
R para análisis de datos

En el tema anterior, presentamos R y cómo comenzar a usar la programación R para analizar datos. En esta parte, explicamos más detalladamente algunos de los conceptos importantes necesarios para el análisis de datos, incluida la lectura de varios tipos de archivos de datos, el almacenamiento de datos y la manipulación de datos. También discutimos cómo crear tus propias funciones y paquetes R.

Después de leer estas notas, tendrás una buena introducción a R y podrás comenzar con el análisis de datos.

2.1 Lectura y escritura de datos

Los datos están disponibles en una variedad de fuentes y en una variedad de formatos. Como científico de datos, tu tarea es leer datos de diferentes fuentes y diferentes formatos, analizarlos e informar los resultados. Una fuente de datos puede ser una base de datos Oracle, un sistema de gestión de SAP, la Web o una combinación de estos. El formato de datos puede ser un archivo plano simple en un formato delimitado por comas, formato Excel o formato de lenguaje de marcado extensible (XML, eXtensible Markup Language). R proporciona una amplia gama de herramientas para importar datos. La figura 2.1 muestra las diversas interfaces de R. Más información sobre la importación y exportación de datos R está disponible en el manual R en línea en <https://cran.r-project.org/doc/manuals/R-data.pdf>.

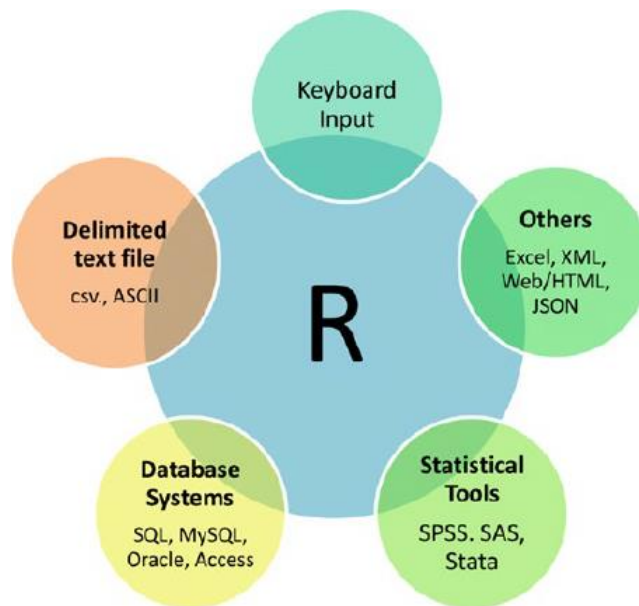


Figura 2.1: Varias interfaces para R.

Como puedes ver, R admite datos de las siguientes fuentes:

- Diversas bases de datos, incluidas MS SQL Server, MySQL, Oracle y Access
- Archivos planos simples como archivos TXT
- Microsoft Excel
- Otros paquetes estadísticos como SPSS y SAS
- Archivos XML y HTML
- Otros archivos, como los archivos de notación de objetos JavaScript (JSON)

Cubrimos algunos de estos formatos de datos importantes en secciones posteriores de este documento y en documentos posteriores.

2.1.1 Leer datos de un archivo de texto

Las funciones `read.table()` y `read.csv()` son las dos funciones compatibles con R para importar datos desde un archivo de texto. La función `read.csv()` se usa específicamente para leer archivos delimitados por comas, mientras que `read.table()` se puede usar para leer cualquier archivo delimitado, pero el delimitador debe mencionarse como parámetro.

Uno de los formatos de entrada más populares para R son los valores separados por comas (CSV, Comma-Separated Values).

Para leer archivos CSV en R, puedes usar `read.csv()`, que importa datos de un archivo CSV y crea un marco de datos (data frame). La sintaxis para `read.csv()` es la siguiente (del archivo de ayuda R):

```
myCSV <- read.csv(file, header = TRUE, sep = ",", quote = "\"",  
                  dec = ".", fill = TRUE, comment.char = "", ...)
```

file: El nombre del archivo. Se debe especificar la ruta completa del archivo (usa `getwd()` para verificar la ruta). Cada línea de un archivo se traduce como cada fila de un marco de datos. El archivo (file) también puede ser una URL completa.

header: Una variable lógica (TRUE o FALSE) que indica si la primera fila del archivo contiene los nombres de las variables. Este indicador se establece en TRUE si la primera línea contiene los nombres de las variables. Por defecto, se establece en FALSE.

sep: El separador es por defecto una coma. No hay necesidad de establecer esta bandera.

fill: Si el archivo tiene una longitud desigual de filas, puedes establecer este parámetro en TRUE para que los campos en blanco se agreguen implícitamente.

dec: El carácter utilizado en el archivo para puntos decimales.

El único parámetro del que debes preocuparte es el archivo (file) con una RUTA (PATH) adecuada. El separador de archivos se establece como una coma de forma predeterminada. Establece el parámetro de encabezado adecuadamente, dependiendo de si la primera fila de un archivo contiene los nombres de las variables.

La figura 2.2 muestra un archivo de texto de muestra con cada elemento de una fila separado por una coma.

