6. Manipulación (avanzada) de Datos

Paula Vargas Pellicer

22/02/2022

Introducción

Hasta ahora, hemos ganado algo de experiencia en los conceptos básicos de la manipulación de datos y hemos empezado a entender como funcionan las canalizaciones/tuberías/pipes (%>%); es momento de aprender otras habilidades nuevas.

La idea detrás de este tutorial es aclarar introducir aspectos más avanzados de la manipulación de datos y tomarnos un momento para aprender algunas funciones nuevas... ten paciencia porque es un tutorial largo y un tanto pesado.

El paquete *dplyr* (uno de los paquetes de *tidyverse*) es tu mejor amigo para resolver varios desafíos que puedes encontrar al manipular datos.

Revisaremos las funciones de dplyr para conocer su diversidad. La idea es que te vuelvas más eficiente con las funciones que ya conoces haciendo pequeños cambios en el código, por ejemplo, *select()* y *mutate()*.

Tómate todo el tiempo necesario para repasar reste tutorial, ¡es el ultimo tutorial de manipulación de datos y las funciones de *dplyr*! También cubriremos algunas otras funciones de otros paquetes de *tidyverse* para mejorar tu confianza en las habilidades de manipulación de datos.

En la parte I, trabajaremos con un conjunto de datos ficticio de animales marinos, para que aprender a combinar tablas sea menos aterrador de lo que sería con grandes bases de datos. Luego, en la parte II, nos sumergiremos en la manipulación de variables y casos basados en la base de datos de Living Planet sobre organismos marinos en Oceanía.

Parte I: Animales del océano

Crea un nuevo script en blanco, establece el directorio de trabajo, agrega los paquetes y carga las bases de datos.

```
# Es buena practica siempre poner la ruta de directorio
setwd("ruta-de-directorio")

# Paquetes
library(dplyr)
```

```
# Aquí podrías cargar todos, sin embargo, hoy los iremos cargando poco a poco
para que veas qué funciones pertenecen a qué paquetes
```

Parte I: Animales del Océano

```
# Carga las bases de datos
animal_p1 <- read.csv("data/animal_p1.csv")

animal_p2 <- read.csv("data/animal_p2.csv")

animal_rp <- read.csv("data/animal_rp.csv")

animal_meal <- read.csv("data/animal_meal.csv")
```

Un poco de antecedentes sobre la Parte I: imagina que has recopilado estos datos con tu compañero de campo cuando visitaron un arrecife de coral en el atolón de Palmyra, una pequeña isla en el medio del Pacífico. Lograste identificar con éxito 10 animales de cuatro tipos diferentes (por el momento nos ahorraremos la taxonomía). Nuestro objetivo es combinar diferentes tablas de datos, que luego usaremos para mostrar el peso promedio y el tipo de comida para cada tipo de animal

Combinando tablas

En `dplyr` hay varias funciones para combinar tablas.

a) Combinando filas con `bind_rows`

Cada animal recibió una identificación única y se pesó. Para empezar, tu tienes tus datos en dos partes: animal_p1 en el que describiste pulpos y peces; y animal_p2 donde solo tienes tortugas.

Los conjuntos de datos están en el mismo formato, las columnas están en el orden: id, animal, peso (esto es importante), por lo que simplemente puedes colocarlos uno encima del otro con `bind_rows()`.

```
# Primero veamos p1 y p2
animal_p1

animal_p2

# Ahora las pegamos, una arriba de la otra
# Al agregar paréntesis alrededor de la expresión, los resultados se van a
imprimir en la consola
(animal <- bind_rows(animal_p1, animal_p2))  # 8 observaciones
```

Ahora quieres juntar tus datos con los de tu compañero de campo. Sus datos están en una tabla separada (`animal_rp`), por lo que tendremos que compararlos y combinarlos con `animal`. Para ello necesitamos una clave (key): una variable que sea común para los dos conjuntos de datos y, por lo tanto, se pueda usar como punto de referencia; puede ser la identificación del animal porque es única para cada uno de ellos.

b) Comparación de datos con operaciones de conjunto

Primero descubriremos en qué se diferencian las tablas de datos. Podemos ver que tanto *animal* como *animal_rp* tienen 8 observaciones (esto lo puedes ver de manera muy rápida en la ventana de Entorno), pero no tenemos idea si son iguales. Si bien la comparación visual sería posible aquí, si tuviéramos miles de filas de datos y varias columnas, sería mucho más difícil. Por lo tanto, podemos usar operaciones de conjuntos para comparar los datos con el código.

