Lectura y escritura de archivos

Luis Alejandro C.

Marzo, 2020

Nota preliminar

Los datos se generan y se comparten en una diversidad de extensiones como `.txt`, `.csv` y `.xlsx` por mencionar algunos de las más conocidas. Asimismo, cada base de datos puede tener sus propias particularidades, como presentar renglones en blanco, utilizar separadores (tabuladores, coma, punto y coma), carecer de nombres de variables, etc.. Por esas razones, además de conocer las funciones para cargar datos en nuestro espacio de trabajo, resulta fundamental conocer la estructura de las bases que queremos manipular de forma tal que empleemos no sólo las funciones correctas sino también las opciones adecuadas.

En esta sección se abordará en primer lugar el uso de `read.table()`, `read.csv()` y `read.csv2()`, disponibles en `base R` y sus similares `read_table()`, `read_csv()` y `read_csv2()` de la librería `readr` así como `fread()` de `data.table`. En segundo lugar se describirá la función `read.xlsx()` de la librería `openxlsx` para leer archivos con la extensión homónima. En tercero se presentará la función `download.file()` que nos facilita entre otras cosas la descarga de archivos **zip** desde Internet. En cuarto se describirán brevemente algunas funciones para escribir archivos en extensiones `.txt`,`.csv` y `.xlsx`. En quinto, nos referiremos a las funciones `saveRDS()` y `readRDS()` para crear y leer archivos directamente desde R y finalmente se verá de manera rápida cómo importar datos a R empleando el argumento `file= "clipboard"`en funciones como `read.table()`.

1. Lectura de archivos de .txt, .csv y .tsv

Las bases de datos pueden encontrarse en formato de texto (`.txt`), separados por tabuladores (`.tsv`) y separados por comas o por puntos y comas (`.csv`). En **base R**, la función `read.table()` nos permite trabajar con **.txt** y **.tsv** mientras que `read.csv()` se emplea para archivos separados por comas y `read.csv2()` para archivos separados por puntos y comas.

Los argumentos clave y comunes entre estas funciones son:

Argumento	Descripción
file	Nombre del archivo a leer (entrecomillado y con la extensión correspondiente)
header	Lógico para indicar si el primer renglón contiene los nombres de las variables. En `read.table()`, `header=FALSE` es la opción por defecto mientras que para `read.csv()` y `read.csv2()`, `header=TRUE` es la opción por defecto
sep	Caracter empleado para separar los registros: coma(`,`), punto y coma (`;`), tabulador (`\t`), barra hacia adelante (`/`), etcétera. Los separadores van entre comillas
stringsAsFactors	`FALSE` si las cadenas de caracteres no serán usadas como factores. `stringsAsFactors=TRUE` por defecto
quote	Caracteres que denotarán citas en cadenas de texto (`""` deshabilita las citas, por ejemplo)
fill	Lógico para indicar si se generan espacios en blanco para renglones con longitud desigual. `fill=FALSE` por defecto en `read.table()` y `fill=TRUE` por defecto en `read.csv()` y `read.csv2()`
strip.white	`TRUE` para eliminar espacios en blanco en cadenas de caracteres sin comillas. `strip.white=FALSE` por defecto

Para `read.table()` destaca el argumento `na.strings` que consiste en un vector de caracteres que serán interpretados como valores omitidos (NA). Por default, los registros en blanco se consideran como NA aunque en ocasiones existen codificaciones específicas. Por ejemplo, en este pequeño archivo de extensión *.txt*, los `NA` se indican con la leyenda "EMPTY".

Por su parte, en `read.csv()` y `read.csv2()` se usa el argumento `dec` que se refiere al caracter con el cual se identifican los decimales. Para la primera función suele ser el punto `.` mientras que para la segunda la coma `,`. No debe olvidarse que el uso de cualquiera de esas dos funciones depende del símbolo que se usa como separador, por lo que se sugiere primero abrir el archivo en un programa como Notepad.

```
# Esquema básico para llamar a las funciones de lectura de archivos
# Para archivos "txt" y "tsv" la función empleada es read.table()
# Los ... al final se refieren a todos los argumentos aplicables
objeto <- read.csv("archivo.csv", header=TRUE, sep=",", ...)
```

Notas:

1. Cuando decimos que el valor de un argumento está "por defecto", quiere decir que no es necesario indicarlo cuando llamamos a una función.
2. Si conocemos la estructura de la base de datos y, sobre todo, de las variables categóricas (factores), podemos dejar `stringsAsFactors` con su valor por defecto (es decir, `TRUE`).