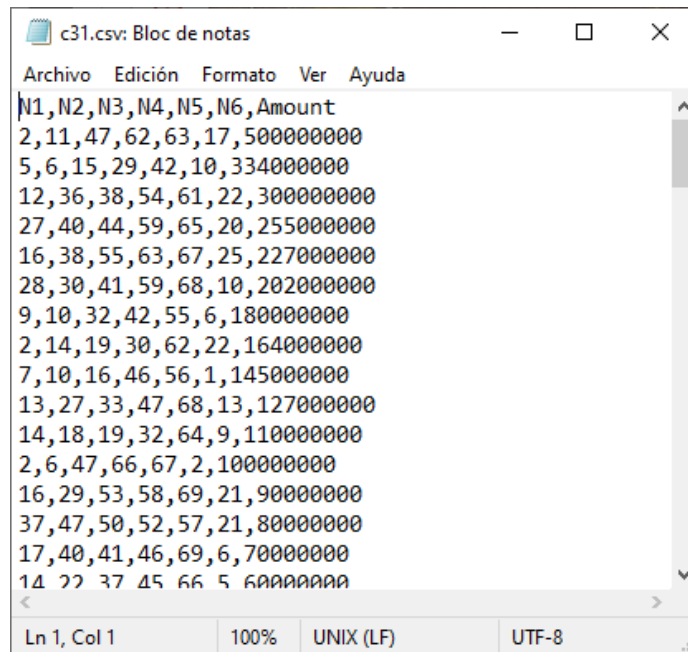


Figura 2.2: Archivo CSV de muestra.

Puedes leer este archivo CSV en R, como se muestra en el siguiente ejemplo:

Nota: Los archivos para lectura deben estar en una carpeta que lee por defecto R, en muchos casos se trata del directorio Documentos (en Windows).

```
> myCSV<-read.csv("c31.csv",header=TRUE)
> str(myCSV)
'data.frame': 104 obs. of 7 variables:
 $ N1 : int 2 5 12 27 16 28 9 2 7 13 ...
 $ N2 : int 11 6 36 40 38 30 10 14 10 27 ...
 $ N3 : int 47 15 38 44 55 41 32 19 16 33 ...
 $ N4 : int 62 29 54 59 63 59 42 30 46 47 ...
 $ N5 : int 63 42 61 65 67 68 55 62 56 68 ...
 $ N6 : int 17 10 22 20 25 10 6 22 1 13 ...
 $ Amount: int 500000000 334000000 300000000 255000000 227000000 202000000 180000000 ...
>
```

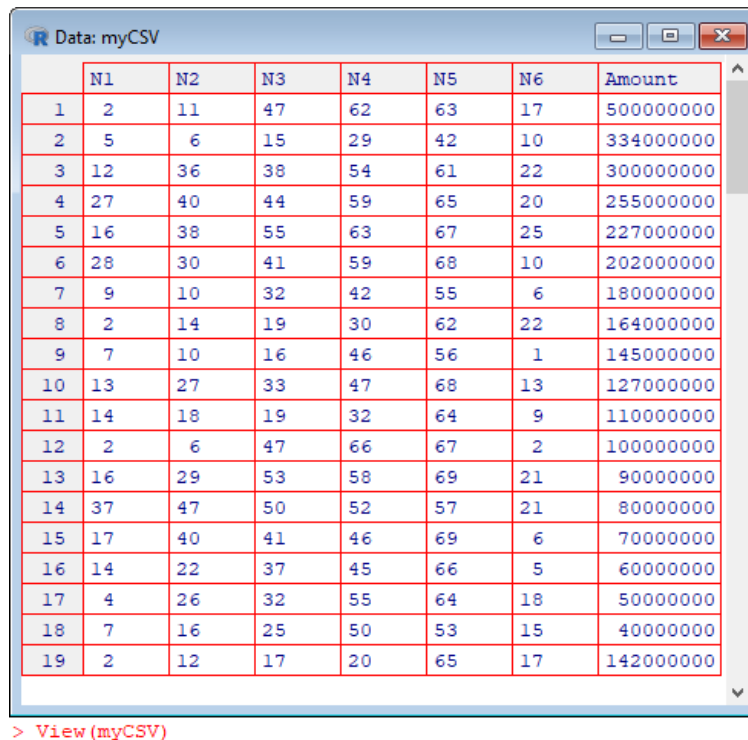
En este archivo CSV, la primera línea contiene encabezados y los valores reales comienzan desde la segunda línea. Cuando lees el archivo mediante `read.csv()`, debes establecer la

opción de encabezado en TRUE para que R pueda comprender que la primera fila contiene los nombres de las variables y pueda leer el archivo en un formato de marco de datos adecuado.

Este ejemplo demuestra cómo `read.csv()` lee automáticamente el archivo con formato CSV en un marco de datos. R también decide el tipo de variable en función de los registros presentes en cada columna. Siete variables están presentes en este archivo, y cada parámetro es un tipo entero. Para ver la tabla del conjunto de datos, simplemente puedes usar el comando `View()` en R:

```
>
> View(myCSV)
>
```

El uso de `View()` abre la tabla de datos en otra ventana, como se muestra en la figura 2.3.



	N1	N2	N3	N4	N5	N6	Amount
1	2	11	47	62	63	17	500000000
2	5	6	15	29	42	10	334000000
3	12	36	38	54	61	22	300000000
4	27	40	44	59	65	20	255000000
5	16	38	55	63	67	25	227000000
6	28	30	41	59	68	10	202000000
7	9	10	32	42	55	6	180000000
8	2	14	19	30	62	22	164000000
9	7	10	16	46	56	1	145000000
10	13	27	33	47	68	13	127000000
11	14	18	19	32	64	9	110000000
12	2	6	47	66	67	2	100000000
13	16	29	53	58	69	21	90000000
14	37	47	50	52	57	21	80000000
15	17	40	41	46	69	6	70000000
16	14	22	37	45	66	5	60000000
17	4	26	32	55	64	18	50000000
18	7	16	25	50	53	15	40000000
19	2	12	17	20	65	17	142000000

> View(myCSV)

Figura 2.3: Vista de un archivo CSV.

