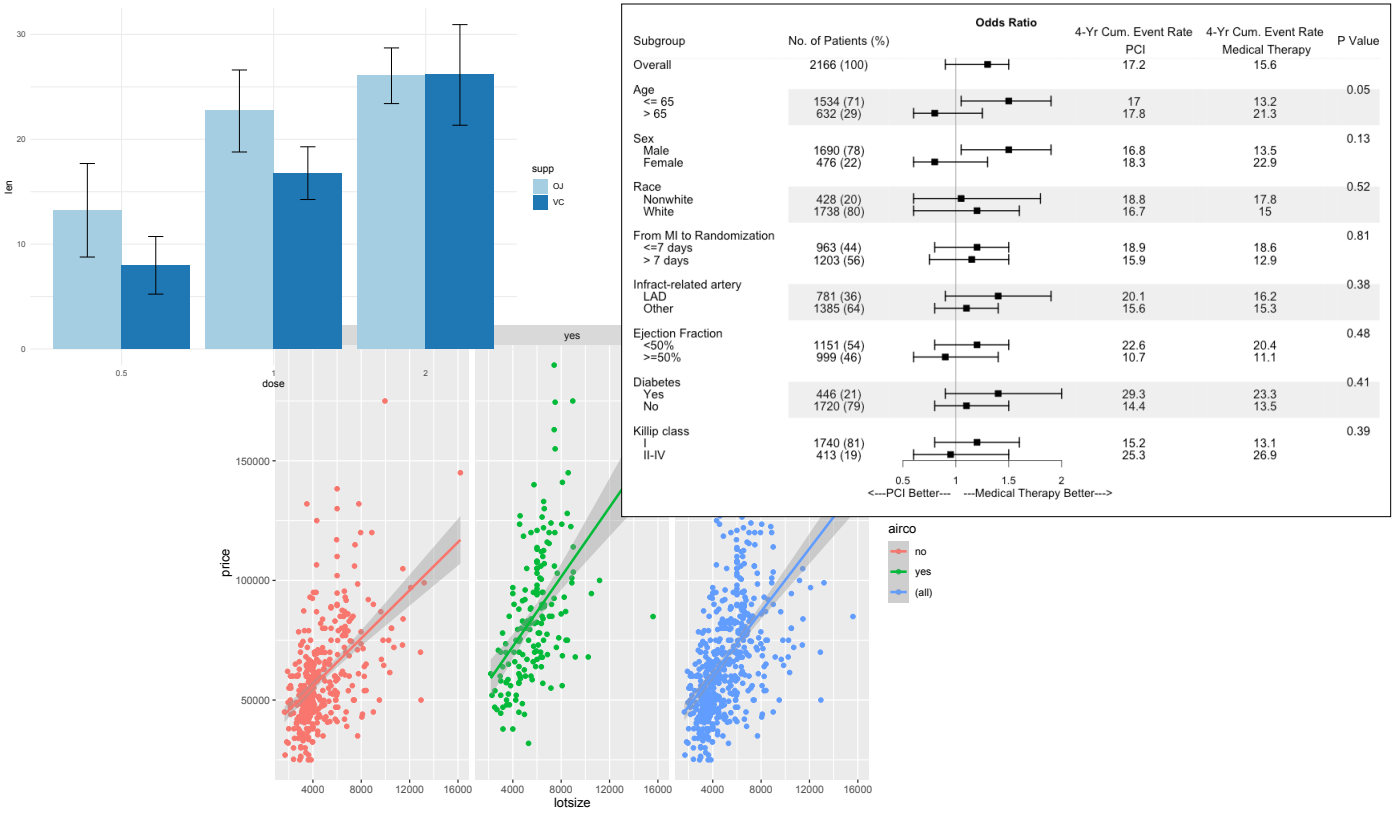
Instituto de Salud Carlos III

Manejo avanzado de datos con R



Instituto de Salud Carlos III

Manejo avanzado de datos

Funciones

Instituto de Salud Carlos III

Índice

- Definición de funciones
- Argumentos
- Control de ejecución: condicionales, loops
- Cuando algo va mal: traceback, browser, debug
- Eficiencia: unix.time, Rprof
- Cosas variadas

Definición de funciones

- Ya hemos definido varias funciones. Aquí una más:

```
my.f2 <- function(x, y) {
        z <- rnorm(10)
        y2 <- z * y
        y3 <- z * y * x
        return(y3 + 25)
        }
```

- Lo que una función devuelve puede ser:
  - un simple número
  - vector
  - Una gráfica
  - Una lista
  - Un mensaje
  - … o todo.

Argumentos (I)

```
otra.f <- function (a, b, c = 4, d = FALSE) {
        x1 <- a * z ...}
```

- Los argumentos "a" y "b" tienen que darse en el orden debido o, si los nombramos, podemos darlos en cualquier orden:

```
otra.f(4, 5)
otra.f(b = 5, a = 4)
```

- Pero los argumentos con nombre siempre se tienen que dar después de los posicionales

```
otra.f(c = 25, 4, 5) # error
```

- Los argumentos **"c"** y **"d"** tienen "**default values**". Podemos especificarlos nosotros, o no especificarlos (i.e., usar los valores por defecto).

- **args**(nombre.funcion) nos muestra los argumentos (de cualquier función).

