El entorno estadístico R

Estructura, lenguaje y sintaxis

PID_00273871

Daniel Liviano Solís
Maria Pujol Jover

Revisión a cargo de Ángel Javier Gil

Universitat Oberta de Catalunya

Daniel Liviano Solís

Profesor de los Estudios de Economía y Empresa de la Universitat Oberta de Catalunya. Doctor en Economía por la Universitat Rovira i Virgili.

Maria Pujol Jover

Profesora de los Estudios de Economía y Empresa de la Universitat Oberta de Catalunya. Doctora en Estudios Empresariales por la Universidad de Barcelona.

La revisión de este recurso de aprendizaje UOC ha sido coordinada por la profesora: Laura Calvet Liñán

Tercera edición: septiembre 2020

Av. Tibidabo, 39-43, 08035 Barcelona
Autoría: Daniel Liviano Solís, Maria Pujol Jover
Producción: FUOC

Índice

Introducción

El objetivo principal de este módulo es introducir al estudiante en el entorno estadístico R. Se podría definir como un lenguaje y un entorno para computación y gráficos estadísticos. Es un proyecto libre, es decir, no requiere el pago de ninguna licencia. R proporciona una amplia variedad de técnicas estadísticas (modelos lineales y no lineales, pruebas estadísticas clásicas, análisis de series temporales, análisis multivariante, etc.). Además, también ofrece una alta potencialidad para elaborar gráficos complejos y de calidad.

El uso de R en las asignaturas cuantitativas que ofrece la UOC permite al estudiante muchas y variadas posibilidades de análisis numérico y estadístico. Algunas se detallan a continuación. R proporciona:

1) Facilidades para almacenar y manejar datos numéricos en diferentes formatos, como vectores, matrices y bases de datos.

2) Una amplia colección de operadores y funciones matemáticas.

3) Una larga lista de paquetes estadísticos con todo tipo de herramientas estadísticas y matemáticas.

4) Facilidades gráficas para el análisis y la visualización de datos.

5) Un lenguaje de programación sencillo y eficaz muy desarrollado, que incluye condicionales, bucles y funciones recursivas.

6) La posibilidad de añadir funcionalidades adicionales mediante la definición de nuevas funciones creadas por el usuario.

7) La posibilidad de extenderse mediante paquetes. Hay cerca de ocho paquetes básicos suministrados con la distribución de R, pero hay muchos más de disponibles en el CRAN (*Comprehensive R Archive Network*).

Se tiene que reconocer que empezar a trabajar con R no es fácil, sobre todo para los estudiantes que no están acostumbrados a utilizar paquetes informáticos de análisis numérico. Es decir, hay una cierta curva de aprendizaje que tiene que superarse con la práctica. Este módulo busca, pues, servir de apoyo en estos primeros pasos siempre complicados, de modo que el estudiante se familiarice con este paquete estadístico y pueda aprovechar toda su potencialidad.

A pesar de que existe una extensión llamada R Commander que permite simular un conjunto de menús similar a Minitab o SPSS y, dadas las características de nuestros estudios, es preferible trabajar directamente con los comandos de

R y ayudarnos con un entorno de trabajo, como por ejemplo RStudio, que nos permitirá integrar en un único documento las instrucciones de R, los resultados obtenidos, incluyendo gráficos, y nuestros comentarios.

Objetivos

1. Instalar correctamente R en el ordenador.

2. Instalar y cargar paquetes adicionales en función de las necesidades del análisis.

3. Utilizar las herramientas de ayuda que ofrece R para resolver dudas.

4. Localizar los manuales disponibles en el web oficial del programa.

5. Identificar los tipos principales de datos y objetos admitidos en R, y poderlos manejar eficientemente.

6. Crear un *script* para escribir un análisis de manera coherente, asignando nombres a nuevas variables.

7. Hacer operaciones matemáticas y estadísticas con vectores y matrices.

8. Dominar los operadores lógicos y las funciones recursivas, ciclos y condicionales.

9. En general, llevar a cabo todo tipo de operaciones matriciales.

10. Leer y escribir ficheros, sobre todo en formato csv.

11. Hacer los primeros pasos con RStudio.

1. Primeros pasos con R

1.1. ¿Qué es R?

Esta es una pregunta simple que no tiene una respuesta fácil. En la definición más amplia, es un lenguaje de programación que permite al usuario programar algoritmos y usar herramientas programadas por otras. Si nos limitamos al análisis estadístico y cuantitativo, podemos afirmar sin temor a exagerar que no tiene límites, es decir, es capaz de hacer cualquier cosa que el usuario pueda imaginar. Con R se pueden escribir funciones, hacer cálculos, aplicar técnicas estadísticas complejas e implementar técnicas *ad hoc* (la manera más sencilla de aplicar técnicas complicadas es aprovechar las que ya están disponibles en la red y, en caso necesario, optar por desarrollarlas nosotros mismos), crear gráficos simples y complejos, e incluso escribir funciones propias. El hecho de que muchos centros de investigación y universidades lo usen hace que existan miles de contribuciones disponibles para todos los usuarios. Además, hay material abundante (libros, manuales, ejemplos, etc.) de alta calidad disponibles gratuitamente. Una ventaja nada despreciable es que, a diferencia de muchos programas estadísticos, no tiene ningún coste, puesto que es un proyecto GNU (software libre).

The R Project for Statistical Computing

En la página web oficial http://www.r-project.org encontraremos toda la información que necesitamos sobre este programa en inglés: archivos de instalación, características, paquetes disponibles, manuales, etc. Para encontrar manuales en castellano, tenemos que visitar la página http://cran.es.r-project.org/ e ir a la sección Contributed.

En este contexto, las ventajas principales de R pueden resumirse en la siguiente lista:

- Es un programa distribuido libremente bajo una licencia GNU.
- Hay mucho material de apoyo: manuales, programas de aprendizaje y ejemplos.
- Además, hay librerías que cubren casi todas las metodologías de la estadística y las matemáticas (optimización, series temporales, programación lineal, ecuaciones diferenciales, inferencia, etc.).
- Permite sistematizar análisis largos y complejos en unas pocas instrucciones sintéticas, por lo que es un programa muy eficiente.

¡Que nadie se asuste! Es normal que, al principio, introducirse en el mundo de la programación cueste un poco. Pero una vez os acostumbréis a este nuevo entorno y lo dominéis, todo será mucho más fácil.

Sin embargo, también podemos mencionar algunos inconvenientes:

- Hay una barrera de entrada o curva de aprendizaje considerable.
- No es el tipo de programa con ventanas y menús, como pueden serlo Minitab, E-Views o SPSS. Sin embargo, hay extensiones de R, como R-Commander, que sí que se basan en el uso de ventanas y menús, y existen entornos como RStudio que permiten trabajar de manera muy eficiente.

R es un lenguaje de programación muy popular en el ámbito de la estadística y las probabilidades, y no solo se utiliza en universidades o centros de investigación sino que también tiene muchos usos en la empresa privada y el sector público. También presenta ventajas respecto a otros lenguajes de programación que incorporan funciones estadísticas, precisamente porque está diseñado específicamente y desde su origen para trabajar eficientemente con vectores y bases de datos, además de contar con una comunidad muy amplia y fiel que constantemente propone nuevas soluciones y a la que podemos dirigirnos si necesitamos ayuda en algún problema concreto. En este contexto, sería importante insistir en no "inventar la rueda" cada vez e intentar adaptarnos al que nos propone R (por ejemplo evitar programar o recurrir a bucles cuando hay funciones propias de R que hacen el cálculo de forma más eficiente).

En lo referente a la documentación, podemos encontrar información general en https://www.r-project.org/help.html, diferentes manuales en https://cran.r-project.org/manuals.html, y material traducido, que puede sernos muy útil, en https://cran.r-project.org/other-docs.html.

Es importante también destacar que hay muchas fuentes de ayuda en internet y que cuando tengamos una idea básica del funcionamiento de R podemos fácilmente acudir a recursos o tutoriales en línea o consultar en foros de ayuda donde no solo se habla de la parte de utilización de R sino también de las técnicas estadísticas que pueden usarse, lo que añade interés al uso de R. Podemos empezar desde https://www.r-project.org/help.html o ir directamente, como es bastante habitual, a la página de Stack Overflow y las discusiones específicas sobre R en https://stackoverflow.com/questions/tagged/r.

1.2. Instalación

El primer paso obligado es instalar el programa, por lo que tendremos que visitar la página web del proyecto:

http://www.r-project.org/

Una vez allí, veremos que hay varias secciones, entre ellas el centro de descargas *CRAN*. Si accedemos a este centro, tendremos en pantalla una lista de *mirrors* (réplicas), es decir, servidores desde donde pueden descargarse los archivos. Seleccionaremos el servidor que esté geográficamente más cerca de nuestra ubicación, y nos aparecerá una página con varias opciones de instalación. La manera más rápida es seleccionar una de las opciones siguientes, que dependerá de nuestro sistema operativo:

- Download R for Linux
- Download R for MacOS X
- Download R for Windows

Los vídeos son útiles

En internet podéis encontrar multitud de vídeos en inglés y en castellano sobre cómo instalar R. Muchos están disponibles en www.youtube.com.

Si tenéis un Mac o una distribución del Linux, deberéis tener en cuenta, además, la versión del sistema operativo que tengáis en vuestro ordenador.

