Estructura de datos

Luciano Selzer

28 June, 2018

Datos Tabulares

Una de las características más poderosa de R es la habilidad de leer datos tabulares – como la que podrías tener en una hoja de cállculo o CSV. Empecemos por hacer un set datos de prueba en la carpeta `data/`, llamada `datos-felinos.csv`.

```
pelaje,peso,gusta_ovillo
atigrado,2.1,1
negro,5.0,0
bicolor,3.2,1
```

Tip: Editando archivos de texto en R

Alternativamente, podés crear `data/datos-felinos.csv` usando un editor de texto (Bloc de Notas, Nano) o con RStudio usando la opción **File -> New File -> Text File**.

Podemos cargarlo en R usando la siguiente función:

```
gatos <- read.csv(file = "data/datos-felinos.csv")
gatos
```

```
    pelaje peso gusta_ovillo
1 atigrado  2.1         TRUE
2    negro  5.0        FALSE
3  bicolor  3.2         TRUE
```

Podemos empezar a exploar nuestro set de datos enseguida, seleccionando columnas con el operador `$`:

```
gatos$peso
```

```
[1] 2.1 5.0 3.2
```

```
gatos$pelaje
```

```
[1] atigrado negro    bicolor
Levels: atigrado bicolor negro
```

Se pueden hacer otras operaciones con las columnas:

```
## Digamos que descubrimos que los pesos tienen un kilo de menos:
gatos$peso + 1
```

```
[1] 3.1 6.0 4.2
```

```
paste("Mi gato es", gatos$pelaje)
```

```
[1] "Mi gato es atigrado" "Mi gato es negro"    "Mi gato es bicolor"
```

Pero que pasa si hacemos esto:

```
gatos$peso + gatos$pelaje
```

```
Warning in Ops.factor(gatos$peso, gatos$pelaje): '+' not meaningful for
factors
```

```
[1] NA NA NA
```

Entender los tipos básicos de datos es fundamental para entender como funciona R.

Tipos de datos

Podemos ver que tipo de dato es cierto objeto con:

```
typeof(gatos$peso)
```

```
[1] "double"
```

Hay 5 tipos básicos de dato:

- double
- integer
- complex
- logical
- character.

```
typeof(3.14)
```

```
[1] "double"
```

```
typeof(1L)
```

```
[1] "integer"
```

```
typeof(1 + 1i)
```

```
[1] "complex"
```

```
typeof(TRUE)
```

```
[1] "logical"
```

```
typeof('banana')
```

```
[1] "character"
```

Otro usuario a añadido datos de otro gato. La información está en el archivo `data/feline-data_v2.csv`.

```
file.show("data/datos_felinos_v2.csv")
```

```
pelaje,peso,gusta_ovillo
atigrado,2.1,TRUE
negro,5.0,FALSE
```

```
bicolor,3.2,TRUE
bicolor,2.3 o 2.4,TRUE
```

Cargamos de nuevo los datos como antes, y chequeamos que tipo de dato encontramos en la columna `peso`:

```
gatos <- read.csv(file = "data/datos_felinos_v2.csv")
typeof(gatos$peso)
```

```
[1] "integer"
```

Ahora nuestros pesos ya no son de tipo doble.

Si intentamos hacer la misma operación matemática que antes vamos a tener problemas:

```
gatos$peso + 1
```

```
Warning in Ops.factor(gatos$peso, 1): '+' not meaningful for factors
```

```
[1] NA NA NA NA
```

¿Qué ha sucedido?

- Cuando R lee un archivo, insiste en que todo en una columna tiene que ser del msimo tipo básico. Si no puede entender que *todo* en la columna es tipo doble, entonces *nadie* en la columna tiene tipo doble. La tabla cargada en nuestro ejemplo de datos es llamada *data.frame* y es nuestro primer ejemplo de algo llamado *estructura de datos* - que significa, una estrucutura que R sabe construir a partir de tipos de datos básicos.

Podemos ver que es una *data.frame* llamando la función `class` en el objecto:

```
class(gatos)
```

```
[1] "data.frame"
```

Para usar existosamente nuestros datos en R, primero necesitamos entender que son los tipos básicos de estructuras y como se comportan. Por ahora quitemos la última linea de nuestros gatos y recarguemos el archivo, mientras investigamos más este comportamiento.

