Master en Big Data. Fundamentos Matemáticos del Análisis de Datos (FMAD).

Práctica 1

Fernando San Segundo

Curso 2021-22. Última actualización: 2021-08-30

COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI
DEPARTAMENTO DE MATEMÁTICA APLICADA

Preliminares

- En esta y en las próximas prácticas vamos a empezar cargando algunas librerías y conjuntos de datos que luego necesitaremos en los ejemplos.

```
library(tidyverse)

fhs = read_csv("data/framingham.csv")
```

Verbos de dplyr

- Esta sección pretende ser una invitación a la lectura del Capítulo 5 de (Wickham and Grolemund 2016) y desde luego no aspira a sustituir esa lectura.
- Aunque ya hemos visto algunos ejemplos de `dplyr` en acción, vamos a recopilar aquí de forma más sistemática, los elementos básicos de la transformación de datos con esa librería. En esencia, la mayoría de las operaciones se organizan en torno a una familia de verbos. Los principales son:
  - `select`
  - `filter`
  - `mutate`
  - `arrange`
  - `summarize`
  - `group_by` En las próximas páginas vamos a ver ejemplos de uso de estos verbos. Los tres primeros han aparecido ya, así que nos detendremos un poco más en los nuevos.

`select` para elegir columnas

- En esta y en las siguientes páginas vamos a usar la tabla 'gapminder, así que empezamos cargándola. Además vamos a ver los nombres de las variables que la componen:

```
library(gapminder)
names(gapminder)
```

```
[1] "country"   "continent" "year"      "lifeExp"
[5] "pop"       "gdpPercap"
```

- Ahora vamos a usar `select` para elegir las columnas de `lifeExp` y `gdpPercap`.

```
gapminder %>%
  select(lifeExp, gdpPercap) %>%
  head(3)
```

```
# A tibble: 3 x 2
  lifeExp gdpPercap
    <dbl>     <dbl>
1    28.8      779.
2    30.3      821.
3    32.0      853.
```

Fíjate en que hemos usado `head` para ver los primeros elementos de la tabla.

Otras posibilidades de `select`

- De la misma forma que `12:20` representa un conjunto consecutivo de números podemos usar `:` para seleccionar un conjunto consecutivo de columnas *por sus nombres*. Y si usamos `-` estaremos excluyendo una columna:

```
gapminder %>%
  select(continent:pop, -year) %>%
  names()
```

```
[1] "continent" "lifeExp"   "pop"
```

Asegúrate de que entiendes por qué se incluyen específicamente esas columnas.

- Además podemos usar una serie de funciones auxiliares que permiten elegir las columnas cuyos nombres cumplan cierto patrón. Esas funciones incluyen: `contain`, `starts_with`, `ends_with`, `matches`, `one_of` (pero hay más). Por ejemplo:

```
gapminder %>%
  select(starts_with("c")) %>%
    names()
```

```
[1] "country"   "continent"
```

Este tipo de funciones auxiliares son muy útiles cuando estemos *limpiando conjuntos sucios* de datos antes del análisis.

`filter` para elegir filas.

- La función `filter` realiza selección por filas en una tabla. Por ejemplo, para ver las observaciones correspondientes a España:

```
gapminder %>%
  filter(country == 'Spain') %>%
  head(4)
```

```
# A tibble: 4 x 6
  country continent  year lifeExp      pop gdpPercap
  <fct>   <fct>     <int>   <dbl>    <int>     <dbl>
1 Spain   Europe     1952    64.9 28549870     3834.
2 Spain   Europe     1957    66.7 29841614     4565.
3 Spain   Europe     1962    69.7 31158061     5694.
4 Spain   Europe     1967    71.4 32850275     7994.
```

- Además de `filter` existen otras funciones que permiten seleccionar por filas. Por ejemplo aquí usamos `top_n` (mira la chuleta de dplyr para ver más posibilidades):

```
gapminder %>%
filter(year == "1997") %>%
top_n(3, gdpPercap)
```

```
# A tibble: 3 x 6
  country       continent  year lifeExp       pop gdpPercap
  <fct>         <fct>     <int>   <dbl>     <int>     <dbl>
1 Kuwait        Asia       1997    76.2   1765345    40301.
2 Norway        Europe     1997    78.3   4405672    41283.
3 United States Americas   1997    76.8 272911760    35767.
```

mutate para crear nuevas variables.