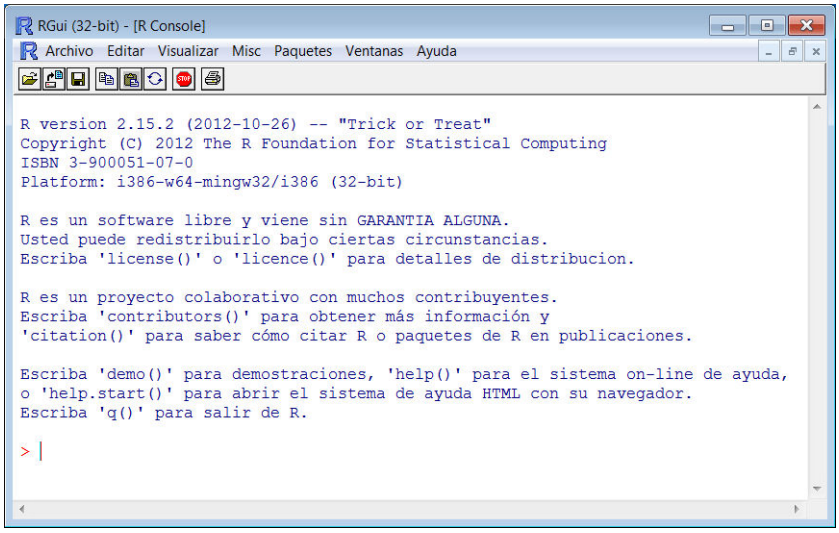
Si suponemos que tenemos un sistema operativo Windows, accedemos a él y descargamos la distribución base, a la que se accede con la opción *install R for the first time*. Seguimos las instrucciones y descargamos el archivo ejecutable, y luego realizamos la instalación sin más complicaciones.

1.3. Iniciar una sesión

Una vez instalado, ya podemos abrir el programa, desde el acceso directo o desde el archivo ejecutable incluido en la carpeta de instalación de R. Sea como fuere, al ejecutar encontraremos la ventana siguiente:

Es importante tener siempre instalada la última versión de R, puesto que las nuevas versiones siempre incluyen mejoras. Además, nos ahorraremos problemas de compatibilidad con los últimos paquetes que instalemos.

Figura 1. Consola de R



Como podemos ver, las instrucciones pueden introducirse directamente en la consola, en el cursor que aparece detrás el símbolo >. Una vez introducidas las instrucciones, si pulsamos *Enter* se ejecutarán los cálculos, igual que con una calculadora:

```
> 2 * 3
[1] 6
```

No obstante, para hacer un análisis largo, lo más conveniente es escribir el conjunto de instrucciones con los cálculos que necesitamos en un archivo de texto, y esto puede hacerse de dos maneras:

- En un *script*, lo que tiene la ventaja que puede abrirse y guardarse como un archivo con formato *.R*, desde la barra de menús del programa. Esta opción, que se muestra en la figura 2, permite ejecutar cada línea de código o toda una selección de líneas mediante la instrucción *CONTROL + R*, y el resultado aparece de color azul en la consola:

Figura 2. Consola de R con editor de textos

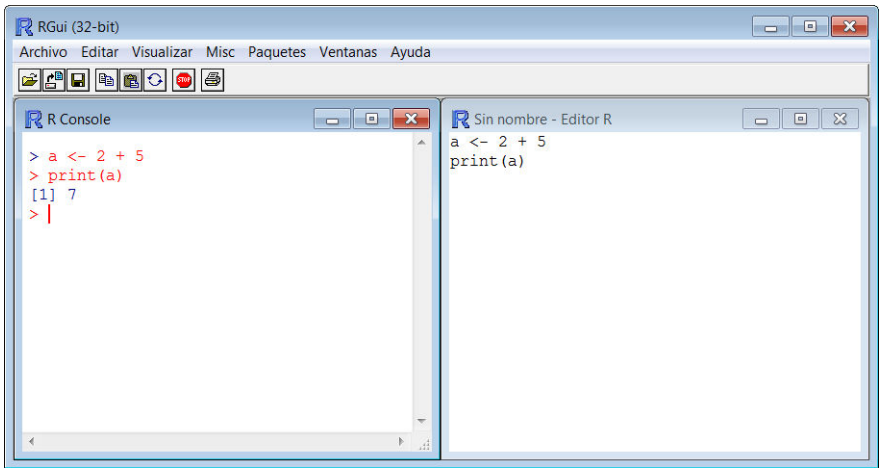


Figura 2

Fijaos que R se basa en la asignación de objetos. Aquí vemos como el resultado de la operación 2 + 5 lo hemos asignado a un objeto al que hemos dado el nombre de `a`.

- En un documento de texto aparte, que una vez finalizado se copia y pega en la consola. El resultado será el mismo que el de la ventana izquierda de la figura 2, es decir, aparecerán las instrucciones y los resultados. Si elegimos esta opción, cualquier editor de textos servirá, aunque aquí recomendamos el editor *gedit*, puesto que destaca la sintaxis de las instrucciones. Más adelante comentaremos como usar RStudio para facilitar nuestro trabajo.

gedit

gedit es un editor de textos compatible con Linux, Mac OS X y Windows. Está diseñado como un editor de textos de propósito general, y enfatiza la simplicidad y facilidad de uso. Además, incluye herramientas para la edición de código fuente y de textos estructurados. Puede descargarse libremente desde la dirección projects.gnome.org/gedit

1.4. Gestión de paquetes

El programa R es modular, es decir, se basa en diferente módulos o paquetes (*packages*) que los usuarios han elaborado. Estos paquetes son unidades específicas que proporcionan funciones y datos para poner en práctica análisis específicos. Actualmente hay tantos paquetes que sería inviable cargarlos todos simultáneamente en una sesión por motivos de memoria. La lista completa de paquetes se encuentra en la página del *CRAN*, en el apartado *Packages*. Allí podremos visualizar una lista de los más de 4.000 paquetes disponibles por orden alfabético con una breve descripción. Es muy útil saber que tenemos a nuestra disposición una lista por categorías: matemáticas, estadística, econometría, etc. Accediendo a cada paquete obtendremos más información sobre este, y también diferentes manuales de uso con las funcionalidades que incluyen.

Antes de intentar implementar nosotros mismos un procedimiento numérico o estadístico complejo, vale la pena comprobar si alguien ya ha hecho el trabajo. Por ello, es muy recomendable visitar el repositorio de paquetes de (http://cran.r-project.org/), en el que podemos buscar paquetes por orden alfabético o por categorías.

En este sentido, cuando se abre una sesión de las funcionalidades básicas (denominadas *base*) se cargan automáticamente, y si el usuario está interesado en una metodología específica tiene que instalar el paquete correspondiente y cargarlo. Esto lo podemos hacer desde el menú de la consola de que se muestra en la figura 1, en la opción *Paquetes*. Esta opción nos permite seleccionar un paquete de la lista que aparece, instalarlo y/o cargarlo. Otra opción es hacer esto manualmente utilizando la consola. Si sabemos el nombre del paquete, para instalarlo introduciremos:

```
>install.packages("paquete")
```

Una vez instalado, al inicio de cada sesión en que necesitemos este paquete será suficiente introducir:

```
>library(paquete)
```

Una vez se ha instalado un paquete, ya queda almacenado en nuestro ordenador, de forma que no hay que volverlo a instalar. Solo tendremos que cargarlo en las sesiones en que lo necesitemos.

Un análisis muy planificado tiene que incluir al principio una lista de todos los paquetes que tienen que cargarse para hacer los análisis que queremos.

1.5. ¡Documentación y ayuda!

Para un usuario novel, la interfaz de R, la sintaxis y en general el modo de uso puede resultar complejo, especialmente en comparación con otros programas estadísticos. No obstante, dispone de muchos manuales y herramientas de apoyo al alcance de los usuarios. Para empezar, en la página del *CRAN* desde la que hemos descargado el archivo de instalación hay una sección llamada *Manuales*. Si accedemos a ella tendremos acceso a los manuales oficiales de R, el primero de los cuales es *An Introduction to R*, un manual completísimo muy bien estructurado que cubre todos los aspectos elementales del programa. Si queremos manuales en otros idiomas, en el enlace *contributed documentation* disponemos de una lista ordenada por idiomas, entre los cuales el castellano.

Además, existe la posibilidad de obtener ayuda específica para cada paquete y función. Simplemente hay que introducir en la consola la instrucción `help`, con el nombre del paquete o la función sobre la que necesitamos ayuda entre los símbolos de paréntesis, y se nos abrirá una ventana con una página completa de ayuda con descripciones y ejemplos.

¡Cuidado con las mayúsculas!

Hay que tener en cuenta que el lenguaje de R distingue entre mayúsculas y minúsculas. De este modo, la instrucción `HELP` resultaría en un mensaje de error.

Por ejemplo si queremos investigar sobre la función que encuentra la media sería suficiente con escribir

```
> help(mean)
```

con lo que obtenemos una descripción detallada de la función, con referencias y pequeños ejemplos.

```
Arithmetic Mean

Description

Generic function for the (trimmed) arithmetic mean.

Usage
mean(x, ...)

## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)

Arguments
```

```
 X

An R object. Currently there are methods for numeric/logical vectors
    and date, date-time and time interval objects. Complex vectors are
     allowed for trim = 0, only.

 trim

the fraction (0 to 0.5) of observations to be trimmed from each end of~
    x~before the mean is computed. Values of trim outside that range
    are taken as the nearest endpoint.

 na.rm

a logical value indicating whether NA values should be stripped before
    the computation proceeds.

 ...

further arguments passed to or from other methods.

Value

If trim is zero (the default), the arithmetic mean of the values in x
    is computed, as a numeric or complex vector of length one. If x is
     not logical (coerced to numeric), numeric (including integer) or
    complex, NA_real_ is returned, with a warning.

If trim is non-zero, a symmetrically trimmed mean is computed with a
    fraction of trim observations deleted from each end before the
    mean is computed.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) The New S
    Language. Wadsworth & Brooks/Cole.
See Also
weighted.mean, mean.POSIXct, colMeans for row and column means.

Examples
x <- c(0:10, 50)
xm <- mean(x)
c(xm, mean(x, trim = 0.10))
```

2. Sintaxis y programación

2.1. Tipos de datos y objetos

R es un lenguaje orientado a objetos o estructuras de datos diferenciados con características específicas. Los tres tipos básicos de datos son:

1) *Datos numéricos:* números reales y complejos (–1, 2, 3).