3. Para consultar todos los argumentos aplicables a las funciones antes vistas, se sugiere acceder a la documentación

Ejercicio: Cargar el archivo "iris.txt" desde nuestro entorno de trabajo y verificar su estructura. ¿Qué argumentos deben incluirse en `read.table()` para que la base de datos pueda ser visualizada correctamente?

```
iris <- read.table("iris.txt")
str(iris)
```

Aunque las anteriores funciones son de uso común, pueden ocupar mucha memoria si se intenta leer archivos de gran tamaño. Por ese motivo, las funciones `read_csv()`, `read_csv2()` y `read_tsv()` de la librería `readr` y `fread()` de `data.table` han cobrado relevancia debido a su sencillez (incorporación de pocos argumentos) y rapidez. Por ejemplo, en el caso de las primeras tres, no es necesario indicar el tipo de separador debido a que se halla implícito en el nombre de la función.

Por ejemplo, la base de datos `iris.txt` puede ser leída por `read_csv()` señalando únicamente la ruta y nombre del archivo:

```
irisrr <- readr::read_csv("iris.txt")
str(irisrr)
```

Pregunta: ¿Por qué utilizamos la notación `readr::read_csv()` y no directamente `read_csv()`?

Además de permitir la lectura local (archivos alojados en la computadora), las funciones soportan la descarga desde sitios web. La Internet Movie Database cuenta con bases de datos que se actualizan diariamente y su descripción puede ser consultada aquí. Como se aprecia, son bases de datos únicas comprimidas (es decir, sólo una base por archivo) en extensión **.gz** y con formato **.tsv**.

Como ejercicio ilustrativo, realizaremos la descarga de la base `title.ratings.tsv.gz` empleando las funciones `read_tsv` y `fread`. El siguiente fragmento de código contiene la instrucción inicial para la descarga e ingresaremos los argumentos considerando la descripción de las bases de datos y la documentación de las funciones que se encuentra aquí y aquí.

Pista: `read_tsv()` requiere dos argumentos y `fread()` tres

```
# Código base
rating_rr <- readr::read_tsv(
  "https://datasets.imdbws.com/title.ratings.tsv.gz")

rating_dt <- data.table::fread(
  "https://datasets.imdbws.com/title.ratings.tsv.gz")
```

2. Lectura de archivos .xlsx

La extensión `.xlsx` comenzó a utilizarse en sustitución de la `xls` a partir del lanzamiento de la suite Microsoft Office 2007. Los archivos con esta extensión tienen compresión zip y se basan en el estándar XML. Debido a esas características, los `.xlsx` ocupan menos espacio en disco y requieren menor tiempo para ser leídos o creados.

En **R** existen múltiples librerías como `XLConnect`, `xlsx` y `gdata`, entre otras, que permiten la lectura y escritura de archivos `.xlsx`. Sin embargo, tienen dependencias externas lo que puede dificultar su manejo para muchos usuarios (por ejemplo, las primeras dos hacen uso de Java y la tercera de PERL). Por ese motivo, se recomienda `openxlsx` que se basa en la librería `Rcpp`.

Para importar archivos desde esta librería, la función principal es `read.xlxs()` cuya documentación se encuentra aquí. El siguiente bloque de código contiene los argumentos que se consideran más importantes para llamar a esta función:

```
objeto <- read.xlsx("archivo.xlsx",

        # Nombre o número de hoja
        # Si no se indica, se leerá la primera hoja
        sheet = ,

        # Fila en la que se inicia la lectura
        # Si no se indica, se lee a partir de la primera fila
        startRow = ,

        # ¿Mantener nombres de columnas?
        # TRUE por default
        colNames = TRUE,

        # ¿Mantener nombres de renglones?
        # FALSE por default
        rowNames = FALSE,

        # ¿Saltarse renglones y columnas vacías?
        # TRUE por default en ambos casos
        skipEmptyRows = TRUE,
        skipEmptyCols = TRUE,

        # ¿Qué cadena de caracteres se utilizará para identificar NA?
        na.strings = "NA")
```

3. Importación de archivos con download.file()

La función `download.file()` facilita la descarga de archivos que se encuentran alojados en sitios web. Tiene dos argumentos principales: la dirección web y el nombre del archivo de destino (incluyendo su ruta). Esta función es particularmente útil si se usa junto con `unzip()` pues nos permite establecer un flujo estándar para la descarga y extracción de archivos **zip**.

