Introducción al procesamiento de datos en R

Instructor: Felipe González, ITAM.
Septiembre, 2008

Resumen. Cómo leer y escribir datos en R. Cómo almacenar datos en R y extraer datos de objetos de R Cómo construir tablas agregadas y reorganizar tablas.

Leer y escribir datos. Leer tablas de datos en los formatos usuales de texto (separado por comas, ancho fijo) es fácil. En el siguiente ejemplo leemos de un archivo separado por comas con encabezado (nombres de las columnas) que está en nuestro sistema. Cambiamos directorio de trabajo, leemos los datos, e imprimimos la tabla:

```
> setwd("/Users/felipe/proyectos/foro_r/datos")
> mi_tabla <- read.table("mis_datos.csv", sep = ",", header = TRUE)
> print(mi_tabla)
  ind  var1   var2
1   1  rojo  0.095
2   2  rojo  3.221
3   3 verde -2.342
```

De manera similar podemos leer archivos de otros formatos usando la librería *foreign*). Por ejemplo, para leer un archivo dbf hacemos:

```
> library(foreign)
> setwd("/Users/felipe/proyectos/foro_r/datos")
> so2_df<-read.dbf("2008SO2P.DBF")
> print(so2_df[1:4,1:3])   #Sólo imprime 4 renglones y 3 columnas
       FECHA HORA MSO2VAL
1 2008-01-01    1   0.010
2 2008-01-01    2   0.016
3 2008-01-01    3   0.013
4 2008-01-01    4   0.012
```

Podemos leer directamente de bases de datos usando la librería *RODBC* y comandos de *SQL* ([1]). En el siguiente ejemplo creamos una base de datos test, cuya fuente es una hoja de Excel:

```
> library(RODBC)
> conex <- odbcConnect("test")
> hojas <- sqlTables(conex)
> print(hojas[, 3:4])
  TABLE_NAME TABLE_TYPE
1    Colores      TABLE
2      Hoja2      TABLE
> query <- "SELECT * FROM Hoja2"
> resultado <- sqlQuery(conex, query)
> close(conex)
> print(resultado)
  Ind Color
1   1     V
2   2     V
3   3     M
```

Todo el archivo se lee a la memoria en los ejemplos anterior, así que no siempre podemos hacer esto con archivos muy grandes. En estos casos, podemos abrir conexiones y leer y procesar líneas secuencialmente (nótese también que con las primeras dos líneas bajamos el archivo del *internet movie database*):

```
> #url_base<-"ftp://ftp.sunet.se/pub/tv+movies/imdb/ratings.list.gz"
> #download.file(url_base,"ratings.list.gz")
> ratings_gz<-gzfile("ratings.list.gz","r")  #Abrir conexión
> encabezado<-readLines(ratings_gz,27)       #Leer líneas 1-27
> lineas<-readLines(ratings_gz,4)            #Leer líneas 28-31
> close(ratings_gz)
> print(lineas)
[1] "New  Distribution  Votes  Rank  Title"
[2] "      0000000115  371002   9.1  Shawshank Redemption, The (1994)"
[3] "      0000000015  316612   9.1  Godfather, The (1972)"
[4] "      0000000016  264079   9.0  Dark Knight, The (2008)"
```

Tablas de datos en la memoria de R pueden guardarse como archivos delimitados:

```
> write.table(lineas, file = "nuevo.csv", sep = ",", row.names = FALSE)
```

y podemos verificar leyendo de nuevo el archivo:

```
> nuevas_lineas <- read.table("nuevo.csv", sep = ",", header = TRUE)
> print(nuevas_lineas)
                                                                    x
1                        New  Distribution  Votes  Rank  Title
2       0000000115  371002   9.1  Shawshank Redemption, The (1994)
3                 0000000015  316612   9.1  Godfather, The (1972)
4               0000000016  264079   9.0  Dark Knight, The (2008)
```

Mientras trabajemos con R, sin embargo, conviene más usar el formato interno de R:

```
> save(lineas,resultado,file="mis_datosR.rdata")
> rm(list=ls()) #Borrar objetos de R en sesión
> load("mis_datosR.rdata")
> load("mis_datosR.rdata")
> head(resultado,2)
  Ind Color
1   1     V
2   2     V
> ls()          #Listar objetos de R en sesión
[1] "lineas"    "resultado"
```

