UNIVERSIDAD ICESI

Departamento de Economía

Facultad de Ciencias Administrativas y Económicas

Economics Lecture Notes

Una introducción a los Loops en R (y algunas alternativas)

Julio Cesar Alonso Cifuentes

Icesi ECONOMICS LN No.14
Febrero 2021

Una introducción a los Loops en R (y algunas alternativas)

Julio Cesar Alonso Cifuentes

Icesi
ECONOMICS LN
No.14
Febrero 2021

Universidad Icesi

Tel: 5552334. Fax: 5551441
Calle 18 # 122-135 Cali, Valle del Cauca, Colombia

Una introducción a los Loops en R (y algunas alternativas)

Julio César Alonso
jcalonso@icesi.edu.co
Cienfi - Departamento de Economía
Universidad Icesi
Cali - Colombia

2 de marzo de 2021

1. Objetivos de Aprendizaje

Al finalizar la lectura de este documento el estudiante estará en capacidad de:

- Explicar en sus propias palabras qué es un loop
- Construir un loop sencillo en R
- Reconocer otras formas de hacer la misma tarea de un loop con funciones en R.

Este documento asume que el lector está familiarizado a un nivel introductorio con R.

2. Introducción

En algunas ocasiones se desea repetir una operación o una linea de código muchas veces. De hecho, una de las tareas que hacemos con frecuencia en la analítica o en investigación es aplicar una función a un gran número de variables (columnas) de un *dataframe*. En esos casos emplear un loop (algunas veces se traduce como bucle en español), ciclo, iterar o simplemente replicar una linea de código es la solución.

Por ejemplo, se puede querer crear gráficos de cada una de las columnas de un *dataframe*, o realizar un cálculo y guardarlo en una fila diferente de un objeto. En vez de copiar, pegar y modificar el código, podemos emplear un "loop"que repita la operación.

Si bien los loops pueden ser poco eficientes, y encontrarás en la Web que es deseable evitar usar los loops en R(R Core Team, 2019) , es importante tener en la “caja de herramientas” el recurso del loop. Es cierto que no es eficiente emplear loops en R, pues este lenguaje de programación soporta la vectorización lo cual implica

la posibilidad de cálculos mucho más rápidos empleando funciones que aprovechan esta característica. Sin embargo, al iniciar en R, es bueno tener una comprensión básica de los loops y de cómo escribirlos. Esto ayudará a entender lo que hay detrás de las tareas de vectorización que con seguridad emplearás cuando seas un usuario mas avanzado.

En otras palabras, es importante conocer los loops al inicio del proceso de aprendizaje, pues éstos reflejan de una buena forma la manera como pensamos en un problema y su solución. No obstante, como no hay almuerzo gratis, la simplicidad de los loops tienen como consecuencia, a veces, que los scripts de R se vuelven muy largo y tardan horas en ejecutarse.

En este documento discutiremos cómo hacer tareas repetitivas de varias maneras. Primero veremos cómo realizar estas tareas repetitivas con loops. Después con funciones que permiten hacer esto de manera más eficiente sacándole provecho a la vectorización, pero de pronto será menos intuitiva la operación.

Este tutorial está dirigido a personas que estén iniciando su camino en R. Si aún no estás familiarizado con la sintaxis básica del lenguaje R, es recomendable realizar un tutorial de los muchos disponible en linea. Por ejemplo en los siguientes links puedes encontrar dos videos que he preparado como introducción a R:

Introducción a R (parte 1): https://youtu.be/GNZr-Kabnb8

Introducción a R (parte 2): https://youtu.be/zQLtJxGt8OQ

3. Los loops

Para entender un poco la naturaleza de este tipo de operaciones repetitivas, consideremos un ejemplo muy sencillo. Supongamos que se desea crear el nombre de 100 variables que se llamen “variable_1”, “variable_2” y así sucesivamente hasta “variable_100”. Y esos nombres los queremos guardar en un vector que contenga todos los nombres.

Esto se puede hacer de manera sencilla digitando 100 veces el nombre correspondiente. Pero esto es tedioso, puede generar errores y no parece algo elegante para un usuario de R. Otra opción es emplear R para que repita 100 veces la operación de pegar los caracteres “variable” y un número que va desde 1 hasta 100, aumentando de uno en uno. Es decir, si definimos i como un número que va de 1 a 100 e iniciamos con el número 1, le pediremos a R que pegue “variable_ ” y el número 1 y lo guarde en la primera posición de un objeto que guardará los nombres (*nombres*). Ahora repitamos esta tarea para el siguiente número (i=2) y así sucesivamente hasta que lleguemos a i=100. ¡Esto es un loop!