---

datos-felinos.csv:

```
pelaje,peso,gusta_ovillo
atigrado,2.1,1
negro,5.0,0
bicolor,3.2,1
```

Y devuelta en RStudio

```
gatos <- read.csv(file="data/datos-felinos.csv")
```

Vectores y Coerción de Tipo

Para entender mejor este comportamiento, conozcamos otro tipo de estructura de datos: el *vector*

```r
mi_vector <- vector(length = 3)
mi_vector
```

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

Un vector es:

- lista ordenada de cosas
- *Todo en el vector deber ser del mismo tipo básico.*
- Por defecto selecciona `logical`

```r
otro_vector <- vector(mode = 'character', length = 3)
otro_vector
```

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

Podés ver si un objecto es un vector:

```r
str(otro_vector)
```

```
 chr [1:3] "" "" ""
```

La salida algo críptica de este comando indica el tipo básico de dato en este vector - en este caso `chr`, caracter; un indicador del númer de cosas en ese vector, en este caso `[1:3]`; y unos ejemplos de que hay en el vector - en este caso una cadena de caracteres vacia.

De forma similar si hacemos:

```r
str(gatos$peso)
```

```
 Factor w/ 4 levels "2.1","2.3 o 2.4",..: 1 4 3 2
```

Vemos que también es un vector - *las columnas de datos que cargamos en el data.frame de R son todos vectores*, y es por eso que R fuerza toda la columna en ser de un mismo tipo básico de dato.

Discusión 1

¿Por qué R es tán dogmático sobre lo que ponemos en nuestras columnas de datos? ¿Cómo esto nos ayuda?

- Al tener todos los datos de la columna como un solo tipo nos permite hacer algunas suposiciones simples sobre ellos; si podes interpretar una entrada como un número, entonces podés interpetar *todas* las entradas como números, entonces no tenemos que comprobar cada una. Esta consistencia, como la consistencia de usar el mismo separador en nuestros archivos, es lo que se refiere a *datos limpios*. A la larga, esa consistencia estricta nos facilita mucho nuestro trabajo con R.

También podes hacer vectores con contenidos explicitos usando la función de concatenación

```r
concat_vector <- c(2,6,3)
concat_vector
```

```
[1] 2 6 3
```

Dado lo que ya hemos aprendido ¿Qué tipo de vector piensa que va a proudcir lo siguiente?

```
quiz_vector <- c(2,6,'3')
```

Esto es algo llamado *coerción de tipo*, y es la fuente de muchas sorpresas y la razón por la que tenemos que estar atentos de los tipos básicos de datos y como R va a interpretarlos. Cuando R encuentra un mezcla de tipos (númericos y caracter en este ejemplo) para ser combinados en un solo vector, va a forzar todos al mismo tipo. Considera lo siguiente:

```
coercion_vector <- c('a', TRUE)
coercion_vector
```

```
[1] "a"    "TRUE"
```

```
otro_coercion_vector <- c(0, TRUE)
otro_coercion_vector
```

```
[1] 0 1
```

Los reglas de coerción son: `logical` -> `integer` -> `numeric` -> `complex` -> `character`, donde -> es leido como es *transofrmado en Podes intertar forzar la coerción contra este flujo usando las funciones `as.`

```
character_vector_ejemplo <- c('0','2','4')
character_vector_ejemplo
```

```
[1] "0" "2" "4"
```

```
character_coerced_to_numeric <- as.numeric(character_vector_ejemplo)
character_coerced_to_numeric
```

```
[1] 0 2 4
```

```
numeric_coerced_to_logical <- as.logical(character_coerced_to_numeric)
numeric_coerced_to_logical
```

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

Si tus datos no se ven como se deben ver probablemente sea debido a la coerción de tipos