El comando `read.table()` es similar a `read.csv()` pero se usa para leer cualquier archivo de texto. El contenido del archivo de texto puede estar separado por un espacio, coma, punto y coma o dos puntos, y debe especificar un separador. El comando tiene otros parámetros opcionales. Para obtener más detalles, escribe `help(read.table)`. La figura 2.4 muestra un archivo de ejemplo que contiene valores separados por una pestaña.

N1	N2	N3	N4	N5	N6	Value
2	11	47	62	63	17	High
5	6	15	29	42	10	Low
12	36	38	54	61	22	Medium
27	40	44	59	65	20	High
16	38	55	63	67	25	Low
28	30	41	59	68	10	Low
9	10	32	42	55	6	Medium
2	14	19	30	62	22	High
7	10	16	46	56	1	Low
13	27	33	47	68	13	Medium
14	18	19	32	64	9	High
2	6	47	66	67	2	Low
16	29	53	58	69	21	Low
37	47	50	52	57	21	Medium
17	40	41	46	69	6	High
14	22	37	45	66	5	Low

Figura 2.4: Archivo de ejemplo.

Aquí está el comando `read.table()` y su salida:

```
> myTable<-read.table("c311.txt",sep="\t",header=TRUE)
>
> str(myTable)
'data.frame': 108 obs. of 7 variables:
 $ N1 : Factor w/ 35 levels "", "1", "10", "11", ...: 12 29 5 17 9 18 33 12 31 6 ...
 $ N2 : int 11 6 36 40 38 30 10 14 10 27 ...
 $ N3 : int 47 15 38 44 55 41 32 19 16 33 ...
 $ N4 : int 62 29 54 59 63 59 42 30 46 47 ...
 $ N5 : int 63 42 61 65 67 68 55 62 56 68 ...
 $ N6 : int 17 10 22 20 25 10 6 22 1 13 ...
 $ Value: Factor w/ 4 levels "", "High", "Low", ...: 2 3 4 2 3 3 4 2 3 4 ...
>
```

En este ejemplo, el archivo de texto está en un formato separado por tabulaciones. La primera línea contiene los nombres de las variables. Además, ten en cuenta que `read.table()` lee el archivo de texto como un marco de datos, y el Valor (Value) se reconoce automáticamente como un factor.

2.1.2 Lectura de datos de un archivo de Microsoft Excel

Hay varios paquetes disponibles para leer un archivo de Excel. Para Windows, usa una conexión Open Database Connectivity (ODBC) con el paquete RODBC. XLConnect es otro paquete que es una solución basada en Java. El paquete gdata está disponible para plataformas Windows, macOS y Linux. A menudo, también se usa el paquete xlsx.

El siguiente ejemplo muestra cómo instalar el paquete RODBC (para Windows). La primera fila del archivo de Excel debe contener los nombres de las variables. El primer paso es descargar el paquete utilizando `install.packages("RODBC")`. El procedimiento se muestra en detalle aquí (debes elegir desde qué mirror descargar):

```
> install.packages("RODBC")
Warning in install.packages("RODBC") :
  'lib = "C:/Program Files/R/R-4.0.2/library"' is not writable
--- Please select a CRAN mirror for use in this session ---
probando la URL 'https://cran.itam.mx/bin/windows/contrib/4.0/RODBC_1.3-17.zip'
Content type 'application/zip' length 878540 bytes (857 KB)
downloaded 857 KB

package 'RODBC' successfully unpacked and MD5 sums checked

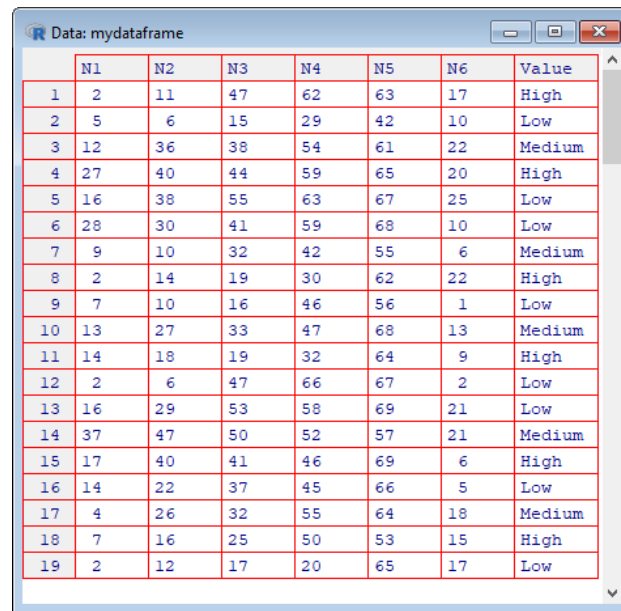
The downloaded binary packages are in
  C:\Users\sans\AppData\Local\Temp\Rtmp0azgmc\downloaded_packages
>
```

Una vez que el paquete se haya instalado correctamente, importe un archivo de Excel a R ejecutando el siguiente conjunto de comandos (sólo con versión 32 bits, utilice la 4.12):

```
> library(RODBC)
> myodbc<-odbcConnectExcel("c311Lot1.xls")
> mydataframe<-sqlFetch(myodbc,"LOT")
>
```

