máximaformación

TEMA 1

SOFTWARE R

MÁSTER DE ESTADÍSTICA APLICADA CON R SOFTWARE. TÉCNICAS CLÁSICAS, ROBUSTAS, AVANZADAS Y MULTIVARIANTES

TEMA 1. SOFTWARE R

INTRODUCCIÓN

Bienvenidos al primer tema del Máster, en el que vamos a introducirlos en el **Software R.** Ilustraremos cada paso con ejemplos fáciles de seguir. Los códigos para completar cada tarea se encuentran adjuntos en la carpeta del TEMA 1.

Al final del capítulo será capaz de manipular sus datos aplicando distintas funciones y de apreciar el significado del siguiente código (Fuente: R for dummies, 2012):

```
knowledge<-apply(theory, 1, sum)
```

En el primer capítulo veremos la descripción e historia del Software R. Conoceremos qué es R y por qué actualmente es la herramienta estadística más potente. A continuación detallaremos cómo decargar e instalar la última versión de R y explicaremos la interfaz de trabajo. En el capítulo 3 observaremos los conceptos básicos del lenguaje para la manipulación y análisis de datos. Para luego estudiar las operaciones básicas, y utilizar R para realizar procesamiento de datos y análisis estadístico básico. Veremos cómo encontrar ayuda sobre las funciones de R, artículos explicativos, y aprender a citar el software y los paquetes con los que trabajo. El sexto capítulo trata sobre la manipulación de datos en R y cómo crear gráficos bonitos para visualizar nuestros datos. Finalmente una serie de ejercicios nos permitirá poner en práctica lo aprendido.

1. Descripción e historia

Numerosas empresas y universidades utilizan R para la manipulación de datos y los análisis estadísticos pero, ¿qué es realmente R?.

En este apartado te explicamos todo acerca del software R.

Figura 1. Logo del Software R

1.1 ¿Qué es R?

R es un **lenguaje y ambiente (o entorno) de programación para el análisis estadístico y gráficos**. Se denomina ambiente o entorno de trabajo a un conjunto herramientas muy flexibles que pueden extenderse fácilmente mediante extensiones llamadas paquetes o librerías (packages) o definiendo nuestras propias funciones. Existe un repositorio oficial de paquetes con finalidades específicas de cálculo o gráfico.

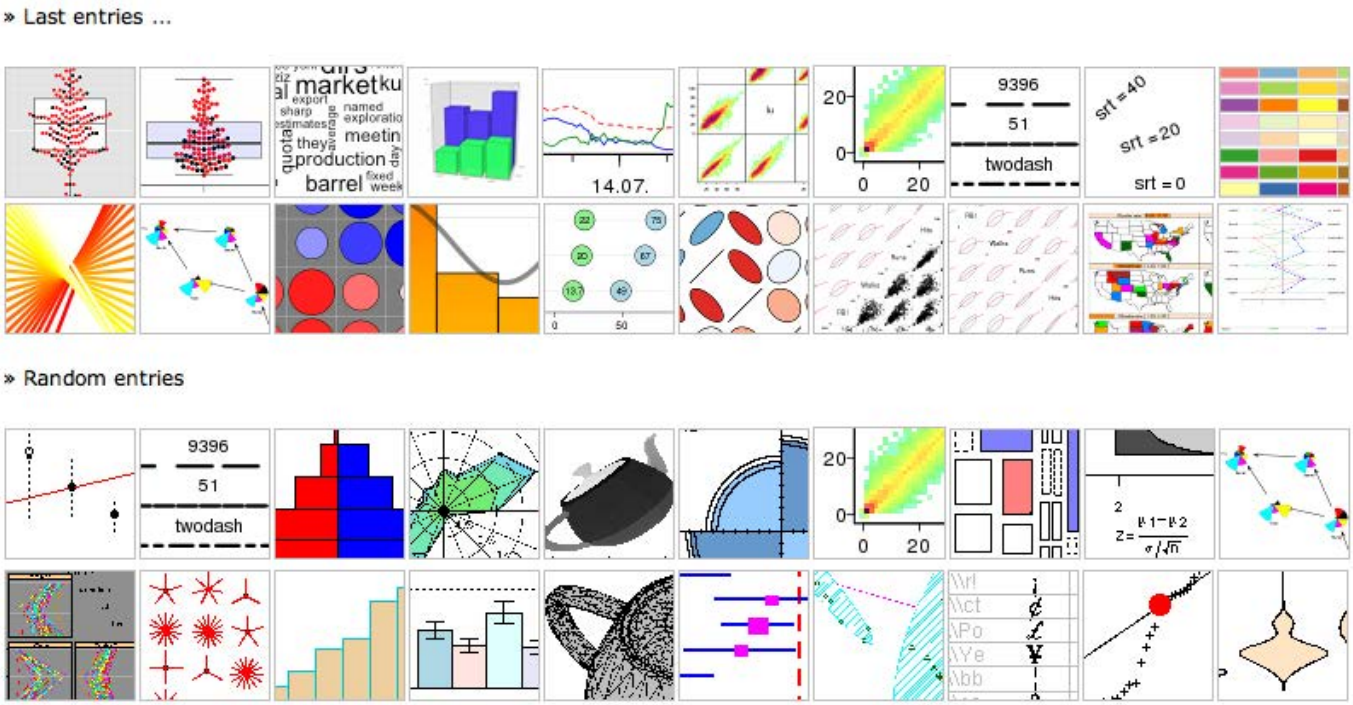


Figura 2. Ejemplos de gráficos en R.

R fue creado en 1993 por Ross Ihaka y Robert Gentleman del Departamento de Estadística de la Universidad de Auckland, New Zealand. Su nombre es en parte al reconocimiento de la influencia del lenguaje S (otro software estadístico desarrollado por de J. Chambers et al., en Bell Laboratories desde finales de 1970) y en parte para hacer gala de sus propios logros (ya que refiere a las iniciales de ambos autores). Sin embargo, a diferencia del lenguaje S, R es un software de **código abierto** (*Open Source*, parte del proyecto GNU, como GNU-Linux o Mozilla Firefox), lo cual significa que cualquiera puede descargar y modificar su código de manera gratuita. R se distribuye bajo la licencia GNU GPL (*General Public License*), esta licencia no tiene restricciones de uso, sólo obliga a que la distribución sea siempre GPL (puedes verlo aquí). Es decir, si cambias o redistribuyes el código de R, esos cambios deben estar disponibles para todo público.

Figura 3. Los creadores de R, Ross Ihaka y Robert Gentleman.

R **permite acceder a su código, modificarlo y mejorarlo**. Gracias a poder acceder a su código, el software deja de ser la **caja negra** típica de otros softwares estadísticos comerciales (como SPSS). Como resultado, R forma parte de un **proyecto colaborativo**, donde la comunidad de usuarios (integrada por programadores de alto nivel) contribuye a desarrollar nuevas funciones y paquetes que rápidamente son accesibles a todo público. Esto hace que R sea estable y confiable.

R está disponible para **todos los sistemas operativos** (Windows, Macintosh y sistemas Unix -como Linux-).

R **trabaja con otros lenguajes**. R permite leer datos de softwares como SPSS, SAS, Stata y otros (mediante el paquete foreign), así como datos de Excel. R conecta con sistemas de **bases de datos** que utilizan Open Database Connectivity protocol (ODBC, mediante el paquete RODBC) o a las bases de datos Oracle (mediante el paquete ROracle). R también permite utilizar código de lenguajes como C++, Java, Python, etc., ya que R se basa en lenguaje Fortran y C.

R es un **lenguaje Orientado a Objetos**: significa que las variables, datos, funciones, resultados, etc., se guardan en la memoria activa del computador en forma de objetos con un nombre específico. El usuario puede modificar o manipular estos objetos con operadores (aritméticos, lógicos, y comparativos) y funciones (que a su vez son objetos). Mientras que programas más clásicos muestran directamente los resultados de un análisis, R guarda estos resultados como un objeto, de tal manera que se puede hacer un análisis sin necesidad de mostrar su resultado inmediatamente. Esto puede ser un poco extraño para el usuario, pero esta característica suele ser muy útil. Otras características de los lenguajes orientados a objetos son la herencia: las subclases heredan las características de las superclases, y el polimorfismo -la misma operación aplicada a diferentes objetos- resulta en diferentes implementaciones. Por ejemplo, la lista de números 1,2,3 se asigna al objeto x, por lo que si queremos ver cómo está formado x debemos llamar a este objeto.

```
x<-1:3 #para crear una secuencia de números basta con usar el operador ":"
x
```

```
## [1] 1 2 3
```

R es un **lenguaje basado en vectores**, lo cual permite aplicar cálculos a un conjunto de valores a la vez sin la necesidad de utilizar una *funciónbucle* (loop). Un vector es una fila o columna de números o caracteres, por ejemplo, el objeto x del párrafo anterior es también un vector. Si queremos sumarle 2 a cada elemento del vector, debemos

escribir:

```
x+2
```

```
## [1] 3 4 5
```

Si queremos sumarle otro vector, escribimos:

```
x+5:7
```

```
## [1]  6  8 10
```

R es un **lenguaje interpretado** (como Java) y **no compilado** (como Fortran o Pascal), lo cual significa que los comandos escritos en el teclado son ejecutados directamente sin necesidad de construir un ejecutable.

1.2 Ventajas y desventajas

A continuación se detallan las ventajas y desventajas de utilizar el software R.

Ventajas
Algunos conceptos que hacen de R un software único, incluyen:

- Es una herramienta muy poderosa para todo tipo de procesamiento y manipulación de datos.
- Es gratuito y de código abierto.
- Ambiente de trabajo muy flexible y extensible.
- Existe una gran comunidad de usuarios y programadores que actualizan y agregan constantemente funciones y paquetes de funciones.
- Gráficos de alta calidad exportables en diversos formatos: PostScript,pdf, bitmap, pictex, png, jpeg, etc..
- Consume pocos recursos informáticos.
- Puede ejecutarse de manera remota (telnet).
- Gran cantidad de información sobre sus funciones y paquetes de funciones.

Desventajas
Las dificultades de R se relacionan con:

- Sintaxis exigente
- Documentación muy amplia y dispersa, que puede resultar difícil.
- Algunos paquetes no han sido muy contrastados.
- Utiliza la línea de comandos y no un interfaz gráfica, esto lleva mucho tiempo de adaptación y práctica.

---

2. ¿Cómo trabajar con R?

Toma nota...

Solo existe una forma de aprender R, ¡utilizándolo!

En este tema intentamos familiarizarte con el uso de R, pero lo importante es que pases un tiempo frente al ordenador practicando y, por qué no, jugando con él.

2.1. Descarga e instalación

De interés...

Puedes descargar R de la página oficial. En esta página también puedes encontrar información sobre las posibilidades que ofrece R, sus paquetes, documentación y manuales online, listas de correo, conferencias y publicaciones. Depende del sistema operativo, pero todas las versiones se pueden encontrar en la página oficial.

- Windows: se debe descargar el ejecutable y ejecutar el fichero. Se instalará el sistema base y los paquetes recomendados.
- GNU-Linux: tenemos dos opciones, 1) obtener el archivo tar, compilar desde las fuentes, descargar los paquetes adicionales e instalar; 2) Obtener los archivos binarios (ej. los archivos deb para Debian, rpm para RedHat, SuSE, Mandrake).
- Mac: tan sólo tienes que descargar e instalar la versión de R para Mac OS desde el CRAN.