Esto se pude realizar de manera muy sencilla en R con la función **for()** . Veamos el código que realiza la tarea descrita arriba antes de ver los detalles de esta función.

```
# se crea el objeto nombres en el que se guardarán los
# nombres
nombres <- NULL

# de inicia el loop
for (i in 1:100) {
    # se crea el nombre de la variable y se guarda en la
    # correspondiente posición
    nombres[i] <- paste("variable_", i)
```

```
}
# se examinan las primeras cinco entradas del objeto nombres
head(nombres, 5)

## [1] "variable_ 1" "variable_ 2" "variable_ 3" "variable_ 4"
## [5] "variable_ 5"

# se examinan las últimas cinco entradas del objeto nombres
tail(nombres, 5)

## [1] "variable_ 96"  "variable_ 97"  "variable_ 98"
## [4] "variable_ 99"  "variable_ 100"
```

La primera linea crea un objeto nulo (*nombres*) en el que guardaremos los nombres de las variables que se creen. La segunda linea le dice a R que emplee la función **for** y defina un objeto *i* que irá de 1 a 100 (lo que se encuentra entre paréntesis). Luego encontramos la linea 3 que es precedida por un corchete abriendo en la linea 2 y uno cerrando en la linea 4. Todo lo que se encuentre entre los corchetes se repetirá las veces que la condición especifique (es decir lo que encontremos entre los paréntesis que siguen al **for**). La linea 3 le indica a R que para cada posible valor de *i*, que debe asignar en la posición *i*-ésima el resultado de pegar (paste) los caracteres "variable_" y el valor de *i* para la respectiva iteración. Una vez termina una iteración, R suma una unidad a *i* y ejecuta esta instrucción de nuevo. El proceso se repite hasta llegar a $i = 100$.

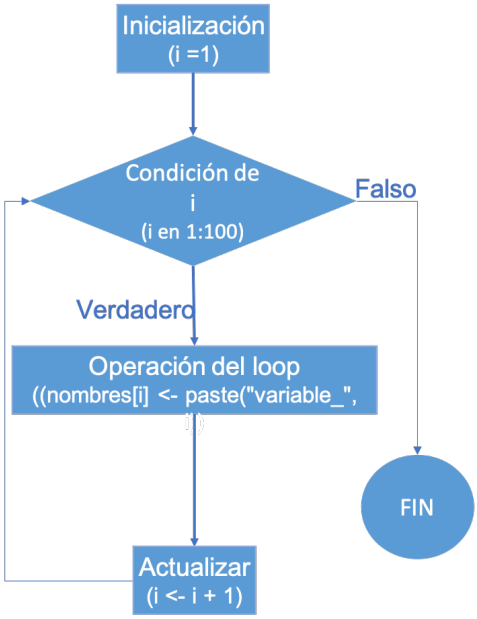
En otras palabras, estas líneas de código se pueden leer de la siguiente manera: "Para (for) cada *i* que esté en la secuencia de 1 a 100 (1:100) se ejecuta el código que se encuentra entre corchetes". Una vez que el loop ha ejecutado el código para cada *i* de la secuencia establecida, el loop se detiene y se ejecuta la siguiente linea que se encuentra por fuera del loop.

Al emplear la función utilizar **for()** solo se tiene dos argumentos importantes. La condición que se escribe entre paréntesis (en este caso que *i* está entre 1 y 100) y la expresión que se escribe entre corchetes. La expresión es el código que se repetirá.

Conceptualmente, un loop es una forma de repetir una secuencia de instrucciones bajo ciertas condiciones. Esto se puede expresar gráficamente como se ve en la Figura 1. El bucle arranca con una inicialización de la variable que se empleará como contador (caja rectangular), antes de continuar se evalúa si la condición se cumple (rombo). Si la condición no se cumple, el proceso termina. Si la condición se cumple, se realiza la operación que se desea replicar (caja rectangular). Después de efectuada la operación, el loop actualiza el contador[1] y se regresa a evaluar si la condición se cumple.

[1] También podría ser reducir el contador.