2) *Caracteres:* cadenas de texto representadas entre comillas ("*x*", "*y*", "*z*").

3) *Datos lógicos:* los que únicamente toman los valores verdadero o falso (*TRUE*, *FALSE*).

Es importante tener en cuenta que R interpreta del mismo modo las comillas simples 'x' y las dobles "x". Por lo tanto, podemos introducir las cadenas de texto de cualquiera de las dos maneras.

Esta orientación a objetos hace indispensable que conozcamos los diferentes tipos de objetos con que podemos trabajar, ya que cada uno tiene características únicas. No obstante, como veremos en este módulo, nos permite que transformamos algunos en otros (por ejemplo, convertir vectores en matrices y viceversa). Las clases principales de objetos están enumeradas y definidas en la Tabla 1.

Tabla 1. Clases de objetos

Objeto	Descripción
Data frame	Estructuras de datos bidimensionales, en los que se recogen diferentes variables por columnas.
Vector	Colección ordenada de datos con una longitud determinada.
Array	Conjunto de datos indexado con dos o más índices, caracterizado por tener dimensión.
Matrix	Caso particular de array cuando hay dos índices, es decir, dos dimensiones (filas y columnas).
List	Objeto que recoge diferentes tipos de elementos (componentes), que pueden ser de clases diferentes.
Function	Pieza de código que puede ser un algoritmo, una función matemática o una estructura lógica.
Factor	Vector que especifica una clasificación discreta de los elementos de otro vector de la misma longitud.

A continuación profundizaremos en cada una de estas estructuras.

2.2. Vectores

2.2.1. Almacenar datos en vectores

Hay dos maneras de introducir la información en el programa. Si introducimos información mediante expresiones sin asignarlas a ningún objeto, estamos utilizando el programa como una simple calculadora. En cambio, si lo almacenamos y le damos un nombre estamos guardando el resultado para utilizarlo posteriormente. Estas dos funcionalidades básicas quedan explicadas a continuación:

1) *Expresión*. La información se introduce en la consola, R la evalúa y la muestra en pantalla, pero no la almacena en la memoria.

```
> 4 + 3
[1] 7
```

2) *Asignación*. En este caso, la expresión se evalúa pero no se muestra, y se almacena con el nombre que escribimos antes del símbolo de asignación <-. Para ver el valor en pantalla, tendremos que introducir el nombre que hemos dado al vector, que es la estructura de datos más simple[1]. La manera más inmediata de introducir los valores en un vector es con la instrucción c, que significa *concatenar*.

Cuestión de memoria

Cuando asignamos un nombre a un objeto, este queda guardado en la memoria temporal. Esto es, podemos recuperar la información de estos objetos siempre que no cerremos la sesión de R.

```
> a <- c(1,2,3,4,5,6)
> a
[1] 1 2 3 4 5 6
```

Si introducimos una asignación entre paréntesis, además de quedar guardada en la memoria temporal también se mostrará en pantalla.

```
> (b <- c(3,6,9))
[1] 3 6 9
```

Una vez tengamos unos cuantos vectores creados, podemos combinarlos de una manera muy flexible. Veamos un ejemplo:

```
> a <- c(1,2,3)
> b <- c(7,8,9)
> ab <- c(a,4,5,6,b)
> ab
[1] 1 2 3 4 5 6 7 8 9
```

El espacio no existe

En el lenguaje de R los espacios no se interpretan. Es decir, las expresiones `a <- 6` y `a<-6` son análogas, y no hay ninguna diferencia entre ellas.

2.2.2. Operaciones básicas

En la Tabla 2 se recogen las operaciones aritméticas básicas tomando como ejemplo los vectores v1 y v2. Estas operaciones están vectorizadas, es decir, se aplican elemento a elemento.

Hay que señalar que en R las comas quedan reservadas para la separación de los diferentes valores o componentes de un mismo objeto o bien para los diferentes parámetros de una función.

Tabla 2. Operaciones aritméticas básicas (I)

```
v1 <- c(1,2,3)
v2 <- c(2,4,6)
```

Descripción	Instrucciones	Resultado
Concatenación	c(v1,v2)	1,2,3,2,4,6
Suma	v1+v2	3,6,9
Resta	v1−v2	−1,−2,−3
Multiplicación	v1*v2	2,8,18
División	v1/v2	0.5,0.5,0.5
Potencia	v1^v2	1,16,729

[1] La función print también muestra el resultado en pantalla.

Cuando se intentan hacer operaciones matemáticas no definidas, como por ejemplo la raíz de un número negativo, aparecerá el valor `NaN`, que significa *Not a Number*. Además, los valores $\pm$ `Inf.` aparecerán cuando el resultado tienda a $\pm\infty$. Veamos más operaciones matemáticas elementales aplicadas elemento a elemento en la Tabla 3, tomando como ejemplo el vector `v3`:

Tabla 3. Operaciones aritméticas básicas (II)

```
v3 <- c(-1,0,1,2)
```

Descripción	Instrucciones	Resultado
Raíz cuadrada	`sqrt(v3)`	`NaN 0.00 1.00 1.41`
Valor absoluto	`abs(v3)`	`1 0 1 2`
Exponencial	`exp(v3)`	`0.36 1.00 2.71 7.38`
Log. base *e*	`log(v3)`	`NaN -Inf 0.00 0.69`
Log. base 10	`log10(v3)`	`NaN -Inf 0.00 0.30`
Factorial (!)	`factorial(v3)`	`NaN 1 1 2`

Tabla 3

Es fundamental tener en cuenta que, en `R`, los decimales se expresan con un punto, y no una coma como se hace en España. Confusiones en este sentido pueden provocar errores.

Con `R` podemos hacer las operaciones estadísticas básicas. En la Tabla 4 se recogen las más usuales, tomando como ejemplo los vectores `v4` y `v5`.

Tabla 4. Operaciones estadísticas básicas

```
v4 <- c(-2,3,5,0,0,3)
v5 <- c(20,14,51,76,21,30)
```

Descripción	Instrucciones	Resultado
Longitud	`length(v4)`	`6`
Máximo	`max(v4)`	`5`
Mínimo	`min(v4)`	`-2`
Suma	`sum(v4)`	`9`
Producto	`prod(v4)`	`0`
Media	`mean(v4)`	`1.5`
Mediana	`median(v4)`	`1.5`
Desviación estándar	`sd(v4)`	`2.58`
Variancia	`var(v4)`	`6.7`
Covariancia	`cov(v4,v5)`	`5.8`
Correlación	`cor(v4,v5)`	`0.09`
Producto escalar	`sum(v4*v5)`	`347`

2.2.3. Secuencias

Una herramienta muy útil es la creación de secuencias, que puede hacerse de diferentes maneras, entre ellas las funciones `seq` (*sequence*), `rev` (*reverse*) y `rep` (*repeat*). La Tabla 5 lo ilustra con algunos ejemplos, partiendo del vector `v6`:

Tabla 5. Secuencias y repeticiones

```
v6 <- c(1,2,3)
```

Instrucciones	Resultado
`rep(v6,2)`	`1 2 3 1 2 3`
`rep(v6,each=2)`	`1 1 2 2 3 3`
`rep(v6,each=2,times=2)`	`1 1 2 2 3 3 1 1 2 2 3 3`
`rep(v6,c(2,3,4))`	`1 1 2 2 2 3 3 3 3`
`1:10`	`1 2 3 4 5 6 7 8 9 10`
`seq(1,10)`	`1 2 3 4 5 6 7 8 9 10`
`10:1`	`10 9 8 7 6 5 4 3 2 1`
`seq(10,1)`	`10 9 8 7 6 5 4 3 2 1`
`seq(4,length=8)`	`4 5 6 7 8 9 10 11`
`seq(0,40,by=5)`	`0 5 10 15 20 25 30 35 40`
`seq(0,1,length=4)`	`0.00 0.33 0.66 1.00`
`seq(from=2,to=8,by=3)`	`2 5 8`
`rev(1:5)`	`5 4 3 2 1`
`2*1:4`	`2 4 6 8`

Tabla 5

El dominio de estas instrucciones nos ahorrará muchas líneas de programación, y nos permitirá programar procedimientos numéricos complejos de una manera simple, compacta y elegante.

2.2.4. Operadores lógicos

R también nos permite operar con vectores lógicos que pueden tomar los valores `TRUE` (verdadero) y `FALSE` (falso). Los operadores principales se muestran en la Tabla 6:

Tabla 6. Operadores lógicos

Descripción	Operador
Más pequeño	`<`
Más grande	`>`
Igual	`==`
No igual	`!=`
Más pequeño o igual	`<=`
Más grande o igual	`>=`
Intersección	`a & b`
Unión	`a|b`
Negación	`!a`

Tabla 6

Podemos interpretar el operador lógico intersección como *a* y *b*, mientras que la unión sería interpretable como *a* o *b*.