2.2 Abrir R. Sistema de ventanas

- Hacer doble clic en el icono. Se abrirá la consola de R llamada Rgui (Gui, Graphical User Interface), con un mensaje de inicio.
- Iniciar R desde una interfaz gráfica (ej. RStudio o Tinn-R).
- Desde una ventana del sistema ejecutar Rterm; parecido a R en Unix o Linux.
- Teclear R en una shell.

Una vez iniciado R, veremos que se trabaja con un sistema de ventanas. La ventana del código o donde escribimos los comandos, la correspondiente a los gráficos y la consola donde observamos las salidas de las órdenes ejecutadas.

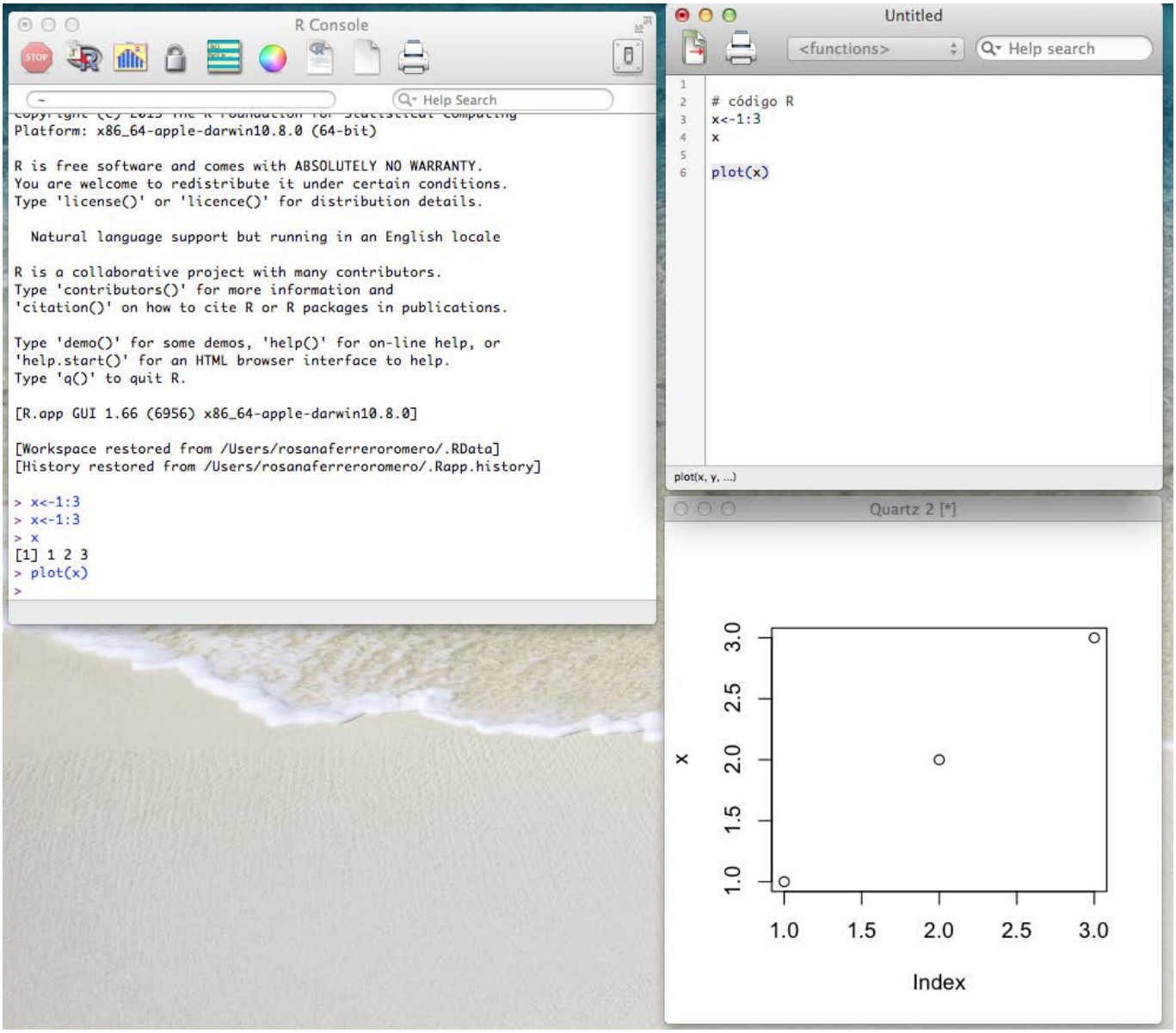


Figura 4. R, sistema de ventanas.

2.3 Editores de textos.

R no es una aplicación, es decir, podemos elegir nosotros mismos qué editor de texto queremos utilizar junto con R. Algunas de las ventajas de los editores de texto es que permiten comprobar si hemos escrito bien el código indicando, por ejemplo, si nos falta ningún paréntesis. Existen varios editores de R disponibles, por ejemplo:

- RStudio. También es de código abierto y está disponible para todos los sistemas operativos. RStudio tiene un resaltador de código que nos indica con diferentes colores las palabras claves, variables y símbolos, haciendo más sencilla la creación de código en R. También tiene la posibilidad de completar el código, por lo que no tienes que recordar todos los comandos. Presenta una pestaña de consulta a la ayuda de R, con buscador propio, nos permite observar el listado de variables y valores que tenemos en nuestra área de trabajo, los paquetes instalados y los gráficos que vayamos haciendo. Además tiene una gran ventaja, permite trabajar con archivos Shiny, Markdown, Sweave, etc., con lo que podremos generar aplicaciones, escribir informes

reproducibles y dinámicos con R, con texto en formato Latex, y exportar nuestros resultados a formatos doc, pdf, html, etc.. ¡Sin dudas, es mi favorito y los animo a que lo prueben!

:: Tinn-R. Es específico de R, disponible para Windows. Es más sencillo que los dos siguientes pero tiene menos prestaciones.

:: Eclipse StatET. Es útil sobretodo si quieres tabajar con proyectos largos de desarrollo de software. Requiere Java.

:: Emacs Speaks Statistics. Disponible para Linux y Windows. Es útil por sus atajos de teclados, en particular para los que estén familiarizados con Linux.

En adelante, trabajaremos con el editor RStudio. Para abrir el programa basta con hacer doble click en el ícono de RStudio. Observaremos 4 áreas de trabajo. En la esquina superior izquierda tienes el editor de texto, debajo a la izquierda la consola (*console*, idéntica al RGui), arriba a la derecha el área de trabajo (*workspace*) y el historial (*history*) y debajo a la derecha tienes varias pestañas que corresponden a tus archivos (*files*), gráficos (*plots*), la lista de paquetes (*package*) y la página de ayuda (*help*).

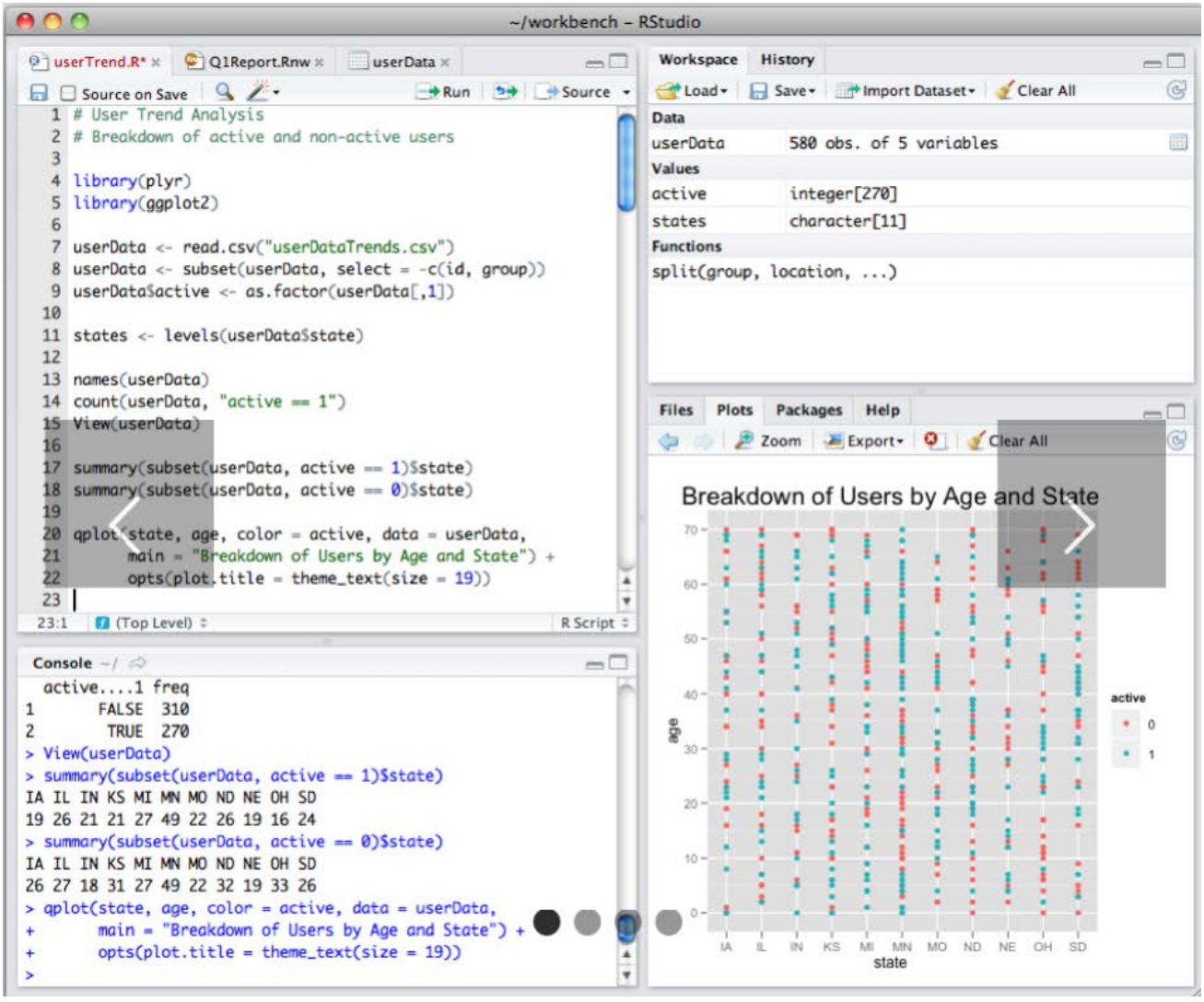


Figura 5. RStudio, el mejor editor de texto para R

Toma nota...

1. Para enviar una línea individual de código del editor a la consola, basta con hacer posicionarse en dicha línea y hacer clic en el botón ***Run*** o recurrir al atajo de teclado ***Ctrl+Enter*.**

2. Para enviar un grupo de líneas de código del editor a la consola debemos seleccionar el grupo de órdenes que queremos enviar y repetir el mismo procedimiento anterior.
3. Para enviar todo el código a la consola utilizamos el botón ***Source*** o el atajo de teclado ***Ctrl+Shift+Enter***.

2.4 ¡Información extra! Shiny y RMarkdown.

