5.Manipulacion_Eficiente de datos

Paula Vargas Pellicer

17/02/2022

La clase pasada aprendimos a crear subconjuntos y modificar datos para satisfacer la mayoría de nuestras necesidades para un manejo de datos ordenado. Hoy profundizaremos en el mundo de `dplyr` con una de las características más famosas: el operador de "tubería" (pipes, en inglés) `%>%`. También exploraremos algunas funciones adicionales de `dplyr` y veremos algunos consejos para recodificar y reclasificar valores.

Vamos a trabajar con un subconjunto de una base de datos grande de árboles dentro de la ciudad de Edimburgo. Esta base de datos conforma un enorme esfuerzo de campo (¡más de 50,000 árboles!) en el Área especial de paisaje alrededor del castillo de Craigmillar.

☛ Crea un nuevo script en blanco y guárdalo con tu nombre y el nombre de la clase (recuerda usar # para comentar y anotar en tu script).

1. Una introdcción a las "tuberías"

El operador de tubería `%>%` funciona como un canal para que la salida de un comando se pase a otra función sin problemas, es decir, sin crear objetos intermediarios (como en Mario Bross). Realmente hace que tu código fluya y evita la repetición.

```
# Paquetes
library(dplyr)     # Manipulación de datos

# Establece tu ruta de archivo
setwd("ruta-de-archivo")   # remplaza con el tuyo

# Carga los datos
trees <- read.csv(file = "trees.csv", header = TRUE)

head(trees)  #
```

- Asegúrate de que se haya importado bien y familiarízate con las variables

Digamos que queremos saber cuántos árboles de cada especie se encuentran en el conjunto de datos. Si hacemos los que hicimos la clase pasada, usaríamos las funciones `group_by()` y `summarise()`.

```r
# Establecer la riqueza

trees.grouped <- group_by(trees, CommonName)    # Crear grupos

trees.summary <- summarise(trees.grouped, count = length(CommonName))  # Aquí contamos el numero de filas por cada grupo. Usamos "CommonName" pero pudimos utilizar cualquiera para hacer el conteo

# O usamos una funcion que justmente cuenta las filas
trees.summary <- tally(trees.grouped)
```

Esto funciona bien, pero observa cómo tuvimos que crear un marco de datos adicional, "`trees.grouped`", antes de lograr el resultado deseado de "`trees.summary`". Para un análisis más grande y complejo, ¡esto saturaría rápidamente tu entorno con muchos objetos que realmente no necesitas!

Aquí es donde entra la tubería. Toma la base de datos creada del lado izquierdo y la pasa a la función del lado derecho.

Las funciones de `tidyverse` "saben" que el objeto que se pasa a través de la tubería es el argumento `data =` de esa función.

```r
trees.summary <- trees %>%                # la base de datos que vamos a pasar
                 group_by(CommonName) %>%   # la variable que vamos a agrupar
                 tally()                    # Cuenta las filas
```

Notas importantes: las canalizaciones por la tubería solo funcionan en objetos de base de datos, y las funciones fuera del `tidyverse` a menudo requieren que especifique la fuente de datos con un punto final.

(Si te gustan los atajos, puedes usar Ctrl + Shift + M para crear el operador %>%)

```r
trees.subset <- trees %>%
                filter(CommonName %in% c('Common Ash', 'Rowan', 'Scots Pine')
) %>%
                group_by(CommonName, AgeGroup) %>%
                tally()
```

¿Que hicimos aquí?

2. Más funciones de dplyr

Una extensión de las funciones principales de `dplyr` es `summarise_all()`:ejecutará una función de resumen de tu elección sobre TODAS las columnas. No es significativo aquí, pero podría serlo si todos los valores fueran numéricos, por ejemplo.

2a. summarise_all() - genera rápidamente un marco de datos de resumen

```
summ.all <- summarise_all(trees, mean)
```

OJO: En el resultado, solo dos de las columnas tienen valores numéricos. ¿Por qué crees que sea esto?

2b. case_when() - reclasificar valores o factores

Pero primero, hay que introducir la función más simple sobre la que se basa, `ifelse()`. Esta es una declaración condicional que evaluará los valores que debe devolver cuando la declaración sea verdadera o falsa. Vamos a hacer un ejemplo muy simple para empezar:

```
vector <- c(4, 13, 15, 6)      # crea un vector

ifelse(vector < 10, "A", "B")  # da las condiciones: si x es inferior a 10,
entonces escribe A, de otra forma, escribe B
```

`case_when()` es una generalización de `ifelse()` que te permite asignar más de dos resultados. Todos los operadores lógicos están disponibles y el nuevo valor se asigna con una tilde ~. Por ejemplo, (los Beatles):

```
vector2 <- c("What am I?", "A", "B", "C", "D")

case_when(vector2 == "What am I?" ~ "I am the walrus",
          vector2 %in% c("A", "B") ~ "goo",
          vector2 == "C" ~ "ga",
          vector2 == "D" ~ "joob")
```

3. Cambiar los niveles de los factores o crear variables categóricas

El uso de `mutate()` junto con `case_when()` es una excelente manera de cambiar los nombres de los niveles de los factores o crear una nueva variable basada en las existentes. Vemos en las columnas de `LatinName` que hay muchas especies de árboles pertenecientes a algunos géneros, como abedules (*Betula*) o sauces (*Salix*), por ejemplo. Es posible que deseemos crear una columna llamada `Genus` usando `mutate()`.