Veamos algunos ejemplos del uso de estos operadores:

```
> a <- c(-2,-1,0,1,2)
> a > 2
[1] FALSE FALSE FALSE FALSE FALSE
> a <= -1
[1] TRUE TRUE FALSE FALSE FALSE
> a > 0 & abs(a) == 2
[1] FALSE FALSE FALSE FALSE TRUE
> a > 0 | abs(a) == 2
[1] TRUE FALSE FALSE TRUE TRUE
```

Los símbolos = y ==

Cuando usamos el operador lógico igualdad, es importante tener en mente que se expresa con un doble símbolo de igualdad (==), diferente del símbolo de igualdad simple utilizado en asignaciones.

2.2.5. Indexación

Para acceder a los elementos de un vector existe la posibilidad de *indexar*, es decir, acceder a los valores de un vector especificándolo en una expresión entre corchetes `[]`. Esto lo hacemos indicando la posición de los elementos o bien usando operadores lógicos. Un ejemplo considerando el vector `v7` se recoge en la Tabla 7:

Tabla 7. Indexación de vectores

`v7 <- c(-3,-2,-1,0,1,2,3)` o de manera más compacta `v7 <- c(-3:3)`

Expresión	Resultado	Descripción
`v7[1]`	`-3`	Primer elemento
`v7[2:4]`	`-2 -1 0`	Del segundo al cuarto elemento
`v7[-(2:length(v7))]`	`-3`	Todos menos aquel des del segundo hasta el último elemento
`v7[c(2,4,6)]`	`-2 0 2`	Elementos segundo, cuarto y sexto
`v7[v7>0]`	`1 2 3`	Elementos más grandes que 0
`v7[v7!=0]`	`-3 -2 -1 1 2 3`	Elementos diferentes de 0
`v7[abs(v7)==3]`	`-3 3`	Elementos con valor absoluto 3
`v7[v7>0 & v7!=2]`	`1 3`	Elementos positivos **y** que no sean 2
`v7[v7<0 | v7==3]`	`-3 -2 -1 3`	Elementos negativos **o** que sean 3

Además, la función `which` nos permite extraer la posición ordinal de los elementos descritos en la condición que se ha introducido. En el ejemplo siguiente, en el primer caso los elementos vector iguales a cero son el primero, el tercero y el sexto. En el segundo caso, el único elemento mayor que cero (`a>0`) y menor o igual que dos (`a<=2`) es el cuarto.

```
> a <- c(0,3,0,1,3,0)
> which(a==0)
[1] 1 3 6
> which(a>0 & a<=2)
[1] 4
```

Del mismo modo, las funciones `which.min` y `which.max` nos dan la posición de los valores mínimos y máximos del vector, respectivamente.

2.3. Matrices

2.3.1. Los objetos `array` y `matrix`

A menudo tendremos la necesidad de almacenar los datos en estructuras más complejas que simples vectores. Esto es, podemos necesitar tener una estructura de filas y columnas, y poder hacer operaciones matriciales. En R, hay dos estructuras (tipos de objetos) que satisfacen esta necesidad:

1) Variable indexada (`array`). Estructura que organiza los datos en k dimensiones, y es muy útil para tratar con datos con varias clasificaciones simultáneas. Por ejemplo, la variable v_{ijt} puede representar las ventas de una empresa en qué i es el sector, j la provincia y t el año, y se almacenaría en una estructura array con dimensión $k = 3$.

2) Matriz (`matrix`). Se trata de un caso particular de *array* cuando hay dos dimensiones ($k = 2$), siendo la primera dimensión las filas y la segunda dimensión, las columnas.

En muchos casos no hay diferencia entre los dos tipos de estructuras. No obstante, algunas operaciones como la transposición y algunos tipos de operaciones algebraicas están reservadas solo a las matrices. Por simplicidad, a continuación nos centraremos en la creación y manipulación de estructuras bidimensionales (matrices). La manera más inmediata de crear una matriz es asignando una dimensión a un vector, puesto que, como hemos visto, los vectores tienen longitud y no dimensión:

```
> a <- 1:8
> a
[1] 1 2 3 4 5 6 7 8
> length(a)
[1] 8
> dim(a)
NULL
```

El resultado `NULL`

Siempre que un objeto esté vacío aparecerá el resultado `NULL`. En este caso, al ser `a` un vector y no una matriz, su dimensión no existirá.

2.3.2. Creación de matrices

Supongamos que queremos convertir el vector a en una matriz de dimensión 4×2, es decir, una estructura con 4 filas y 2 columnas. En este caso, podemos asignar una dimensión:

```
> dim(a)<-c(4,2)
```

```
> a
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8
> dim(a)
[1] 4 2
```

Es importante destacar que, por defecto, R siempre empieza a colocar los valores de arriba abajo y de izquierda a derecha, como puede observarse en el ejemplo anterior.

Una manera más directa de crear matrices es con la función matrix. Esta función tiene tres argumentos: el conjunto de valores que compondrán la matriz, el número de filas y el número de columnas:

```
> matrix(1:8,2,4)
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
```

En caso de necesitar una estructura de datos con más de dos dimensiones, habrá que usar la función `array`. La diferencia principal respecto a la función `matrix` es que el número de dimensiones se introduce mediante un vector. A modo de ejemplo, crearemos una estructura $2 \times 2 \times 2$:

```
> array(1:4,c(2,2,2))

, , 1
     [,1] [,2]
[1,]    1    3
[2,]    2    4

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

El reciclaje es importante

Fijaos que, cuando se acaban de colocar todos los elementos del vector 1:4 en la estructura resultante y esta todavía no está completa, se vuelve a empezar hasta que la estructura está completa. Esta funcionalidad se denomina reciclaje.

Una característica interesante de la construcción de matrices es el **reciclaje**, que consiste en que si el número total de elementos de la matriz (filas por columnas) es superior al número de valores iniciales, estos se repiten hasta completar la matriz. En el ejemplo anterior, hemos querido transformar un vector con 4 elementos en una estructura con $2 \times 2 \times 2 = 8$ elementos, esto es, el doble. Por eso, R ha tenido que repetir los elementos del vector inicial dos veces para completar la estructura final. Veamos otro ejemplo de reciclaje:

```
> matrix(1:3,3,3)
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    2    2    2
[3,]    3    3    3
```

Además, hay la opción de que la matriz se construya invirtiendo el orden, es decir, de izquierda a derecha y de arriba abajo. Se consigue añadiendo una opción al final de la función:

```
> matrix(1:8,2,4,byrow=TRUE)
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
```

¿Por filas o columnas?

Utilizando la función `byrow` invertimos el orden de construcción de matrices, empezando por las filas en vez de las columnas, que es la opción por defecto.

También es posible especificar únicamente el número de filas y/o columnas de la matriz que tiene que construirse:

```
> matrix(1:6,nrow=2)
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> matrix(1:4,ncol=2)
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

Es muy fácil combinar escalares, vectores y matrices. La función `rbind` une varias estructuras por filas, mientras que `cbind` hace lo mismo por columnas. Veamos cómo se hace:

```
> a<-matrix(0,2,2)
> a
     [,1] [,2]
[1,]    0    0
[2,]    0    0
> b<-matrix(1,1,2)
> b
     [,1] [,2]
[1,]    1    1

> rbind(a,b)
     [,1] [,2]
[1,]    0    0
[2,]    0    0
[3,]    1    1
```

Como hemos visto, estas dos funciones nos ofrecen muchas posibilidades:

```
> cbind(1,1:4)
     [,1] [,2]
[1,]    1    1
[2,]    1    2
[3,]    1    3
[4,]    1    4
```

Otro ejemplo de reciclaje

En este caso, vemos que unir un escalar y un vector hace que el valor del escalar se repita tantas veces como la longitud del vector. Esto hace que la estructura resultante tenga dos columnas y tantas filas como elementos vectores.

La matriz identidad se crea con la función `diag`, especificando el número de filas y columnas:

```
> diag(2)
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

La matriz identidad

Recordad que la matriz identidad siempre es cuadrada, es decir, tiene el mismo número de filas y de columnas.

Además, la función `diag` también puede usarse para acceder a los elementos de la diagonal principal de una matriz:

```
> a<-matrix(5*1:4,2,2)
> a
     [,1] [,2]
[1,]    5   15
[2,]   10   20
> diag(a)
[1]  5 20
```

2.3.3. Submatrices e indexación

De manera análoga al caso de los vectores, también es posible acceder a subconjuntos de matrices mediante corchetes `[]`. Estos se usan con tres propósitos:

1) Tener acceso a filas y columnas específicas.

2) Crear submatrices.

3) Modificar directamente partes de una matriz.

Esto nos ofrece una gran potencialidad para realizar todo tipo de operaciones con matrices, y hacer que estas sean fácilmente replicables, modificables y combinables. A partir de la matriz `a`, la Tabla 8 muestra ejemplos de creación de submatrices.