Como comenté anteriormente, RStudio te permite crear aplicaciones shinny y observar el resultado rápidamente. La siguiente figura muestra un ejemplo:

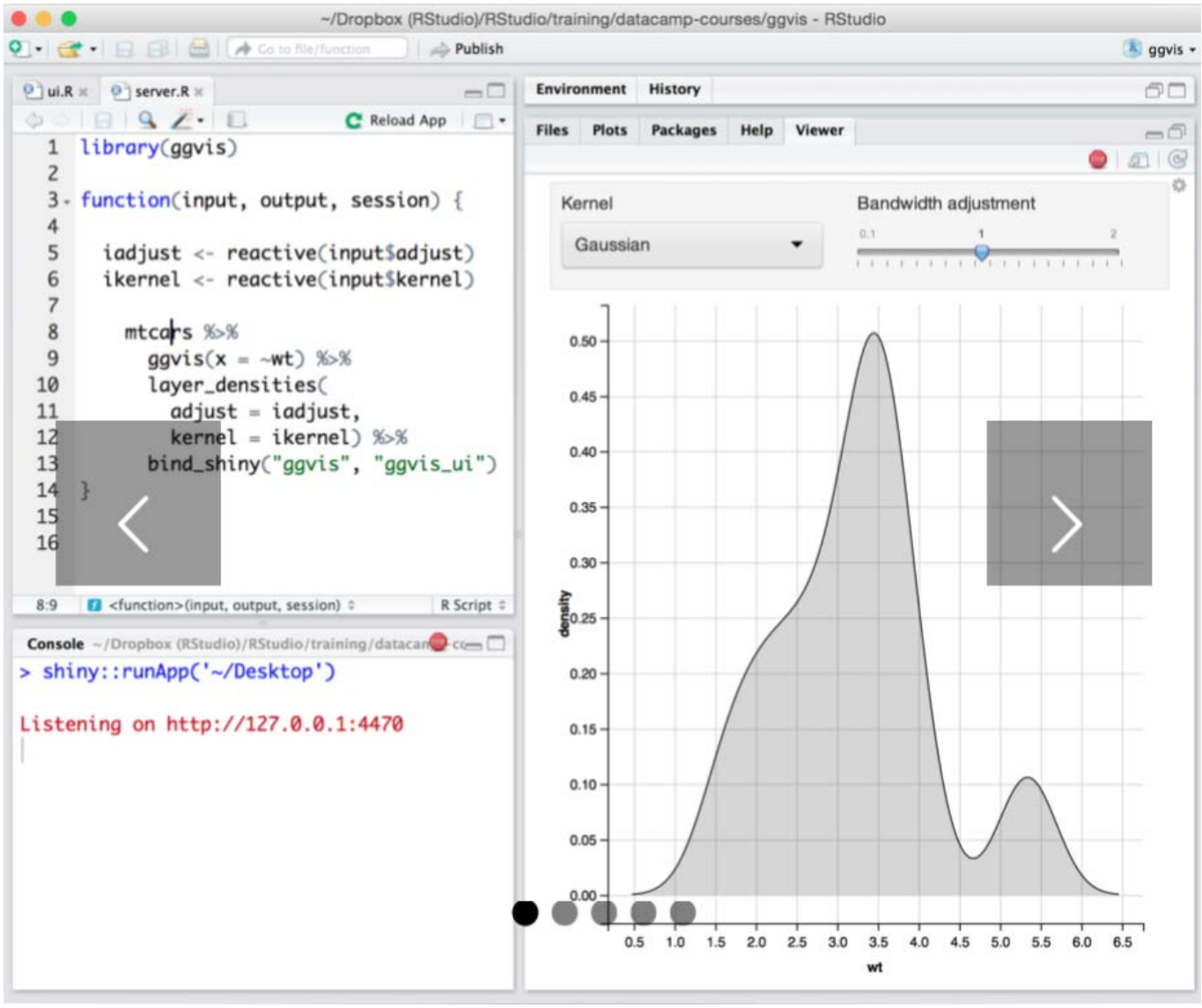


Figura 6. Shiny, para crear aplicaciones.

También podemos ver un ejemplo de los archivos RMarkdown, que permite editar textos y ver los resultados de nuestro código R en distintos formatos, por ejemplo en pdf, word o html.

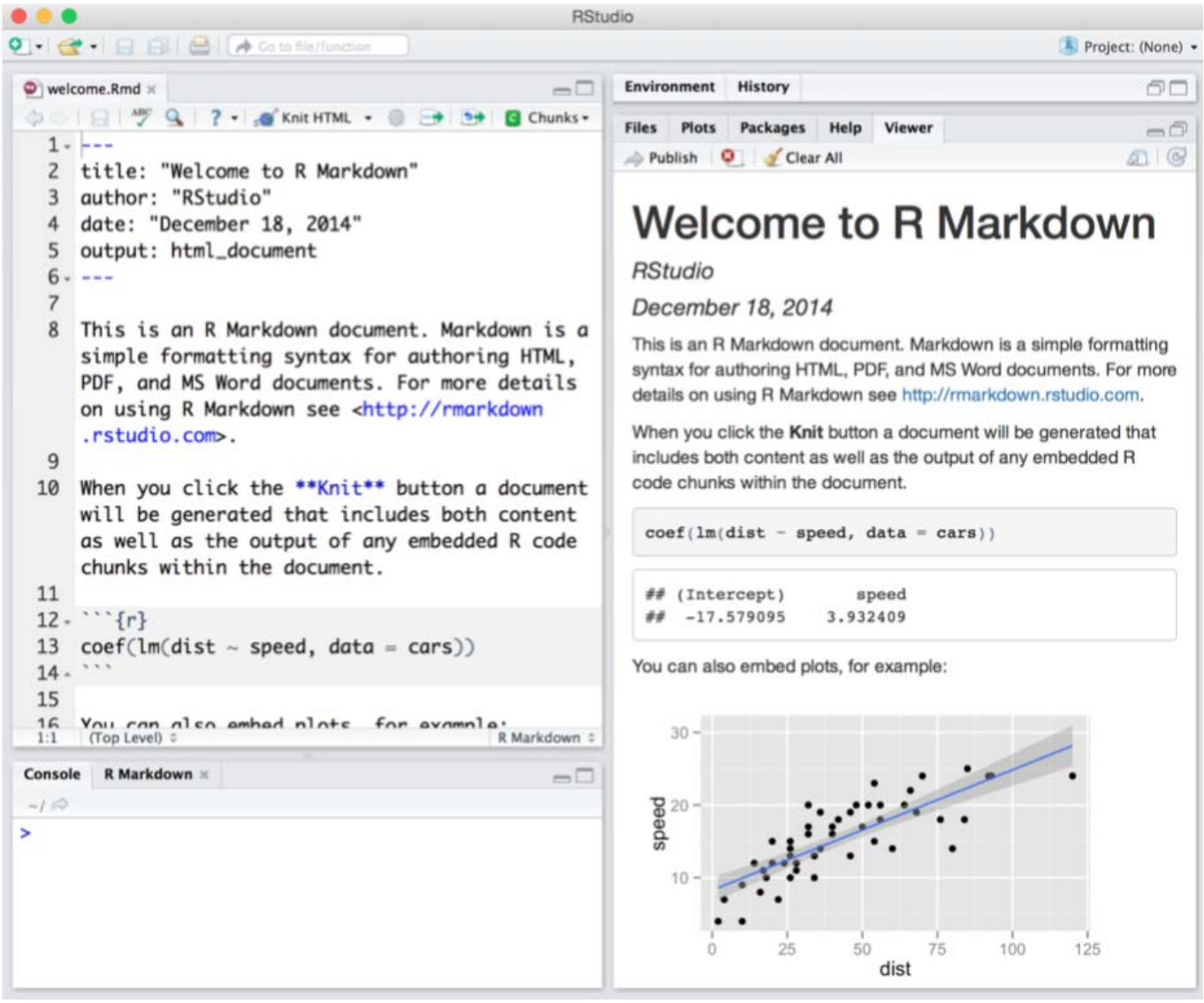


Figura 7. RMarkdown, para dar formato al texto.

2.5 Sintaxis básica

El signo > (command prompt) en la consola indica que el sistema está a la espera de que el usuario realize alguna entrada. Si queremos obtener el cociente de dividir 1 entre 3, basta con escribir:

```
1/3
```

```
## [1] 0.3333333
```

Para hacer un comentario que no se ejecute en R, utilizamos el símbolo almohadilla. Este nos sirve, por ejemplo, para describir las distintas etapas de trabajo dentro código:

```
# primer paso: crear el vector "y"
y<-1:10
# segundo paso: sumarle 3 y dividir por 2
(y+3)/2
```

```
##  [1] 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0 6.5
```

Podemos utilizar las flechas verticales del teclado en la consola de R para recorrer el historial de ordenes, mientras que las flechas horizontales permiten el movimiento dentro de la línea.

El operador de asignación $<-$ o el de igualdad $=$ permiten crear objetos en R. El nombre de un objeto debe comenzar con una letra (A-Z y a-z), y puede incluir dígitos (0-9) y puntos (.). R discrimina entre letras mayúsculas y minúsculas para el nombre de un objeto, de tal manera que x y X se refiere a objetos diferentes. R no interpreta los espacios excepto para la asignación. Por ejemplo, para crear el objeto x1 con los siguientes elementos 1,2,3,4,5 podemos marcar cualquiera de las siguientes opciones:

```
x1<-1:5
1:5->x1
x1=1:5
```

Para observar los objetos creados en nuestra área de trabajo podemos escribir la función $ls()$ (es la abreviación de $list$) u $objects()$.

```
ls()
```

```
## [1] "x"  "x1" "y"
```

```
objects()
```

```
## [1] "x"  "x1" "y"
```

Si queremos borrar objetos de la memoria podemos escribir $rm(x1)$, mientras que si queremos eliminar todos los objetos del área de trabajo, escribiremos $rm(list = ls())$.

```
rm(x1)
```

La función $save$ nos permite guardar los objetos en un nuevo archivo RData. Podemos seleccionar específicamente qué archivos guardar y el nombre del archivo que queremos crear o simplemente guardar todo lo que se encuentre en nuesta área de trabajo.

```
#guardar los objetos x e y en un archivo xy.Rdata
save(x, y, file = "xy.RData")

# sin especificar el nombre del archivo ni los objetos a guardar
# nos guarda todo lo que tengamos en el workspace.
save.image("xy2.RData")
```

Vayamos a la carpeta de trabajo y observemos que se han creado dos archivos de extensión RData.
Si quisiéramos acceder posteriormente a estos objetos, basta con utilizar la función $load$, con el nombre del archivo (y la ruta donde se encuentra).

Si lo que queremos guardar es el historial de órdenes que hemos ejecutado en R, utilizaremos la función $savehistory$, y $loadhistory$ para cargar este archivo en R.

```
#guarda el trabajo de una sesión.
 savehistory()

#recupera el trabajo de una sesión.
 loadhistory()
```

```
#la extensión .RHistory indica un archivo de historia de órdenes.
```

Para salir de R podemos utilizar el comando $q()$, donde nos preguntará si queremos guardar el workspace (también podemos utilizar $q(save = no)$).