- Usemos mutate para añadir una columna que calcule el gdp (en millones de dolares) multiplicando pop por gdpPercap. Aprovechamos para usar sample_n, una función emparentada con filter:

```
gapminder %>%
  mutate(gdp = pop * gdpPercap / 10^6) %>%
  filter(year == 1982) %>%
  sample_n(4)
```

```
# A tibble: 4 x 7
  country    continent  year lifeExp    pop gdpPercap    gdp
  <fct>      <fct>     <int>   <dbl>  <int>     <dbl>  <dbl>
1 Nicaragua  Americas   1982    59.3 2.98e6     3470. 1.03e4
2 Guatemala  Americas   1982    58.1 6.40e6     4820. 3.08e4
3 Turkey     Europe     1982    61.0 4.73e7     4241. 2.01e5
4 West Bank~ Asia       1982    64.4 1.43e6     4336. 6.18e3
```

- Hay otras funciones relacionadas con mutate, como add_column, rename, etc.
- Si quieres aplicar una función a todos los elementos de una columna puedes usar mutate_at. Por ejemplo, para calcular el logaritmo en base 10 del gdp, ejecuta:

```
gapminder %>%
  mutate(gdp = pop * gdpPercap / 10^6) %>%
  mutate_at("gdp", log10) %>%
  head(4)
```

summarize y group_by para describir los datos

- Vamos a ver como usar summarize para explorar nuestros datos. En un primer ejemplo sencillo vamos a calcular la longitud media de los pétalos en la tabla iris:

```
    iris %>%
      summarise(mediana = median(Petal.Length), desvMediana = mad(Petal.Length))
```

```
  mediana desvMediana
1    4.35     1.85325
```

  Por cierto ¿como harías esto con R básico? Busca información sobre la función aggregate y sobre la familia de funciones apply de R (por ejemplo en el Capítulo 21 de (Wickham and Grolemund 2016), o las Secciones 3.3 y 4.4 de (Matloff 2011).)

- Eso está bien, pero sabemos que iris contiene datos de tres especies y lo natural es preguntar si hay diferencias *significativas* (volveremos pronto sobre esa palabra) entre las longitudes de los pétalos de cada una de esas especies. Así que queremos calcular las medias por especie, que son *medias agrupadas*. Ahí es donde interviene group_by:

```
    iris %>%
      group_by(Species) %>%
      summarise(mediana = median(Petal.Length), desvMediana = mad(Petal.Length))
```

```
# A tibble: 3 x 3
  Species    mediana desvMediana
  <fct>        <dbl>       <dbl>
1 setosa         1.5       0.148
2 versicolor     4.35      0.519
3 virginica      5.55      0.667
```

Grupos con más de un factor.

- En el ejemplo anterior hemos agrupado las observaciones de la tabla `iris` usando únicamente el factor `Species`. Pero no es necesario limitarse a un único factor. Por ejemplo en la tabla `mpg`

```
    mpg %>%
      group_by(manufacturer, cyl) %>%
      summarise(urbano = mean(cty), n = n()) %>%
  head(6)
```

```
`summarise()` has grouped output by 'manufacturer'. You can override using the `.groups` argument.
```

```
# A tibble: 6 x 4
# Groups:   manufacturer [2]
  manufacturer   cyl urbano     n
  <chr>        <int>  <dbl> <int>
1 audi             4   19.1     8
2 audi             6   16.4     9
3 audi             8   16       1
4 chevrolet        4   20.5     2
5 chevrolet        6   17.7     3
6 chevrolet        8   13.6    14
```

- **Ejercicio:** ¿qué cambia si usas el orden inverso `group_by(manufacturer, cyl)` en el anterior código?
- **Ejercicio:** piensa qué hace la función `n()` en este código ()

Funciones que podemos usar con `summarize`

- Para que podamos usar una función dentro de `summarize` tiene que ser una función vectorial (que actúa sobre una columna de la tabla, vista como vector) cuyo resultado sea un valor simple (como un número o un booleano). Te recomendamos consultar la discusión de la Sección 5.6.4 de (Wickham and Grolemund 2016).
- Una de las funciones más útiles de ese tipo es la función `count`. Fíjate en el resultado de este código y compáralo con el anterior.

```
mpg %>%
  group_by(manufacturer) %>%
  count(cyl) %>%
  head(8)
```

```
# A tibble: 8 x 3
# Groups:   manufacturer [3]
  manufacturer   cyl     n
  <chr>        <int> <int>
1 audi             4     8
2 audi             6     9
3 audi             8     1
4 chevrolet        4     2
5 chevrolet        6     3
6 chevrolet        8    14
7 dodge            4     1
8 dodge            6    15
```

Observa en particular que no hemos necesitado agrupar por `cyl` explícitamente.