Veamos un ejemplo extrayendo microdatos de la Ciudad de México correspondientes a la Encuesta Intercensal 2015. En primer lugar, ingresaremos al sitio web. Posteriormente, nos situamos en la pestaña "Microdatos" y recorremos hacia abajo para buscar "Distrito Federal". En el ícono correspondiente a `CSV` damos clic derecho y seleccionamos la opción **"Copiar ruta del enlace"**. A continuación, abrimos

un nuevo script y copiamos el siguiente código y sustituimos `"https://URL"` por la ruta obtenida en el paso anterior. **No debe olvidarse que la ruta va entrecomillada**. Finalmente, ejecutamos el contenido del script.

```
# Primer argumento: Sitio web de la descarga
# Segundo argumento: Nombre del archivo zip
# Nota: Los archivos se guardarán en el directorio de trabajo
cdmx <- download.file("https://URL","cdmx")
# Descomprimimos el archivo con unzip()
cdmx_zip <- unzip("cdmx")
# Podemos remover el archivo zip para limpiar el directorio de trabajo
file.remove("cdmx")
# ¿Cuál es la estructura de nuestro archivo descomprimido?
str(cdmx_zip)
```

Si el código funcionó correctamente, observaremos esta salida en la consola:

```
chr [1:2] "./TR_PERSONA09.CSV" "./TR_VIVIENDA09.CSV"
```

Esto significa que el zip contiene dos archivos con extensión `.csv` los cuales podemos llevar a nuestro entorno de trabajo empleando alguna de las funciones antes vistas. Si quisiéramos descargar múltiples bases de datos similares (por ejemplo, para comparar estados), podemos recurrir al uso de iteraciones con la librería `foreach`. Una explicación sobre este procedimiento se encuentra aquí.

4. Escritura de archivos

Después de realizar manipulaciones en una base de datos, usualmente deseamos guardar los archivos en formatos útiles para otras personas. Con **base R** tenemos las funciones con raíz `write.` para escribir en formatos `.txt`, `.csv` y `.csv2`. Estas funciones toman dos argumentos básicos: el nombre del objeto así como el nombre y la extensión del archivo que será alojado en nuestro equipo:

```
# Para generar archivos de texto (.txt)
write.table(archivo, "nombre.txt")

# Para archivos separados con coma (.csv)
# Para archivos separados por punto y coma, usar write.csv2
write.csv(archivo, "nombre.csv")
```

Las librerías `readr` y `data.table` también ofrecen opciones para escribir archivos con extensión `.txt` y `.csv`, las cuales son más rápidas que sus equivalentes en **base R**, por lo cual se recomiendan cuando los archivos tienen un gran tamaño.

```
# Con readr
readr::write_csv(archivo, "nombre.txt")
readr::write_csv(archivo, "nombre.csv")

# Con data.table
```

```
data.table::fwrite(archivo, "nombre.txt")
data.table::fwrite(archivo, "nombre.csv")
```

Importante: A diferencia de las funciones base R, las de las librerías `readr` y `data.table` no conservan los nombres de los renglones (row.names) cuando se emplean para exportar bases de datos. Si desean mantenerse, se sugiere asignarlos a una variable específica.