Primero, estableces una conexión ODBC a la base de datos XLS especificando el nombre del archivo. Entonces llamas a la tabla. Aquí, `c311Lot1.xls` es un archivo de Excel en formato XLS, `LOT` es el nombre de la hoja de trabajo dentro del archivo de Excel. `File.myodbc` es el objeto ODBC que abre la conexión ODBC, y `mydataframe` es el marco de datos. Todo el proceso se muestra en la figura 2.5. Se puede usar un procedimiento similar para importar datos desde una base de datos de Microsoft Access.

```
> str(mydataframe)
'data.frame': 108 obs. of 7 variables:
 $ N1 : num 2 5 12 27 16 28 9 2 7 13 ...
 $ N2 : num 11 6 36 40 38 30 10 14 10 27 ...
 $ N3 : num 47 15 38 44 55 41 32 19 16 33 ...
 $ N4 : num 62 29 54 59 63 59 42 30 46 47 ...
 $ N5 : num 63 42 61 65 67 68 55 62 56 68 ...
 $ N6 : num 17 10 22 20 25 10 6 22 1 13 ...
 $ Value: chr "High" "Low" "Medium" "High" ...
>
```



> View(mydataframe)

	N1	N2	N3	N4	N5	N6	Value
1	2	11	47	62	63	17	High
2	5	6	15	29	42	10	Low
3	12	36	38	54	61	22	Medium
4	27	40	44	59	65	20	High
5	16	38	55	63	67	25	Low
6	28	30	41	59	68	10	Low
7	9	10	32	42	55	6	Medium
8	2	14	19	30	62	22	High
9	7	10	16	46	56	1	Low
10	13	27	33	47	68	13	Medium
11	14	18	19	32	64	9	High
12	2	6	47	66	67	2	Low
13	16	29	53	58	69	21	Low
14	37	47	50	52	57	21	Medium
15	17	40	41	46	69	6	High
16	14	22	37	45	66	5	Low
17	4	26	32	55	64	18	Medium
18	7	16	25	50	53	15	High
19	2	12	17	20	65	17	Low

Figura 2.5: Trabajando con datos importados.

Para leer el formato Excel 2007 XLSX, puedes instalar el paquete `xlsx` usando `install.packages("xlsx")` y usar el comando `read.xlsx()` de la siguiente manera.

Nota: Hay que instalar igualmente dos paquetes de los cuales depende este paquete (`install.packages("rJava")` y `install.packages("xlsxjars")`).

```
> library(xlsx)
> myxlsx<- "c311Lot.xlsx"
> myxlsxdata<-read.xlsx(myxlsx,1)
```

El primer argumento de `read.xlsx()` es el archivo de Excel; 1 se refiere a la primera hoja de trabajo en Excel, que se guarda como un marco de datos. Existen numerosos paquetes disponibles para leer el archivo de Excel, incluidos `gdata`, `xlsReadWrite` y `XLConnect`, pero el paquete `xlsx` es el más popular.

Sin embargo, se recomienda convertir el archivo de Excel en un archivo CSV y usar `read.table()` o `read.csv()` en lugar de los demás. El siguiente código proporciona ejemplos del uso de otros paquetes y la lectura del archivo:

```
> install.packages("XLConnect")
> library(XLConnect)
> myE1<-loadWorkbook("c311Lot1.xls")
> mydata<-readWorksheet(myE1,sheet="Lot",header=TRUE)

> install.packages("gdata")
> Data<-read.xls("c311Lot1.xls",sheet=1,header=TRUE)
```

2.1.3 Lectura de datos de la web

En estos días, a menudo queremos leer datos de la Web. Puedes descargar una página web utilizando el comando `readLines()` y guardarla como una estructura R para su posterior análisis.

Recuerda que los datos web pueden ser datos no estructurados y que necesitan un procesamiento previo adicional antes del análisis. Puedes usar `grep()` y otras expresiones regulares para manipular datos web. Para obtener más información, puede consultar el sitio web de ProgrammingR en <https://www.programmingr.com/>.

Veamos algunos ejemplos interesantes.

Usaremos en el curso, data sets del UCI Machine Learning Repository en <http://archive.ics.uci.edu/ml/index.php>.

Primero, cargamos los datos (ten en cuenta que este no es un archivo CSV estándar, sino que usa el punto y coma como separador de columnas) de la siguiente manera:

```
>data<-read.csv("https://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality-white.csv", sep=";")
```

Hay cerca de 5,000 observaciones en el conjunto de datos. Aquí hay un resumen de los datos que proporciona una descripción general:

```
> summary(data)
```

Importando un archivo *.txt de Internet

```
> data=read.table(file="http://www.sthda.com/upload/deathlon.txt",  
+ header=T, row.names=1, sep="\t")  
> print(data)
```

2.2 Uso de estructuras de control en R

Al igual que otros lenguajes de programación, R admite estructuras de control que te permiten controlar el flujo de ejecución. Las estructuras de control permiten que un programa se ejecute con cierta “lógica”. Sin embargo, muchas veces estas estructuras de control tienden a ralentizar el sistema. En cambio, se utilizan otras funciones integradas al manipular los datos (que discutiremos más adelante en estas notas). Algunas de las estructuras de control comúnmente soportadas son las siguientes:

if y else: Prueba una condición
 while: Ejecuta un ciclo cuando una condición es verdadera
 for: Repite un número fijo de veces
 repeat: Ejecuta un ciclo continuamente (debe salir del ciclo salir)
 next: Opción para omitir una iteración de un ciclo
 break: Rompe la ejecución de un ciclo