- La Sección 5.7.1. de (Wickham and Grolemund 2016) describe otras operaciones interesantes que podemos hacer usando `group_by`.

Listas.

- Como referencias para este apartado puedes usar (Boehmke 2016, capítulo 11), (Matloff 2011, capítulo 4).
- A diferencia de los vectores, las listas sirven para guardar elementos heterogéneos (incluidas sublistas). La forma más sencilla de crear una lista es usando `list`:

```
(planeta = list(nombre = "Marte", exterior = TRUE,
                radio = 3389.5, satelites = list("Fobos", "Deimos")))
```

```
$nombre
[1] "Marte"

$exterior
[1] TRUE

$radio
[1] 3389.5

$satelites
$satelites[[1]]
[1] "Fobos"

$satelites[[2]]
[1] "Deimos"
```

Accediendo a los elementos de una lista.

- R usa $ o doble corchete [[ ]] para identificar los elementos de la lista.

```
planeta[[1]]
```

```
[1] "Marte"
```

```
planeta$exterior
```

```
[1] TRUE
```

```
planeta$satelites[[1]]
```

```
[1] "Fobos"
```

  La salida es del tipo de objeto que hay en esa posición de la lista.
- Pero fíjate en la diferencia si usamos un único corchete:

```
planeta[1]
```

```
$nombre
[1] "Marte"
```

```
planeta["exterior"]
```

```
$exterior
[1] TRUE
```

  En este caso la salida *siempre es una lista*.

Funciones `list`, `append` y `c`.

- Atención a esta diferencia:

```
(l1 = list("A", "B"))
```

```
[[1]]
[1] "A"

[[2]]
[1] "B"
```

```
(l2 = list(c("A", "B")))
```

```
[[1]]
[1] "A" "B"
```

La función `list` siempre crea *listas anidadas*. Por ejemplo este comando (no se muestra la salida) crea una lista con dos componentes y el primero es `l2`:

```
(l3 = list(l2, "C"))
```

- Las funciones `append` y `c` *adjuntan* elementos. Estos comandos son equivalentes:

```
l4 = append(l2, "D")
(l4 = c(l2, "D"))
```

```
[[1]]
[1] "A" "B"

[[2]]
[1] "D"
```

También se pueden añadir elementos por nombre, como en

```
planeta$distSol = 227.9
```

Otras propiedades y operaciones con listas.

- La función `length` produce el número de elementos de una lista. Y con `names` se obtienen los nombres de sus elementos (si se han dado nombres).
- **Ejercicio:** Prueba a usar `names` y `length` con varias de las listas que hemos creado. Ejecuta `(sesion = sessionInfo())` para ver lo que hace esa función. Y luego explora como acceder a las componentes usandosesion$‘
- Para eliminar elementos de una lista basta con hacerlos `NULL`.

```
l4[3] = NULL
l4
```

```
[[1]]
[1] "A" "B"

[[2]]
[1] "D"
```

- La función `unlist` *aplana* una lista dando como resultado un vector:

```
unlist(l1)
```

```
[1] "A" "B"
```

- **Ejercicio:** ¿qué se obtiene al aplicar `unlist` a la siguiente `lista`?
  `lista = list(letters[1:3], matrix(1:12, nrow = 3), TRUE)`.

Estructuras de control en R. Bloques if/else.