Para guardar archivos con extensión `.xlsx`, empleamos la librería `openxlsx` y su función `write.xlsx`. Los argumentos básicos de esta función son los mismos que las anteriores aunque se puede especificar un nombre para la hoja en la que se guardará el archivo. Si se desea que la hoja creada tenga un formato tipo tabla, se incorpora el argumento `asTable=TRUE`.

```
openxlsx::write.xlsx(archivo, "nombre.xlsx", sheetName="nombre_hoja",
                     asTable=TRUE)
```

Esta función nos permite además crear múltiples hojas individuales utilizando una lista de nombres. Tras ejecutar el siguiente código, revisemos

```
lista_dbs <- list("iris_db" = iris, "cars_db" = mtcars,
                  "air_db" = airquality)

write.xlsx(lista_dbs, "dbs.xlsx")
```

5. Utilizando formatos nativos: saveRDS() y readRDS()

`saveRDS()` y `readRDS()` son funciones de **R** que permiten guardar y leer objetos con extensión `.rds` propia del lenguaje. Supongamos que creamos un objeto cualquiera (una base de datos, un modelo estadístico, una función, etc.) llamado `cualquier_cosa`. Cuando usamos `saveRDS()`, el primer argumento es el objeto a guardar y el segundo el nombre que le daremos:

```
saveRDS(cualquier_cosa, "cualquier_cosa_uno.rds")
```

Para llevar de vuelta el objeto a nuestro entorno de trabajo, realizamos otra asignación con un nuevo nombre (por ejemplo `cualquier_cosa_dos`):

```
cualquier_cosa_dos <- readRDS("cualquier_cosa_uno.rds")
```

El hecho de que tengamos que establecer un nuevo nombre para nuestro objeto original (de `cualquier_cosa` a `cualquier_cosa_dos`) implica que con `saveRDS()` **R** sólo guarda una "representación" del mismo, es decir, de su contenido y no de su nombre. Con ello, la sobreescritura no resulta problemática y evitamos conflictos que pudieran darse en caso de que, por ejemplo, lleguemos a tener nuevamente un objeto llamado `cualquier_cosa`. Por este motivo, recurrir a `saveRDS()` y `readRDS()` se considera una mejor práctica que emplear `save()` y `load()`.

Para hacer más claras estas ideas, recurramos a un ejemplo. En primer lugar, veremos qué ocurre con `save()` y `load()`:

```r
# No es aconsejable pero es útil para este ejemplo:
rm(list=ls())

# Creamos un objeto simple
prueba <- mtcars[c(1:5),c(1:2)]
# Guardamos el objeto con el nombre "prueba_uno"
save(prueba, file="prueba_uno.rda")
# Cargamos y asignamos el nombre prueba_dos
prueba_dos <- load(file="prueba_uno.rda")
# Verificamos cuáles son los objetos en nuestro entorno de trabajo
ls()
```

```
## [1] "prueba"     "prueba_dos"
```

```r
# ¿Se trata de objetos iguales?
identical(prueba, prueba_dos, ignore.environment = TRUE)
```

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

Como se aprecia, `prueba` y `prueba_dos` son diferentes. Verifiquemos la estructura de cada objeto con `str()` para identificar la fuente de la disimilitud:

```r
str(prueba)
```

```
## 'data.frame':    5 obs. of  2 variables:
##  $ mpg: num  21 21 22.8 21.4 18.7
##  $ cyl: num  6 6 4 6 8
```

```r
str(prueba_dos)
```

```
##  chr "prueba"
```

Podemos concluir entonces que las asignaciones con `load()` tienen como resultado la creación de una cadena de caracteres y no la copia de un objeto. Por tanto, si nombramos `prueba` a otro objeto, perderemos de forma definitiva el que creamos en primera instancia. Con `saveRDS()` y `readRDS()`esto no ocurre:

```r
# Limpiamos nuevamente el entorno de trabajo
rm(list=ls())
# Creamos el mismo objeto y lo guardamos como "test_uno"
test <- mtcars[c(1:5),c(1:2)]
saveRDS(test, "test_uno.rds")
# Cargamos y asignamos el nombre "test_dos"
test_dos <- readRDS("test_uno.rds")
# Verificamos cuáles son los objetos en nuestro entorno de trabajo
ls()
```

```
## [1] "test"     "test_dos"
```

```
# ¿Se trata de objetos iguales?
identical(test, test_dos, ignore.environment = TRUE)
```

```
## [1] TRUE
```

En resumen, cuando usamos `load()`, el nombre del archivo que será llevado al entorno de trabajo es el del objeto guardado en primera instancia, y no el de extensión `.rda`. Por el contrario, con `readRDS()` creamos una copia del archivo con extensión `.rds` y no necesitamos recordar el nombre del objeto original.