```
> a <- matrix(1:9*3,3,3)
```

```
> a
     [,1] [,2] [,3]
[1,]    3   12   21
[2,]    6   15   24
[3,]    9   18   27
```

Tabla 8. Indexación de matrices

Expresión	Resultado	Descripción
`a[,3]`	`21 24 27`	Todas las filas, 3ª columna
`a[2,]`	`6 15 24`	2ª fila, todas las columnas
`a[-1,-3]`	`6 15` `9 18`	Todas las filas menos la 1ª y todas las columnas menos la 3ª.
`a[2:3,1]`	`6 9`	De la 2ª a la 3ª fila, primera columna

Es interesante comprobar que las submatrices creadas, en caso de tener una sola fila o columna, se degradan a vectores, por lo que pierden la dimensión[2]. Para que estas submatrices sigan teniendo dimensión, tenemos que especificarlo desactivando la opción `drop`:

```
> a
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> b <- a[,1,drop=FALSE]
> b
     [,1]
[1,]    1
[2,]    2
> dim(b)
[1] 2 1
```

Veamos ahora como se haría para introducir elementos nuevos en una matriz. Supongamos, por ejemplo, que queremos sustituir la primera columna de valores por una columna de ceros:

```
> a<-matrix(1:8,2,4)
> a[,1]<-0
> a
     [,1] [,2] [,3] [,4]
[1,]    0    3    5    7
[2,]    0    4    6    8
```

Modificando una matriz

En este caso, hemos utilizado `a[,1]<-0` para transformar la primera columna, inicialmente compuesta por los elementos 1 y 2, en una columna con dos ceros.

[2] En inglés, este hecho se denomina *deprecation*.

2.3.4. Operaciones matriciales básicas

R ofrece muchas posibilidades para aplicar operaciones y funciones a vectores y matrices[3]. En la Tabla 9 se incluyen las funciones básicas aplicables en matrices:

Tabla 9. Funciones con matrices

Descripción	Función
Dimensiones	`dim`
Traspuesta	`t`
Determinante	`det`
Inversa	`solve`
Valores y vectores propios	`eigen`
Valores singulares	`svd`
Factorización de Choleski	`chol`
Factorización QR	`qr`
Rango	`rank`
Número de filas	`nrow`
Número de columnas	`ncol`
Suma de las filas	`rowSums`
Suma de las columnas	`colSums`
Media de las filas	`rowMeans`
Media de las columnas	`colMeans`

Para finalizar el apartado dedicado a matrices, vale la pena mencionar que hay algunas funciones que permiten, por un lado, identificar objetos y, por el otro, transformarlos en objetos de otras clases. Este primer ejemplo es ilustrativo:

```
> a <- c(1,3,6)
> is.matrix(a)
[1] FALSE
> is.vector(a)
[1] TRUE
> class(a)
[1] "numeric"
```

La familia de funciones `is.` responde a la pregunta de si se trata de un objeto determinado por un valor lógico, mientras que la función `class` devuelve la clase del objeto. Finalmente, la familia de funciones `as.` se usa para transformar un tipo de objeto en otro. El ejemplo siguiente lo muestra:

Convertir matrices en vectores

En este caso, hemos conseguido transformar la matriz `A` en el vector `a`.

```
> (A <- matrix(1,2,2))
     [,1] [,2]
[1,]    1    1
[2,]    1    1
> (a <- as.vector(A))
[1] 1 1 1 1
```

[3] El módulo 2 de este manual incluye un apartado de álgebra vectorial y matricial dedicado a operaciones algebraicas específicas.

2.4. Funciones

Una herramienta muy útil y poderosa es el comando `function`, que también es una clase de objeto. Podemos crear desde simples funciones matemáticas a algoritmos complejos con los que podemos hacer multitud de cálculos de una manera estructurada y sintética. La forma de función más sencilla tiene la estructura siguiente:

```
nombre <- function (argumentos) expresión
```

Como ejemplo, supongamos que queremos estudiar la función matemática $y = x^3$ y evaluarla en el vector de valores iniciales $x_0 = (-4, -3, \ldots, 3, 4)$. Los haremos creando nuestra propia función `cubic` y evaluándola en el vector de valores iniciales `x0`:

```
> cubic <- function(x) x^3
> x0 <- (-4:4)
> cubic(x0)
[1] -64 -27  -8  -1   0   1   8  27  64
```

Un ejemplo de una función con más de un argumento es el de la fórmula financiera del interés simple $C_f = C_i\,(1+rt)$. Según esta fórmula, el capital final (C_f) que tiene que devolverse en una operación financiera con un importe inicial C_i depende de la duración de la operación en años (t) y del tipo de interés en tanto por un (r). ¿Qué importe final resulta de una operación a nueve meses ($t = 0.75$) con un tipo nominal del 5 % ($r = 0.05$) y un capital inicial $C_i = 1000$?

```
> op.fin <- function(Ci,r,t) Ci*(1+r*t)
> op.fin(1000,0.05,0.75)
[1] 1037.5
```

Para funciones más complejas a menudo es necesaria más de una línea para introducir las fórmulas. En este caso, suponiendo que necesitemos n líneas, la estructura será la siguiente:

```
nombre <- function(argumentos){
expresión 1
...
expresión n
return(resultado)
}
```

Con el objeto de ilustrar este tipo de estructura crearemos una función para calcular la covarianza muestral. Esto es algo innecesario; podríamos usar la función `cov` que nos proporciona R. De todos modos, lo hacemos para que sirva de ejemplo de funciones más elaboradas:

$$Cov(X,Y) = \frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})\,(y_i - \bar{y})$$

```
> mi.covar <- function(x,y){
+     N <- length(x)
+     demean.x <- x-mean(x)
+     demean.y <- y-mean(y)
+     sumat <- sum(demean.x*demean.y)
+     return(sumat/(N-1))
+ }
```

Funciones multiusos

Crear funciones propias nos será muy útil para llevar a cabo un conjunto de cálculos que tengamos que usar más de una vez, de manera estructurada y ordenada.

2.5. Ciclos y condicionales

R ofrece una serie de herramientas para hacer asignaciones múltiples y condicionales. `ifelse` permite hacer operaciones elemento por elemento a un vector sujeto a una condición. Supongamos que queremos aplicar la siguiente transformación a un vector:

$$f(x) = \begin{cases} log(x) & \text{si} \quad x > 0 \\ 0 & \text{si} \quad x \leq 0 \end{cases}$$

```
> a <- c(-1,0,2,5)
> ifelse(a>0,log(a),0)
[1] 0.0000000 0.0000000 0.6931472 1.6094379
```

Ciclos y condicionales

Para usar eficientemente ciclos y condicionales es fundamental dominar los operadores lógicos, explicados en este módulo.

Uno de los componentes fundamentales de cualquier lenguaje de programación es el uso de estructuras iterativas, es decir, de funciones que repiten una o más expresiones en un ciclo. El comando principal que realiza esta función es `for`. La estructura general de esta función es, para un ciclo de *n* iteraciones:

```
for (i in 1:n){
expresión(ns)
...
}
```

Un ejercicio sencillo para ilustrar un ciclo es la suma acumulativa. Partiendo de un vector $\vec{v} = (v_1, \cdots, v_n)$, crearemos un vector $\vec{s} = (s_1, \cdots, s_n)$ en el que $s_j = \sum_{i=1}^{j} v_i$. Antes de implementar el ciclo crearemos un vector *s* vacío que iremos completando iterativamente:

```
> v <- 1:10
> n <- length(v)
> s <- rep(0,n)

> for (i in 1:n){
+     p <- v[1:i]
+     s[i] <- sum(p)
+ }

> print(s)
 [1]  1  3  6 10 15 21 28 36 45 55
```

Ciclos iterativos

Utilizaremos los ciclos cuando queramos trabajar con bucles o realizar cálculos iterativos.

Obviamente, en un análisis real no usaremos este cálculo iterativo, puesto que la función implementada `cumsum` ya nos ofrece esta funcionalidad de una manera más eficiente.

2.6. La familia `apply`

En la práctica, es importante no hacer un uso excesivo de los ciclos iterativos, puesto que no son eficientes en cuanto al uso de la memoria y pueden retardar el tiempo de análisis que se requiera. Por eso, la norma es que se crearán estructuras iterativas siempre que no haya alternativas, y el grupo de funciones `apply` es una de ellas.

Para ilustrar este concepto, supongamos que tenemos una matriz A en la que las filas corresponden a observaciones y las columnas son las variables X, Y y Z:

$$A = \begin{pmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ \vdots & \vdots & \vdots \\ x_n & y_n & z_n \end{pmatrix}$$

La función `apply` nos permite aplicar cualquier operación a las filas y a las columnas, independientemente. Supongamos que queremos aplicar la operación $\sqrt{x_i^2 + y_i^2 + z_i^2}$ a cada observación (es decir, a cada fila). El primer paso será definir esta operación (que denominaremos `hipot`), y el segundo aplicarla a las filas de la matriz A:

La función `apply`

La instrucción `apply(A,1,hipot)` se lee como aplica a la dimensión 1 (filas) de la matriz A la operación `hipot`. El resultado será un vector con n elementos.

```
> hipot <- function(x) sqrt(sum(x*x))
> apply(A,1,hipot)
```

Si quisiéramos un vector con la media aritmética de cada columna, es decir $(\bar{x},\bar{y},\bar{z})$, tendríamos que introducir:

```
> apply(A,2,mean)
```

Para explicar la funcionalidad de la función de la misma familia `tapply`, es necesario introducir los **factores**, que son una clase de objeto que establece una clasificación discreta de una o más variables. Supongamos que queremos estudiar la estatura de los alumnos de una escuela, para lo que disponemos de dos vectores: uno con la estatura de cada estudiante y otro con el género (masculino *m* y femenino *f*):

```
> altura <- c(131,125,126,140,152,119)
> genero <- c('m','m','f','m','f','f')
```

Introducción de caracteres

Al introducir un vector de caracteres, como `género`, los valores del vector tienen que ir entre comillas, para lo que son válidos los símbolos `"` y `'` indistintamente.

En este caso es conveniente convertir el vector `genero` en un factor con dos categorías: masculina y femenina. Para ello, usamos la función `factor` para obtener un vector de factores con las categorías *f* = 1 y *m* = 2, que llamaremos `f.genero`. Veamos que son objetos de diferente clase:

```
> f.genero <- factor(genero)
> class(genero)
[1] "character"
> class(f.genero)
[1] "factor"
```

La instrucción `levels` devuelve las categorías que el vector incluye:

```
> levels(f.genero)
[1] 'f' 'm'