Figura 1: Esquema de un loop (ejemplo de R entre paréntesis)



Ahora veamos otros ejemplos de loops que emplean este principio básico

3.1. Loops con condiciones no numéricas

En R es posible construir loops que iteran sobre un grupo de objetos y no sobre una secuencia numérica. Por ejemplo, consideremos el siguiente código:

```
# se cree un objeto con las condiciones que corresponderán a
# nombres
n1 <- c("Juan", "Ana", "Pedro", "Javier", "Paula")

for (i in n1) {

    print(paste("El nombre", i, "tiene", nchar(i), "letras."))
}

## [1] "El nombre Juan tiene 4 letras."
## [1] "El nombre Ana tiene 3 letras."
## [1] "El nombre Pedro tiene 5 letras."
## [1] "El nombre Javier tiene 6 letras."
## [1] "El nombre Paula tiene 5 letras."
```

En este caso el loop ejecuta el cálculo del número de caracteres (**nchar()**) en cada uno de los elementos que hacen pare del objeto `n1`. No solamente se calcula el número de letras, sino que este se pega con otras palabras para generar el resultado deseado de imprimir en la consola tanto el nombre como el número de letras.

3.2. Loops anidados

También podemos crear un loop al interior de otro. Por ejemplo, supongamos que queremos crear una matriz de 40×30 cuyas entradas sean iguales a la suma de la respectiva columna y fila. Esto se puede hacer con el siguiente código

```
# se crea una matriz vacía de 40 x 30
m <- matrix(nrow = 40, ncol = 30)
# primer loop que itera en las filas
for (i in 1:dim(m)[1]) {
    # segundo loop que itera en las columnas
    for (j in 1:dim(m)[2]) {

        # se guarda en la correspondiente posición la suma de la
        # columna y la fila
        m[i, j] = i + j
    }
}

# se muestran las primeras 5 filas y columnas de la matriz

m[1:5, 1:5]

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

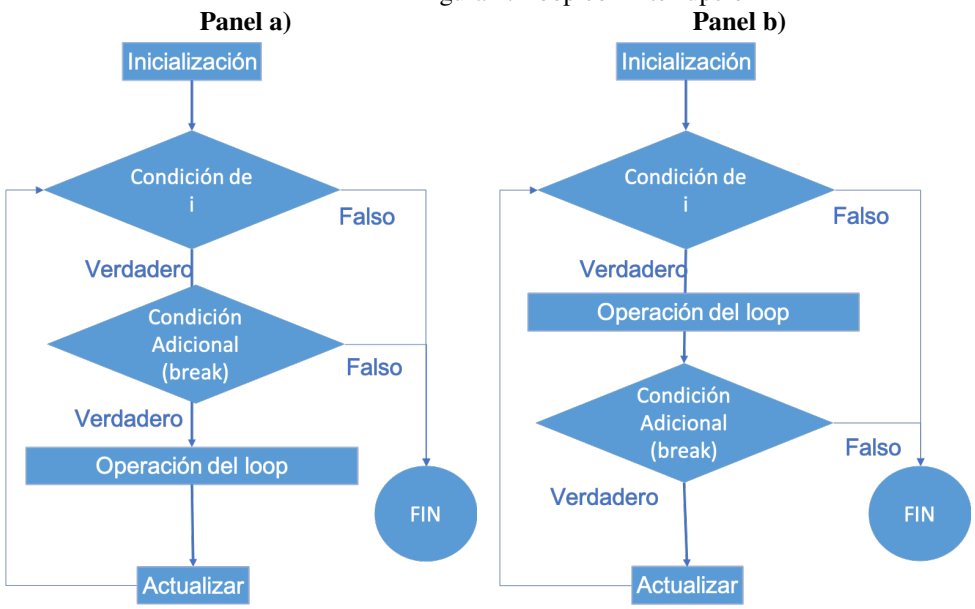
3.3. Loops con opciones de interrupción

En algunas ocasiones podemos necesitar que un loop pare su ejecución si se cumple una condición que no necesariamente esté relacionado con el contador. Continuando con la matriz construida en el ejemplo anterior, supongamos que se quisiera encontrar la suma de cada elemento de la fila hasta que la suma no supere 1000.