---

Pero la coerción de tipos también puede ser ¡muy útil! Por ejemplo, en nuestros ejemplo de `gatos` los datos de `gusta_ovillo` es númerico, pero sabemos que los ceros y unos representan `FALSE` y `TRUE` (una forma común de representarlos). Deberiamos usar el tipo `logical` aquí, el cual tiene dos estados: `TRUE` o `FALSE`, que es exactamente lo que nuestros datos representa. Podemos coercionar esa columna a `logical` usando la función `as.logical`:

```
gatos$gusta_ovillo
```

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

```
gatos$gusta_ovillo <- as.logical(gatos$gusta_ovillo)
gatos$gusta_ovillo
```

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

Concatenear también puede adjuntar cosas a un vector existente:

```
ab_vector <- c('a', 'b')
ab_vector
```

```
[1] "a" "b"
```

```
concat_example <- c(ab_vector, 'SWC')
concat_example
```

```
[1] "a"   "b"   "SWC"
```

También se pueden hacer series de números

```
mySeries <- 1:10
mySeries
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

```
seq(10)
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

```
seq(1,10, by = 0.1)
```

```
 [1]  1.0  1.1  1.2  1.3  1.4  1.5  1.6  1.7  1.8  1.9  2.0  2.1  2.2  2.3
[15]  2.4  2.5  2.6  2.7  2.8  2.9  3.0  3.1  3.2  3.3  3.4  3.5  3.6  3.7
[29]  3.8  3.9  4.0  4.1  4.2  4.3  4.4  4.5  4.6  4.7  4.8  4.9  5.0  5.1
[43]  5.2  5.3  5.4  5.5  5.6  5.7  5.8  5.9  6.0  6.1  6.2  6.3  6.4  6.5
[57]  6.6  6.7  6.8  6.9  7.0  7.1  7.2  7.3  7.4  7.5  7.6  7.7  7.8  7.9
[71]  8.0  8.1  8.2  8.3  8.4  8.5  8.6  8.7  8.8  8.9  9.0  9.1  9.2  9.3
[85]  9.4  9.5  9.6  9.7  9.8  9.9 10.0
```

Podemos ver otras cosas sobre los vectores

```
sequence_example <- seq(10)
head(sequence_example, n = 2)
```

```
[1] 1 2
```

```
tail(sequence_example, n = 4)
```

```
[1]  7  8  9 10
```

```
length(sequence_example)
```

```
[1] 10
```

```
class(sequence_example)
```

```
[1] "integer"
```

```
typeof(sequence_example)
```

```
[1] "integer"
```

Finalmente, podes darle nombres a los elementos de vector:

```
names_example <- 5:8
names(names_example) <- c("a", "b", "c", "d")
names_example
```

```
a b c d
5 6 7 8
```

```
names(names_example)
```

```
[1] "a" "b" "c" "d"
```

Ejercicio 1

Empieza haciendo un vector de 1 hasta 26. Multiplica el vector por 2, y dale al vector resultante nombres desde de la A hasta la Z (pista: hay un vector incluido llamado `LETTERS`).

Data Frames

Dijimos que las columnas en el data.frame son vectores:

```
str(gatos$peso)
```

```
 Factor w/ 4 levels "2.1","2.3 o 2.4",..: 1 4 3 2
```

```
str(gatos$gusta_ovillo)
```

```
 logi [1:4] TRUE FALSE TRUE TRUE
```

Tiene sentido, pero que hay de:

```
str(gatos$pelaje)
```

```
 Factor w/ 3 levels "atigrado","bicolor",..: 1 3 2 2
```

Factores

Otra estructura de datos importante son los llamados *factores*.

Los factores generalmente se ven como datos tipo caracter, pero son usado tipicamente para representar información categórica.

Por ejemplo, hagamos un vector de cadena nombrando los colores de los gatos para todos los gatos en el estudio:

```
pelajes <- c('atigrado', 'carey', 'carey', 'negro', 'atigrado')
pelajes
```

```
[1] "atigrado" "carey"    "carey"    "negro"    "atigrado"
```

```
str(pelajes)
```

```
 chr [1:5] "atigrado" "carey" "carey" "negro" "atigrado"
```

Podemos convertir nuestro vector en factor así:

```
Categorias <- factor(pelajes)
class(Categorias)
```

```
[1] "factor"
```

```
str(Categorias)
```

```
 Factor w/ 3 levels "atigrado","carey",..: 1 2 2 3 1
```