2.2.1 if-else

La estructura if-else es la función más común utilizada en cualquier lenguaje de programación, incluido R. Esta estructura te permite probar una condición verdadera o falsa. Por ejemplo, supongamos que estás analizando un conjunto de datos y tienes que trazar un gráfico de x vs. y basado en una condición, por ejemplo, si la edad de una persona es mayor de 65 años. En tales situaciones, puedes usar una declaración if-else. La estructura común de if-else es la siguiente:

```

Si (<condición>) {
  ## hacer algo
}
else {
  ## hacer nada
}

```

Puedes tener múltiples declaraciones if-else. Para obtener más información, puedes consultar las páginas de ayuda de R. Aquí hay una demostración de la función if-else en R:

```

> #Generate uniform random number
> x<-runif(1,0,20)
> x
[1] 14.62051
> if (x>10){
+ print(x)
+ }else{
+ print("x is less than 10")
+ }
[1] 14.62051
>

```

2.2.2 Ciclos for

Los *ciclos for* son similares a otras estructuras de programación. Durante el análisis de datos, los *ciclos for* se utilizan principalmente para acceder a un arreglo o lista. Por ejemplo, si accedes a un elemento específico en un arreglo y realizas la manipulación de datos, puedes usar un ciclo for. Aquí hay un ejemplo de cómo se usa un ciclo for:

```

> x<-c("Juan","Beni","Jorge","David")
> x
[1] "Juan" "Beni" "Jorge" "David"
> for (i in 2:3){
+ ## Imprime solo 2 elementos
+ print(x[i])
+ }
[1] "Beni"
[1] "Jorge"
>

```

A veces, la función `seq_along()` se usa junto con el ciclo `for`. La función `seq_along()` genera un número entero basado en la longitud del objeto. El siguiente ejemplo usa `seq_along()` para imprimir cada elemento de `x`:

```

> x
[1] "Juan" "Beni" "Jorge" "David"
> for (i in seq_along(x))
+ {
+ print(x[i])
+ }
[1] "Juan"
[1] "Beni"
[1] "Jorge"
[1] "David"
>

```

2.2.3 Ciclos while

El ciclo R `while` tiene una condición y un cuerpo. Si la condición es verdadera, el control ingresa al cuerpo del ciclo. El ciclo continúa la ejecución hasta que la condición sea verdadera; sale después de que la condición falla. Aquí hay un ejemplo:

```

> count=0
> while (count<10){
+ print(count)
+ count=count+1
+ }
[1] 0
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
>

```

`repeat()` y `next()` no se usan comúnmente en análisis estadísticos o de datos, pero tienen sus propias aplicaciones. Si estás interesado en obtener más información, puedes consultar las páginas de ayuda de R.

2.2.4 Funciones de ciclo

Aunque los ciclos `for` and `while` son herramientas de programación útiles, el uso de llaves y funciones de estructuración a veces puede ser engorroso, especialmente cuando se trata de grandes conjuntos de datos. Por lo tanto, R tiene algunas funciones que implementan ciclos en una forma compacta para hacer que el análisis de datos sea más simple y efectivo. R admite las siguientes funciones, que veremos desde la perspectiva del análisis de datos:

`apply()`: Evalúa una función en una sección de un arreglo y devuelve los resultados en un arreglo.

`lapply()`: Recorre una lista y evalúa cada elemento o aplica la función a cada elemento.

`sapply()`: Una aplicación fácil de usar de `lapply()` que devuelve un vector, matriz o arreglo.

`tapply()`: Usualmente se usa sobre un subconjunto de un conjunto de datos

Estas funciones se utilizan para manipular y cortar datos de matrices, arreglos, listas o marcos de datos. Estas funciones atraviesan un conjunto de datos completo, ya sea por fila o por columna, y evitan construcciones de ciclo. Por ejemplo, estas funciones se pueden llamar para hacer lo siguiente:

- Calcular la media, la suma o cualquier otra manipulación en una fila o columna.
- Transformar o realizar subconjuntos.

La función `apply()` es más simple en su forma, y su código puede tener muy pocas líneas (en realidad, una línea) mientras ayuda a realizar operaciones efectivas. Las otras formas más complejas son `lapply()`, `sapply()`, `vapply()`, `mapply()`, `rapply()` y `tapply()`. El uso de estas funciones depende de la estructura de los datos y del formato de la salida que necesitas para ayudar a tu análisis.

Para comprender cómo funciona la función de ciclo, utilizaremos el conjunto de datos de Cars, que es parte de la biblioteca R. Este conjunto de datos contiene la velocidad de los automóviles y las distancias tomadas para detenerse. Estos datos se registraron en la década de 1920. Este marco de datos tiene dos variables: velocidad (`speed`) y dist. Ambos son numéricos y contienen un total de 50 observaciones.

2.2.4.1 `apply()`

Con la función `apply()`, puedes realizar operaciones en cada fila o columna de una matriz o marco de datos o lista sin tener que escribir ciclos. El siguiente ejemplo muestra cómo usar `apply()` para encontrar la velocidad promedio y la distancia promedio de los autos:

```
> apply(cars, 2, mean)
speed dist
15.40 42.98
> head(cars)
  speed dist
1     4    2
2     4   10
3     7    4
4     7   22
5     8   16
6     9   10
>
```

La función `apply()` se usa para evaluar una función sobre un arreglo[] y a menudo se usa para operaciones en filas o columnas:

```
> str(apply)
function (X, MARGIN, FUN, ...)
>
```

La función `apply()` toma lo siguiente:

X: Un arreglo

MARGIN: Un vector entero para indicar una fila o columna

FUN: El nombre de la función que estás aplicando

En el ejemplo anterior, calculamos la media de cada fila y columna. Pero la función podría ser cualquier cosa; puede ser SUM o average o tu propia función. El concepto es que recorre toda la matriz y proporciona resultados en el mismo formato. Esta función elimina todos los ciclos engorrosos, y los resultados se logran en una línea.