Argumentos (II)

- **"z"** es una "**free variable**": ¿cómo se especifica su valor?

- **Lexical scoping**. Ver documento *Frames, environments,and scope in R and S-PLUS* de J. Fox (en http://cran.r-project.org/doc/contrib/Fox-Companion/appendix.html) y sección 10.7 en *An introduction to R*. También **demo**(scoping).

- **". . . "** permite pasar argumentos a otra función:

```
> f3 <- function(x, y, label = "la x", ...) {
plot(x, y, xlab = label, ...)
}

> f3(1:5, 1:5)
> f3(1:5, 1:5, col = "red")
```

Control de ejecución if (I)

- **if** (*condicion.logica*) *instruccion* donde "*instruccion*" es cualquier expresión válida (incluida una entre { }).

- **if** (*condicion.logica*) *instruccion* **else** *instruccion.alternativa*.

```
> f4 <- function(x) {
        if(x > 5) print("x > 5")
        else {
        y <- runif(1)
        print(paste("y is ", y))
        }
        }
```

Control de ejecución if (II)

- **ifelse** es una versión vectorizada:

```
> odd.even <- function(x) {
        ifelse(x %% 2 == 1, "Odd", "Even")
        }

> mtf <- matrix(c(TRUE, FALSE, TRUE, TRUE),
nrow = 2)

>ifelse(mtf, 0, 1)
```

Control de ejecución: while, for

- **while** (*condicion.logica*) *instrucción*
- **for** (*variable.loop* in *valores*) *instrucción*

```
for(i in 1:10) {
        cat("el valor de i es", i, "nn")
        continue.loop <- TRUE
        x <- 0
        while(continue.loop) {
                x <- x + 1
                print(x)
                if( x > 10) {continue.loop <- FALSE}
        }
}
```

- **repeat**, **switch** también están disponibles.
- **break**, para salir de un loop.

familia apply

Cada vez que vayamos a usar un **"loop"** explícito, intentemos substituirlo por algún miembro de *la ilustre familia apply*

- apply
- tapply
- sapply
- lapply (no la vemos)

Instituto de Salud Carlos III

apply

```
> apply(datos[,c(6,7)],2,function(x) x/10)
 peso   altura
[1,] 0.5958221 1.507163
[2,] 0.5995427 1.492075
[3,] 0.7920674 1.689795
```

tapply

```
> tapply(datos$edad,datos$sexo, mean)
Hombre  Mujer
39.64  44.70
```

sapply

```
> sapply(datos[,c(6)], function(x) x/10)
[1] 0.5958221 0.5995427 0.7920674 0.8078347 0.8076036...
```

Cuando algo va mal (I)

- Cuando se produce un **error**, **traceback**() nos informa de la secuencia de llamadas antes del crash de nuestra función (útil cuando se produce mensajes de error incomprensibles).

- Cuando se producen **errores** o la función da **resultados incorrectos o warnings indebidos** podemos seguir la ejecución de la función.

- **browser** interrumpe la ejecución a partir de ese punto y permite seguir la ejecución o examinar el entorno; con "n" paso a paso, si otra tecla sigue ejecución normal. "Q" para salir.

- **debug** es como poner un "broswer" al principio de la función, y se ejecuta la función paso a paso. Se sale con "Q".

```
> debug(my.buggy.function)

> undebug(my.buggy.function)
```

Cuando algo va mal (II)

- **Ejemplo:**

```
my.f2 <- function(x, y) {
z <- rnorm(10)
y2 <- z * y
y3 <- z * y * x
return(y3 + 25)
}

my.f2(runif(3), 1:4)
debug(my.f2)
my.f2(runif(3), 1:4)
undebug(my.f2)
# insertar un browser() y correr de nuevo
```

Eficiencia

- Nuestro objetivo aquí no es producir las funciones más eficientes, sino funciones que hagan lo que deben.
- Pero a veces útil saber cuanto dura la ejecución de una función: **unix.time(my.f2(runif(10000), rnorm(1000)))**.
- **Rprof**: un profiler para ver cuantas veces es llamada cada función y cuanto tiempo se usa en esa función.
- Se puede ver el status con **gc**(), que sirve también para despejar las cosas después de operaciones con manejo de grandes objetos.

Cosas variadas (I)

source y BATCH: Para la ejecución no interactiva de código.

- Con source abrimos una sesión de R y hacemos **>** **source("mi.fichero.con.codigo.R")**.
- Con **BATCH**: **% R CMD BATCH mi.fichero.con.codigo.R**.
- **source** es en ocasiones más útil porque informa inmediatamente de errores en el código.
- **BATCH** no informa, pero no requiere tener abierta una sesión (se puede correr en el background).
- Puede que necesitemos explícitos print statements o hacer **source**(my.file.R, echo = TRUE).
- **sink** es el inverso de source (manda todo a un fichero).

Cosas variadas (II)

- Se pueden crear paquetes, con nuestras funciones, que se comporten igual que los demás paquetes. Ver *Writing R extensions*.
- R puede llamar código compilado en C/C++ y FORTRAN. Ver .C, .Call, .Fortran.Lexical scoping importante en programación más avanzada.
- No hemos mencionado el "computing on the language" (ej., do.call, eval, etc).
- R es un verdadero "object-oriented language". Dos implementaciones, las S3 classes y las S4 classes.