2.6 Directorio de trabajo

Antes de comenzar a trabajar en R conviene observar nuestro directorio de trabajo, es decir, aquella carpeta donde guardaremos nuestros resultados. Para observar en qué carpeta estamos trabajando utilizaremos la función $getwd$, mientras que para cambiar o seleccionar un nuevo directorio utilizaremos la función $setwd$.

```
# Para ver en qué directorio estamos trabajando
getwd()
```

```
## [1] "/Users/rosanaferreroromero/Desktop/maxima/curso R/TEMAS/tema1"
```

```
# Para seleccionar el directorio en Mac se utiliza una única barra "/", mientras que en Windows se utilizan barras dobles "//" o simple pero con inclinación opuesta  "\". Sin embargo, es mejor utilizar las dobles barras debido a que R interpreta "\" como un caracter de salida.
setwd("~/Desktop/maxima/curso R/TEMAS/tema1")
```

Para ver y seleccionar las opciones de la sesión:

```
# ver todas las opciones
help(options)

# opciones actuales
options()

#si queremos que de ahora en más los números tengan 2 dígitos
options(digits=3)
```

Ser ordenado.

Se recomienda crear una carpeta para cada nuevo análisis, donde subdividir las áreas en:

1. data
2. script
3. figures
4. workspace

2.7 Librerías o paquetes

R consta de un sistema base (vienen con la instalación del programa) y un sistema adicional (deben ser instalados por el usuario) de paquetes, que extienden las funciones del programa. Los paquetes son una colección de funciones programadas previamente sobre temas específicos.

Para instalar un paquete adicional en R, tenemos dos opciones:

1) podemos ir a la ventana que se encuentra debajo a la derecha y pinchar en la pestaña de “Packages”, luego en “Install” y buscar en el repositorio CRAN (Comprehensive R Archive Network) de R el paquete que deseamos o primero descargarlo y luego buscar el archivo comprimido del paquete en nuestro directorio; y

2) podemos utilizar la función $install.packages()$ desde la consola de R.

Si no usas RStudio, R te pedirá que especifiques un $mirror$ para la descarga. Es decir, tienes que elegir un servidor que se encuentre cerca de tu localidad para comenzar la descarga. Si usas RStudio y quieres elegir el servidor, puedes ir a “Tools” y luego a “Options”, para seleccionar el que más te interese.

Finalmente, para cualquiera de los dos procedimientos de descarga, los paquetes deben ser cargados o activados ($loading$) para volverlos disponibles por el programa; esto se realiza desde la barra de herramientas con Paquetes, Cargar paquete o desde la consola con la función $library$. Hay que:

```
# para instalar el paquete vegan
 install.packages("vegan")

# Carga o activa (loading) el paquete vegan
library(vegan)

# lo mismo que antes
require(vegan)
```

Si queremos desactivar un paquete (por ejemplo, porque sus funciones tienen el mismo nombre que otras ya activadas y nos genera tener que especificar el paquete que queremos utilizar), podemos utilizar la función .

```
detach(package:vegan)
```

Para actualizar las últimas versiones de un paquete podemos utilizar la función o en la pestaña de Packages presionar en “Updates”.

Además, disponemos de las siguientes funciones:

```
# para obtener el directorio donde se localiza la librería
.libPaths()

# lista de todos los paquetes disponibles en nuestro programa
   library()

# lista de todos los paquetes activados
  search()

# información del paquete
  library(help="spatial")

# lista completa de los contenidos del paquete.
  objects(grep("spatial",search()))
```

2.8 Cambios en la consola

Si quisiéramos cambiar la apariencia de la consola de R, basta con seleccionar en la barra de herramientas de

RStudio la pestaña RStudio, luego “Preferences” y “Appearence”.

También te puede interesar ver otras opciones de personalización de R.

---

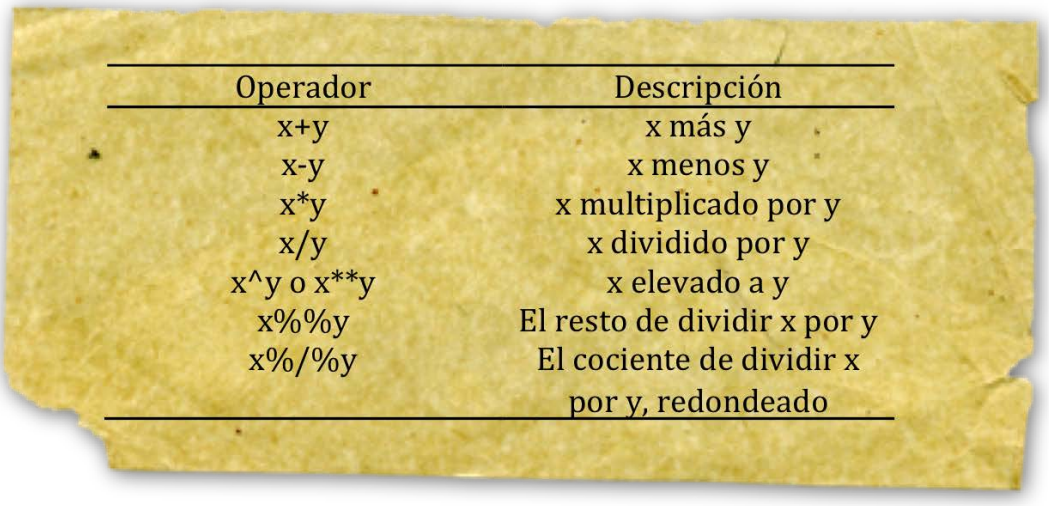
3. Operaciones básicas

Este capítulo te permite conocer qué **utilidades** tiene R. A modo de resumen, podemos utilizar R como una calculadora para operaciones aritméticas y funciones matemáticas o podemos querer ir más allá y aprender el lenguaje de R.

Esta última opción es sin dudas la más interesante, por ello, primero te enseñaremos las reglas básicas de sintaxis para que puedas elaborar tus propios códigos y luego ahondaremos en los pilares de R: los objetos.

3.1 R como calculadora: aritmética y matemáticas básicas

El listado de las funciones aritméticas básicas disponibles en R podemos observarlo escribiendo .



Operador	Descripción
x+y	x más y
x-y	x menos y
x*y	x multiplicado por y
x/y	x dividido por y
x^y o x**y	x elevado a y
x%%y	El resto de dividir x por y
x%/%y	El cociente de dividir x por y, redondeado

Figura 8. Operadores aritméticos básicos.

Entonces, si queremos utilizar R para cálculos matemáticos simples, basta con escribir:

```
# operaciones aritméticas básicas
  2 + 3 # suma
```

```
## [1] 5
```

```
  2 - 3 # resta
```

```
## [1] -1
```

```
  2*3   # multiplicación
```

```
## [1] 6
```

```
  2/3   # división
```

```
## [1] 0.6666667
```

```
  2^3   # potenciación
```

```
## [1] 8
```

Hay que tener en cuenta que R utiliza la precedencia en las operaciones, esto quiere decir que primero se realizan las funciones exponenciales, luego la multiplicación y división según el orden de los operadores que hayan (paréntesis) y finalmente las sumas y restas también en el orden en el que estén presentes. Aquí tienen algunos ejemplos:

```
#conviene utilizar paréntesis
  4^2 - 3*2
```

```
## [1] 10
```

```
 (4^2) - (3*2)
```

```
## [1] 10
```

```
  1 - 6 + 4
```

```
## [1] -1
```

```
  (1 - 6) + 4
```

```
## [1] -1
```

```
  2^-3
```

```
## [1] 0.125
```

```
  2^(-3)
```

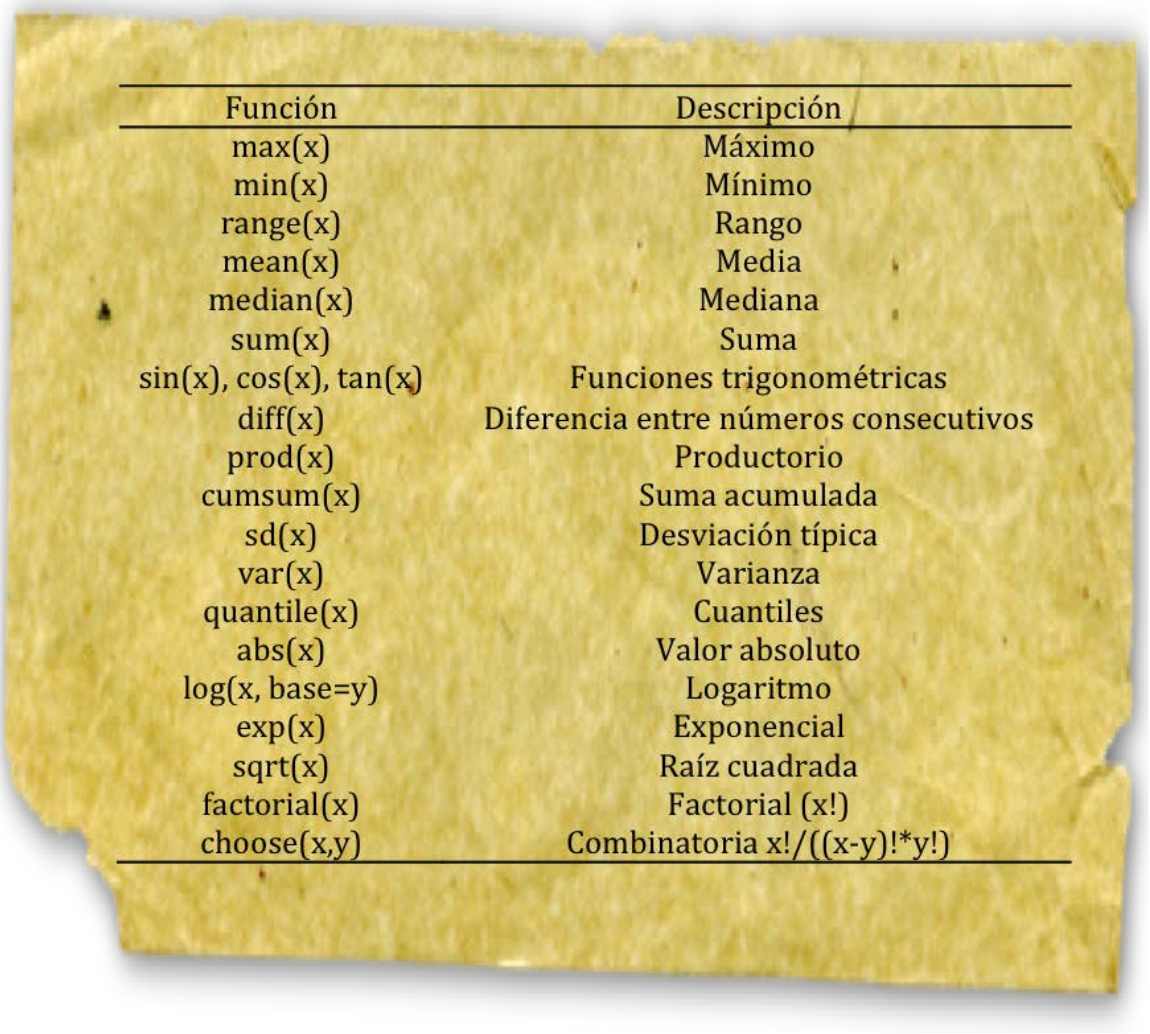
```
## [1] 0.125
```

Si queremos realizar cálculos con vectores, lo primero es construir los vectores (en este caso x e y) y luego aplicar las herramientas aritméticas.

```
x<-1:3
y<-2:4
+ x
- x
```

```
x + y
x - y
x * y
x / y
x ^ y
x ** y
x %% y
x %/% y
```

Por supuesto también podemos utilizar funciones matemáticas.