El uso de `save()` y `load()` suele sustentarse en la ventaja de que permiten trabajar con múltiples objetos. Sin embargo, con `saveRDS()` y `readRDS()` basta alojar dichos objetos en una lista y usar el operador de indexación `[[ ]]` después de llevar el objeto a nuestro entorno de trabajo:

```
# Procedimiento con save() y load()
# Al final, sólo tendremos dos objetos en el entorno de trabajo
base_uno <- mtcars[c(1:5),c(1:2)]
base_dos <- airquality[c(1:5),c(1:2)]
save(base_uno, base_dos, file="bases.rda")
load(file="bases.rda")
ls()

# Limpiamos el espacio de trabajo
rm(list=ls())

# Procedimiento con saveRDS() y readRDS()
base_uno <- mtcars[c(1:5),c(1:2)]
base_dos <-  airquality[c(1:5),c(1:2)]
saveRDS(list(base_uno, base_dos), file="bases.rds")
basesrds <- readRDS("bases.rds")
ls()
# Tenemos 3 objetos. Usamos str() para ver qué hay en basesrds
str(basesrds)
# Para acceder a los objetos, usamos [[ ]] como vimos en el módulo II
basesrds[[1]]
basesrds[[2]]
```

El almacenamiento de archivos en formato nativo permite también un importante de ahorro de espacio en disco. Del ejemplo de la sección 4 quedó guardado en nuestro directorio de trabajo el archivo `TR_PERSONA09.CSV`. Lo traeremos a nuestro entorno actual y lo guardaremos utilizando la función `saveRDS()`:

```
# Asignamos primero un nombre para que quede como objeto
vivrds <- readr::read_csv("TR_VIVIENDA09.csv")
# Guardamos el objeto en formato nativo
saveRDS(vivrds, "viv_rds.rds")
```

Ahora vamos a comparar los tamaños de cada archivo:

```
# Obtenemos la información con file.info()
info_file <- as.data.frame(file.info(c("TR_VIVIENDA09.csv", "viv_rds.rds")))
# Guardamos la columna correspondiente al tamaño
info_file <- info_file[,1, drop=F]
# ¿Cuántas veces es más grande el .csv respecto al .rds?
info_file[1,]/info_file[2,]
# ¿Cuál es el resultado?
```

Para un análisis a mayor profundidad sobre `save()`, `load()`, `saveRDS()` y `readRDS()` se sugiere revisar la documentación aquí, aquí y aquí así como el capítulo número 5 del libro "Efficient R Programming".

6. Nota final: file= "clipboard"

Una forma menos conocida de importar archivos desde hojas de cálculo, bloc de notas o páginas de internet es `read.table()` o `read.csv()` con el argumento `file= "clipboard"`. Esto nos permite "copiar y pegar" contenidos de forma directa en R. El procedimiento es sencillo:

1. Seleccionamos los datos y hacemos `Ctrl` + `C`
2. Realizamos una asignación en R de la forma `objeto <- read.table(file = "clipboard", ...)` donde **...** se refiere a los argumentos aplicables. (También se puede utilizar la notación compacta vista en el segundo módulo)
3. Ejecutamos el código con `Run` o `Ctrl` + `R` y verificamos que la información haya sido copiada correctamente.

Ejercicio: Siguiendo el procedimiento descrito, copiar la base de datos `mtcars` que se encuentra en este sitio y llevarla a R bajo el nombre `mtcarsgh` haciendo uso de `read.table()`.

- ¿Cuál es el tipo de separador que debe aplicarse?
- ¿El argumento `header` es `TRUE` o `FALSE`?
- ¿Cuál es la estructura de la base de datos?

Al usar `file="clipboard"` debemos ser cuidadosos porque existe un límite por defecto de 32 kilobytes para el texto que puede ser exportado a **R**. Sin embargo, este puede ser modificado desde el argumento. Por ejemplo, `file=clipboard-128` expande el límite a 128 kilobytes, `file=clipboard-1024` a 1024 kilobytes, etc.. Para archivos de gran tamaño lo recomendable es valerse de las funciones descritas en este módulo.