Ahora R se ha dado cuenta que hay tres posibles categorias en nuestros datos - pero también hizo algo sorprendente; en vez de imprimir las cadenas que le dimos, obtuvimos un montón de números. R ha reemplazado nuestra categorias facilmente interpretables por nosotros con indices númericos debajo del capot.

```
typeof(pelajes)
```

```
[1] "character"
```

```
typeof(Categorias)
```

```
[1] "integer"
```

Ejercicio 2

¿Hay algún factor en nuestro data.frame `gatos`?¿Cuál es su nombre? Intenta usar `?read.csv` para averiguar como mantener el texto como vector caracter en vez de factores; luego escribe un comando o dos para mostrar que el factor en `gatos` es en verdad un vector caracter cuando es cargo de estas forma.

En las funciones modeladoras, es importante saber cuales son los niveles de base. Se asume que es el primer factor, pero por defecto los factores son etiquetados en orden alfabético. Podes cambiar esto especificando los niveles:

```
mydata <- c("case", "control", "control", "case")
factor_ordering_example <- factor(mydata, levels = c("control", "case"))
str(factor_ordering_example)
```

```
 Factor w/ 2 levels "control","case": 2 1 1 2
```

En este caso, le hemos explicitado a R que “control” debería ser representado por 1, y “case” por 2. Esta designación es muy importante al interpretar los resultados de los modelos estadísticos.

Listas

Otra estructura de datos que querrás en tu manga será la `list` (lista). Una lista tiene menos restricciones que otras estructuras, porque podés poner cualquier cosa en ella:

```
list_example <- list(1, "a", TRUE, 1 + 4i)
```

```
list_example
```

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

[[2]]
[1] "a"

[[3]]
[1] TRUE

[[4]]
[1] 1+4i
```

```
another_list <- list(title = "Research Bazaar", numbers = 1:10, data = TRUE )
another_list
```

```
$title
[1] "Research Bazaar"

$numbers
 [1]  1  2  3  4  5  6  7  8  9 10

$data
[1] TRUE
```

Ahora podemos entender algo sorprendente de nuestra data.frame; ¿que sucede si ejecutamos?:

```
typeof(gatos)
```

```
[1] "list"
```

En nuestro ejemplo `gatos`, tenemos una variable *integer*, una *double* y una *logical*. Como hemos visto ya, cada columna del data.frame es un vector

```
gatos$pelaje
```

```
[1] atigrado negro    bicolor  bicolor
Levels: atigrado bicolor negro
```

```
gatos[,1]
```

```
[1] atigrado negro    bicolor  bicolor
Levels: atigrado bicolor negro
```

```
typeof(gatos[,1])
```

```
[1] "integer"
```

```
str(gatos[,1])
```

```
 Factor w/ 3 levels "atigrado","bicolor",..: 1 3 2 2
```

Cada columna es una *observación* de cada tipo de variable, por si misma una data.frame, y por lo tanto puede estar compuesta de elementos de diferentes tipos.

```
gatos[1,]
```

```
    pelaje peso gusta_ovillo
1 atigrado  2.1         TRUE
```

```
typeof(gatos[1,])
```

```
[1] "list"
```

```
str(gatos[1,])
```

```
'data.frame':   1 obs. of  3 variables:
 $ pelaje      : Factor w/ 3 levels "atigrado","bicolor",..: 1
 $ peso        : Factor w/ 4 levels "2.1","2.3 o 2.4",..: 1
 $ gusta_ovillo: logi TRUE
```

Ejercicio 3

Hay varios formas sutilmente diferentes para llamar a las variables:

- `gatos[1]`
- `gatos[[1]]`
- `gatos$pelaje`
- `gatos["pelaje"]`
- `gatos[1, 1]`
- `gatos[, 1]`
- `gatos[1, ]`

Pista : Usa la función `typeof()` para examinar que se devuelve en cada caso.

Podemos pensar que la data.frame es una lista de vectores. Un corchete simple `[1]` devuelve la primer rebanda de la lista de vectores, como otra lista. En este caso la primera columna del data.frame.