```
# se crea el objeto sumas en el que se guardarán la suma de
# todos los elementos de la fila
suma <- NULL

for (i in 1:dim(m)[1]) {
    # condición para interrumpir
    if (sum(m[i, ]) > 1000) {
        break
    }
    suma[i] <- sum(m[i, ])
}

suma

##  [1] 495 525 555 585 615 645 675 705 735 765 795 825 855 885 915
## [16] 945 975
```

La opción de interrupción se puede incluir en el loop por medio de un condicional que ejecute la condición **break**. Noten que la opción de parar se puede ejecutar antes de la operación de la iteración o después. En la Figura 2 se presentan dos esquemas de la forma como se puede interrumpir el loop, dependiendo de lo deseado.

Figura 2: Loop con interrupción



3.4. Loops con opciones de saltar una iteración

En otras ocasiones quisiéramos hacer que el lopp salte algunas iteraciones. Es decir, a diferencia de lo discutido en la subsection anterior, podremos necesitar que el loop no pare si se cumple una condición sino que simplemente omita una iteración. Eso se puede realizar con un condicional que ejecute la condición **next** para seguir a la siguiente iteración. Por ejemplo, supongamos que aún queremos sumar cada una de las filas de la matriz (como en la subsección anterior), pero ahora solo queremos tener en cuenta aquellas sumas que sean múltiplos de 25.

Esto se puede realizar con el siguiente código:

```
# se crea el objeto suma.mu25 en el que se guardará la suma
# de todos los elementos de la fila y que es múltiplo de 25
suma.mul25 <- NULL
# se crea el objeto fila.suma.mu25 en el que se guardará el
# número de fila cuya suma de todos los elementos de la fila
# es múltiplo de 25

fila.suma.mul25 <- NULL


# install.packages('schoolmath')
library(schoolmath)
# paquete que permite emplear la función is.decimal()

for (i in 1:dim(m)[1]) {
    # condición para saltar a siguiente iteración
```

```
    if (is.decimal(sum(m[i, ])/25)) {
        next
    }
    fila.suma.mul25[i] <- i
    suma.mul25[i] <- sum(m[i, ])

}
# se crea un data.frame con los resultados
resultado <- data.frame(fila.suma.mul25, suma.mul25)
# se muestran los resultados
resultado
```

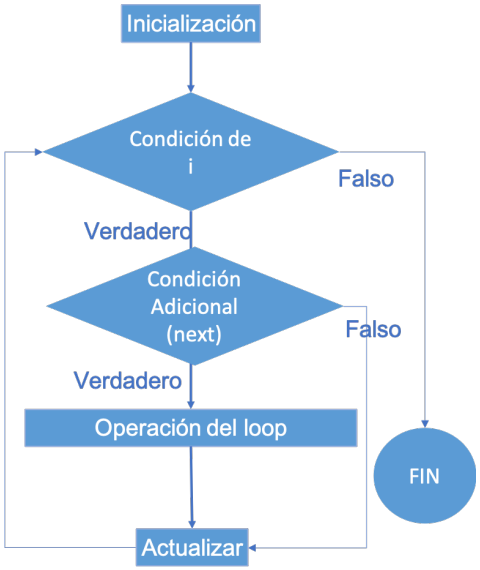
```
##    fila.suma.mul25 suma.mul25
## 1               NA         NA
## 2                2        525
## 3               NA         NA
## 4               NA         NA
## 5               NA         NA
## 6               NA         NA
## 7                7        675
## 8               NA         NA
## 9               NA         NA
## 10              NA         NA
## 11              NA         NA
## 12              12        825
## 13              NA         NA
## 14              NA         NA
## 15              NA         NA
## 16              NA         NA
## 17              17        975
## 18              NA         NA
## 19              NA         NA
## 20              NA         NA
## 21              NA         NA
## 22              22       1125
## 23              NA         NA
## 24              NA         NA
## 25              NA         NA
## 26              NA         NA
## 27              27       1275
## 28              NA         NA
## 29              NA         NA
## 30              NA         NA
## 31              NA         NA
## 32              32       1425
## 33              NA         NA
## 34              NA         NA
## 35              NA         NA
## 36              NA         NA
## 37              37       1575
```

```
# se omiten los NA
```

```r
na.omit(resultado)