Listas en R. Los datos en R se almacenan en objetos. Los objetos con los que más usualmente trabajamos son *listas*. Las listas son estructuras de datos recursivas: son colecciones ordenadas de listas. Podemos crear una lista con la función *list*:

```
> manzana <- list(nombre = "manzana", peso = 36.2, tipo = "fruta")
> pera <- list(nombre = "pera", tipo = "fruta", peso = 25)
> print(manzana)
$nombre
[1] "manzana"

$peso
[1] 36.2

$tipo
[1] "fruta"
> frutas <- list(uno = manzana, dos = pera)
> names(manzana)
[1] "nombre" "peso"   "tipo"
> names(frutas)
[1] "uno" "dos"
```

donde la función `names` nos da los nombres de las componentes de la lista. Usando los nombres podemos extraer componentes de la lista de la siguiente forma:

```
> manzana$tipo
[1] "fruta"
> manzana$peso
[1] 36.2
> pera$peso
[1] 25
> manzana$peso + pera$peso
[1] 61.2
```

aunque también podemos extraer las componentes con la notación de doble corchete:

```
> pera[[1]]
[1] "pera"


> manzana[[2]] + pera[[2]]


Error in manzana[[2]] + pera[[2]] :
  non-numeric argument to binary operator
```

La última línea nos da un error: no podemos sumar la componente 2 de `manzana` con la componente 2 de `pera`. Este tipo de errores son comunes, y pueden ser muy frustrantes para los principiantes de R. Cuando aparecen estos errores, conviene checar los tipos de cada uno de los objetos sobre los que estamos operando con la función `mode`, que nos dice el tipo de las componentes de cada objeto.

```
> mode(pera$tipo)
[1] "character"
> mode(manzana$peso)
[1] "numeric"
> mode(manzana)
[1] "list"
```

y el problema es que la suma no está definida para un objeto de tipo *character* y uno de tipo *numeric*.

Modo y clase de objetos Los objetos *atómicos* son aquellos cuyas componentes son todas de tipo *numeric*,*character*,*logical*, o *complex*. En el ejemplo de arriba, vemos por ejemplo que `pera$tipo` y `manzana$peso` son atómicos. Los objetos atómicos más importantes son los *vectores* y las *matrices*. Empezamos construyendo un vector numérico y uno de caracteres:

```
> x <- c(1.5, 2.2, -5)
> a <- c("manzana", "pera", "plátano")
> mode(x)
[1] "numeric"
> mode(a)
[1] "character"
```

Podemos juntar estos dos vectores en una lista

```
> mi_lista <- list(medida = x, fruta = a)
> mode(mi_lista)
[1] "list"
> print(mi_lista$medida)
[1]  1.5  2.2 -5.0
> print(mi_lista)
$medida
[1]  1.5  2.2 -5.0

$fruta
[1] "manzana" "pera"    "plátano"
```

y ya casi tenemos la estructura de datos más usual de R. Para crearla usamos la función `data.frame`:

```
> mis_datos <- data.frame(mi_lista)
> mode(mis_datos)
[1] "list"
> names(mis_datos)
[1] "medida" "fruta"
> mode(mi_lista)
[1] "list"
> names(mi_lista)
[1] "medida" "fruta"
```

Nótese que cuando aunque el tipo de `mis_datos` sigue siendo lista, cuando hicimos `print(mis_datos)` no obtuvimos el mismo resultado que cuando hicimos `print(mi_lista)`. La razón es que al usar la función `data.frame` convertimos a `mi_lista` en un objeto de una *clase* distinta:

```
> class(mi_lista)
[1] "list"
> class(mis_datos)
[1] "data.frame"
```

y la función `print` funciona de distinta manera según la clase de su argumento.

En general, todos los objetos además de tener un tipo o modo tienen una clase. La clase es estructura adicional que determina, entre otras cosas, cómo aplican las funciones a ese objeto.

Obtener datos de objetos de R. Una gran parte de los objetos usuales de R son listas. Esto incluye los *marcos de datos* o *data frames*, y los objetos que dan como resultado distintos análisis. Veamos unos ejemplos:

```
> setwd("/Users/felipe/proyectos/foro_r/datos")
> so2_df <- read.table("2008SO2P2.csv", sep = ",", header = TRUE)
> mode(so2_df)
[1] "list"
> class(so2_df)
[1] "data.frame"
```