- *setqual(x, y)* devuelve VERDADERO si todas las observaciones en x e y son idénticas.
- *intersect(x, y)* encuentra observaciones presentes tanto en x como en y.
- *setdiff(x, y)* encuentra observaciones presentes en x, pero no en y.
- *union(x, y)* encuentra observaciones únicas en x e y (o usa *union_all()* para retener también duplicados).

```
setequal(animal_p1, animal_p2)

# FALSE, esto quiere decir que las tablas no son idénticas

# ¿Cuántas observaciones tienen en común?
# Ojo, Aquí no estás creando un objeto, por lo que el resultado se mostrará
en la consola pero nada será guardado

intersect(animal, animal_rp)  # 6 observaciones en común

# ¿Qué observaciones tienes tú y tu compañero de campo no tiene?



setdiff(animal, animal_rp)  # id no. 2 y 5

# ¿Cuáles tiene él que tú no tienes?



setdiff(animal_rp, animal)  # id no. 6 y 10

# Vamos a conectarlas con `union()` quitando cualquier duplicado

(animal_weight <- union(animal, animal_rp) %>%
  arrange(id))  # pone la columna id en orden numérico
```

Perfecto, ahora tenemos el conjunto de datos *animal_weight* con 10 observaciones únicas (en el rango de 1 a 10). Normalmente no necesitarías usar todas las operaciones de configuración como las anteriores, simplemente podríamos usar *union()*. Sin embargo, las otras funciones pueden ser útiles si tienes un objetivo de estudio diferente (por ejemplo, podrías usar *intersect()* para encontrar solo observaciones que hayan sido confirmadas por dos investigadores), o simplemente para conocer mejor sus datos.

c) Combinar tablas con uniones que mutan

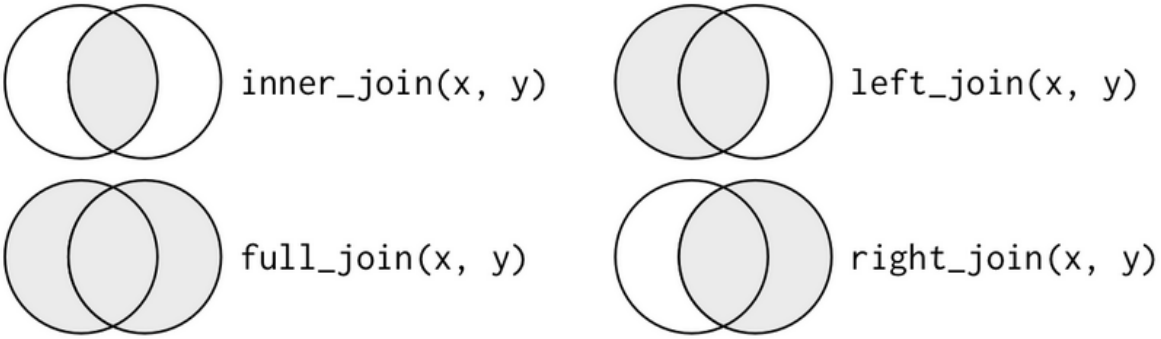
Ahora, queremos combinar *animal_weight* con otro conjunto de datos: *animal_meal*, que contiene información sobre la última comida que se observó para cada animal.

```
animal_meal  # 21 observaciones
```

Ya no tenemos el nombre del animal, pero aún tenemos la clave, la identificación única, en función de la cual podremos combinar las dos tablas agregando columnas entre sí.