##    fila.suma.mul25 suma.mul25
## 2                2        525
## 7                7        675
## 12              12        825
## 17              17        975
## 22              22       1125
## 27              27       1275
## 32              32       1425
## 37              37       1575
```

En la Figura 3 se presentan dos esquemas de la forma como se puede saltar la interacción del loop.

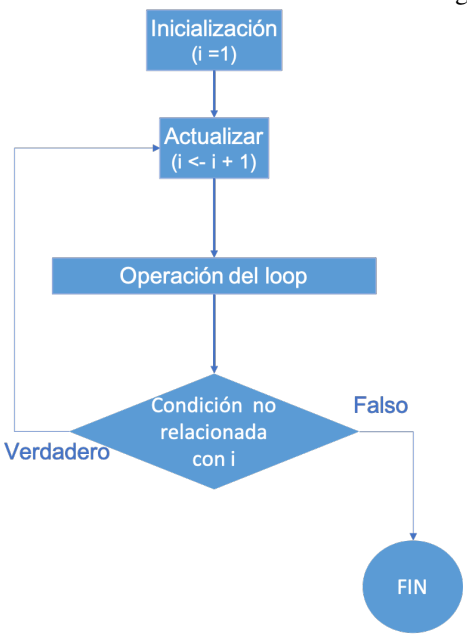
Figura 3: Loop con saltos de iteración



3.5. Loops con repeat

En algunas ocasiones puede ser necesario realizar loops para los cuales no se conoce cuántas veces se tendrá que iterar pues la condición para parar no depende del contador de iteración sino de una condición que puede estar no asociada al contador (ver Figura 4).

Figura 4: Loop con repeat()



Veamos un ejemplo con la matriz `m` que creamos anteriormente. Supongamos que queremos calcular la suma de las filas hasta que la raíz cuadrada de esta suma sea inferior a 30. Esto se puede hacer de la siguiente manera.

```
# se crea el objeto suma2 en el que se guardará la suma de
# todos los elementos de la fila
suma2 <- NULL

i <- 0
repeat {
    # en este caso el contador (si se necesita se debe
    # actualizar)
    i <- i + 1
    suma2[i] <- sum(m[i, ])
    # condición para salir del loop
    if (sqrt(sum(m[i, ])) > 30) {
        print("Fin del Loop; condición alcanzada")
        break
    }

}

## [1] "Fin del Loop; condición alcanzada"

i

## [1] 15

# se examinan las primeras cinco sumas
head(suma2, 5)

## [1] 495 525 555 585 615
```

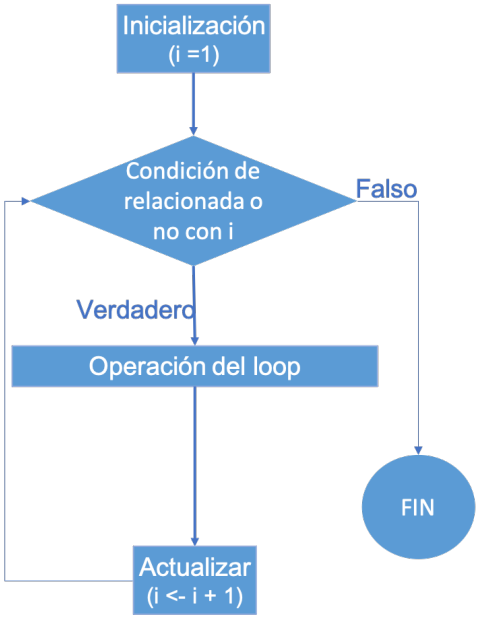
```r
# se examinan las últimas cinco sumas
tail(suma, 5)

## [1] 855 885 915 945 975
```

Noten que esto es similar, pero no exactamente igual a un loop con break, la diferencia está en que implícitamente no existe una condición asociada al contador como si existe en el loop con break.

3.6. Loops con while

Una alternativa similar a la anterior es realizar loops que repiten las iteraciones mientras (while) se cumpla una condición. Es decir, repeat genera iteraciones hasta que una condición se cumpla y while genera iteraciones mientras una condición se cumpla. Estas dos formas de hacer loops pueden ser similares y dependiendo del caso, una puede ser mas conveniente que otra. En la Figura 5 se presenta un esquema de este tipo de loops.

Figura 5: Loop con while()



Veamos el ejemplo de la sección anterior empleando esta función.

```r
# se crea el objeto suma3 en el que se guardará la suma de
# todos los elementos de la fila
suma3 <- NULL

i <- 0
while (sqrt(sum(m[i, ])) < 30) {
    # en este caso el contador (si se necesita se debe
    # actualizar)
    i <- i + 1
    suma3[i] <- sum(m[i, ])
}
```