El siguiente ejemplo calcula una medida cuantil para cada fila y columna:

```
> apply(cars, 2, quantile)
      speed dist
0%      4    2
25%     12   26
50%     15   36
75%     19   56
100%    25  120
>
```

2.2.4.2 lapply()

La función `lapply()` genera los resultados como una `list.lapply()` y se puede aplicar a una lista, marco de datos o vector. La salida siempre es una lista que tiene el mismo número de elementos que el objeto que se pasó a `lapply()`:

```
> str(lapply)
function (X, FUN, ...)
>
```

Toma los siguientes argumentos:

X: Conjunto de datos

FUN: Función

El siguiente ejemplo muestra la función `lapply()` para el mismo conjunto de datos de Cars. El conjunto de datos de Cars es un marco de datos con dos variables `dist` y `speed`; Ambos son numéricos. Para encontrar la media de velocidad y `dist`, puedes usar `lapply()`, y la salida es una lista:

```
>
> str(cars)
'data.frame': 50 obs. of 2 variables:
 $ speed: num 4 4 7 7 8 9 10 10 10 11 ...
 $ dist : num 2 10 4 22 16 10 18 26 34 17 ...
>
> lap<-lapply(cars,mean)
> lap
$ speed
[1] 15.4

$ dist
[1] 42.98

> str(lap)
List of 2
 $ speed: num 15.4
 $ dist : num 43
>
```

2.2.4.3 `sapply()`

La principal diferencia entre `sapply()` y `lapply()` es el resultado de salida. El resultado de `sapply()` puede ser un vector o una matriz, mientras que el resultado de `lapply()` es una lista. Según el tipo de análisis de datos que esté realizando y el formato de resultado que necesites, puedes utilizar las funciones adecuadas. Como resolvemos muchos problemas de análisis en el curso, verás el uso de diferentes funciones en diferentes momentos. El siguiente ejemplo demuestra el uso de `sapply()` para el mismo ejemplo de conjunto de datos de coche:

```
> sap<-sapply(cars,mean)
> sap
speed dist
15.40 42.98
> str(sap)
Named num [1:2] 15.4 43
- attr(*, "names")= chr [1:2] "speed" "dist"
```

2.2.4.4 `tapply()`

`tapply()` se usa sobre subconjuntos de un vector. La función `tapply()` es similar a otras funciones `apply()`, excepto que se aplica sobre un subconjunto de un conjunto de datos:

```
> str (tapply)
```

```
function (X, INDEX, FUN = NULL, ..., simplify = TRUE)
```

X: Un vector

INDEX: Un factor o una lista de factores (o se convierten en factores)

FUN: Una función para aplicar

- ... contiene otros argumentos para pasar a FUN
- `simplify`, TRUE o FALSE para simplificar el resultado

Para demostrar la función de `tapply()`, consideremos el conjunto de datos de `Mtcars`. Este conjunto de datos se extrajo de una revista Motor Trend de 1974. Los datos incluyeron detalles del consumo de combustible y 10 aspectos del diseño y desempeño del automóvil para 32 automóviles (modelos 1973-1974). La Tabla 2-1 muestra los parámetros del conjunto de datos.

Tabla 2.1: Conjunto de datos de `Mtcars`.

[,1]	mpg	Miles per gallon
[2]	cyl	Número de cilindros
[,3]	disp	Desplazamiento (en pulgadas cúbicas)
[,4]	hp	Potencia bruta
[,5]	drat	Relación del eje trasero
[,6]	wt	Peso (lb/1000)
[,7]	qsec	tiempo de cuarto de milla
[,8]	vs	V/S
[,9]	am	Transmisión (0 = automática, 1 = manual)
[,10]	gear	Número de marchas adelante
[,11]	carb	Numero de carburadores

Data source: Henderson and Velleman, "Building Multiple Regression Models Interactively," Biometrics, 37 (1981), pp. 391–411.

La figura 2.6: muestra algunos de los datos del conjunto de datos de `Mtcars`.