Nota: si no tuviéramos ninguna clave, pero supiéramos que las filas en ambas tablas están en el mismo orden (O si el orden no importara y simplemente quisiéramos agregar comida aleatoria a cada animal), podíamos usar *bind_cols()*. En este caso, generaría varios NA ya que el número de observaciones para las dos tablas no es igual.

Hay muchas funciones diferentes en *dplyr* para combinar tablas, pero cada una es diferente y podría adaptarse mejor a diferentes necesidades. Las siguientes se utilizan para combinar variables de dos tablas de formas ligeramente diferentes.



- `inner_join(x, y)` mantiene las observaciones que aparecen en ambas tablas.
- `left_join(x, y)` mantiene todas las observaciones en x y solo agrega coincidencias de y.
- `right_join(x, y)` mantiene todas las observaciones en y y solo agrega coincidencias de x. (Nota: es lo mismo que `left_join(y, x)`.)
- `full_join(x, y)` mantiene todas las observaciones en x e y; si no hay ninguna coincidencia, devuelve NA.

Veamos cómo funcionan

```
(animal_joined <- left_join(animal_weight, animal_meal,
                            by = c("id" = "IDs")))

# hay que indicar a qué columnas en ambas talbas se refieren a `id`

# O podemos usar el operador %>%
(animal_joined <- animal_weight %>%
  left_join(animal_meal, by = c("id" = "IDs")))
```

Aquí usamos *left_join()* para mantener *animal_weight* como la base a la cual la información se añade. De esta manera solo mantenemos los 10 id's de los animales que nos interesan y les añadimos sus dietas.

¿Por qué, a tortuga con el *id==2* se le asignó *<NA>*?

```
inner_join(animal_weight, animal_meal, by = c("id" = "IDs"))

# ¿Qué hace este comando?

right_join(animal_weight, animal_meal, by = c("id" = "IDs"))

# Tenemos todos los id's para dieta, pero varios NAs para `animal` y `weight`
¿Por que?

full_join(animal_weight, animal_meal, by = c("id" = "IDs"))

# Aquí ¿qué estamos haciendo, en donde se introducen los NAs?
```

Tal vez ahora estarás notando que pudimos haber usado *full_join()* para añadir filas en el primer ejemplo: a) conectar filas con *bind_rows()*

```
full_join(animal_p1, animal_p2, by = c("id", "animal", "weight"))
```

d) Combinar filas con uniones que filtran

semi_join(x,y) encuentra todas las observaciones en x que tienen un igual en y
anti_join(x,y) quita todas las observaciones en x que tienen un igual en y

```
semi_join(animal_weight, animal_meal, by = c("id" = "IDs"))

# ¿Qué sucede aquí?

anti_join(animal_weight, animal_meal, by = c("id" = "IDs"))

# ¿y aquí?
```

Ya no vamos a ocupar estos comandos por ahora, pero es bueno conocerlos.

Ejercicio 1

A tu compañero de campo le dio tiempo de hacer mas observaciones y te entrega una nueva base de datos. Haz una tabla final que junte *animal_new* con *animal_joined*

Ejercicio 2

Haz una grafica de barras para visualizar los tipos de dietas por cada tipo de animal y otra de cajas para visualizar los pesos

Parte II : Oceanía LPI

Ahora vamos a utilizar datos de verdad: una base de datos extensa con miles de filas. vamos a echarle un ojo a las poblaciones de especies marinas de Oceanía entre 1970 y 2014

```
#Paquetes
library(tidyr)
library(readr)

# carga la base de datos LPI
marine <- read.csv("data/LPI_marine.csv")

# Vamos a organizar los datos un poco
marine2 <- marine %>%
  # cambia a formato largo con `tidyr::gather()`
  gather(key = year, value = pop, c(25:69)) %>%
  # nos deshacemos de las X's para la columna del año con `readr::parse_numbe
r()`
  mutate(year = parse_number(as.character(year)),
  # define `pop` como numérico, cualquier dato faltante se convertirá en NA
         pop = as.numeric(pop)) %>%
  # elimina todo lo que sea NA
  drop_na(pop)

# OJO!!!!!! Ignora el mensaje de advertencia (warning message on NAs), esto
aparece porque solo mantenemos valores numéricos para la columna `pop`

glimpse(marine2)  # échale un ojo a la base que creaste

View(marine2) # o vela completa
```