> cbind(f.genero,altura)
     f.genero altura
[1,]        2    131
[2,]        2    125
[3,]        1    126
[4,]        2    140
[5,]        1    152
[6,]        1    119
```

Un análisis estadístico de la variable *altura* tendrá que diferenciar entre *f* y *m*, para lo que se usa la función `tapply`. Calcularemos la media aritmética y la desviación estándar de la altura diferenciando por género:

```
> tapply(altura,f.genero,mean)
       f        m
132.3333 132.0000

> tapply(altura,f.genero,sd)
        f         m
17.387735  7.549834
```

La función `tapply`

Utilizaremos la función `tapply` cuando queramos aplicar un cálculo a una variable diferenciando por grupos o segmentos, esto es, según las categorías de un factor.

2.7. Bases de datos

Una base de datos (*data frame*) es similar a una matriz, puesto que es una estructura bidimensional en la que las filas son las observaciones y las columnas son las variables, cada una con un nombre específico. Es muy recomendable tener las variables almacenadas en este formato por varios motivos:

- Es muy fácil extraer subgrupos (*subsets*) de las variables utilizando operadores lógicos.
- Muchas funciones estadísticas, como los estimadores econométricos, solo admiten datos en este formato.
- Es posible extraer variables de la base de datos mediante la función `attach`.

Hay diferentes maneras de crear una base de datos. La primera es importando datos externos de un archivo de texto limpio[4]. La segunda es agrupando vectores existentes, como mostramos en el siguiente ejemplo (continuación del anterior):

```
> g <- c('m','m','f','m','f','f')   # Género
> v1 <- c(131,125,126,140,152,119)  # Altura
> v2 <- c(48,53,45,40,49,50)        # Peso
>
> datos <- data.frame(genero=g,altura=v1,peso=v2)
> datos
  genero altura  peso
1      m    131   48
2      m    125   53
3      f    126   45
4      m    140   40
5      f    152   49
6      f    119   50
```

Incorporar comentarios mediante el símbolo #

Muy a menudo nos interesará introducir aclaraciones y comentarios en nuestras líneas de código para comprenderlo mejor. Para que R no los interprete y dé un error, los introduciremos después del símbolo #.

Como vemos, en la primera columna se identifica cada elemento de la muestra, mientras que cada columna está encabezada por el nombre de la variable. Es importante destacar que, alternativamente, la función `data.frame` también admite una matriz.

Supongamos que queremos hacer un análisis solo de los chicos (*m*) que pesan menos de 50 kilos. Para ello, es muy práctico crear otra base de datos que sea un subgrupo de la base de datos original. Denominaremos a este subgrupo `datos2`, y como vemos solo incluye dos observaciones, las que satisfacen las condiciones establecidas:

La función `subset`

La función `subset` nos permite crear una nueva base de datos que contenga una selección condicional de observaciones de la base de datos original.

```
> datos2<-subset(datos,genero=="m" & peso<50)
> datos2
  genero altura  peso
```

[4] Más adelante se explica este procedimiento con detalle.

```
1    m    131   48
4    m    140   40
```

Si queremos operar con las variables incluidas en una base de datos, tenemos que referenciarlas con el símbolo $, ya que las variables no están en el espacio de trabajo individualmente, solo como elementos de una base de datos. Por ejemplo, si necesitamos la correlación entre la altura y el peso, tenemos que introducir la instrucción siguiente:

```
> cor(datos$altura,datos$peso)
[1] -0.3202351
```

Esto puede llegar a ser bastante pesado. Por eso, si la intención es hacer operaciones con las variables de la base de datos, es recomendable *verterlas* previamente en el espacio de trabajo mediante la función `attach`, con lo que ya podremos hacer referencia a las variables individualmente:

```
> attach(datos)
> cor(altura,peso)
[1] -0.3202351
```

La función `attach`

Esta función es aplicable cuando queremos realizar operaciones y cálculos individuales con las variables incluidas en una base de datos.

2.8. Listas

Las listas (*lists*) son comparables a un cajón de sastre, puesto que permiten almacenar objetos de clases diferentes en una misma estructura. Esto es muy útil cuando un mismo análisis tiene como resultado diferentes objetos dispares, por ejemplo vectores o matrices de longitudes o dimensiones diferentes. Supongamos que queremos almacenar en una estructura la matriz *A*, su descripción, su inversa y el determinante:

```
> descr <- c("Matriz A")
> A <- matrix(c(3,5,6,1),2,2)
> inv <- solve(A)
> deter <- det(A)
```

Las listas

Esta clase de objeto nos permitirá agrupar en una única estructura elementos que son, a la vez, objetos de clases diferentes.

La forma de almacenarlo es mediante la función `list`. Para acceder a los diferentes elementos de una lista por su nombre, usaremos el símbolo $ de la siguiente manera:

```
> MatA <- list(descripcion=descr,
+              matriz=A,
+              inversa=inv,
+              deter=deter)

> MatA$descripcion
```

```
[1] "Matriz A"

> MatA$matriz
     [,1] [,2]
[1,]    3    6
[2,]    5    1

> MatA$inversa
            [,1]       [,2]
[1,] -0.03703704  0.2222222
[2,]  0.18518519 -0.1111111

> MatA$deter
[1] -27
```

Los símbolos > y + en la consola

Como hemos visto, en la consola tenemos que introducir cada línea de código después del símbolo >. No obstante, algunas instrucciones de código son tan largas que ocupan más de una línea, como en el caso de la lista `MatA`. En este caso, en la línea siguiente, R nos mostrará el símbolo + para recordarnos que la instrucción de la línea previa está incompleta.

También se puede acceder a los elementos de una lista no por el nombre, sino ordinalmente. Para ello, usaremos el símbolo `[[]]`. Veamos una comparación de las dos maneras de acceder a los elementos de la lista:

```
> MatA$matriz
     [,1] [,2]
[1,]    3    6
[2,]    5    1
> MatA[[4]]
[1] -27
```

Diferentes maneras de acceder a los elementos de una lista

Fijaos que, en este ejemplo, las instrucciones `MatA$matriu` y `MatA[[2]]` darían el mismo resultado.

2.9. Lectura y escritura de ficheros csv

Muchas veces encontraremos datos almacenados en ficheros de texto y necesitaremos importarlos a R para poder tratarlos estadísticamente. De los muchos formatos existentes comentaremos uno muy utilizado en varios repositorios de datos y presente también como opción en muchas hojas de cálculo; se trata del formato csv (*Comma Separated Values*) y comentaremos la función *read.csv*.

El contenido de un fichero csv (que es un fichero de texto) es como el siguiente (en un sencillo ejemplo):

```
Nom: Nombre de la persona
Ciu: Ciudad de nacimiento
Alt: Altura en metros
Nom;Ciu;Alt
David;Palma;1,75
María;Cáceres;1,77
```

Como podemos observar puede haber algunas filas con la descripción del contenido de las variables (en este caso las tres primeras filas), después tenemos una fila con los nombres de las variables (Nom;Ciu;Alt: es el llamado *header*) y después propiamente los datos, que en este caso usan el punto y coma (";") como separador de las variables y la coma (",") como separador decimal. Además hay que tener en cuenta la codificación del fichero para que lea correctamente los acentos y los caracteres especiales (en este caso supondremos que está escrito en UTF-8). Supondremos que el nombre del fichero es NCA.csv.

Las instrucciones que usaremos para leer el fichero serán *setwd* (*Set working directory*) para indicar el directorio de trabajo (hay que escribir la ruta entera entre comillas) y *read.table* de esta forma (o bien escribiendo directamente el fichero con toda la ruta entera) por lo que la información del fichero se copia al *data.frame* de nombre *test*:

```
> setwd("Directorio")
> test<-read.table("NCA.csv", header=TRUE,
+           sep=";", dec=",", skip=3,
+           fileEncoding = "UTF-8")
```

Observad que "skip" indica que las tres primeras filas son información y no datos, y cómo indicamos el separador decimal y el separador de variables.

Os recomendamos leer atentamente la documentación de la función para ver todas las opciones.

Más adelante explicaremos cómo importar datos con RStudio.

2.9.1. Inspección de los datos leídos

Una vez leídos nuestros datos del fichero (y almacenados en *test*) es recomendable realizar una primera inspección de las variables, su tipo y su contenido; pueden ayudarnos las siguientes instrucciones:

- *str(test*) muestra la estructura de la tabla leída (nombres de las variables y tipos).
- *names(test)* muestra las variables.
- *levels(factor1*) muestra los niveles de una variable categórica (de nombre *factor1)*.
- *head(test,n=5*) muestra las 5 primeras filas del *data.frame* de nombre *test* o de una variable cualquiera.
- *tail(test,n=5)* muestra las 5 últimas filas del *data.frame*.

En el caso del fichero NCA.csv tenemos:

```
> str(test)
'data.frame': 2 obs. of  3 variables:
 $ Nom: Factor w/ 2 levels "David","María": 1 2
```

```
 $ Ciu: Factor w/ 2 levels "Cáceres","Palma": 2 1
 $ Alt: num  1.75 1.77

> names(test)
[1] "Nom" "Ciu" "Alt"

> levels(test$Ciu)
[1] "Cáceres" "Palma"


> head(test)
    Nom     Ciu  Alt
1 David   Palma 1.75
2 María Cáceres 1.77
```

Si efectuamos alguna modificación en nuestra base de datos podemos guardar el resultado en un fichero para aprovechar el trabajo más adelante; para hacerlo simplemente hay que usar por ejemplo la función *write.csv*, indicando el *data.frame* que queremos exportar y el fichero de destino.