Función	Descripción
max(x)	Máximo
min(x)	Mínimo
range(x)	Rango
mean(x)	Media
median(x)	Mediana
sum(x)	Suma
sin(x), cos(x), tan(x)	Funciones trigonométricas
diff(x)	Diferencia entre números consecutivos
prod(x)	Productorio
cumsum(x)	Suma acumulada
sd(x)	Desviación típica
var(x)	Varianza
quantile(x)	Cuantiles
abs(x)	Valor absoluto
log(x, base=y)	Logaritmo
exp(x)	Exponencial
sqrt(x)	Raíz cuadrada
factorial(x)	Factorial (x!)
choose(x,y)	Combinatoria x!/((x-y)!*y!)

Figura 9. Funciones matemáticas básicas.

Por ejemplo:

```
exp(3) #exponencial
```

```
## [1] 20.08554
```

```
sqrt(2) # raíz cuadrada
```

```
## [1] 1.414214
```

```
log(1) # logaritmo
```

```
## [1] 0
```

```
  sin(1); cos(1); tan(1) # trigonométricas.
```

```
## [1] 0.841471
```

```
## [1] 0.5403023
```

```
## [1] 1.557408
```

```
# El símbolo ";" sirve para separar operaciones

  max(1:3) #máximo
```

```
## [1] 3
```

```
  min(1:3)  #mínimo
```

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

```
  range(1:3)  #rango(mínimo, máximo)
```

```
## [1] 1 3
```

```
  mean(1:3)  #media
```

```
## [1] 2
```

```
  sum(1:3)  #suma
```

```
## [1] 6
```

```
  diff(1:3)  #diferencia entre números consecutivos
```

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

```
  cumsum(1:3) #suma acumulada
```

```
## [1] 1 3 6
```

```
  sd(1:3)  #desviación típica
```

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

```
  quantile(1:3)  # cuantiles
```

```
##   0%  25%  50%  75% 100%
##  1.0  1.5  2.0  2.5  3.0
```

Notación científica.

R decide automáticamente cuando utilizar notación científica. Los siguientes ejemplos enseñan cómo se representan números muy pequeños o muy grandes. Si utilizamos notación científica, el número $0,000172$ se debe escribir como $1,72x10-4$, y en R se escribe $1.72e-4$. A su vez, el número 11131, se escribe en notación científica $1,33x10 \quad 4$, y en R se escribe $1.33e4$.

```
1.72e-4
```

```
## [1] 0.000172
```

```
1.33e4
```

```
## [1] 13300
```

Si queremos especificar el número de cifras significativas que queremos observar, basta con escribir:

```
signif(0.1723,digits=2)
```

```
## [1] 0.17
```

Recordemos que los ceros a la izquierda del punto no son cifras significativas.

También tenemos la posibilidad de redondear una cifra, solo tenemos que especificar el número de cifras (o dígitos) que queremos obtener. Por ejemplo:

```
round(1.12345,digits=2)
```

```
## [1] 1.12
```

```
round(1.12534,digits=2)
```

```
## [1] 1.13
```

Aprende Practicando...

EJERCICIO: Explora las funciones *floor(x)*, *ceiling(x)* y *trunc(x)*.

3.2 R como lenguaje: sintaxis

Operadodes de sintaxis

Nuevamente mediante la función ***help*** podemos acceder al listado de todos los operadores de sintaxis.

```
# R utiliza operadores unitarios y binarios, para obtener el listado de ellos escribe:
 help(Syntax)
```

Destacamos en especial los siguientes operadores:

```
# $ @   para extraer elementos de los objetos
data(iris)
iris$Sepal.Length
```

```
##   [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4
##  [18] 5.1 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5
##  [35] 4.9 5.0 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0
##  [52] 6.4 6.9 5.5 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8
##  [69] 6.2 5.6 5.9 6.1 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4
##  [86] 6.0 6.7 6.3 5.6 5.5 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8
## [103] 7.1 6.3 6.5 7.6 4.9 7.3 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7
## [120] 6.0 6.9 5.6 7.7 6.3 6.7 7.2 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7
## [137] 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8 6.7 6.7 6.3 6.5 6.2 5.9
```

```
# [ [[   para indexar o referenciar o acceder a los elementos de un objeto
# debes tener en cuenta que la fórmula es: objeto[filas,columnas]
head(iris) #muestra el inicio del objeto
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

```
head(iris[,"Sepal.Length"]) #muestra el inicio de la variable que le pedimos, entre comillas
```

```
## [1] 5.1 4.9 4.7 4.6 5.0 5.4
```

```
head(iris[,1]) #lo mismo que en el paso anterior pero ahora llamamos por el número de columna, no por su
nombre
```

```
## [1] 5.1 4.9 4.7 4.6 5.0 5.4
```

```
iris[1:3,] #enseña las tres primeras filas
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
```

```
iris[1:2,1:2] # enseña las dos primeras filas y las dos primeras columnas
```

```
##   Sepal.Length Sepal.Width
## 1          5.1         3.5
## 2          4.9         3.0
```

```
# podemos cambiar valores dentro de un vector
```

```
x<-1:5
x[2]<-NA
x
```

```
## [1]  1 NA  3  4  5
```

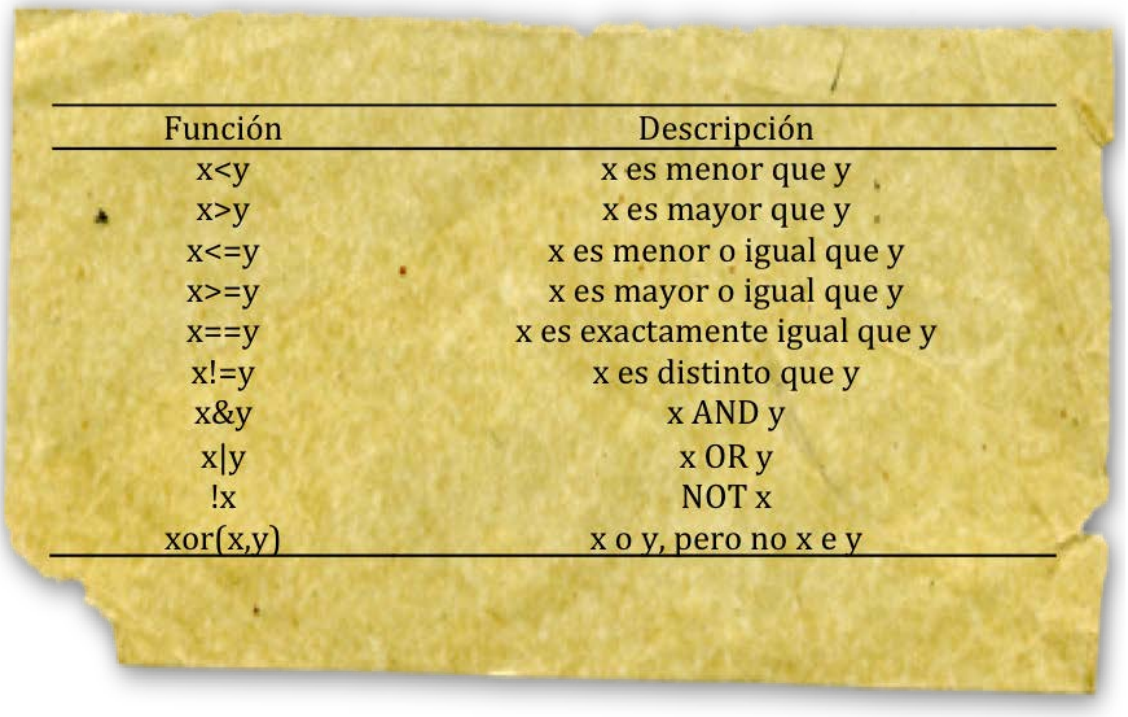
```
# :  generador de secuencia.
1:6
```

```
## [1] 1 2 3 4 5 6
```

Operaciones comparativas y lógicas.

Prueba con *help(Comparison)* y *help(Logic)*.

```
help(Comparison)
help(Logic)
```



Función	Descripción
x<y	x es menor que y
x>y	x es mayor que y
x<=y	x es menor o igual que y
x>=y	x es mayor o igual que y
x==y	x es exactamente igual que y
x!=y	x es distinto que y
x&y	x AND y
x|y	x OR y
!x	NOT x
xor(x,y)	x o y, pero no x e y

Figura 10. Funciones comparativas.

Algunos operadores ya los hemos visto, pero ponlos en práctica.

```
u<-c(3,6,NA,1)
u[u<2] # retiene aquellos valores de u que son menores que 2
```

```
## [1] NA  1
```

```
u[!is.na(u)] # retiene aquellos valores que no están ausentes NA
```

```
## [1] 3 6 1
```

```
any(u<2) # algún elemento de u es menor a 2?
```

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

```
all(u<2) # todos los elementos de u son menores que 2?
```

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

```
v<-c(1,6,3)
v>5 #qué valores de v son mayores que 5?
```

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

```
which(v>5) #cuáles valores son mayores que 5?
```

```
## [1] 2
```

```
w<-c(2,1,1)
v<w #qué valores de v son menores que w, elemento a elemento?
```

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

```
z<-v<w
which(z) #cuáles cumplen la orden z?
```

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

3.3 R como lenguaje: objetos en R

(Casi) todo en R es un objeto, ¡pero existen distintos tipos de objetos!.

Objeto	Descripción
vector	numérico, carácter, complejo o lógico
factor	numérico o carácter
matriz	numérico, carácter, complejo o lógico
array	numérico, carácter, complejo o lógico
data.frame	numérico, carácter, complejo o lógico. Permite varios tipos posibles en el mismo objeto.
lista	numérico, carácter, complejo, lógico, expresión o función Permite varios tipos posibles en el mismo objeto.

Figura 11. Tipos de objetos.

Vector

Un vector es una colección ordenada de elementos del mismo tipo.

Podemos crear un vector de distintas maneras. Por ejemplo, utilizando el operador de concatenación c:

```
x<-c(1,2,3) #números
x
```

```
## [1] 1 2 3
```

```
y<-c("a","b","c") #caracteres
y
```

```
## [1] "a" "b" "c"
```

```
z<-c(TRUE,TRUE,FALSE) #caracteres lógicos
z
```

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

```
# podemos preguntarle a R si son vectores
 is.vector(x)
```

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

```
 is.vector(y)
```

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

```
 is.vector(z)
```

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

Matrices

Son generalizaciones multidimensionales del vector, con elementos del mismo tipo. Es decir, están formados por varias columnas (o filas) de vectores. Para crear una matriz en R solo tienes que utilizar la función $matrix$ con los valores que queramos agregar y el número de columnas o filas que queremos generar. Además puedes especificar cómo quieres que se rellene la matriz, si con los valores agregados por filas o columnas (argumento $byrow = TRUE$ o $FALSE$).

Ejemplo:

```
b<-matrix(1:9,nrow=3) #ordena por columnas
b
```

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

```
b[1,2]
```

```
## [1] 4
```

```
#si quisiéramos que ordenada por filas
matrix(1:9,nrow=3,byrow=TRUE)
```

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

Las funciones ***rownames*** y ***colnames*** nos permiten agregarles nombres a las filas y columnas, respectivamente.

En la figura 12 puedes observar un listado de operaciones que podemos realizar con las matrices.

Función	Descripción
A%*%B	producto de matrices
t(A)	transpuesta de la matriz
solve(A)	inversa de la matriz
solve(A,b)	solución del sistema de ecuaciones Ax=b
svd(A)	descomposición en valores singulares
qr(A)	descomposición QR
eigen(A)	valores y vectores propios
diag(A)	matriz diagonal
sum(diag(A))	traza de una matriz
diag(1)	matriz identidad
all(B==t(A))	comprobar si una matriz es simétrica
det(A)	determinante de la matriz
A%o%B==outer(A,B)	producto exterior de dos matrices
rbind(A,B)	concatena por filas las matrices
cbind(A,B)	concatena por columnas las matrices
apply(A,1,fun)	calcula una función por filas (1)
apply(A,2,fun)	calcula una función por columnas (2)

Figura 12. Operaciones con matrices.

Array

También son generalizaciones multidimensionales del vector, con elementos de un solo tipo. Un vector tiene una dimensión(o ninguna según R), una matriz tiene dos dimensiones, y un array tiene dos o más dimensiones.