```
i

## [1] 15

# se examinan las primeras cinco sumas
head(suma3, 5)

## [1] 495 525 555 585 615

# se examinan las últimas cinco sumas
tail(suma3, 5)

## [1] 795 825 855 885 915
```

4. Alternativas a los loops

Como se mencionó en la introducción, R es un lenguaje de programación que emplea vectorización. Es decir, que convierte operaciones repetidas con números simples (escalares) en operaciones simples sobre vectores o matrices. En otras palabras, la forma natural de hacer cálculos en R es emplear vectores y matrices y no escalares.

Esto hace que de manera nativa cuando aplicamos una función a un vector o array, R evalúa la función de manera iterativa para cada elemento. Recordemos el ejemplo anterior en el que con un loop calculábamos el número de letras en 5 nombres que estaban en un array de nombres (Ver sección 3.1). Esto se puede hacer de manera mas eficiente empleando una función sencilla que explota la vectorización de R.

En el siguiente código construimos la función **cutree(num.letras)** que efectúa lo mismo que hacíamos en el loop en cada iteración (asegúrate que revisas el código del loop y entiendes como se construye la función)

```
num.letras <- function(i) {
    res <- (paste("El nombre", i, "tiene", nchar(i), "letras."))
    return(res)
}
```

Ahora, en vez de emplear un loop para ejecutar esta función para cada uno de los elementos del objeto `n1` (Ver sección 3.1), podemos aplicar la función al objeto `n1`.

```
n1

## [1] "Juan"   "Ana"    "Pedro"  "Javier" "Paula"

num.letras(n1)

## [1] "El nombre Juan tiene 4 letras."
## [2] "El nombre Ana tiene 3 letras."
## [3] "El nombre Pedro tiene 5 letras."
## [4] "El nombre Javier tiene 6 letras."
## [5] "El nombre Paula tiene 5 letras."
```

Noten que el resultado es el mismo que hacer un loop. No obstante si esta operación la estuviésemos haciendo con un objeto con miles de nombres, la función se ejecutará mucho más rápido que el loop.

Por otro lado, ya existen funciones en R que permiten aprovechar esta arquitectura de R vectorial que permite repetir la aplicación de funciones (cálculos u operaciones) de manera mucho más rápida que empleando un loop. A continuación veremos algunas de estas funciones.

La función **apply()** de la base de R genera un vector o lista de resultados fruto de aplicar una función a las filas o columnas de una matriz. La estructura de esta función es la siguiente:

apply(X, MARGIN, FUN, ...)

donde:

- **X**: es un objeto de clase matriz o *array*.
- **MARGIN**: este argumento permite determinar si la función se repetirá para cada columna (`MARGIN = 2`) o cada fila (`MARGIN = 1`).
- **FUN**: la función que se iterará. Por ejemplo, si queremos calcular la media (**cutree()**) entonces `FUN = mean`.

Veamos un ejemplo sencillo. Supongamos que tenemos un data.frame con 100 observaciones y 20 variables. Construyamos este data.frame de manera aleatoria (empleando una distribución estándar normal).

```
set.seed(1234)
data <- matrix(rnorm(n = 100 * 20), nrow = 100, ncol = 20)
data <- data.frame(data)
str(data)

## 'data.frame': 100 obs. of  20 variables:
##  $ X1 : num  -1.207 0.277 1.084 -2.346 0.429 ...
##  $ X2 : num  0.415 -0.475 0.066 -0.502 -0.826 ...
##  $ X3 : num  0.485 0.697 0.186 0.701 0.312 ...
##  $ X4 : num  -0.58 -0.9533 -0.1794 1.0098 0.0236 ...
##  $ X5 : num  -1.2268 0.0362 -0.4214 -0.8994 0.4174 ...
##  $ X6 : num  0.985 -1.225 0.71 -0.109 1.783 ...
##  $ X7 : num  0.3135 0.6125 -1.6911 0.7847 0.0119 ...
##  $ X8 : num  0.455 1.425 -0.209 -1.231 -0.361 ...
##  $ X9 : num  -1.0239 -1.3877 -0.0492 1.811 -0.0995 ...
##  $ X10: num  -2.874 -0.427 -1.487 0.589 -0.443 ...
##  $ X11: num  -1.205 0.301 -1.539 0.635 0.703 ...
##  $ X12: num  0.1133 -0.5651 0.0807 -0.6958 -1.0962 ...
##  $ X13: num  -1.402 2.384 0.873 -1.536 1.134 ...
##  $ X14: num  1.051 0.367 1.973 0.445 -0.488 ...
##  $ X15: num  -2.006 -1.56 1.963 -0.178 1.358 ...
##  $ X16: num  1.794 -1.365 -0.707 -0.556 -0.31 ...
##  $ X17: num  -0.2493 -1.2219 -0.0725 1.3649 -0.4575 ...
##  $ X18: num  -0.45 1.444 -0.155 -1.022 1.538 ...
##  $ X19: num  -0.532 -0.499 -1.15 0.126 0.473 ...
##  $ X20: num  1.124 0.698 0.181 -0.17 -0.235 ...
```

Ahora, supongamos que deseamos calcular la media de cada una de las variables (columnas). Si emplearemos un loop el código sería como el siguiente:

```r
medias <- NULL

for (i in 1:dim(data)[2]) {
    medias[i] <- mean(data[, i])
}

medias
```

```
##  [1] -0.1567617424  0.0412431799  0.1546036721 -0.0081051362
##  [5] -0.0217858703 -0.1368770057 -0.0878617963 -0.0008371926
##  [9]  0.0181244354 -0.0677145538 -0.0842019763 -0.0075419570
## [13]  0.1336330727  0.0857410357  0.0307662985 -0.0710467482
## [17] -0.2244129083  0.1651090007  0.1291796133 -0.0121430062
```

Pero si empleamos la función **apply()**, esto se puede hacer mucho más rápido y en una sola linea. El código sería de la siguiente manera:

```r
apply(data, MARGIN = 2, mean)
```

```
##            X1            X2            X3            X4
## -0.1567617424  0.0412431799  0.1546036721 -0.0081051362
##            X5            X6            X7            X8
## -0.0217858703 -0.1368770057 -0.0878617963 -0.0008371926
##            X9           X10           X11           X12
##  0.0181244354 -0.0677145538 -0.0842019763 -0.0075419570
##           X13           X14           X15           X16
##  0.1336330727  0.0857410357  0.0307662985 -0.0710467482
##           X17           X18           X19           X20
## -0.2244129083  0.1651090007  0.1291796133 -0.0121430062
```

También se puede aplicar dos o mas funciones al mismo tiempo. Por ejemplo calculemos la media y la desviación estándar al mismo tiempo para cada variable.

```r
Est.descrip <- apply(data, 2, function(i) c(mean(i), sd(i)))
Est.descrip
```

```
##              X1         X2        X3           X4          X5
## [1,] -0.1567617 0.04124318 0.1546037 -0.008105136 -0.02178587
## [2,]  1.0044053 1.03218735 0.9601544  1.050309000  1.11667656
##              X6         X7            X8         X9         X10
## [1,] -0.1368770 -0.0878618 -0.0008371926 0.01812444 -0.06771455
## [2,]  0.9289288  0.9190978  0.9849229772 0.92358054  1.04139508
##              X11          X12       X13        X14       X15
## [1,] -0.08420198 -0.007541957 0.1336331 0.08574104 0.0307663
## [2,]  0.91382267  0.977667908 0.9479492 0.95130108 0.8989082
##              X16        X17      X18       X19         X20
## [1,] -0.07104675 -0.2244129 0.165109 0.1291796 -0.01214301
## [2,]  1.00025480  1.0094924 1.042450 1.0756262  0.95765779
```

Noten que en el tercer argumento de **apply()** se especifica por medio de la *i* cuál será el argumento de la función sobre el que se iterará. El resultado de aplicar la función en un array numérico o matricial dependiendo del caso. Es decir,

```r
medias <- apply(data, MARGIN = 2, mean)
class(medias)

## [1] "numeric"

Est.descrip <- apply(data, 2, function(i) c(mean(i), sd(i)))
class(Est.descrip)

## [1] "matrix" "array"
```

En algunas oportunidades se deseará trabajar con listas y obtener como resultado un objeto de tipo lista, esto se puede hacer con una función de la misma familia de **apply()**: **lapply()**. Esta función **lapply()** tiene los siguientes argumentos:

lapply(X, FUN)

donde:

- **X**: es un vector o lista que será iterado.
- **FUN**: la función que se iterará.

Veamos un ejemplo algo más sofisticado. Por ejemplo, supongamos que se quiere sacar 10 muestras aleatorias de tamaño 1, 2 y 3 sin remplazo de las cinco personas que se encuentran en el objeto `n1`. Y cada una de las muestras se guarda en los elementos de una lista. Esto se puede hacer, sin emplear un loop, fácilmente de la siguiente manera:

```r
n1

## [1] "Juan"   "Ana"    "Pedro"  "Javier" "Paula"

class(n1)

## [1] "character"

set.seed(12324)
muestras <- lapply(1:3, function(i) sample(n1, i, replace = FALSE))
muestras

## [[1]]
## [1] "Javier"
##
## [[2]]
## [1] "Ana"    "Javier"
##
## [[3]]
## [1] "Pedro" "Ana"   "Juan"

class(muestras)