Hemos cambiado la base de datos a formato largo porque queremos que cada observación esté en una fila separada.

Ahora intentaremos crear la menor cantidad de objetos posibles para que no saturemos el ambiente de R, usando `dplyr()`

Manipular variables

Extracción de variables

```
marine2 %>%
  pull(Species) %>%
  glimpse()  # Que vemos aqui

marine2 %>%
  select(Species) %>%
  glimpse()  # y aqui?
```

`select()` es útil pues es muy flexible para crear nuevas tablas

Podemos crear una nueva tabla con las columnas elegidas y en el orden preferido.

```
# Selecciona solo las columnas que necesitas, en el orden que quieras
marine2 %>%
  select(id, pop, year, Country.list) %>%
  glimpse()
```

Así como también puedes darle nuevos nombres a las columnas de la nueva tabla.

```
#
marine2 %>%
  select("Country list" = Country.list,
         method = Sampling.method) %>%
  glimpse()
```

Si quieres reordenar algunas columnas y dejar el resto sin cambios, puedes usar `Everything()`

```
marine2 %>%
  select(id, year, pop, everything()) %>%
  glimpse()  # ¿Qué pasó aquí?
```

También puedes indicar el rango de columnas que deseas conservar usando `star_col:end_col` (usando nombres o números de columna).

```
marine2 %>%
  select(Family:Species, 24:26) %>%
  glimpse()
```

Elimina las columnas que no necesitas con `-` (recuerda que si quieres eliminar varias columnas, usa `select(-c())` para que `-` se aplique a todas).

```
marine2 %>%
  select(-c(2:22, 24)) %>%
  glimpse()
```

Define las columnas elegidas en un vector hecho de antemano y luego las llamas con `!!`.

```
marine_cols <- c("Genus", "Species", "year", "pop", "id")

marine2 %>%
  select(!!marine_cols) %>%
  glimpse()
```

Además, puedes usar `select()` con estas funciones: `starts_with("x")` coincide con los nombres que comienzan con "x" `ends_with("x")` coincide con los nombres que terminan en "x" `contains("x")` coincide con los nombres que contienen "x"

```
marine2 %>%
  select(starts_with("Decimal")) %>%
  glimpse()
```

También puedes seleccionar columnas según su tipo de datos utilizando *select_if()*. Los tipos de datos comunes que se llamarán son: *is.character*, *is.double*, *is.factor*, *is.integer*, *is.logical*, *is.numeric*.

```
marine2 %>%
  select_if(is.numeric) %>%
  glimpse()
```

O puedes mezclar varias formas de llamar a las columnas dentro de *select()*:

```
marine2 %>% select(id,  # pon id primero
                   Class:Family,# agrega una columna entre `Class` y `Family`
                   genus = Genus,  # renombra `Genus` a minúscula
                   starts_with("Decimal"),  # agrega columnas que empiezan
con "Decimal"
                   everything(),  # agrega todas las demás columnas
                   -c(6:9, system:Data.transformed)) %>%  # borra columnas en
estos rangos
  glimpse()
```

Probablemente no es el modo más eficiente de llegar al punto final, pero prueba el punto de la flexibilidad de todas estas funciones

Ahora que hemos aprendido todas las variedades de extracción de variables con *select()*, mantengamos las columnas que queremos usar en otras tareas (¡y ahora sí de manera eficiente!).

```
marine3 <- marine2 %>%
  select(id, Class, Genus, Species, year, pop,
         location = Location.of.population,
         lat = Decimal.Latitude,
         lon = Decimal.Longitude) %>%
  glimpse()
```