La función *read.table* produce un marco de datos. Las componentes son

```
> names(so2_df)
[1] "FECHA.D"       "HORA.N.2.0"    "MSO2VAL.N.7.3" "MSO2SUR.N.7.3"
[5] "MSO2TAC.N.7.3" "MSO2EAC.N.7.3" "MSO2LLA.N.7.3"
```

y podemos extraerlas como explicamos arriba, quizá creando nuevos objetos, y luego aplicarles funciones:

```
> sta_ursula <- so2_df$MSO2SUR.N.7.3
> head(sta_ursula, 20)
 [1] 0.013 0.023 0.015 0.010 0.007 0.006 0.005 0.004 0.008 0.008 0.004 0.003
[13] 0.003 0.002 0.002 0.002 0.002 0.002 0.002 0.002
> summary(sta_ursula)
     Min.   1st Qu.    Median      Mean   3rd Qu.      Max.      NA's
0.000e+00 2.000e-03 5.000e-03 5.977e-03 7.000e-03 1.100e-01 1.920e+02
> length(sta_ursula)
[1] 5112
> mode(sta_ursula)
[1] "numeric"
> length(so2_df)
[1] 7
```

Los resultados de análisis los podemos tratar de la misma forma:

```
> densidad<-density(sta_ursula,na.rm=TRUE) #Estimación con kernel de densidad
> names(densidad)
[1] "x"         "y"         "bw"        "n"         "call"      "data.name"
[7] "has.na"
> head(densidad$x)    #Qué contiene la componente x
[1] -0.0018400768 -0.0016176107 -0.0013951447 -0.0011726786 -0.0009502126
[6] -0.0007277465
> class(densidad)
[1] "density"
> mode(densidad)
[1] "list"
> print(densidad)     #¿Qué hace print con este objeto?
Call:
       density.default(x = sta_ursula, na.rm = TRUE)

Data: sta_ursula (4920 obs.);        Bandwidth 'bw' = 0.0006134

       x                 y
 Min.   :-0.00184   Min.   :2.675e-06
 1st Qu.: 0.02658   1st Qu.:8.101e-02
 Median : 0.05500   Median :2.467e-01
 Mean   : 0.05500   Mean   :8.791e+00
 3rd Qu.: 0.08342   3rd Qu.:1.411e+00
 Max.   : 0.11184   Max.   :1.265e+02
```

En resumen, 1) la estructura de datos más común en R es la lista: los conjuntos de datos son listas de clase *dataframe*, y es común que los resultado de los análisis sean listas con distintas clases, 2) Las funciones´de R muchas veces son polimórficas: dan resultados distintos según la clase de los objetos en los que se evalúan, y 3) Cuando queremos extraer de objetos que tienen modo de lista, usamos la notación de $ mostrada arriba (¡aunque no es la única y manera, y no siempre es la mejor!)

Más de funciones en R. Generalmente los objetos creados en R tienen *funciones accesoras* ([2],[1]). Cuando éstas existen, son la manera preferida de extraer datos de objetos. Por ejemplo, consideramos las funciones accesoras de `lm`, que representa un modelo lineal.

```
> modelo_lineal <- lm(stack.loss ~ Air.Flow, data = stackloss)
> class(modelo_lineal)
[1] "lm"
> print(modelo_lineal)
Call:
lm(formula = stack.loss ~ Air.Flow, data = stackloss)

Coefficients:
(Intercept)     Air.Flow
    -44.132        1.020
```

Podemos ver qué funciones están definidas para los objetos `lm`:

```
> ## La función apropos busca entre TODOS los objetos de R según
> apropos('.*\\.lm$') #una expresión regular
 [1] "anova.lm"        "anovalist.lm"    "hatvalues.lm"    "kappa.lm"
 [5] "model.frame.lm"  "model.matrix.lm" "plot.lm"         "predict.lm"
 [9] "print.lm"        "residuals.lm"    "rstandard.lm"    "rstudent.lm"
[13] "summary.lm"
```

Asi que por ejemplo, para extraer los residuales obtenidos podemos hacer

```
> residuales <- residuals.lm(modelo_lineal)
> head(residuales)
         1          2          3          4          5          6
 4.5072796 -0.4927204  4.6088262  8.8728473 -1.1271527 -1.1271527
```

Ésta última línea de código no funcionaría si el objeto `modelo_lineal` fuera de otra clase para la que también tiene sentido obtener residuales. Es mejor llamar la *versión genérica* de la función, que es `residuals` ([1]), y así aseguramos que la siguiente línea de código funciona aunque usemos otro tipo de modelos (R se encarga de utilizar la función apropiada):

```
> residuales <- residuals(modelo_lineal)
> head(residuales)
         1          2          3          4          5          6
 4.5072796 -0.4927204  4.6088262  8.8728473 -1.1271527 -1.1271527
```