## [1] "list"
```

Otra función de esta familia que puede ser muy interesante son **spply()**, **vapply()** y **replicate()**. Esta última función es muy útil cuando se generan números aleatorios. La función **replicate()** tiene los siguientes argumentos principales:

replicate(n, expr)

donde:

- **n**: el número de iteraciones o replicas.
- **expr**: la expresión que se evaluará en cada iteración.

Por ejemplo, la creación del data.frame `data` se podría hacer de una manera más eficiente si empleamos esta función de la siguiente manera:

```
set.seed(1234)
data.2 <- replicate(20, rnorm(100))
data.2 <- data.frame(data.2)
str(data.2)

## 'data.frame': 100 obs. of  20 variables:
##  $ X1 : num  -1.207 0.277 1.084 -2.346 0.429 ...
##  $ X2 : num  0.415 -0.475 0.066 -0.502 -0.826 ...
##  $ X3 : num  0.485 0.697 0.186 0.701 0.312 ...
##  $ X4 : num  -0.58 -0.9533 -0.1794 1.0098 0.0236 ...
##  $ X5 : num  -1.2268 0.0362 -0.4214 -0.8994 0.4174 ...
##  $ X6 : num  0.985 -1.225 0.71 -0.109 1.783 ...
##  $ X7 : num  0.3135 0.6125 -1.6911 0.7847 0.0119 ...
##  $ X8 : num  0.455 1.425 -0.209 -1.231 -0.361 ...
##  $ X9 : num  -1.0239 -1.3877 -0.0492 1.811 -0.0995 ...
##  $ X10: num  -2.874 -0.427 -1.487 0.589 -0.443 ...
##  $ X11: num  -1.205 0.301 -1.539 0.635 0.703 ...
##  $ X12: num  0.1133 -0.5651 0.0807 -0.6958 -1.0962 ...
##  $ X13: num  -1.402 2.384 0.873 -1.536 1.134 ...
##  $ X14: num  1.051 0.367 1.973 0.445 -0.488 ...
##  $ X15: num  -2.006 -1.56 1.963 -0.178 1.358 ...
##  $ X16: num  1.794 -1.365 -0.707 -0.556 -0.31 ...
##  $ X17: num  -0.2493 -1.2219 -0.0725 1.3649 -0.4575 ...
##  $ X18: num  -0.45 1.444 -0.155 -1.022 1.538 ...
##  $ X19: num  -0.532 -0.499 -1.15 0.126 0.473 ...
##  $ X20: num  1.124 0.698 0.181 -0.17 -0.235 ...

sum(data != data.2)

## [1] 0
```

Noten que el resultado es el mismo.

Existen otras funciones y paquetes que emplean la vectorización para evitar la construcción de loops. Por ejemplo el paquete *dplyr*(Wickham, François, Henry y Müller, 2020) tiene funciones que permiten vectorizar acciones concretas como agregar.

5. Comentarios finales

Los loops son una buena herramienta para realizar tareas repetitivas, pero en algunos casos no son eficientes en términos computacionales. Si decides emplear loops en tu código, intenta incluir la menor cantidad de lineas

de código y cálculos posible dentro del loop. Recuerda que todo lo que esté dentro del loop se repetirá varias veces y quizás no sea necesario todo lo que tienes en el loop.

Otra buena práctica es utilizar una o más llamadas a funciones dentro del loop y no escribir todas las lineas de código que se podía tener en el loop. Las llamadas a funciones facilitarán a otros usuarios el seguimiento del código. Ten en cuenta, que existen funciones de R que permiten vectorizar las iteraciones. En general si tu código tiene loops anidados para realizar operaciones con matrices o arrays, probablemente es posible vectorizar tus operaciones para optimizar tu código para sacarle provecho a un lenguaje basado en matrices como R.

Por otro lado, el costo de computo es cada vez mas bajo. Y a veces el tiempo necesario para pensar la mejor forma de convertir tus loops a operaciones vectorizadas es alto. Tu tendrás que hacer el análisis costo beneficio sobre optimizar tu código disminuyendo unos segundo el cálculo versus el tiempo de pensar la solución. Así aún observaras que muchos investigadores y científicos de datos experimentados aún emplean loops en sus códigos.

Referencias

R Core Team. (2019). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria. Recuperado desde https://www.R-project.org/

Wickham, H., François, R., Henry, L. & Müller, K. (2020). *dplyr: A Grammar of Data Manipulation*. R package version 1.0.2. Recuperado desde https://CRAN.R-project.org/package=dplyr