```
(a<-array(1:24, dim=c(2,3,4)))
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
##
## , , 2
```

```
##
##      [,1] [,2] [,3]
## [1,]    7    9   11
## [2,]    8   10   12
##
## , , 3
##
##      [,1] [,2] [,3]
## [1,]   13   15   17
## [2,]   14   16   18
##
## , , 4
##
##      [,1] [,2] [,3]
## [1,]   19   21   23
## [2,]   20   22   24
```

```
a[1,2,3] #fila 1, columna 2, tabla 3
```

```
## [1] 15
```

```
# utilizamos la opción drop para ver las dimensiones reales de lo que le pedimos a R.
a[,2,3, drop=FALSE] #columna 2, tabla 3
```

```
## , , 1
##
##      [,1]
## [1,]   15
## [2,]   16
```

```
a[,2,3]
```

```
## [1] 15 16
```

```
is.array(a) #podemos preguntar a R si a es un array
```

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

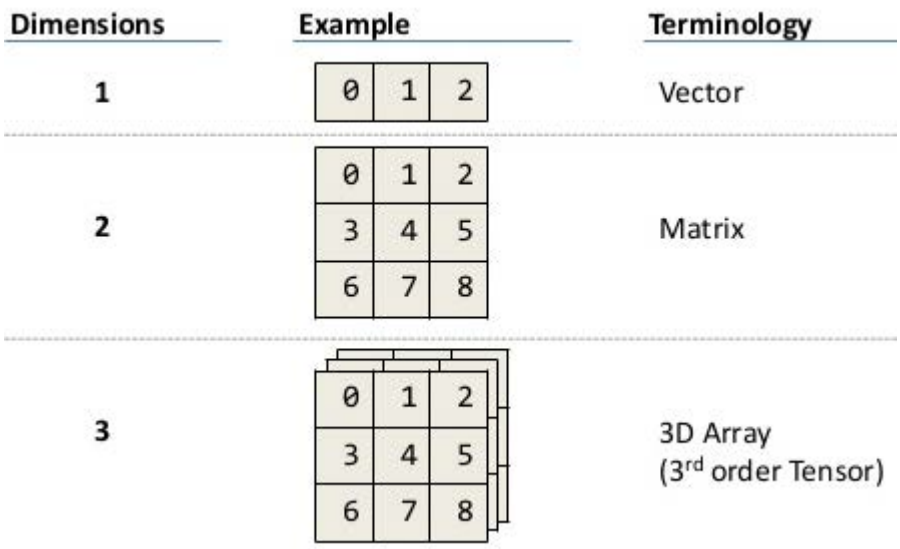


Figura 13. Un vector, una matriz y un array.

Data frame

Similar al array, pero admite columnas de diferentes tipos.

Por ejemplo:

```
data<-data.frame(ID=c("gen0","genB","genZ"),
                 subj1=c(10,25,33),
                 subj2=c(NA,34,15),
                 oncogen=c(TRUE,TRUE,FALSE),
                 loc=c(1,30,125))
data
```

```
##     ID subj1 subj2 oncogen loc
## 1 gen0    10    NA    TRUE   1
## 2 genB    25    34    TRUE  30
## 3 genZ    33    15   FALSE 125
```

Distintas formas de crear un dataframe:

```
#transformando una matriz. Todos los valores son del mismo tipo.
y=matrix(1:9,ncol=3)
y
```

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

```
as.data.frame(y)
```

```
##   V1 V2 V3
## 1  1  4  7
## 2  2  5  8
## 3  3  6  9
```

```
#uniendo vectores de diferente tipo.
x=runif(10)
y=letters[1:10]
z=sample(c(rep(T,5),rep(F,5)))
new=data.frame(y,z,x)
new
```

```
##    y     z         x
## 1  a  TRUE 0.6442188
## 2  b FALSE 0.3191041
## 3  c  TRUE 0.7681431
## 4  d FALSE 0.7122142
## 5  e FALSE 0.6615698
## 6  f  TRUE 0.8615854
## 7  g  TRUE 0.7786770
## 8  h FALSE 0.2479793
## 9  i FALSE 0.2790006
## 10 j  TRUE 0.1275188
```

```
#NOTA: R transforma los vectores de tipo caracter en factor cuando construye un data frame. Para evitarlo
debes utilizar el argumento stringsAsFactors.
```

```
str(new)
```

```
## 'data.frame':    10 obs. of  3 variables:
##  $ y: Factor w/ 10 levels "a","b","c","d",..: 1 2 3 4 5 6 7 8 9 10
##  $ z: logi  TRUE FALSE TRUE FALSE FALSE TRUE ...
##  $ x: num  0.644 0.319 0.768 0.712 0.662 ...
```

```
new2=data.frame(y,z,x, stringsAsFactors=FALSE)
str(new2)
```

```
## 'data.frame':    10 obs. of  3 variables:
##  $ y: chr  "a" "b" "c" "d" ...
##  $ z: logi  TRUE FALSE TRUE FALSE FALSE TRUE ...
##  $ x: num  0.644 0.319 0.768 0.712 0.662 ...
```

Factor

Es un tipo de vector para datos cualitativos (o categóricos). Las variables cuantitativas se representan en R mediante un vector, mientras que para las variables cualitativas se utilizan factores.

Por ejemplo:

```
x<-factor(c(1,2,2,1,1,2,1,2,1))
x
```

```
## [1] 1 2 2 1 1 2 1 2 1
## Levels: 1 2
```

```
factor(c("A", "B"))
```

```
## [1] A B
## Levels: A B
```

Cuando necesitamos convertir los factores en texto o en números utilizamos las funciones *as.character* y *as.numeric*, respectivamente.

```
(spain<-c("Madrid", "Barcelona"))
```

```
## [1] "Madrid"    "Barcelona"
```

```
(spain.f<-factor(spain))
```

```
## [1] Madrid    Barcelona
## Levels: Barcelona Madrid
```

```
as.character(spain.f)
```

```
## [1] "Madrid"    "Barcelona"
```

```
as.numeric(spain.f)
```

```
## [1] 2 1
```

Si queremos construir un factor ordenado, por ejemplo para representar una variable ordinal, tenemos que especificar la opción *ordered*.

```
res.examen<-c("Bajo","Medio","Bajo","Alto","Medio")
factor(res.examen, levels=c("Bajo","Medio", "Alto"), ordered=TRUE)
```

```
## [1] Bajo  Medio Bajo  Alto  Medio
## Levels: Bajo < Medio < Alto
```

List

Es un vector generalizado, es decir, sus componentes pueden ser también listas, o pueden ser de distinto tipo. No tienen estructura.

Las listas son muy útiles cuando queremos agrupar distintos tipos de objetos y de distinto tamaño.

Por ejemplo:

```
(y<-list(1:3,"A"))
```

```
## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1] "A"
```

```
# para acceder a sus elementos utilizo la indexación
y[1] # nos devuelve una lista con los elementos seleccionados
```

```
## [[1]]
## [1] 1 2 3
```

```
y[[2]] #nos devuelve los elementos
```

```
## [1] "A"
```

```
# también podemos darle nombre
z=list(primero=1:3,segundo="A")
z["primero"]
```

```
## $primero
## [1] 1 2 3
```

```
z[["primero"]]
```

```
## [1] 1 2 3
```

```
z$primero
```

```
## [1] 1 2 3
```

```
z$segundo
```

```
## [1] "A"
```

```
#cambiamos elementos
z[[2]]<-"B"
z
```

```
## $primero
## [1] 1 2 3
##
## $segundo
## [1] "B"
```

```
#combinamos
c(y,z)
```

```
## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1] "A"
##
## $primero
## [1] 1 2 3
##
## $segundo
## [1] "B"
```

```
#eliminamos elementos
z$segundo<-NULL
```

Funciones código

Para escribir nuestra propia función en R, basta con seguir la siguiente sintaxis: ***function_name<-function(argument) statment***. Con la palabra ***function*** definimos la función, entre paréntesis especificamos los argumentos, mientras que entre corchetes escribimos el cuerpo de la función (las órdenes a ejecutar).

Por ejemplo:

```
#creamos una función que ha de devolver siempre el doble de aquel elemento que le demos.
f1<-function(x){2*x}

# llama a la función anterior, dándole un vector desde 1 a 10.
 f1(1:10)
```

```
##  [1]  2  4  6  8 10 12 14 16 18 20
```

```
 is.function(f1) # pregunta lógica.
```

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

Otros ejemplos:

```r
#creamos una función que aplica la relación A*B/C y devuelve el objeto formado.
 funcion1<-function(A,B,C){x1<-A*B/C; x1}

#podemos consultar los argumentos de la función
 args(funcion1)
```

```
## function (A, B, C)
## NULL
```

```r
#aplicamos la función creada con los argumentos A=1, B=2, C=3.
 funcion1(1,2,3)
```

```
## [1] 0.6666667
```

La función siempre retorna la última expresión evaluada. Si se quiere obtener otra expresión, se puede utilizar la función ***return*** o ***print,*** que escriben el valor de una variable.