Supongamos que necesitas averiguar el consumo promedio de gasolina (mpg) de cada cilindro. Usando la función `tapply()`, la tarea se ejecuta en una línea, como sigue:

```
> tapply(mtcars$mpg,mtcars$cyl,mean)
      4      6      8
26.66364 19.74286 15.10000
>
```

```

> mtcars
      mpg  cyl  disp  hp drat   wt  qsec vs am gear carb
Mazda RX4           21.0    6  160.0  110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag       21.0    6  160.0  110 3.90 2.875 17.02  0  1    4    4
Datsun 710          22.8    4  108.0   93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive      21.4    6  258.0  110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout   18.7    8  360.0  175 3.15 3.440 17.02  0  0    3    2
Valiant             18.1    6  225.0  105 2.76 3.460 20.22  1  0    3    1
Duster 360          14.3    8  360.0  245 3.21 3.570 15.84  0  0    3    4
Merc 240D            24.4    4  146.7   62 3.69 3.190 20.00  1  0    4    2
Merc 230             22.8    4  140.8   95 3.92 3.150 22.90  1  0    4    2
Merc 280             19.2    6  167.6  123 3.92 3.440 18.30  1  0    4    4
Merc 280C            17.8    6  167.6  123 3.92 3.440 18.90  1  0    4    4
Merc 450SE           16.4    8  275.8  180 3.07 4.070 17.40  0  0    3    3
Merc 450SL           17.3    8  275.8  180 3.07 3.730 17.60  0  0    3    3
Merc 450SLC          15.2    8  275.8  180 3.07 3.780 18.00  0  0    3    3
Cadillac Fleetwood   10.4    8  472.0  205 2.93 5.250 17.98  0  0    3    4
Lincoln Continental  10.4    8  460.0  215 3.00 5.424 17.82  0  0    3    4
Chrysler Imperial    14.7    8  440.0  230 3.23 5.345 17.42  0  0    3    4
Fiat 128             32.4    4   78.7   66 4.08 2.200 19.47  1  1    4    1
Honda Civic          30.4    4   75.7   52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla       33.9    4   71.1   65 4.22 1.835 19.90  1  1    4    1
Toyota Corona        21.5    4  120.1   97 3.70 2.465 20.01  1  0    3    1
Dodge Challenger     15.5    8  318.0  150 2.76 3.520 16.87  0  0    3    2
AMC Javelin          15.2    8  304.0  150 3.15 3.435 17.30  0  0    3    2
Camaro Z28           13.3    8  350.0  245 3.73 3.840 15.41  0  0    3    4
Pontiac Firebird     19.2    8  400.0  175 3.08 3.845 17.05  0  0    3    2
Fiat X1-9            27.3    4   79.0   66 4.08 1.935 18.90  1  1    4    1
Porsche 914-2        26.0    4  120.3   91 4.43 2.140 16.70  0  1    5    2
Lotus Europa         30.4    4   95.1  113 3.77 1.513 16.90  1  1    5    2
Ford Pantera L       15.8    8  351.0  264 4.22 3.170 14.50  0  1    5    4
Ferrari Dino         19.7    6  145.0  175 3.62 2.770 15.50  0  1    5    6
Maserati Bora        15.0    8  301.0  335 3.54 3.570 14.60  0  1    5    8
Volvo 142E           21.4    4  121.0  109 4.11 2.780 18.60  1  1    4    2

```

Figura 2.6: Algunos datos de muestra del conjunto de datos de Mtcars.

De manera similar, si deseas averiguar la potencia promedio (hp) para la transmisión automática y manual, simplemente usa `tapply()` como se muestra aquí:

```

> tapply(mtcars$hp,mtcars$am,mean)
      0      1
160.2632 126.8462
>

```

Como puedes ver en este ejemplo, la familia de funciones `apply()` es poderosa y te permite evitar los ciclos `for` y `while` tradicionales. Dependiendo del tipo de análisis de datos que desees realizar, puedes utilizar las funciones respectivas.

2.2.4.5 cut()

A veces, en el análisis de datos, es posible que debas dividir las variables continuas para colocarlas en diferentes contenedores. Puedes hacer esto fácilmente usando la función `cut()` en R.

Tomemos el conjunto de datos de Orange; nuestra tarea es agrupar los árboles según la edad. Esto se puede lograr usando `cut()` como se muestra aquí:

```
> Orange
  Tree age circumference
1    1  118             30
2    1  484             58
3    1  664             87
4    1 1004            115
5    1 1231            120
6    1 1372            142
7    1 1582            145
8    2  118             33
9    2  484             69
10   2  664            111
11   2 1004            156
12   2 1231            172
13   2 1372            203
14   2 1582            203
15   3  118             30
16   3  484             51
```

A continuación, crea cuatro grupos según la edad de los árboles. El primer parámetro es el conjunto de datos y la edad, y el segundo parámetro es el número de grupos que se desea crear. En este ejemplo, los naranjos se agrupan en cuatro categorías según la edad, y hay cinco árboles de entre 117 y 483 años:

```
> c1<-cut(Orange$age,breaks=4)
> table(c1)
c1
      (117,484]      (484,850]      (850,1.22e+03] (1.22e+03,1.58e+03]
           10              5              5              15
>
```

Los intervalos se definen automáticamente mediante cut() y pueden no ser los previstos. Sin embargo, usando el comando seq(), puedes especificar los intervalos:

```
> seq(100,2000,by=300)
[1] 100 400 700 1000 1300 1600 1900
> c2<-cut(Orange$age,breaks=seq(100,2000,by=300))
> table(c2)
c2
      (100,400]      (400,700]      (700,1e+03]      (1e+03,1.3e+03]
           5           10           0           10
(1.3e+03,1.6e+03] (1.6e+03,1.9e+03]
          10           0
>
```

2.2.4.6 split()

A veces, en el análisis de datos, es posible que debas dividir el conjunto de datos en grupos. Esto se puede lograr cómodamente usando la función split(). Esta función divide el conjunto de datos en grupos definidos por el argumento. La función unsplit() invierte los resultados de split().

La sintaxis general de split() es la siguiente; x es el vector y f es el argumento split:

```
> str(split)
function (x, f, drop = FALSE, ...)
```


En el siguiente ejemplo, el conjunto de datos de Orange se divide en función de la edad. La diferencia es que el conjunto de datos se agrupa según la edad. Hay siete grupos de edad y el conjunto de datos se divide en siete grupos:

```
> c3<-split(Orange,Orange$age)
> c3
$`118`
  Tree age circumference
1     1 118             30
8     2 118             33
15    3 118             30
22    4 118             32
29    5 118             30

$`484`
  Tree age circumference
2     1 484             58
9     2 484             69
16    3 484             51
23    4 484             62
30    5 484             49

$`664`
  Tree age circumference
3     1 664             87
10    2 664            111
17    3 664             75
24    4 664            112
31    5 664             81

$`1004`
```

2.2.5 Escribir tus propias funciones en R

Al igual que en cualquier otro lenguaje de programación, puedes escribir tus propias funciones en R.

