R Programming

Martín Macías

Diciembre de 2015

1 Estableciendo el Directorio de trabajo

La idea es hacer el seguimiento de los comandos en `R` para establecer el directorio de trabajo.

- Con la opción `getwd()` se obtiene el directorio de trabajo actual:

```
  getwd()

[1] "/Users/Martin/Desktop/GitHub/datasciencecoursera"
```

- Para ver los archivos crpetas que hay en el directorio de trabajo actual:

```
  dir()

[1] "datasciencecoursera.Rproj" "HelloWorld.md"
[3] "R Programming.Rnw"         "R_Programming.bbl"
[5] "R_Programming.pdf"         "R_Programming.Rnw"
[7] "R_Programming.tex"         "README.md"
[9] "y.R"
```

- El comando `ls()` muestra lo que exista en mi espacio de trabajo:

```
  ls()

character(0)
```

2 Data types

2.1 Vectores y listas

2.1.1 Creando vectores

La función `c()` se usa para crear vectores de objetos:

```r
x <- c(0.5, 0.6)       # Numérico
x <- c(TRUE, FALSE)    # Lógico
x <- c(T, F)           # Lógico
x <- c("a", "b","c")   # Caracter
x <- 9:29              # Entero
x <- c(1+0i, 2+4i)     # Complejo
```

Usando la función `vector()`

```r
x <- vector("numeric", length = 10)
x
```

```
 [1] 0 0 0 0 0 0 0 0 0 0
```

2.1.2 Mezclando objetos

Miremos cómo se mezclan objetos en un vector

```r
y <- c(1.7, "a")     # Caracter
y <- c(TRUE, 2)      # Numérico
y <- c("a", TRUE)    # Caracter
```

2.1.3 Concatenación explícita

Los objetos pueden concatenarse explícitamente de una clase a otra usando las funciones `as.*`, si están disponibles:

```r
x <- 0:6
class(x)
```

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

```r
as.numeric(x)
```

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

```r
as.logical(x)
```

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

```r
as.character(x)
```

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

La concatenación sinsentido resulta en `NAs`

```r
  x <- c("a", "b", "c")
  class(x)
```

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

```r
  as.numeric(x)
```

```
Warning:  NAs introduced by coercion
```

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

```r
  as.logical(x)
```

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

```r
  as.complex(x)
```

```
Warning:  NAs introduced by coercion
```

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

2.2 Listas

Las listas son tipos de vectores especiales que pueen contener elementos de diferentes clases.

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

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

[[2]]
[1] "a"

[[3]]
[1] TRUE

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

2.3 Matrices

Las matrices son vectores con un atributo de dimensión. El atributo de dimensión es, en sí mismo, un vector entero de longitud 2 `(nrow, ncol)`

```
  m <- matrix(nrow = 2, ncol = 3)
  m

     [,1] [,2] [,3]
[1,]   NA   NA   NA
[2,]   NA   NA   NA

  dim(m)

[1] 2 3

  attributes(m)

$dim
[1] 2 3
```

Las matrices se contruyen en el sentido de las columnas, es decir, en forma de zig zag invertido

```
  m <- matrix(1:6, nrow = 2, ncol = 3)
  m

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

Las matrices también pueden crearse directamente de vectores añadiendo el atributo de dimensión

```
  m <- 1:10
  m

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

  dim(m) <- c(2, 5)
  m

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

Las matrices también pueden crearse mediante la unión de filas `rbind()` o la unión de columnas `cbind`

```
  x <- 1:3
  y <- 10:12
  cbind(x ,y)
```

```
     x  y
[1,] 1 10
[2,] 2 11
[3,] 3 12
```

```
  rbind(x, y)
```

```
  [,1] [,2] [,3]
x    1    2    3
y   10   11   12
```

2.4 Factores

Los factores se usan para representar datos categóricos. Los factores pueden ser ordenados o desordenados.

- Los factores son utilizados especialmente para modelar funciones como `lm()` y `glm()`
- Usar factores con labels es mejor que usar enteros puesto que los factores son autodescriptivos; tener una variable que tenga como valores ¡¡Masculino¿¿ y ¡¡Femenino¿¿ es mejor que tener una variable con 1 y 2.

```
  x <- factor(c("yes", "yes", "no", "yes", "no"))
  x
```

```
[1] yes yes no  yes no
Levels: no yes
```

```
  table(x)    # Hace conteo de niveles que hay
```

```
x
 no yes
  2   3
```

```
  unclass(x)  # Despoja de la clase que tenga el vector
```

```
[1] 2 2 1 2 1
attr(,"levels")
[1] "no"  "yes"
```

El orden de los niveles se puede determinar usando el argumento `levels` en `factor()`. Esto puede ser importante en modelación lineal porque el primer nivel se usa como nivel de base.

```r
  x <- factor(c("yes", "yes", "no", "yes", "no"),
              levels = c("yes", "no"))
  x
```

```
[1] yes yes no  yes no
Levels: yes no
```

2.5 Datos faltantes

Los datos faltantes se denotan por `NA` o `NaN` para operaciones matemáticas indefinidas.