```r
check <- function(x){
   if(x>0){result <- "Positive"}
   else if(x<0){result <- "Negative"}
   else{result <- "Zero"}
   result
}

check(1)
```

```
## [1] "Positive"
```

```r
check(-10)
```

```
## [1] "Negative"
```

```r
check(0)
```

```
## [1] "Zero"
```

```r
check <- function(x){
   if(x>0){return("Positive") }
   else if(x<0){return("Negative")}
   else {return("Zero")}
}

check(1)
```

```
## [1] "Positive"
```

```r
check(-10)
```

```
## [1] "Negative"
```

```r
check(0)
```

```
## [1] "Zero"
```

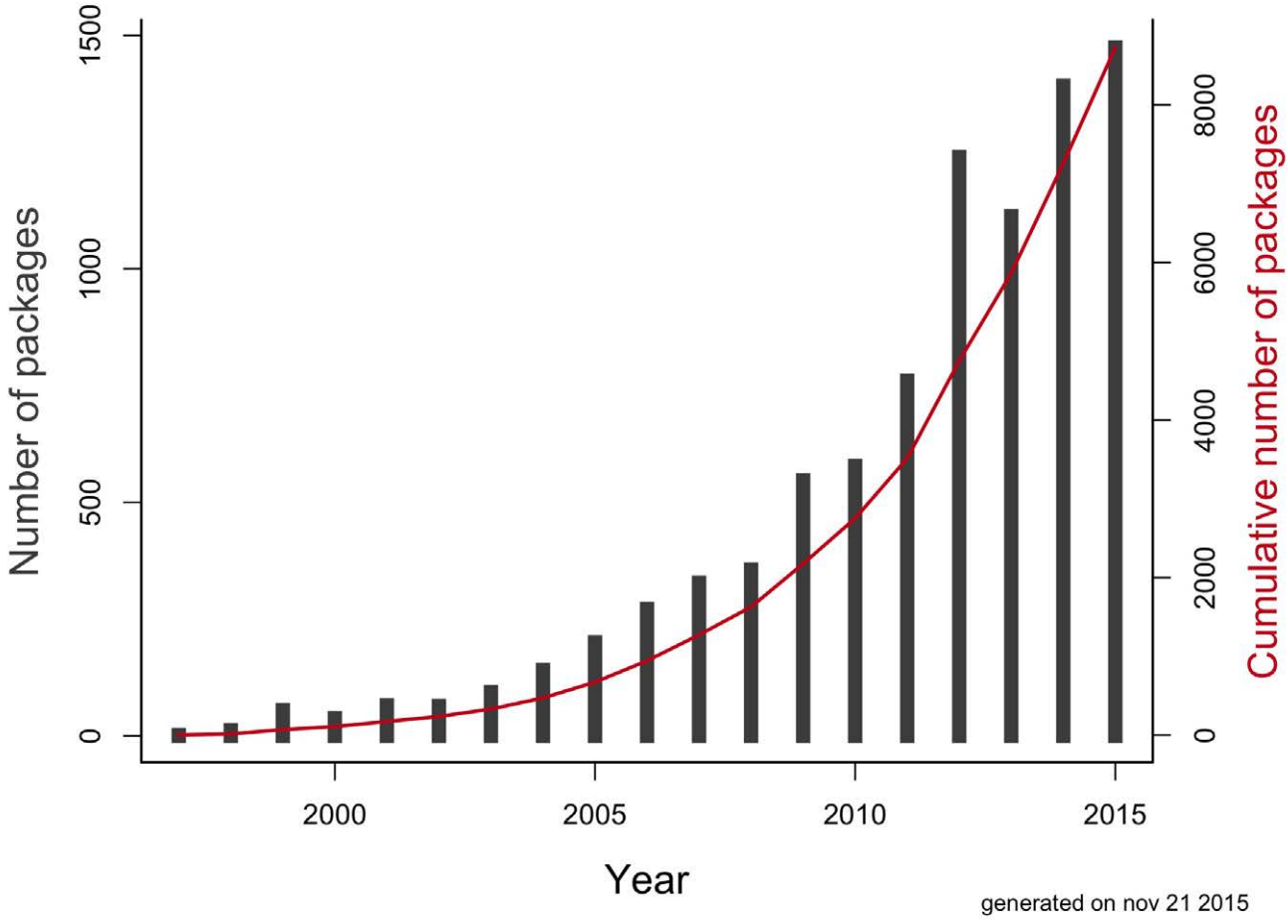
También podemos utilizar la función *cat*, que tiene mayor versatilidad. Por ejemplo:

```
x<-2
cat(x)

cat("María")

cat("María tiene", x, "hijos", ".")
cat("María tiene", x, "hijos", "\b.") #\b quita el último espacio
cat("María tiene\n", x, "hijos", "\b.") #\n divide la expresión en dos líneas
```

Función	Descripción
\\	agrega una barra inclinada \
\"	agrega unas comillas dobles "
\'	agrega una comilla simple '
\t	tabulador
\b	resta un espacio
\n	línea nueva
\#	agrega el símbolo numeral #
\r	return

Figura 14. Expresiones de control en R

4. Ayuda, viñetas y documentación, citas, y comunidad R



4.1 Solicitar ayuda

Para trabajar de manera adecuada con el software R es importante utilizar su documentación.

```
# para abrir una página html de ayuda general
help.start()

# busca directamente la ayuda relativa a una función específica
 help("summary")
# también busca la función "summary"
 ?summary
 ??summary

# abre una página html con funciones o paquetes donde aparece la palabra "median"
 help.search("median")

# indica en la consola qué paquete tiene la función median
find("median")

# indica en la consola la lista de funciones que contienen el término "lm"
  apropos("lm")

# nos da un ejemplo de cómo trabaja la función lm
 example(lm)

# para buscar en los manuales de ayuda y en las listas de correo
RSiteSearch("median")
```

Si lo que queremos es acceder a demostraciones sobre las funciones de R, o a los datos que vienen junto con los paquetes instalados, basta con teclear:

```
# lista las demostraciones disponibles
 demo(graphics)

# todas las hojas de datos (data.frame) de todas las librerías disponibles
 data()

# datos de la librería car
 data(package="car")

# carga en la memoria en uso el objeto car
  data(cars)
```

4.2 Viñetas

Como hemos mencionado anteriormente, R consta de un sistema base de paquetes y un conjunto adicional de paquetes que extienden su funcionalidad.

Puedes acceder a ellos en la dirección CRAN.

Además, puedes acceder a información sobre los paquetes utilizando la función vignette:

```
# lista de toda la información pdf disponible en los paquetes activados
 vignette(all = FALSE)

# lista de toda la información pdf disponible en los paquetes instalados
  vignette(all = TRUE)

# abre un archivo pdf de introducción al paquete grid
 vignette(grid)
```

4.3 Citar R

```
# Muestra los datos de R y como éste ha de ser citado.
citation()
```

```
##
## To cite R in publications use:
##
##   R Core Team (2015). R: A language and environment for
##   statistical computing. R Foundation for Statistical Computing,
##   Vienna, Austria. URL http://www.R-project.org/.
##
## A BibTeX entry for LaTeX users is
##
##   @Manual{,
##     title = {R: A Language and Environment for Statistical Computing},
##     author = {{R Core Team}},
```

```
##     organization = {R Foundation for Statistical Computing},
##     address = {Vienna, Austria},
##     year = {2015},
##     url = {http://www.R-project.org/},
##   }
##
## We have invested a lot of time and effort in creating R, please
## cite it when using it for data analysis. See also
## 'citation("pkgname")' for citing R packages.
```

```
#Indica cómo se han de citar los paquetes empleados.
citation("vegan")
```

```
##
## To cite package 'vegan' in publications use:
##
##   Jari Oksanen, F. Guillaume Blanchet, Roeland Kindt, Pierre
##   Legendre, Peter R. Minchin, R. B. O'Hara, Gavin L. Simpson,
##   Peter Solymos, M. Henry H. Stevens and Helene Wagner (2015).
##   vegan: Community Ecology Package. R package version 2.3-2.
##   http://CRAN.R-project.org/package=vegan
##
## A BibTeX entry for LaTeX users is
##
##   @Manual{,
##     title = {vegan: Community Ecology Package},
##     author = {Jari Oksanen and F. Guillaume Blanchet and Roeland Kindt and Pierre Legendre and Peter R.
Minchin and R. B. O'Hara and Gavin L. Simpson and Peter Solymos and M. Henry H. Stevens and Helene Wagner},
##     year = {2015},
##     note = {R package version 2.3-2},
##     url = {http://CRAN.R-project.org/package=vegan},
##   }
##
## ATTENTION: This citation information has been auto-generated from
## the package DESCRIPTION file and may need manual editing, see
## 'help("citation")'.
```

4.4 Referencias

- Michael J. Crawley. 2007. The R Book. John Wiley & Sons Ltd.
- Michael J. Crawley. 2005. Statistics: An Introduction using R. Wiley. ISBN 0 470 02297 3.
- John Verzani. 2005. Using R for Introductory Statistics. Chapman & Hall CRC, Boca Raton, FL, ISBN 1 584 88450 9. aquí.
- Emmanuel Paradis. 2002. R for Beguinners. aquí.

En **español:**

- R para Principiantes, la versión en español de R for Beginners, traducido por Jorge A. Ahumada (PDF).
- Versión en español de An Introduction to R por Andrés González y Silvia González (PDF).

- Estadística Básica con R y R-Commander. aquí.
- Introducción a R y R-Commander. aquí.
- Gráficos Estadísticos con R por Juan Carlos Correa y Nelfi González.
- Cartas sobre Estadística de la Revista Argentina de Bioingeniería por Marcelo R. Risk.
- Introducción al uso y programación del sistema estadístico R por Ramón Díaz-Uriarte, transparencias preparadas para un curso de 16 horas sobre R, dirigido principalmente a biólogos y especialistas en bioinformática.
- Lista de correo R-help-es en español. Lista de correo oficial de R en español.

4.5 Mensajes de error y advertencias

Cuando cometemos un error en el código, R nos indica un mensaje o advertencia que, si se presta atención, nos da información muy útil acerca de cómo solucionar el problema. ¡De los errores se aprende!

5. Manipulación de datos en R

5.1 Importar y exportar dato

Para leer datos desde un fichero podemos utilizar la función ***read.table*** que lee datos separados por espacios en blanco, tabuladores o saltos de línea. Para exportar datos la función básica es ***write.table().*** Veamos cómo trabajar con estas funciones:

```
# creamos el objeto que queremos exportar
x <- data.frame(a = 1:5,b="hola",c = pi)

# para escribir un archivo
write.table(x, file = "foo.xls", row.names = FALSE)
write.csv(x, file = "foo.csv", row.names = FALSE)
write.table(x, file = "foo.txt", row.names = FALSE)
# También podemos especificar más argumentos
# write.table( data.frame ,"name.txt", sep="t",quote=F)
# el argumento sep nos permite especificar el tipo de tabulación:
# t para tabulación, es el valor por defecto


# para leerlos nuevamente en R
read.table("foo.xls", header=TRUE)
```

```
##   a    b        c
## 1 1 hola 3.141593
## 2 2 hola 3.141593
## 3 3 hola 3.141593
## 4 4 hola 3.141593
```

```
## 5 5 hola 3.141593
```

```
read.csv("foo.csv")
```

```
##   a    b        c
## 1 1 hola 3.141593
## 2 2 hola 3.141593
## 3 3 hola 3.141593
## 4 4 hola 3.141593
## 5 5 hola 3.141593
```

```
read.table("foo.txt", header=TRUE)
```

```
##   a    b        c
## 1 1 hola 3.141593
## 2 2 hola 3.141593
## 3 3 hola 3.141593
## 4 4 hola 3.141593
## 5 5 hola 3.141593
```

```
#En el caso de que los decimales estén determinados por una coma, debemos especificarlo en R, ya que R lee los decimales con puntos.
```

Hay que recordar que si trabajas con RStudio tienes la opción de leer datos csv o txt fácilmente desde la barra de herramientas.

Como ya hemos mencionado, si queremos guardar o leer *datos en formato de R* software utilizaremos las funciones ***save*** y ***load***.

