de tipeo. `select(datos, edad)` en lugar de `datos[, "edad"]`.
3. **El resultado es siempre un nuevo `data.frame`.**
    - `dplyr` nunca modifica tu base de datos original. Cada operación genera una nueva tabla. Esto promueve un flujo de trabajo seguro y reproducible.

Esta gramática consistente es lo que permite que las funciones se encadenen de manera fluida y predecible.

El Operador Pipe %>%: Escribiendo Código como se Piensa

El *pipe* (tubería) es el operador `%>%` y es la pieza central que hace que el código del Tidyverse sea tan legible.

- **¿Qué hace?** Toma el resultado del código a su izquierda y lo pasa como el **primer argumento** de la función a su derecha.
- **¿Por qué es útil?** Nos permite encadenar operaciones de izquierda a derecha, como si estuviéramos leyendo una frase, en lugar de anidar funciones.

Sin pipe (difícil de leer "de adentro hacia afuera"):

```
summarise(group_by(filter(casen, region == 13), sexo), edad_promedio = mean(edad, na.rm = TRUE))
```

Con pipe (claro y secuencial "de izquierda a derecha"):

```
casen %>%
  filter(region == 13) %>%
  group_by(sexo) %>%
  summarise(edad_promedio = mean(edad, na.rm = TRUE))
```

Esto se lee como: "Toma la base CASEN, **luego** filtra por la RM, **luego** agrupa por sexo, **luego** calcula la edad promedio".

El Pipe Nativo de R: |>

A partir de la versión 4.1.0, R introdujo su propio operador de *pipe* "nativo": `|>`. Su función es muy similar al `%>%` del Tidyverse.

Sintaxis: `datos |> funcion()`

Ejemplo:

```r
# Usando el pipe nativo de R
casen |>
  filter(region == 13) |>
  group_by(sexo) |>
  summarise(edad_promedio = mean(edad, na.rm = TRUE))
```

- El pipe nativo `|>` es un poco más rápido pero, por ahora, ligeramente **menos flexible** que `%>%`. Algunas funciones y paquetes más antiguos del Tidyverse aún no son totalmente compatibles con él.
- Se puede escribir el pipe pulsando CTRL + SHIFT + M. Se peude cambiar el pipe escrito con dicho atajo en las opciones de RStudio.

2. Los Verbos Fundamentales de `dplyr`

Los 5 Verbos Principales

`dplyr` se basa en un conjunto pequeño de "verbos" o funciones que resuelven la gran mayoría de las tareas de manipulación de datos. Cada verbo realiza una acción específica y predecible.

1. `select()`: **Selecciona columnas** (variables).
2. `filter()`: **Filtra filas** (observaciones) según condiciones.
3. `arrange()`: **Reordena las filas**.
4. `mutate()`: **Crea nuevas columnas** (variables).
5. `summarise()`: **Resume** los datos en un solo valor.

Hoy nos enfocaremos en estos cinco.

`select()`: Trabajando con Columnas

`select()` nos permite quedarnos con las variables que nos interesan, descartando el resto.

Sintaxis: `datos %>% select(variable1, variable2, ...)`

```
# Seleccionar solo las variables region, sexo y edad
casen_seleccion <- casen %>%
  select(region, sexo, edad)

# También podemos excluir variables con el signo menos (-)
# Nos quedamos con todas las variables EXCEPTO folio y id_vivienda
casen_sin_ids <- casen %>%
  select(-folio, -id_vivienda)
```

`select()` tiene “helpers” muy útiles como `starts_with("y_")` para seleccionar todas las variables que empiezan con “y_”, o `everything()` para mover columnas de lugar.

La Lógica del Filtrado: Operadores de Comparación

Antes de usar `filter()`, necesitamos entender cómo construir las "condiciones lógicas". El primer paso es usar **operadores de comparación** para crear una pregunta que R pueda responder con `TRUE` o `FALSE`.