```
> write.csv(test,"NCA2.csv", row.names = FALSE)
```

2.10. Fusión de bases de datos

A veces necesitaremos manipular datos recogidos en diferentes ficheros. Veremos un caso muy sencillo a continuación: imaginemos por ejemplo que tenemos otro fichero NS.csv que contiene el nombre y el sueldo de las personas involucradas; para poder tener en un solo *data.frame* toda la información lo que haremos será leer el segundo fichero y después usaremos la función *merge* para integrar las dos bases de datos según el campo clave común, que en este caso será "Nombre". Procederemos así:

Primero leemos el fichero y guardamos el resultado en `test2`:

```
> test2<-read.table("NS.csv", header=TRUE,
+                sep=";", dec=",", skip=2,
+                fileEncoding = "UTF-8")
```

Mostramos el contenido de `test2`:

```
> test2
    Nom  Sueldo
1 María     2500
2 David     2400
```

Ahora combinamos las dos bases de datos con *merge* y obtenemos el siguiente resultado:

```
> merge(test,test2,by="Nom")
  Nom          Ciu          Alt        Sueldo
1 David        Palma        1.75       2400
2 María        Cáceres      1.77       2500
```

3. Rstudio y R Markdown

3.1. Introducción

Como ya se ha comentado, R es un lenguaje muy adecuado para trabajar temas estadísticos y de probabilidad, pero, además de hacer los cálculos, es muy importante trabajar en un entorno amigable y poder también realizar una presentación adecuada y eficiente de los resultados y es en este momento en el que el popular IDE (*Integrated Development Environment*) llamado RStudio puede ayudarnos de una manera muy sencilla y cómoda. Sin entrar en detalles técnicos, RStudio es un entorno integrado que permite (además de otras muchas opciones) tener en un solo documento las instrucciones de R, los resultados que proporciona R (incluyendo los gráficos), nuestros comentarios y exportarlo todo en forma de documento Word o PDF. Para que os hagáis una idea podéis mirar las siguientes imágenes donde se ve el aspecto del código introducido en RStudio y el resultado obtenido (en este caso exportado a PDF usando la opción de fichero *R Markdown*):

Figura 3. Código en *RStudio*

```
---
title: "Intro a RStudio"
author: "XXX"
date: "XX/XX/XXX"
output:
  pdf_document: default
  html_document: default
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
```

Título de sección

Primer ejemplo

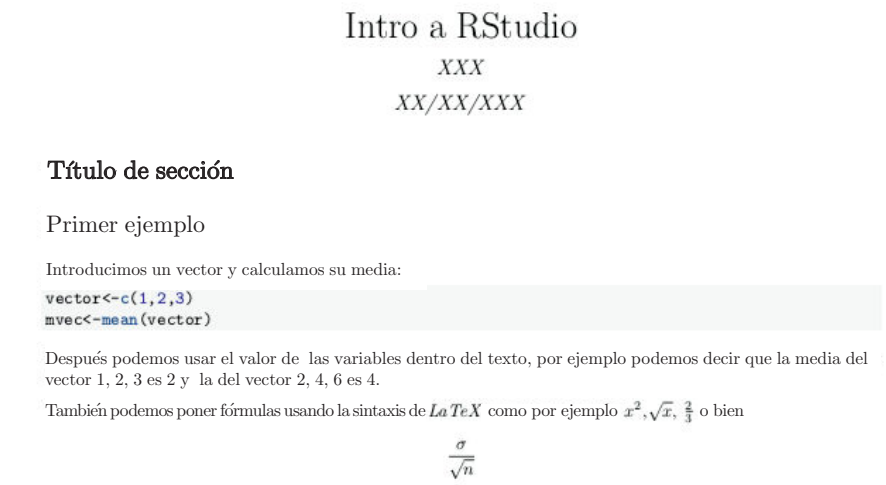
Introducimos un vector y calculamos su media:

```{r}
vector<-c(1,2,3)
mvec<-mean(vector)
```

Después podemos usar el valor de las variables dentro del texto, por ejemplo podemos decir que la media del vector `r vector` es `r mvec` y la del vector `r 2*vector` es `r mean(2*vector)`.

También podemos poner fórmulas usando la sintaxis de *LaTeX* como por ejemplo x^2,$\sqrt{x}$, $\frac{2}{3}$ o bien
\[\frac{\sigma}{\sqrt{n}}\]
```

Figura 4. Resultado en pdf



Intro a RStudio

XXX

XX/XX/XXX

Título de sección

Primer ejemplo

Introducimos un vector y calculamos su media:

```
vector<-c(1,2,3)
mvec<-mean(vector)
```

Después podemos usar el valor de las variables dentro del texto, por ejemplo podemos decir que la media del vector 1, 2, 3 es 2 y la del vector 2, 4, 6 es 4.

También podemos poner fórmulas usando la sintaxis de *LaTeX* como por ejemplo $x^2, \sqrt{x}, \frac{2}{3}$ o bien

$$\frac{\sigma}{\sqrt{n}}$$

Observad que *RStudio*, a partir de un fichero *R Markdown*, ejecuta los comandos de R, los reproduce (opcionalmente) en el documento final e inserta en el documento los resultados de los comandos, todo con opciones relativamente sencillas para controlar el formato del documento, sin tener que recorrer cada vez a copiar-pegar o a capturar pantallas. Es evidente que cada vez que "compilamos – **knit**" el fichero *R Markdown* dentro de *Rstudio*, los resultados se actualizan y vuelve a generarse el fichero actualizado.

A continuación comentaremos la instalación de RStudio y el uso básico de ficheros *R Markdown*.

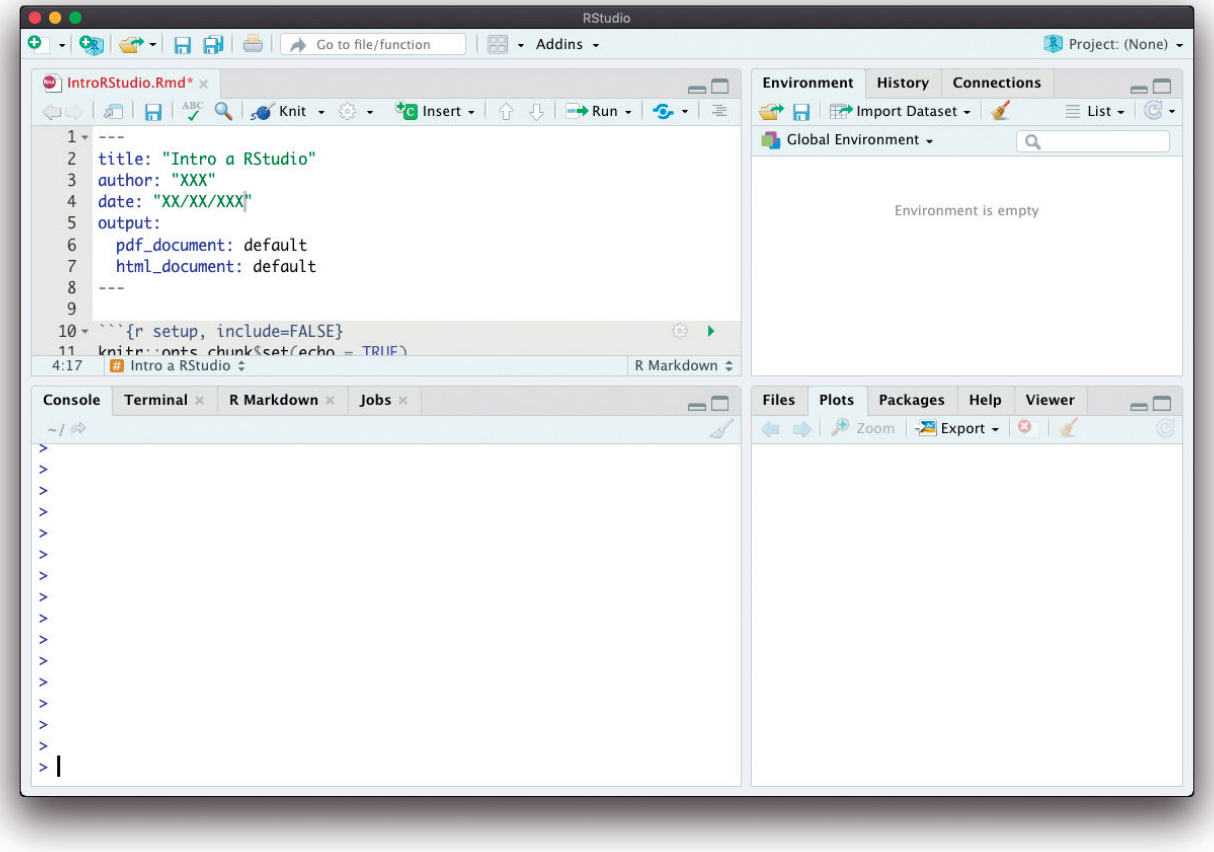
3.2. Instalación

Se descarga desde la página https://rstudio.com/products/rstudio/ (la versión *Open Source*) según vuestro sistema operativo y se siguen las instrucciones de instalación. Lo mejor es hacerlo después de tener R instalado porque, evidentemente, trabajan conjuntamente.

3.3. Primeros pasos

Cuando entremos a *RStudio* encontraremos una pantalla dividida en cuatro secciones:

Figura 5.



1) En la parte superior izquierda encontramos el editor en el que podremos trabajar con diferentes tipos de fichero y lenguajes de programación (no solo R o *R Markdown*, también C++, Python...)

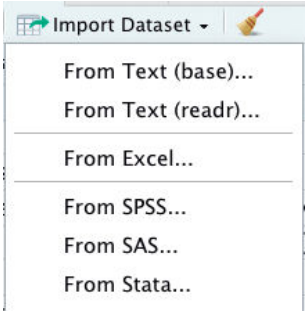
2) En la parte superior derecha encontramos diferentes pestañas, entre las que destacamos la de *Environment*, donde se visualizarán las variables y los *data.sets* que estamos utilizando (además de las opciones de importar ficheros –csv y Excel también– y grabar el entorno).