- Como referencias para este apartado puedes usar (Boehmke 2016, capítulo 19), (Matloff 2011, capítulo 7).
- **Bloques if/else.** La estructura básica de estos bloques es:

```
if (condición) {
  ...
  sentencias que se ejecutan si condicion = TRUE
  ...
}  else {
  ...
  sentencias que se ejecutan si condicion = FALSE
  ...
}
```

  Si necesitas condiciones anidadas puedes cambiar `else` por `else if` y añadir a continuación otra condición para crear un nuevo nivel de la estructura.
- La estructura `if` está pensada para ejecutarse sobre una *única* condición que produzca un *único* valor `TRUE/FALSE`. Existe también una función vectorializada, llamada `ifelse` que se puede aplicar a un vector de condiciones. Un ejemplo:

```
ifelse(((1:5) < 3), yes = "A",  no = "B")
```

```
[1] "A" "A" "B" "B" "B"
```

Bucles for.

- El bucle for se utiliza cuando queremos repetir un bloque de comando y conocemos de antemano el número máximo de repeticiones. Su estructura básica es similar a esta:

```
for(k in valores_k) {
  ...
  cuerpo del bucle, se repite a lo sumo length(valores_k) veces
  ...
}
```

  La variable `k` (el nombre es arbitrario) es el *contador* del bucle for. El vector `valores_k` contiene los valores que toma `k` en cada iteración.

- Si en alguna iteración queremos interrumpir el bucle cuando se cumple alguna condición (y no hacer ninguna iteración más), podemos combinar `if` con la función `break`. Si lo que queremos es solamente pasar a la siguiente iteración usamos `next` en lugar de `break`.
- A menudo se usa un bucle for para *"rellenar"* un objeto como un vector o matriz. Es importante recordar que R es poco eficiente haciendo *"crecer"* estos objetos. En esos casos es mucho mejor comenzar creando el objeto completo, con todas sus posiciones, e ir asignado valores a posiciones en cada iteración (R lo inicializa a 0).

Ejemplo de bucle for con `next` y `break`.

- El siguiente código ilustra un bucle for con el uso de next y break. Ejecútalo varias veces para ver como se comporta según los valores de sus parámetros.

```
valores = numeric(10) # Creamos un vector del tamaño previsto
for (k in 1:10){
  sorteo = sample(1:20, 1)
  print(paste0("k = ", k, ", sorteo = ", sorteo))
  if (k %in% 5:6){
    next # saltamos dos valores
  } else if (sorteo  == 1){
    print("Resultado del sorteo es 1, fin del bucle")
    break # paramos si un valor aleatorio es 1
  }
  valores[k] = sorteo # se ejecuta cuando no se cumplan las condiciones
}
valores
```

- **Ejercicio:** ¿Qué valores asigna R a los elementos de los vectores creados respectivamente con `x = logical(10)` y con `v = character(10)`?

Otros bucles: while y repeat.

- En R también existen estos dos tipos de bucles, comunes a muchos lenguajes. Conviene insistir en que suele ser más eficiente evitar el uso de bucles.
- El bucle `while` tiene esta estructura:

```
while (condición){
    ...
    cuerpo del bucle: eventualmente debe hacer condición TRUE o usar break
    ...
}
```

- El bucle `repeat` tiene esta estructura:

```
repeat {
    ...
    cuerpo del bucle, que debe usar break
    ...
}
```

  Insistimos: a diferencia de otros lenguajes, en R un bucle `repeat` debe usar explícitamente `break` para detenerse.

- El código de este tema contiene ejemplos de bucle `while` y `repeat` con `break`, que puedes ejecutar varias veces. Observa las diferencias en el comportamiento de ambos bucles. Busca también información sobre el uso de `next` en los bucles de R.

Funciones de R.

- Aunque R básico y todas las librerías disponibles nos ofrecen miles de funciones para las más diversas tareas, pronto llegará el día en que necesitarás escribir una función para resolver un problema específico.
- Para escribir una función de R podemos usar este esquema básico

```
nombreFuncion = function(argumento1, argumento2, ...){
  ...
  ...

  líneas de código del cuerpo de la función
  ...
  ...
}
```

Como se ve la función tiene un *nombre*, una lista de *argumentos* y un *cuerpo* que contiene las líneas de código R que se ejecutarán al llamar a la función.