Renombrar variables

Todavía tenemos algunas letras mayúsculas en *marine3*. Prefiero tener todos los nombres de variables en minúsculas para una mayor claridad (y para reducir la posibilidad de escribirlos mal). Como no lo hicimos con *select()* cuando creamos *marine3*, ahora podemos cambiarles el nombre con *rename()* o *rename_with()*.

```
marine3 %>%
  rename(class = Class,
         genus = Genus,
         species = Species) %>%  # renombra solo las columnas seleccionadas
  glimpse()
```

Si solo queremos que los nombres de las columnas se cambien con una función, podemos usar *rename_with()*, que es útil en este caso, ya que podemos cambiarles el nombre con *tolower*.

```
marine3 %>%
  rename_with(tolower) %>%
  glimpse()

# Si no quieres renombrar todas, puedes especificarlo con `.cols =`
```

Podríamos haber logrado lo mismo de arriba seleccionando todas las columnas y aplicando *tolower*.

```
marine4 <- marine3 %>%
  select_all(tolower) %>%
  glimpse()
```

Si quisiéramos cambiar el nombre de algunas de ellas y mantenerlas, podríamos haber usado *select_at()* que especifica columnas con *vars()*.

```
marine3 %>%
  select_at(vars(Genus, Species), tolower) %>%
  glimpse()
```

Si, dentro de un mes (o mañana), no estás segur@ acerca de algún aspecto de estas funciones, por ejemplo, ¿qué es esto *vars()?* o ¿cuál es la diferencia entre *select_with()* o *rename_with()?*, siempre puedes verificarlo fácilmente en RStudio o en google.

... Pero la mayoría de las veces adivinarás la función de manera intuitiva, por ejemplo, *all()* sirve cuando vas a aplicar la función a todas las columnas; *at()* cuando vas a aplicar la función a ciertas columnas, especifícalas con *vars()*; *if()* funciona si vas a aplicar la función a columnas de una determinada característica; *with()* si vas a aplicar la función a las columnas e incluir otra función dentro de ella

Haciendo nuevas variables

La familia de funciones *mutate()* se puede utilizar para crear nuevas variables aplicando funciones vectorizadas a columnas enteras. En primer lugar, vamos a crear una variable *genus_species* conectando género y especie con un guion bajo.

```
marine5 <- marine4 %>%
  mutate(genus_species = paste(genus, species, sep = "_")) %>%
  glimpse()
```

También podemos usar *ifelse()* o *case_when()*, y los operadores lógicos. La función *mutate()* toma cada fila de la tabla y calcula el nuevo valor en la nueva columna con la función de nuestra elección.

```
marine6 <- marine5 %>%
  mutate(region = case_when(lat > 0 & lon >= 0 ~ "NE",
                            lat <= 0 & lon >= 0 ~ "SE",
```

```
                              lat > 0 & lon < 0 ~ "NW",
                              lat <= 0 & lon < 0 ~ "SW")) %>%
  glimpse()

unique(marine6$region)  # Tenemos poblaciones en ambos lados del ecuador y el meridiano 180
```

Otra función es *transmute()*, que crea las nuevas columnas y elimina el resto. Usémosla con los ejemplos de variables agregadas anteriormente (*genus_species* y *region*).

```
marine4 %>%
  transmute(genus_species = paste(genus, species, sep = "_"),
            region = case_when(lat > 0 & lon >= 0 ~ "NE",
                               lat <= 0 & lon >= 0 ~ "SE",
                               lat > 0 & lon < 0 ~ "NW",
                               lat <= 0 & lon < 0 ~ "SW")) %>%
  glimpse()
```