- `is.na()` se usa para probar si los objetos son `NA`
- `is.nan()` se usa para probar si los objetos son `NaN`
- Los valores `NA` también tienen clase. Pueden ser enteros `NA`, caracter `NA`, etc.
- Un valor `NaN` es también `NA` pero el recíproco no es cierto

```r
  x <- c(1, 2, NA, 10, 3)
  is.na(x)
```

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

```r
  is.nan(x)
```

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

```r
  x <- c(1, 2, NaN, NA, 4)
  is.na(x)
```

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

```r
  is.nan(x)
```

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

2.6 Data frames

Los data frames se usan para almacenar datos tabulados.

- Se representan como un tipo especial de lista donde cada elemento de la lista debe tener la misma longitud.

- Cada elemento de la lista se puede pensar como una columna y la longitud de cada elemento de la lista es el número de filas.
- A diferencia de las matrices, los data frames pueden almacenar diferentes clases de objetos en cada columna (así como las listas): los elementos de las matrices deben ser de la misma clase.
- Los data frames también tienen un atributo especial llamado `row.names`
- Los data frames se crean usualmente mediante `read.table()` o `read.csv()`
- Los data frames pueden convertirse en matrices mediante `data.matrix()`

```
  x <- data.frame(foo = 1:4, bar = c(T, T, F, F))
  x

  foo   bar
1   1  TRUE
2   2  TRUE
3   3 FALSE
4   4 FALSE

  nrow(x)

[1] 4

  ncol(x)

[1] 2
```

2.7 Atributos de nombre

Los objetos en R también pueden tener nombres, los cuales son muy útiles para esribir código legible y objetos autodescriptivos.

```
  x <- 1:3
  names(x)

NULL

  names(x) <- c("foo", "bar", "norf")
  x

 foo  bar norf
   1    2    3

  names(x)

[1] "foo"  "bar"  "norf"
```

Las listas también pueden tener nombres.

```
  x <- list(a = 1, b = 2, c = 3)
  x

$a
[1] 1

$b
[1] 2

$c
[1] 3
```

Para las matrices funciona de igual forma.

```
  m <- matrix(1:4, nrow = 2, ncol = 2)
  dimnames(m) <- list(c("a", "b"), c("c", "d"))
  m

  c d
a 1 3
b 2 4
```

3 Leer datos tabulados

Existen unas pocas funciones principales para leer datos en `R`.

- `read.table` y `read.csv` para leer datos tabulados.
- `readLines` para leer líneas de texto.
- `source` para leer archivos de `R` (`inverse` de `dump`)
- `dget` para leer archivos de `R` (`inverse` de `dput`)
- `load` para leer de espacios de trabajo guardados
- `unserialize` para leer objetos de `R` en forma binaria.

3.1 `read.table`

La función `read.table` es una de las funciones más comúnmente usadas para leer datos. Tienen unos pocos argumentos:

- `file`, el nombre de un archivo o conexión.
- `header`, tipo lógico e indica si el archivo tiene línea de cabecera

- `sep`, tipo cadena e indica cómo están separadas las columnas.
- `colClasses`, un vector de caracteres que indica la clase de cada columna en el conjunto de datos.
- `nrows`, número de filas en el conjunto de datos.
- `comment.char`, una cadena de caracteres que indica el caracter del comentario.
- `skip`, número de líneas que deben omitirse desde el principio.
- `stringsAsFactors`, ¿deberían las variables caracter ser codificadas como factores?

Nota: `read.table` trabaja con archivos separados por espacios mientras que `read.csv` trabaja con archivos separados por comas.

3.2 Usando `read.table`

El argumento `colClasses` acelera el uso de la función `read.table` hasta casi el doble. Para usar esta opción, debe conocerse la clase de cada columna de su data frame. Si todas las columnas son numéricas, por ejemplo, entonces debe ajustar el argumento así: `colClasses = "numeric"`.

```
initial <- read.table("datatable.txt", nrows = 100)
classes <- sapply(initial, class)
tabAll <- read.table ("datatable.txt", colClasses = classes)
```

Si configura `nrows` podrá ahorrar memoria. Puede usar la herramienta de Unix `wc` para calcular las líneas en un archivo.

3.3 Formatos de datos textuales

- `dumping` y `dputing` son muy útiles porque el formato textual resultante es editable y en caso de que se corrompan, potenciablemente recuperable.
- A diferencia de trabajar con una tabla o con un archivo csv, las funciones `dumping` y `dputing` conservan los metadatos (sacrificando algo de legibilidad) para que otro usuario no tenga que especificar todo de nuevo.
- Los formatos textuales pueden trabajar mucho mejor con programas de control de versiones, los cuales pueden rastrear cambios significativos es archivos de texto.
- Los archivos textuales pueden ser más duraderos; en casod e que se corrompan, es más fácil hallar el problema dentro del archivo.
- **Desventaja:** El formato no es eficiente en cuanto a ahorro de espacio.

3.4 dput-ing objetos en R

```
  y <- data.frame(a = 1, b = "a")
  dput(y)

structure(list(a = 1, b = structure(1L, .Label = "a", class = "factor")), .Names = c("a",
"b"), row.names = c(NA, -1L), class = "data.frame")

  dput(y, file="y.R")
  new.y <- dget("y.R")
  new.y

  a b
1 1 a
```

3.5 dump-ing objetos en R

```
  x <- "foo"
  y <- data.frame(a = 1, b = "a")
  dump(c("x", "y"), file = "data.R")
  rm(x, y)
  source("data.R")
  y

  a b
1 1 a

  x

[1] "foo"
```

4 Conclusion

"I always thought something was fundamentally wrong with the universe"