Las funciones se escriben si es necesario repetir una secuencia de tareas varias veces en el programa. Las funciones también se escriben si el código debe compartirse con otros o con el público en general. Las funciones también se escriben si no hay funciones ya definidas como parte de los comandos del lenguaje de programación. Una función reduce la complejidad de la programación al crear una abstracción del código, y solo es necesario especificar un conjunto de parámetros. Las funciones en R pueden tratarse como un objeto.

En R, las funciones se pueden pasar como argumento a otras funciones, como `apply()` o `sapply()`.

Las funciones en R se almacenan como un objeto, al igual que los tipos de datos, y se definen como `function()`. Los objetos de función son tipos de datos con un objeto de clase definido como función. Este ejemplo muestra cómo definir una función simple:

```
> myFunc<-function(){
+ print("Mi primera funcion")
+ }
> myFunc()
[1] "Mi primera funcion"
>
```

Esta función no acepta argumentos. Para hacerlo más interesante, puedes agregar un argumento de función. En el cuerpo de esta función, agregamos un ciclo for, para imprimir tantas veces como desee el usuario. El usuario determina el número de veces que se repite especificando el argumento en la función. Esto se ilustra con los dos ejemplos siguientes:

```
> myFun<-function(num)
+ {
+ for (i in seq_len(num))
+ {
+ cat("My FUnction:",i,"\n")
+ }
+ }
>

> myFun(2)
My FUnction: 1
My FUnction: 2
>
```

En este ejemplo, myFun() es una función que toma un argumento. Puedes pasar el parámetro al llamar a la función. En este caso, como puedes ver en la salida, la función imprime el número de veces que ha realizado un ciclo en función del número pasado a la función como argumento.

2.3 Trabajar con bibliotecas y paquetes R

R es de código abierto. Más de 2,500 usuarios han contribuido con varias funciones, conocidas como paquetes, a R. Algunas funciones forman parte de la instalación de forma predeterminada, y otras deben descargarse e instalarse explícitamente desde <http://cran.r-project.org/web/packages>. Estos paquetes contienen varias características o funciones desarrolladas en R y puestas a disposición del público. Son funciones R que ya están compiladas y están disponibles en formato binario (o formato R). El directorio donde residen estos códigos se llama biblioteca.

Después de instalar los paquetes, llama a estas funciones cargando la función library() para que esté disponible en tu sesión de trabajo.

Para ver qué bibliotecas están instaladas en tu entorno R, puedes ejecutar library() en la línea de comando. Para ver la ruta donde están instaladas estas bibliotecas, puedes ejecutar el comando libPaths(). Aquí están los comandos ejecutados:

```
> .libPaths()
[1] "C:/Users/sans_/Documents/R/win-library/4.0"
[2] "C:/Program Files/R/R-4.0.2/library"
>
```

La Figura 2.7 muestra la salida, enumerando las bibliotecas instaladas.



Figura 2.7: Lista de bibliotecas instaladas.

Puedes instalar paquetes ejecutando el comando `install.packages()`. Este ejemplo instala el paquete `rattle`:

```
> install.packages("rattle")
Installing package into 'C:/Users/sans_/Documents/R/win-library/4.0'
(as 'lib' is unspecified)
also installing the dependencies 'ps', 'processx', 'callr', 'prettyunits', 'backports'

There is a binary version available but the source version is later:
  binary source needs_compilation
backports 1.1.7 1.1.8 TRUE

Binaries will be installed
probando la URL 'https://cran.itam.mx/bin/windows/contrib/4.0/ps_1.3.3.zip'
Content type 'application/zip' length 701379 bytes (684 KB)
downloaded 684 KB
```

Nota: La guía de estilo R de Google está disponible en <https://google.github.io/styleguide/Rguide.xml>.

2.4 Resumen

Los datos deben leerse de manera efectiva en R antes de que puedas comenzar cualquier análisis. En esta parte del curso, comenzamos con ejemplos que muestran que varios formatos de archivo de diversas fuentes se pueden leer en R (incluidos archivos de base de datos, archivos de texto, archivos XML y archivos de otras herramientas estadísticas y analíticas).

Además, exploramos cómo se pueden leer varios formatos de archivos de datos en R utilizando comandos tan simples como `read.csv()` y `read.table()`. También checamos ejemplos de importación de datos de archivos de MS Excel y de la Web.

A través de ejemplos detallados, exploramos el uso de estructuras de ciclo como `ifelse`, `while` y `for`. También aprendiste acerca de funciones recursivas simples disponibles en R, como `apply()`, `lapply()`, `sapply()` y `tapply()`. Estas funciones pueden reducir considerablemente la complejidad del código y los posibles errores que pueden ocurrir con las estructuras de ciclo tradicionales como `if-else`, `while` y `for`. Además, analizamos el uso de las funciones `cut()` y `split()`.

Las funciones definidas por el usuario son útiles cuando necesitas compartir código o reutilizar una función una y otra vez. Viste una forma sencilla y fácil de crear funciones definidas por el usuario en R.

Finalmente, miraste los paquetes. Viste cómo usar la función `library()` para determinar qué paquetes ya están instalados en R y cómo usar la instalación. La función `package()` es para instalar paquetes adicionales en R.