La función genérica `print`, por ejemplo, tiene muchas formas:

```
> salida<-apropos("print\\..*$")
> length(salida)
[1] 45
> print(salida[c(12,13)])  #Aquí sólo mostramos dos posibilidades.
[1] "print.coefmat"   "print.condition"
```

Vectores, matrices, y subíndices Como mencionamos arriba, las estructuras atómicas comunes son vectores y matrices. Podemos construir vectores con la función `c` y matrices con `matrix`

```
> x<-seq(from=0,to=1,by=0.1)
> print(x)
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
> mode(x)
[1] "numeric"
> length(x)
[1] 11
> y_mat<-matrix(x,nrow=3,ncol=4) #Qué pasa si x sólo contiene 11 elementos?
> print(y_mat)
     [,1] [,2] [,3] [,4]
[1,]  0.0  0.3  0.6  0.9
[2,]  0.1  0.4  0.7  1.0
[3,]  0.2  0.5  0.8  0.0
```

Además de `seq` o `matrix` hay muchas otras funciones para generar datos en R. Por ejemplo, podemos simular 1000 observaciones de una densidad normal estándar:

```
> z <- rnorm(1000, mean = 0, sd = 1)
> mode(z)
[1] "numeric"
> head(z)
[1] -1.7251833 -0.9623469 -0.8209533 -0.0684846  0.9406443  0.6670199
> mean(z)
[1] -0.006817222
> sum(z)
[1] -6.817222
> sd(z)
[1] 1.017570
```

Podemos extraer o redefinir entradas usando corchetes:

```
> x[1]
[1] 0
> x[5] <- 100
> print(x)
 [1]   0.0   0.1   0.2   0.3 100.0   0.5   0.6   0.7   0.8   0.9   1.0
```

Estos operadores también son polimórficos: los índices pueden ser vectores numéricos, por ejemplo

```
> ind <- seq(from = 2, to = 10, by = 2)
> ind
[1]  2  4  6  8 10
> x[ind]
[1] 0.1 0.3 0.5 0.7 0.9
```

o vectores lógicos

```
> cuales <- rep(FALSE, 11)
> cuales[3] <- TRUE
> cuales[6] <- TRUE
> cuales
 [1] FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
> x[cuales]
[1] 0.2 0.5
```

Lo mismo aplica para las matrices:

```
> print(y_mat)
     [,1] [,2] [,3] [,4]
[1,]  0.0  0.3  0.6  0.9
[2,]  0.1  0.4  0.7  1.0
[3,]  0.2  0.5  0.8  0.0
> y_mat[1, 3]
[1] 0.6
> y_mat[, 3]
[1] 0.6 0.7 0.8
> y_mat[c(2, 3), 2]
[1] 0.4 0.5
> y_mat[c(2, 3), 1:3]
     [,1] [,2] [,3]
[1,]  0.1  0.4  0.7
[2,]  0.2  0.5  0.8
```

Para marcos de datos podemos hacer más utlizando las componentes (que se interpretan como variables o atributos)

```
> print(mis_datos)
  medida   fruta
1    1.5 manzana
2    2.2    pera
3   -5.0 plátano
> mis_datos$medida[2]    #Extraer una entrada
[1] 2.2
> mis_datos[2,]         #Extraer un renglón
  medida fruta
2    2.2  pera
> mis_datos[2,"fruta"]  #Extraer una entrada
[1] pera
Levels: manzana pera plátano
> mis_datos["fruta"]   #Extraer columnas
    fruta
1 manzana
2    pera
3 plátano
> mis_datos[2:3,"fruta"] #Extraer varios renglones de una columna
[1] pera    plátano
Levels: manzana pera plátano
```

Nótese que algunas líneas arriba dieron como salida adicional un renglón comenzado con `Levels:`. La razón es que usualmente queremos que los vectores de caracteres sean de la clase particular `factor`. La clase `factor` es la más natural para las variables categóricas, y `data.frame` los crea automáticamente.

```
> mode(mis_datos[2:3, "fruta"])
[1] "numeric"
> class(mis_datos[2:3, "fruta"])
[1] "factor"
```

Ojo: aunque el lenguaje R tiene las estructuras usuales de ciclos (for...next, por ejemplo), no conviene extraer datos usando ciclos -esto puede ser sumamente lento. La manera rápida es usar subíndices como se explicó arriba.