Operador	Significado	Ejemplo
`==`	**Igual a**	`region == 13` (¿La región es igual a 13?)
`!=`	**No igual a**	`sexo != 1` (¿El sexo es distinto de 1?)
`>`	**Mayor que**	`edad > 60` (¿La edad es mayor a 60?)
`>=`	**Mayor o igual que**	`esc >= 12` (¿La escolaridad es 12 años o más?)
`<`	**Menor que**	`ingreso < 500000` (¿El ingreso es menor a 500.000?)
`<=`	**Menor o igual que**	`num_hijos <= 2` (¿El número de hijos es 2 o menos?)
`%in%`	**Pertenece a un conjunto**	`region %in% c(5, 8)` (¿La región es 5 u 8?)

Combinando Condiciones: Operadores Lógicos

A menudo, nuestras preguntas de investigación requieren combinar varias condiciones. Para esto, usamos los **operadores lógicos**.

Operador	Significado	Lógica	Ejemplo
`&`	**Y (AND)**	Devuelve `TRUE` solo si **ambas** condiciones son verdaderas.	`sexo == 2 & edad > 60` (Personas que son mujeres **Y** mayores de 60).
`|`	**O (OR)**	Devuelve `TRUE` si **al menos una** de las condiciones es verdadera.	`region == 5 | region == 8` (Personas que son de la región 5 **O** de la región 8. Es equivalente a `region %in% c(5, 8)`).
`!`	**NO (NOT)**	Invierte una condición lógica. Devuelve `TRUE` si la condición es falsa.	`!is.na(ingreso)` (Personas cuyo ingreso **NO** es un valor perdido).

El orden de las operaciones importa. Al igual que en matemáticas, es una buena práctica usar paréntesis `()` para agrupar condiciones complejas y asegurar que R las evalúe en el orden que queremos.

`filter()`: Aplicando la Lógica para Seleccionar Filas

Ahora sí, la función `filter()` toma estas condiciones lógicas y las usa para seleccionar solo las filas (observaciones) de nuestra base de datos donde la condición completa es `TRUE`.

Sintaxis: `datos %>% filter(condicion_logica_1 & condicion_logica_2)`

```r
# Ejemplo 1: Filtrar para quedarse solo con personas de la Región Metropolitana (código 13)
casen_rm <- casen %>%
  filter(region == 13)

# Ejemplo 2: Filtrar por mujeres (código 2) mayores de 60 años
mujeres_mayores <- casen %>%
  filter(sexo == 2 & edad > 60)

# Ejemplo 3: Filtrar por personas de las regiones de Valparaíso (5) o Biobío (8)
# Usamos %in% por ser más compacto que usar el operador | (o)
casen_v_viii <- casen %>%
  filter(region %in% c(5, 8))
```

`arrange()`: Reordenando Filas

`arrange()` nos permite ordenar las filas de nuestra tabla de datos según los valores de una o más columnas.

Sintaxis: `datos %>% arrange(variable_de_orden)`

```
# Ordenar la base de datos desde la persona de menor edad a la de mayor edad
casen_ordenada_edad <- casen %>%
  arrange(edad)

# Para ordenar en forma descendente, usamos la función helper desc()
# Ordenar por ingreso del hogar, del más alto al más bajo
casen_ordenada_ingreso <- casen %>%
  arrange(desc(ytotcorh))
```

Podemos ordenar por múltiples variables. Por ejemplo, `arrange(region, desc(edad))` ordenaría primero por región y, dentro de cada región, por edad de forma descendente.

`mutate()`: Creando y Modificando Variables

`mutate()` es el verbo que nos permite añadir nuevas columnas o modificar las existentes. Es una de las funciones más poderosas y creativas.

Sintaxis: `datos %>% mutate(nombre_nueva_variable = operacion)`

```
# Crear una variable con la edad en décadas
casen_con_decadas <- casen %>%
  mutate(edad_decadas = edad / 10)

# Crear una variable que indique si la persona es mayor de edad (variable lógica)
casen_con_mayoria_edad <- casen %>%
  mutate(mayor_de_edad = edad >= 18)
```

En la próxima clase, veremos cómo usar `mutate()` con funciones más complejas como `case_when()` para realizar recodificaciones avanzadas.