Ejemplo

- Crearemos una función `genPasswd` que genere contraseñas aleatorias. Los argumentos serán la longitud de la contraseña `size` y 3 booleanos `upp`, `low` y `nmb` que sirven para incluir o no respectivamente mayúsculas, minúsculas y números. Todos ellos menos `size` tienen valores por defecto.

```
genPasswd = function(size, upp = TRUE, low = TRUE, nmb = TRUE){

  # El vector pool guarda el juego de caracteres del password
  pool = character()

  # Generamos pool según las opciones
  if(upp) pool = c(pool, LETTERS)
  if(low) pool = c(pool, letters)
  if(nmb) pool = c(pool, 0:9)

  # Sorteamos los símbolos que aparecen en el password
  passwd = sample(pool, size, replace = TRUE)
  # Y lo reducimos a un string con paste
  paste(passwd, sep = "", collapse = "")
}
```

La función se ejecuta como cualquier otra función de R (*pero cuidado:* si tratas de ejecutarla sin darle un valor a `size` habrá un error.):

```
genPasswd(size = 15)
```

```
[1] "zoqHXuYiu4ayyo4"
```

- **Ejercicio:** lee la ayuda de la función `paste` (y después la de `paste0`). Es una función extremadamente útil para trabajar con texto.

Acceso a las componentes de una función.

- La función `formals` produce como resultado una lista con los argumentos de cualquier función. Prueba a ejecutar:

```
formals(genPasswd)
```

- La función `body` permite acceder (¡y modificar!) el cuerpo de la función:

```
body(genPasswd)
```

```
{
    pool = character()
    if (upp)
        pool = c(pool, LETTERS)
    if (low)
        pool = c(pool, letters)
    if (nmb)
        pool = c(pool, 0:9)
    passwd = sample(pool, size, replace = TRUE)
    paste(passwd, sep = "", collapse = "")
}
```

Observa el resultado si ahora haces

```
body(genPasswd) = "No me apetece trabajar...invéntate tú el password"
genPasswd(12)
```

Puedes leer más sobre funciones en el Capítulo 18 de (Boehmke 2016).

Manejo de datos ausentes. La función `is.na`

- Hasta ahora hemos tocado sólo tangencialmente el tema de los datos ausentes, pero es sin duda uno de los quebraderos de cabeza más habituales que te encontrarás al trabajar con un nuevo conjunto de datos.
- En R los datos ausentes se representan con el símbolo `NA`. Y disponemos de varias funciones para detectarlos. La más básica es `is.na`. Por ejemplo:

```
x = c(2, 3, -5, NA, 4, 6, NA)
is.na(x)
```

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

Esta función es muy útil cuando se combina con otras como `which` que ya conoces o como `all` y `any`. Estas dos últimas actúan sobre un vector de booleanos y valen `TRUE` si todos o alguno, respectivamente, de los valores del vector son `TRUE`.

- Por ejemplo, podemos saber si `fhs$glucose` tiene algún valor ausente con

```
any(is.na(fhs$glucose))
```

```
[1] TRUE
```

Más sobre datos ausentes: `complete.cases` y `na.rm`

- Una función relacionada es `complete.cases`, Aplicada a una tabla (`data.frame`) nos dirá para cada fila si esa fila tiene o no datos ausentes.

```
head(complete.cases(fhs), 17)
```

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

El primer `FALSE` corresponde a la fila 15 de `fhs` que tiene un valor ausente en la columna `glucose`, como ya sabemos.

- La presencia de datos ausentes puede hacer que muchas funciones produzcan `NA` como resultado (o peor, que no funcionen correctamente). Por ejemplo, una media aritmética:

```
mean(fhs$glucose)
```

```
[1] NA
```

Muchas funciones de R disponen de un argumento `na.rm` para excluir los valores `NA` de la operación que se realice:

```
mean(fhs$glucose, na.rm = TRUE)
```

```
[1] 81.96366
```

Puedes encontrar más información en la Sección 7.4 de R for Data Science, la Sección 5.12 de (Peng 2015) y el Capítulo 14 de (Boehmke 2016).

Referencias

Boehmke, B. C. (2016). *Data Wrangling with R* (p. 508). Springer. https://doi.org/10.1007/978-3-319-45599-0

Matloff, N. S. (2011). *The art of R programming : tour of statistical software design* (p. 373). No Starch Press. https://doi.org/10.1080/09332480.2012.685374

Peng, R. D. (2015). *R Programming for Data Science* (p. 132). Leanpub. https://doi.org/10.1073/pnas.0703993104

Wickham, H., & Grolemund, G. (2016). *R for data science: import, tidy, transform, visualize, and model data*. O'Reilly Media, Inc.