3) La parte inferior izquierda tiene, entre otros, la *Console* del programa que usamos, en este caso R, para ejecutar comandos directamente y la pestaña correspondiente al tipo de fichero que tengamos abierto, en el supuesto de que nos ocupa *R Markdown*.

4) La parte inferior izquierda contiene un visualizador para los gráficos, para la ayuda...

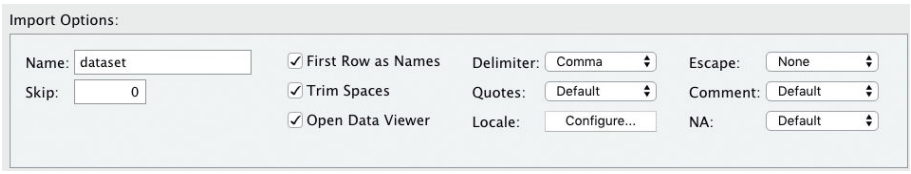
Hay que destacar en la parte superior derecha las opciones de importación de ficheros tanto Excel como de tipo texto:

Figura 6.



En particular, dentro de la opción *From Text (readr)...* encontramos un desplegable que permite, una vez dado el nombre del fichero, especificar las opciones de lectura explicadas anteriormente (con un poco de práctica y con ficheros sencillos vale la pena aprender los correspondientes comandos):

Figura 7.



Además en la barra de menús propia del programa encontramos las opciones habituales para editar código y, además, dentro de ***Tools*** tendremos todo lo relativo a la gestión de los paquetes (o ***packages***).

3.4. R Markdown: primeros pasos

Como ya se ha comentado, *R Markdown* proporciona una manera relativamente sencilla de integrar los comandos, los resultados, los gráficos y los comentarios de nuestro trabajo con R.

Para empezar, en RStudio escogeremos *File->New File->R Markdown*, por lo que después de indicar el título, el autor o la autora y si queremos generar el resultado final en HTML, PDF (en caso de disponer de una instalación de TeX) o Word, nos aparece un modelo de fichero ya preparado para ser generado y comprobar que todo está correctamente instalado. Para hacerlo escogeremos la opción

```
Knit (y la flecha que hay al lado)
```

indicando el tipo de fichero que queremos generar, por ejemplo Word (primero nos pedirá guardar el fichero en el disco, con la extensión **Rmd**). A continuación, veremos actividad en la consola *R Markdown* y se nos creará el correspondiente fichero Word con las instrucciones y los resultados.

3.5. R Markdown: el lenguaje

R Markdown ofrece diferentes posibilidades de una forma relativamente sencilla de usar. Indicaremos a continuación las más habituales y suficientes para hacer un pequeño informe estadístico. Escribiremos en la columna de la izquierda las instrucciones usadas y a la derecha el resultado en Word (a pesar de que en PDF o HTML son similares), después de haber generado el correspondiente fichero con ***Knit***.

Para más información podéis consultar la *R Markdown Reference Guide* (https://rstudio.com/wp-content/uploads/2015/03/rmarkdown-reference.pdf) o la correspondiente *Cheat-Sheet* (desde https://rstudio.com/resources/cheatsheets/).

Secciones y niveles de texto

Tabla 10.

R Markdown	Resultado
# Sección de nivel 1	**Sección de nivel 1**
## Sección de nivel 2	**Sección de nivel 2**

Se usa por lo tanto # o ##...

Listas numeradas

Tabla 11.

R Markdown	Resultado
1. Un	1. Un
1. Dos	2. Dos
1. Tres	3. Tres

Observad que no es necesario poner los números "reales": es suficiente poner "1.".

Listas alfabéticas

Tabla 12.

R Markdown	Resultado
a. Un	a) Un
a. Dos	b) Dos
a. Tres	c) Tres

Como antes, basta con poner siempre "a)" para generar la correspondiente lista.

Otras listas

Tabla 13.

R Markdown	Resultado
+ Un	• Un
+ Dos	• Dos
+ Tres	• Tres

Texto en cursiva y negrita

Tabla 14.

R Markdown	Resultado
Hola	*Hola*
Hola	**Hola**

Código R

Podemos introducir código apropiado escrito en R (los llamados ***chunks***) de dos maneras:

En un bloque separado sin enseñar las instrucciones

Tabla 15.

R Markdown	Resultado
``r echo=FALSE vec<-c(1,2,3) mean(vec)```	## [1] 2

Observad que hay que usar 3 ``` para abrir y cerrar el bloque.

En un bloque separado mostrando las instrucciones

Tabla 16.

R Markdown	Resultado
``{r} vec<-c(1,2,3) mean(vec)```	vec<-c(1,2,3) mean(vec) ## [1] 2

Observad que se muestran las instrucciones y el resultado precedido de ##.

Dentro de la misma línea y mezclado con el texto

Tabla 17.

R Markdown	Resultado
La media de 1, 2, 3 vale `r mean(3,4,5)`	La media de 1, 2, 3 vale 3

Aquí con una "`" es suficiente y observad cómo obtenemos directamente el resultado de la media, sin tener que calcularlo ni tener que copiar el resultado.

Combinación de los dos sistemas para pasar variables al texto

Tabla 18.

R Markdown	Resultado
``{r} vec<-c(2,4,6) mv<-mean(vec) ``` La media del vector `r vec` es `r mv` y la del vector `r vec-mv` es `r mean(vec-mv)`	vec<-c(2,4,6) mv<-mean(vec) La media del vector 2, 4, 6 es 4 y la del vector –2, 0, 2 es 0

Observad cómo se definen y se usan las variables. Dentro de los bloques de R podemos definir funciones, variables, hacer operaciones, bucles... De paso hemos comprobado que la media del vector *vec-mv*, obtenido restando a todos los componentes de *vec* la media del mismo vector, es 0.

Fórmulas matemáticas

En el fichero *R Markdown* podemos escribir fórmulas matemáticas usando la misma sintaxis que LaTeX; a continuación damos algunos ejemplos sencillos de como quedan las fórmulas en el documento generado en Word:

Tabla 19.

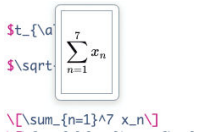
R Markdown	Resultado
`$x^2$`	x^2
`$t_{\alpha}$`	t_α
`$\sqrt{x}$`	$\sqrt{x}$

y otros más complicados combinando las anteriores:

Tabla 20.

R Markdown	Resultado
`\[\sum_{n=1}^7 x_n\]`	$\sum_{n=1}^{7} x_n$
`\[\dfrac{1}{n-1}\sum_{k=1}^{10}\] \[\frac{x^2}{\sqrt{k+2}}\]`	$\frac{1}{n-1}\sum_{k=1}^{10}\frac{x^2}{\sqrt{k+2}}$
`\[\left( \frac{2}{3} \right)\]`	$\left(\frac{2}{3}\right)$

Observad que en el propio editor de *RStudio* puede verse el resultado de las fórmulas, simplemente poniendo la marca del ratón encima:



Esta manera de escribir fórmulas puede parecer muy complicada al comienzo pero con un poco de práctica supone un gran ahorro de tiempo.

Ejemplo final

Reproducimos a continuación un fichero *R Markdown* que podría servir como modelo simplificado de entrega de un informe estadístico y el resultado final que se obtendría en Word:

````
---
title: "Prueba"
author: "XXX"
date: ""
output: word_document
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
```

# Pregunta 1
Dados los valores 2, 4, 5, se pide

1. Calculad su media.
1. Calculad su desviación típica.

## Solución

1. Introducimos los datos y hacemos los cálculos (como
    estamos dentro de una lista, hay que poner **exactamente
    cuatro** espacios al comienzo de la primera línea del
    bloque de R para respetar la numeración)

     ```{r}
vec<-c(2,4,5)
mvec<-mean(vec)
svec<-sd(vec)
```
La media es `r mvec`. Recordamos que se calcula como
\[\bar{x}= \frac{\sum_{i=1}^n x_i}{n}\]

1. La desviación típica es `r svec`.
````

Después de realizar *Knit to Word* obtenemos lo siguiente:

Prueba

XXX

Pregunta 1

Dados los valores 2, 4, 5, se pide

1) Calculad su media.

2) Calculad su desviación típica.

Solución

1. Introducimos los datos y hacemos los cálculos (como estamos dentro de una lista, hay que poner **exactamente cuatro** espacios al comienzo de la primera línea del bloque de R para respetar la numeración)

```
vec<-c(2,4,5)
mvec<-mean(vec)
svec<-sd(vec)
```

La media es 3.6666667. Recordamos que se calcula como

$$\overline{x} = \frac{\sum_{i=1}^{n} x_i}{n}$$

2. La desviación típica es 1.5275252.

Bibliografia

Gibernans Bàguena, J.; Gil Estallo, À. J.; Rovira Escofet, C. (2009). *Estadística*. Barcelona: Material didàctic UOC.

Grolemund, G. (2014). *Hands-On Programming with R: Write your own functions and simulations*. O'Reilly Media https://rstudio-education.github.io/hopr/

R notes for professionals book: http://books.goalkicker.com/RBook/

R-Bloggers (hub de blogs sobre temática R): http://www.r-bloggers.com/

R-Statistics (compilación de funcionalidades R): http://www.r-statistics.com/

R-Studio (promotores del entorno de desarrollo IDE): http://www.rstudio.com/

R-Studio at UOC: https://vimeo.com/channels/816639/138296660

Rseek (buscador web): http://rseek.org/