De manera similar a *select()*, *mutate()* también tiene variaciones *_at*, *_all* y *_if*. El mecanismo es generalmente el mismo, por lo que no veremos todos los ejemplos. Echemos un vistazo a *mutate_at()*. Indicaremos las variables con *vars()*, y luego cambiaremos todos los valores de estas variables a minúsculas. Ten en cuenta que con el ejemplo anterior de minúsculas, *select()* se ocupó de los nombres de las columnas (es decir, cambió los nombres a minúsculas) mientras que *mutate()* se ocupa específicamente de los valores de las columnas elegidas.

```
marine6 %>%
  mutate_at(vars(class, genus, location), tolower) %>%
  glimpse()
```

Aparte de *mutate()*, otra forma de crear nuevas variables puede ser con *add_*, por ejemplo, *add_column()*.

```
# `add_column()` es del paquete `tibble`
library(tibble)

marine6 %>%
  add_column(observation_num = 1:4456) %>%  # le damos a cada fila un numero de observacion
  glimpse()
```

También hay funciones de resumen: *count()* y *tally()* que tienen una variación mutante como *add_count()* y *add_tally()*. Aquí veremos cuantas observaciones únicas anuales tenemos para cada especie.

```
marine6 %>%
  select(genus_species, year) %>%
  group_by(genus_species) %>%
  add_tally(name = "observations_count") %>%
  glimpse()
```

```
marine6 %>%
  select(genus_species, year) %>%
  add_count(genus_species, name = "observations_count") %>%
  glimpse()
```

Manipulación de casos

Ahora, hablaremos sobre la manipulación de casos, eso es trabajar con filas.

Extracción de casos

En esta sección, hablaremos sobre cómo filtrar el conjunto de datos para devolver un subconjunto de todas las filas. Podemos filtrar a una categoría específica o a unas pocas categorías con `filter()`.

Operadores:

> mayor que

>= mayor o igual que

< menos que

<= menor o igual que

== exactamente igual a

!= no igual a

a|b a O b xor(a, b)

solo a O solo b a & b a Y b

is.na() solo NA

!is.na()

todos menos NA %in% en uno de los valores especificados

Ejemplos Puedes identificar que hace cada uno

```
marine6 %>%
  filter(class == "Mammalia") %>%
  glimpse()

marine6 %>%
  filter(class %in% c("Mammalia", "Aves")) %>%
  glimpse()

marine6 %>%
  filter(class == "Mammalia" | class == "Aves") %>%
  glimpse()
```

```r
marine6 %>%
  filter(class != "Actinopteri") %>%
  glimpse()

marine6 %>%
  filter(!class %in% c("Mammalia", "Aves")) %>%
  glimpse()

marine6 %>%
  filter(pop >= 10 & pop <= 100) %>%
  glimpse()

marine6 %>%
  filter(between(pop, 10, 100)) %>%
  glimpse()

marine6 %>%
  filter(!is.na(pop)) %>%
  glimpse()
```

Ahora vamos a ver la importancia de los paréntesis, fíjate como cambia el resultado

```r
marine6 %>%
  filter((class == "Mammalia" | pop > 100) & region != "SE") %>%
  glimpse()  # 38 filas

# Argumento 1: la clase es Mammalia O la población es mayor a 100
# Y
# Argumento 2: en cada caso, la región no puede ser SE


marine6 %>%
  filter(class == "Mammalia" | (pop > 100 & region != "SE")) %>%
  glimpse()  # 96 filas

# Argumento 1: clase es Mammalia
# O
# Argumento 2: la población es mayor a 100 y la región no puede ser SE
```

Nuevamente, `filter()` tiene las variaciones `_at()`, `_all()` y `_if()`, pero ya no vamos a poner más ejemplos.

Además de la familia `filter()`, también tenemos algunas otras funciones para extraer casos. `distinct()` se puede usar para eliminar todas las filas duplicadas. Advertencia: asegúrate de querer hacer eso, puedes verificar duplicados con la función base R `duplicated()`.

```r
marine6 %>%
  distinct() %>%
  glimpse()  # se mantienen 4456 filas, entonces no hay duplicados
```