Aplicando funciones a vectores,listas y matrices. Aplicar una función a cada elemento de un vector es fácil, pues las funciones de R generalmente están definidas para hacerlo:

```
> xx <- rnorm(1000, 0, 1)
> yy <- exp(xx)
> head(yy)
[1] 0.3434247 4.1124959 4.9654200 2.1851607 1.0047623 0.2000262
```

Para listas usamos la función `sapply`:

```
> mi_lista <- list(x = c(1, -2, 3), y = c(4, 10))
> sapply(mi_lista, mean)
       x         y
0.6666667 7.0000000
> sapply(mi_lista, sort)
$x
[1] -2  1  3

$y
[1]  4 10
> mi_lista <- list(x = c(1, -2, 3), y = c(4, 5, 10), z = c("a",
+     "b", "c"))
> sapply(mi_lista, mode)
          x           y           z
  "numeric"   "numeric" "character"
```

Esta función es útil para trabajar con marcos de datos:

```
> mis_datos<-data.frame(mi_lista,stringsAsFactors=FALSE)
> #los factores se almacenan con modo numeric
> sapply(mis_datos,mode)
          x           y           z
  "numeric"   "numeric" "character"
> numericos<-mis_datos[,sapply(mis_datos,mode)=="numeric"]
> numericos
   x  y
1  1  4
2 -2  5
3  3 10
```

Aplicar funciones a matrices o marcos de datos elemento por elemento sigue el mismo patrón de los vectores.

```
> exp(numericos)
```

```
          x          y
1  2.7182818    54.59815
2  0.1353353   148.41316
3 20.0855369 22026.46579
```

Pero muchas veces queremos operar a lo largo de renglones o columnas. La función `apply` es útil en estos caso. Por ejemplo, para calcular medianas o medias de los renglones o columnas:

```
> apply(y_mat, 1, median)
[1] 0.45 0.55 0.35
> apply(y_mat, 2, median)
[1] 0.1 0.4 0.7 0.9
> apply(numericos, 2, mean)
        x         y
0.6666667 6.3333333
```

Trabajando con cadenas de texto. Hay funciones poderosas disponibles en R. Es posible, por ejemplo, trabajar con expresiones regulares para búsqueda y sustitución. Aquí sólo recordamos el ejemplo de las calificaciones de las películas que vimos antes (desempaquetado es $\approx$ 15 MB):

```
> ratings <- read.table("ratings.list", skip = 28, fill = TRUE,
+     sep = "")
> head(ratings, 1)
          V1     V2  V3        V4          V5  V6     V7 V8 V9 V10
1 0000000115 371002 9.1 Shawshank Redemption, The (1994)
```

El argumento de `sep` indica que los separadores de campos son bloques de espacio blanco (espacios, tabs, nueva línea), y `fill` indica que si una línea tiene menos campos que el máximo, debe rellenar con campos en blancos. Usaremos otra función para crear el nombre de la película correctamente.

```
> paste(c("a","b","cdef"),collapse=" ")
[1] "a b cdef"
> nombres_sep<-ratings[,4:10]
> nombres<-apply(nombres_sep,1,paste,collapse=" ")
> #Nótese que en la función anterior pasamos arg collapse a paste
> ratings2<-ratings[,1:3]
> ratings2$nombres<-nombres
> head(ratings2,3)
          V1     V2  V3                                nombres
1 0000000115 371002 9.1 Shawshank Redemption, The (1994)
2 0000000015 316612 9.1           Godfather, The (1972)
3 0000000016 264079 9.0          Dark Knight, The (2008)
```

¿Podemos extraer el año del nombre de la película? Es fácil con expresiones regulares

```
> a<-gsub("^(.*)\\(([0-9]+)\\)(.*)$","\\2",nombres,perl=TRUE,useBytes=T)
> head(a)
[1] "1994" "1972" "2008" "1974" "1966" "1994"
> a_limpia<-encodeString(a)   #Eliminar caracteres no válidos
> años<-as.numeric(a_limpia)
> head(años)
[1] 1994 1972 2008 1974 1966 1994
> length(años)
[1] 269060
> sum(is.na(años))  #Datos incorrectos o formatos no estándar -???
[1] 34813
> ratings2$año<-años
> table(ratings2[ratings2$año>2000,"año"])
 2001  2002  2003  2004  2005  2006  2007  2008
 9144  9314 10170 10924 12473 10906  7987  2309
```