```
gatos[[1]]
```

```
[1] atigrado negro    bicolor  bicolor
Levels: atigrado bicolor negro
```

Los corchetes dobles `[[1]]` devuelven el contenido del item de la lista. En este caso los contenidos de la primera columna, un *vector* de tipo *factor*.

```
gatos$pelaje
```

```
[1] atigrado negro    bicolor  bicolor
Levels: atigrado bicolor negro
```

Este ejemplo usa el signo `$` para llamar a los items por su nombre. *pelaje* es la primera columna de la data.frame, un *vector* de tipo *factor*.

```
gatos["pelaje"]
```

```
    pelaje
1 atigrado
2    negro
3  bicolor
4  bicolor
```

Aquí usamos el corchete simple `["pelaje"]` reemplazando el número de indice con el nombre de la columna. Como en el primer ejemplo, el objeto que devuelve es una lista *list*

```
gatos[1, 1]
```

```
[1] atigrado
Levels: atigrado bicolor negro
```

Este ejemplo usa un corchete simple, pero esta vez damos el valor de fila y columna. El objeto que se devuelve es el valor de la fila 1 y columna. El objeto es *integer* porque es parte de un *vector* de tipo *factor*, R muestra la etiqueta "atigrado"

```
gatos[, 1]
```

```
[1] atigrado negro    bicolor  bicolor
Levels: atigrado bicolor negro
```

Como en el ejemplo anterior usamos corchetes simples y proveemos las coordenadas de fila y columna. La fila no está especificada, R interpreta este valor ausente como todo los valores del este *vector columna*.

```
gatos[1, ]
```

```
    pelaje peso gusta_ovillo
1 atigrado  2.1         TRUE
```

De nuevo usamos el corchete simple con las coordenadas de fila y columa. La columna no está especificada. Se devuelve una *lista* que contiene todos los valores de la primer columna.

Matrices

Último pero no menos está la matriz. Podemos declarar una matriz llena de ceros:

```
matrix_example <- matrix(0, ncol = 6, nrow = 3)
matrix_example
```

```
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    0    0    0    0    0    0
[2,]    0    0    0    0    0    0
[3,]    0    0    0    0    0    0
```

Y como en otras estructuras de datos, podemos preguntar cosas sobre nuestra matriz:

```
class(matrix_example)
```

```
[1] "matrix"
```

```
typeof(matrix_example)
```

```
[1] "double"
```

```
str(matrix_example)
```

```
 num [1:3, 1:6] 0 0 0 0 0 0 0 0 0 0 ...
```

```
dim(matrix_example)
```

```
[1] 3 6
```

```
nrow(matrix_example)
```

```
[1] 3
```

```
ncol(matrix_example)
```

```
[1] 6
```

Ejercicio 4

¿Cual crees que será el resultado de `length(matrix_example)`? Prueba el código.
¿Estabas en lo correcto?¿Por qué?

Ejercicio 5

Crea otra matriz, esta vez que contenga los numeros de 1 a 50, con 5 columnas y 10 filas.
¿Cómo lleno la función `matrix` la matriz? ¿Por columnas o por filas? ¿Cuál es su opción por defecto? Fijate si podés averiguar como cambiar esto (pista: lee la documentación de `matrix`).

Ejercicio 6

Crea una lista de longitud 2 que contenga un vector caracter para cada parte de lo que hemos visto

- Data types
- Data structures

Llena cada vector con los nombres de los tipos de datos y las estructuras de datos que hemos visto hasta ahora.

Ejercicio 7

Considera la salida de R de la siguiente matriz: [,1] [,2] [1,] 4 1 [2,] 9 5 [3,] 10 7

¿Cual fue el comando que se uso para crearla? Examina cada comando e intenta pensar cual es el correcto antes de tipearlos. Piensa en que tipo de matriz producirá cada uno.

1. `matrix(c(4, 1, 9, 5, 10, 7), nrow = 3)`
2. `matrix(c(4, 9, 10, 1, 5, 7), ncol = 2, byrow = TRUE)`
3. `matrix(c(4, 9, 10, 1, 5, 7), nrow = 2)`
4. `matrix(c(4, 1, 9, 5, 10, 7), ncol = 2, byrow = TRUE)`