`summarise()`: Colapsando Datos en Resúmenes

`summarise()` (o `summarize`) reduce la base de datos a una sola fila que contiene resúmenes estadísticos.

Sintaxis: `datos %>% summarise(nombre_resumen = funcion(variable))`

```r
# Calcular la edad promedio y la escolaridad máxima de toda la muestra
resumen_general <- casen %>%
  summarise(
    edad_promedio = mean(edad, na.rm = TRUE),
    maxima_escolaridad = max(esc, na.rm = TRUE)
  )

resumen_general
```

Por sí solo, `summarise()` es útil para obtener una visión global. Sin embargo, su verdadero poder se desata cuando lo combinamos con `group_by()`.

3. Análisis por Grupos: `group_by()` + `summarise()`

Obteniendo estadísticos por grupo

La combinación de `group_by()` y `summarise()` es el flujo de trabajo estándar para el análisis descriptivo en el Tidyverse. Nos permite responder preguntas del tipo: "¿Cuál es [estadístico] por [grupo]?".

La Lógica "Dividir-Aplicar-Combinar":

1. **`group_by(variable_de_agrupacion)`:** Esta función no cambia visiblemente los datos, pero añade una "meta-información" que le dice a `dplyr` que las operaciones siguientes deben realizarse por separado para cada grupo.
2. **`summarise(...)`:** Luego, `summarise` calcula el resumen estadístico para cada uno de estos grupos virtuales.
3. **Resultado:** `dplyr` devuelve una nueva tabla con los resultados para cada grupo.

Calculando Estadísticas por Grupo

Pregunta de investigación: ¿Cuál es el promedio de años de escolaridad (`esc`) por sexo?

```r
casen %>%
  group_by(sexo) %>%  # 1. Agrupamos los datos por la variable sexo
  summarise(
    escolaridad_promedio = mean(esc, na.rm = TRUE) # 2. Calculamos la media para cada grupo
  )
```

```
## # A tibble: 2 × 2
##    sexo escolaridad_promedio
##   <dbl>                <dbl>
## 1     1                 11.4
## 2     2                 11.0
```

Para que esto sea más claro, primero tendríamos que convertir `sexo` a un factor con `as_factor()`.

Análisis por Múltiples Grupos

Podemos agrupar por más de una variable para análisis más detallados.

Pregunta de investigación: ¿Cuál es el ingreso promedio del hogar (`ytotcorh`) por región y por zona (urbana/rural)?

```
casen %>%
  filter(pco1 == 1)  %>% # Filtramos solo jefe de hogar (pco1 == 1)
  group_by(region, zona) %>% # Agrupamos por dos variables
  summarise(
    ingreso_hog_promedio = mean(ytotcorh, na.rm = TRUE)
  ) %>%
  arrange(region, desc(ingreso_hog_promedio)) # Ordenamos para facilitar la lectura
```

Este tipo de tabla es el punto de partida para muchos análisis sociológicos, permitiendo comparar subgrupos de la población de manera sistemática.

Cierre y Próximos Pasos

Resumen de la sesión de hoy:

- Hemos aprendido la lógica y la sintaxis de `dplyr` para manipular datos de forma intuitiva.
- Dominamos los cinco verbos clave: `select()`, `filter()`, `arrange()`, `mutate()` y `summarise()`.
- Entendimos el poder del operador *pipe* `%>%` para escribir código secuencial y legible.
- Aplicamos la combinación `group_by()` + `summarise()` para realizar nuestros primeros análisis descriptivos por subgrupos.

En el práctico de hoy:

- Aplicarán todos estos verbos para filtrar, transformar y resumir la Encuesta CASEN 2022, respondiendo a preguntas sociológicas específicas.

Adelanto de la próxima clase:

- Profundizaremos en `mutate()` usando `case_when()` para realizar recodificaciones complejas y aprenderemos a construir nuestras propias variables, como índices.