Agregando y Reacomodando marcos de datos. Hay diversas funciones, como `table`, para agregar datos (ver [2]) La librería *reshape* nos provee de las funciones `melt` y `cast` con las que podemos hacer las tareas de agregación a distintos niveles y de reacomodo de nuestros datos. Pensemos que hay dos tipos de variables en nuestro conjunto de datos [3]: identificadores y mediciones. En realidad, muchas veces podemos pensar que hay una medición única. Por ejemplo, consideramos los siguientes datos de un experimento sensorial (este ejemplo es de [3]):

```
> library(reshape)
> head(french_fries, 2)
   time treatment subject rep potato buttery grassy rancid painty
61    1         1       3   1    2.9       0      0    0.0    5.5
25    1         1       3   2   14.0       0      0    1.1    0.0
```

donde vemos cuatro mediciones que tienen qué ver con el sabor de papas fritas (papa, mantequilla, pasto, ráncido y pintura). El resto de las variables son el dia de la prueba de sabor (30), el tipo de aceite que se usó (3), el sujeto las probó (12), y la freidora que se usó (2). Podemos pensar que hay una sola medición, incluyendo las variables como identificadores:

```
> papas_melt <- melt(french_fries, id = 1:4)
> head(papas_melt, 3)
  time treatment subject rep variable value
1    1         1       3   1   potato   2.9
2    1         1       3   2   potato  14.0
3    1         1      10   1   potato  11.0
> tail(papas_melt, 3)
     time treatment subject rep variable value
3478   10         3      78   2   painty   1.4
3479   10         3      86   1   painty  10.5
3480   10         3      86   2   painty   9.4
```

A partir de este reacomodo podemos producir distintas tablas usando la función `cast`:

```
> cast(papas_melt,subject~variable,length) #número de mediciones
   subject potato buttery grassy rancid painty
1        3     54      54     54     54     54
2       10     60      60     60     60     60
3       15     60      60     60     60     60
4       16     60      60     60     60     60
5       19     60      60     60     60     60
6       31     54      54     54     54     54
7       51     60      60     60     60     60
8       52     60      60     60     60     60
9       63     60      60     60     60     60
10      78     60      60     60     60     60
11      79     54      54     54     54     54
12      86     54      54     54     54     54
> #Medianas a lo largo de tiempo, replicación y tratamiento:
> mi_tabla<-cast(papas_melt,subject~variable,median,na.rm=TRUE)
>  head(mi_tabla)
  subject potato buttery grassy rancid painty
1       3   4.00     0.5   0.00   0.85   0.65
2      10  10.20     6.8   0.00   2.40   0.00
3      15   3.40     0.3   0.20   2.60   1.00
4      16   6.05     2.6   0.55   2.45   0.10
5      19   9.10     1.7   0.00   5.45   1.75
6      31   8.75     0.0   0.00   5.90   3.40
> #Sumas lo largo de tiempos y replicaciones, aunque no tiene sentido:
> head(cast(papas_melt,subject+treatment~variable,sum,na.rm=TRUE))
  subject treatment potato buttery grassy rancid painty
1       3         1  111.9     6.7    3.4   37.9   56.0
2       3         2  121.3    10.6    1.9   56.5   44.6
3       3         3   95.3    13.8    1.7   51.4   51.6
4      10         1  199.1   135.0   11.7   80.4   27.5
5      10         2  199.9   139.6    9.5   43.0   16.4
6      10         3  200.6   129.0    2.9   62.2   13.8
> #Número de observaciones por individuo
> cast(papas_melt,~subject,length)
  value   3  10  15  16  19  31  51  52  63  78  79  86
1 (all) 270 300 300 300 300 270 300 300 300 300 270 270
```

Y podemos también reacomodar nuestra base sin agregar. Por ejemplo, para comparar las replicaciones podemos hacer:

```
> nuevos <- cast(papas_melt, subject + treatment + variable ~ rep)
```

o simplemente

```
> nuevos <- cast(papas_melt, . ~ rep)
> head(nuevos, 2)
  value    1    2
1 (all) 1740 1740
```

Nótese que en este último ejemplo no es necesario dar una función para agregar, pues todos los identificadores son usados.

Referencias

[1] P. Spector, Data Manipulation with R, Use R!, Springer, 2008.

[2] W.N. Venables, and D.M. Smith, An Introduction to R, Network Theory Limited, 2004.

[3] H. Wickham, Reshaping Data in R, Proceedings of the 4th International Workshop on directions in Statistical Computing, 2005.

[4] W.N. Venables, and B.D. Ripley, Modern Applied Statistics with S, 4th. ed, Springer, 2003