Haremos esto mediante una búsqueda de caracteres con la función `grepl()`, que busca patrones en los datos y especifica qué devolver para ese patrón. Antes de hacer eso, hay que hacer una lista completa de especies que se encuentran en los datos

```
unique(trees$LatinName)  # Muestra el nombre de todas las especies

# Crear una nueva columna con el nombre del género
trees.genus <- trees %>%
               mutate(Genus = case_when(
                  grepl("Acer", LatinName) ~ "Acer",
```

```
                  grepl("Fraxinus", LatinName) ~ "Fraxinus",
                  grepl("Sorbus", LatinName) ~ "Sorbus",
                  grepl("Betula", LatinName) ~ "Betula",
                  grepl("Populus", LatinName) ~ "Populus",
                  grepl("Laburnum", LatinName) ~ "Laburnum",
                  grepl("Aesculus", LatinName) ~ "Aesculus",
                  grepl("Fagus", LatinName) ~ "Fagus",
                  grepl("Prunus", LatinName) ~ "Prunus",
                  grepl("Pinus", LatinName) ~ "Pinus",
                  grepl("Sambucus", LatinName) ~ "Sambucus",
                  grepl("Crataegus", LatinName) ~ "Crataegus",
                  grepl("Ilex", LatinName) ~ "Ilex",
                  grepl("Quercus", LatinName) ~ "Quercus",
                  grepl("Larix", LatinName) ~ "Larix",
                  grepl("Salix", LatinName) ~ "Salix",
                  grepl("Alnus", LatinName) ~ "Alnus")
                )
```

Obviamente esto tampoco es muy eficiente, por lo que hay una manera aun más fácil de lograr esto: Si te fijas, el género es siempre la primera palabra de la columna `LatinName` y siempre está separado de la siguiente palabra por un espacio. Podríamos usar la función de `separate()` del paquete `tidyr` para dividir la columna en varias columnas nuevas llenándolas con las palabras que componen los nombres de las especies, y mantener solo la primera.

```
library(tidyr)


trees.genus.2 <- trees %>%
                  tidyr::separate(LatinName, c("Genus", "Species"), sep = " "
, remove = FALSE) %>%
                  dplyr::select(-Species)
```

Ahora otro ejemplo de cómo podemos reclasificar un factor. El factor Altura (`Height`) tiene 5 niveles que representan rangos de alturas, pero digamos que queremos en realiad tres categorías para nuestros propósitos de análisis. Entonces creamos una nueva variable de categoría de altura `Height.cat`:

```
trees.genus <- trees.genus %>%   #
                mutate(Height.cat =   # nueva columna
                           case_when(Height %in% c("Up to 5 meters", "5 to 10 m
eters") ~ "Short",
                                     Height %in% c("10 to 15 meters", "15 to 20
meters") ~ "Medium",
                                     Height == "20 to 25 meters" ~ "Tall")
                       )
```

Reordenar niveles de factores

Hemos visto cómo podemos cambiar los nombres de los niveles de un factor, pero ¿qué sucede si deseamos cambiar el orden en que se muestran? R siempre los mostrará en orden alfabético, lo cual no siempre es muy útil.

Por ejemplo, si graficamos el número de árboles en cada una de nuestras nuevas categorías de altura, es posible que queramos que las barras digan, de izquierda a derecha: 'Bajo', 'Medio', 'Alto'. Sin embargo, por defecto, R los ordenará 'Medianos', 'Cortos', 'Altos' (Por el orden alfabético en ingles).

Para solucionar esto, puedes especificar el orden explícitamente e incluso agregar etiquetas si deseas cambiar los nombres de los niveles de los factores. Aquí, los ponemos en mayúsculas para ilustrar.

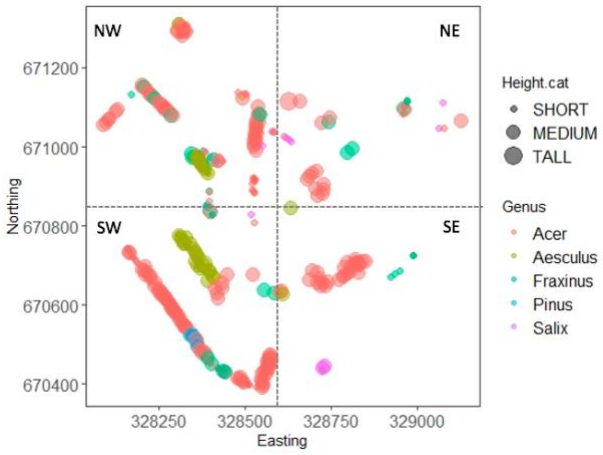
```
levels(trees.genus$Height.cat)  # Muestra los niveles de factor en el orden predeterminado

trees.genus$Height.cat <- factor(trees.genus$Height.cat,
                                 levels = c('Short', 'Medium', 'Tall'),
                                 labels = c('SHORT', 'MEDIUM', 'TALL')
                                 )

levels(trees.genus$Height.cat)  # el nuevo orden
```

Ejercicio

Haz un resumen de las diferentes especies que se encuentran dentro del castillo de Craigmillar, pero dividido en cuatro cuadrantes (NE, NW, SE, SW). (Puedes comenzar usando el objeto `trees.genus` creado anteriormente).



Pista: Para crear los cuadrantes necesitas hacer un poco de matemáticas; encuentra las coordenadas del centro que dividirán los datos (agregando la mitad del rango en longitud y latitud al valor más pequeño)

☛ Calcula la riqueza de especies (el número de especies diferentes) en cada cuadrante.

☛ Calcula la abundancia del género *Acer* (% del número total de árboles) en cada cuadrante.