```
# creamos el objeto que queremos guardar
y=rnorm(10) #genera 10 números aleatorios con distribución normal típica

# para escribir un archivo RData
save(y, file="y.RData")
# para guardar todos los objetos del área de trabajo utilizamos save.image(file="nombre")

# para leerlos nuevamente en R
load("y.RData")
```

Como hemos mencionado anteriormente, si queremos utilizar *bases de datos* incluidos en las librerías, basta con escribir:

```
data(ToothGrowth)
head(ToothGrowth)
```

```
##    len supp dose
## 1  4.2   VC  0.5
## 2 11.5   VC  0.5
## 3  7.3   VC  0.5
## 4  5.8   VC  0.5
## 5  6.4   VC  0.5
## 6 10.0   VC  0.5
```

```
# si queremos acceder los datos de un paquete en particular
data(melanoma, package="lattice")
head(melanoma)
```

```
##   year incidence
## 1 1936       0.9
## 2 1937       0.8
## 3 1938       0.8
## 4 1939       1.3
## 5 1940       1.4
## 6 1941       1.2
```

```
#otra posibilidad es cargar primero la librería y luego solicitar los datos
library(lattice)
data(melanoma)
```

1.Desde Excel

Para leer archivos *xlsx*, es decir aquellas versiones de archivo posterior a Excel 2007, podemos utilizar el paquete *xlsx* o el paquete *gdata*.

```
library(xlsx)
# si queremos leer una hoja específica del documento, utilizamos la opción "1" o sheetName"
mydata <- read.xlsx("c:/myexcel.xlsx", 1)
mydata <- read.xlsx("c:/myexcel.xlsx", sheetName = "mysheet")

#más rápido pero tenemos que definir manualmente las clases de las columnas
mydata <- read.xlsx2("c:/myexcel.xlsx", sheetName = "mysheet")
data = read.xlsx2("myfile.xlsx", 1,
  colClasses = c(rep("character", 2), rep("numeric", 3))

#escribir datos en excel
write.xlsx(mydata, "c:/mydata.xlsx")

library(gdata)
read.xls("mydata.xlsx",sheet=1,header=TRUE)
```

El paquete (*library(xlsReadWrite)*) sirve solo para archivos pre Excel 2007.

```
read.xls("mydata.xls")
```

2.Datos r (archivo de sintaxis, órdenes o comandos de R)

```
dump( , , .r, append=T ó F)

source( .r) # ejecuta las órdenes de R que están en el archivo \_\_.r
```

3.Datos desde internet.

```
data01<- read.table(url("http://personality-
project.org/r/datasets/maps.mixx.epi.bfi.data"),dec=",",header=T)
head(data01)
```

4.Datos desde otros programas estadísticos (e.g. Minitab, S PLUS, SAS, SPSS, Stata, Systat, etc.).

También podemos escribir información en R que pueda ser leída por otros programas estadísticos, acceder a una base de datos.

```
library(foreign)
write.foreign(df, datafile, codefile, package = c(SPSS, Stata, SAS), ...)

?read.ntp # ayuda para leer datos desde archivos Minitab
```

```
library(RODBC) # el paquete utiliza Open DataBase Connectivity (ODBC)
```

Desde SPSS

Hay que guardar el conjunto de datos de SPSS en formato transportable `por` y luego:

```
library(Hmisc)
mydata <- spss.get("c:/mydata.por")

library(foreign)
read.spss("mydata")
write.foreign(mydata, "c:/mydata.txt", "c:/mydata.sps",   package="SPSS")
```

Desde SAS

Hay que guardar el conjunto de datos de SAS en formato transportable xpt y luego:

```
library(Hmisc)
mydata <- sasxport.get("c:/mydata.xpt")

library(foreign)
read.xport("mydata")
write.foreign(mydata, "c:/mydata.txt", "c:/mydata.sas",   package="SAS")
```

Desde Stata

```
library(foreign)
mydata <- read.dta("c:/mydata.dta")
write.dta(mydata, "c:/mydata.dta")
```

Consideraciones sobre los datos.
Las tablas de datos deberán tener siempre las variables en las columnas y las observaciones en las filas.
Eliminar todos los espacios no necesarios en nombres y demás caracteres.
Poner nombres cortos tanto en variables como en observaciones. En estas últimas, todas deberán ser del mismo tamaño.
Comprobar que no hay celdas vacías.
Eliminar todo símbolo inútil.
Eliminar todas las filas y columnas sobrantes.

5.2 Características de los objetos

Estructura

```
str(x)
```

Atributos

Ejemplo:

```
# matriz de 3x2 (filasxcolumnas)con nombres en las columnas (a y pi).
x <- cbind(a=1:3, pi=pi)
x
```

```
##      a       pi
## [1,] 1 3.141593
## [2,] 2 3.141593
## [3,] 3 3.141593
```

```
#nos indica los atributos de x: dimensión y nombres
 attributes(x)
```

```
## $dim
## [1] 3 2
##
## $dimnames
## $dimnames[[1]]
## NULL
##
## $dimnames[[2]]
## [1] "a"  "pi"
```

```
# dimensión 3x2, que no tiene nombres en las filas (dimnames[[1]]) y que si tiene  nombres en las columnas (dimnames[[2]]).

x<-1:15
y<-matrix(5,3,4)
z<-c(TRUE, FALSE)

attributes(x)
```

```
## NULL
```

```
attributes(y)
```

```
## $dim
## [1] 3 4
```

```
attributes(z)
```

```
## NULL
```

```
 w<-list(a=1:3,b=5)
attributes(w)
```

```
## $names
## [1] "a" "b"
```

```
 f1<-function(x) {return(2*x)}
attributes(f1)
```

```
## $srcref
## function(x) {return(2*x)}
```

```
is.function(f1)
```

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

Modo (mode).

Lógico (TRUE o FALSE), entero (discretos), real (continuo), carácter (nombre), etc.

Ejemplo:

```
x<-1:15
y<-matrix(5,3,4)
z<-c(TRUE, FALSE)

mode(x)
```

```
## [1] "numeric"
```

```
mode(y)
```

```
## [1] "numeric"
```

```
mode(z)
```

```
## [1] "logical"
```

Tipo (type).

Un objeto puede ser almacenado bajo diferentes formas. Los tipos de objetos más comunes son los objetos dobles, enteros, complejos, lógicos, carácter y listas.

- **Doble (double).**
  R selecciona por defecto datos de tipo doble para representar números. Los datos de este tipo son valores numéricos continuos. Para verificar si un determinado dato es doble la instrucción a emplear sería is double.

```
x <- 10
 is.double(x)
```

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

```
 y <- "a"
 is.double(y)
```

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

- **Enteros (Integer).**
  Se trata de variables numéricas de naturaleza no continua (ej. número de hijos). Para definir un valor numérico como entero habría que recurrir a la función as integer.

- **Complejos (complex).**
  Aunque R reconoce los números complejos este tipo de formato no se utiliza apenas en el análisis de datos.

- **Lógicos (logical).**
  Los datos de tipo lógico sólo contienen dos valores FALSE (falso) y TRUE (verdadero). Son generados por R tras evaluar expresiones lógicas.

Ejemplo:

```
x1<-1:5
x2<-c(1,2,3,4,5)
x3<-"patata"

typeof(x1)
```

```
## [1] "integer"
```

```
typeof(x2)
```

```
## [1] "double"
```

```
typeof(x3)
```

```
## [1] "character"
```

Nombres (names).

Son Etiquetas. Poner un nombre en minúscula no es lo mismo que ponerlo en mayúscula (R es case-sensitive: $x != X$). Hay nombres reservados, por lo que hay que evitarlos (ej: *function*, *if*).

Ejemplo:

```
z <-list(a=1,b=c,c=1:3)
 names(z)
```

```
## [1] "a" "b" "c"
```

Dimensiones (dim).
Refiere al número de filas y columnas (puede ser cero).

Ejemplo:

```
x=matrix(1:6,2)
length(x) #largo o tamaño muestral
```

```
## [1] 6
```

```
dim(x) #dimensión del objeto
```

```
## [1] 2 3
```

Clase (class).
Lista de las clases del objeto (vector *alfa*-numérico)

Ejemplo:

```
x <- 10
class(x) # numeric
```

```
## [1] "numeric"
```

Valores perdidos NA (not available), NaN (not a number), Inf (infinite).

Los vectores pueden tomar alguno de los siguientes valores especiales: 1) NA significa "valor no disponible" ("not available"), 2) NaN significa "valor no numérico" ("not a number"), y 3) Inf significa "valor no finito" ("infinite"). Las funciones $is.na$, $is.nan$ e $is.infinite$ detectan qué elemento de un vector son NA, NaN o Inf, respectivamente.

Ejemplo:

```
x<-c(1:3,NA)
x
```

```
## [1]  1  2  3 NA
```

```
x+3
```

```
## [1]  4  5  6 NA
```

```
is.na(x) # tiene NA?
```

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

```
which(is.na(x))  #qué elemento es NA?
```

```
## [1] 4
```

```
x[!is.na(x)] # x sin los valores NA
```

```
## [1] 1 2 3
```

```
x[is.na(x)]<-0 #sustituir NA por 0

is.na(5/0) #el resultado es un valor perdido NA?
```

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

```
is.nan(0/0)  #el resultado no es indeterminado NaN?
```

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

```
is.infinite(-5/0) #el resultado es infinito Inf?
```

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

Cuando utilizamos operaciones sobre vectores con valores ausentes, el resultado será NA a no ser que especifiquemos que no los considere en los cálculos. Por ejemplo:

```
x
```

```
## [1] 1 2 3 0
```

```
max(x)
```

```
## [1] 3
```

```
mean(x,na.rm=TRUE) #na.rm=T, quitar NA al evaluar la media
```

```
## [1] 1.5
```

También tenemos las funciones $na.omit$ y $na.exclude$ para omitir o exluir los valores. Las diferencias entre ambas funciones se pueden ver en funciones de predicción o en obtención de residuales.

```
y<-matrix(c(1,2,3,NA,4,NA),nrow=3)
y
```

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

```
na.omit(x) #omitir los NAs
```

```
## [1] 1 2 3 0
```

```
na.exclude(x)
```

```
## [1] 1 2 3 0
```

```
 # diferencias entre ambas funciones
 y<-c(NA,NA,NA,10,8,1,9,2)
 x<-c(2,3,1,7,2,8,3,2)

#la función lm calcula la regresión lineal simple de y en función de x
model.omit<-lm(y~x, na.action=na.omit)
model.exclude<-lm(y~x, na.action=na.exclude)

#valores ajustados
fitted(model.omit)
```

```
##        4        5        6        7        8
## 5.451807 6.506024 5.240964 6.295181 6.506024
```

```
fitted(model.exclude)
```

```
##        1        2        3        4        5        6        7        8
##       NA       NA       NA 5.451807 6.506024 5.240964 6.295181 6.506024
```

```
#residuales
resid(model.omit)
```

```
##         4         5         6         7         8
##  4.548193  1.493976 -4.240964  2.704819 -4.506024
```

```
resid(model.exclude)
```

```
##         1         2         3         4         5         6         7
##        NA        NA        NA  4.548193  1.493976 -4.240964  2.704819
##         8
## -4.506024
```

Practica con Ejercicios...

EJERCICIOS:

1.Sea $x = c(0, 4, NA, NaN, Inf)$, aplica las siguientes operaciones y describe el resultado: $1/x\ x - x\ 1/x - (x - x)$ $is.na(x)$ $*is.nan(x)$

2.Prueba las siguientes operaciones: $5/0; -5/0; 0/0\ is.na(5/0); is.infinite(-5/0); is.nan(0/0)$

5.3 Combinar vectores, matrices y data frames

5.3.1 Funciones rbind, cbind y merge.

Estas funciones sirven para combinar conjuntos de datos (Figura 15).

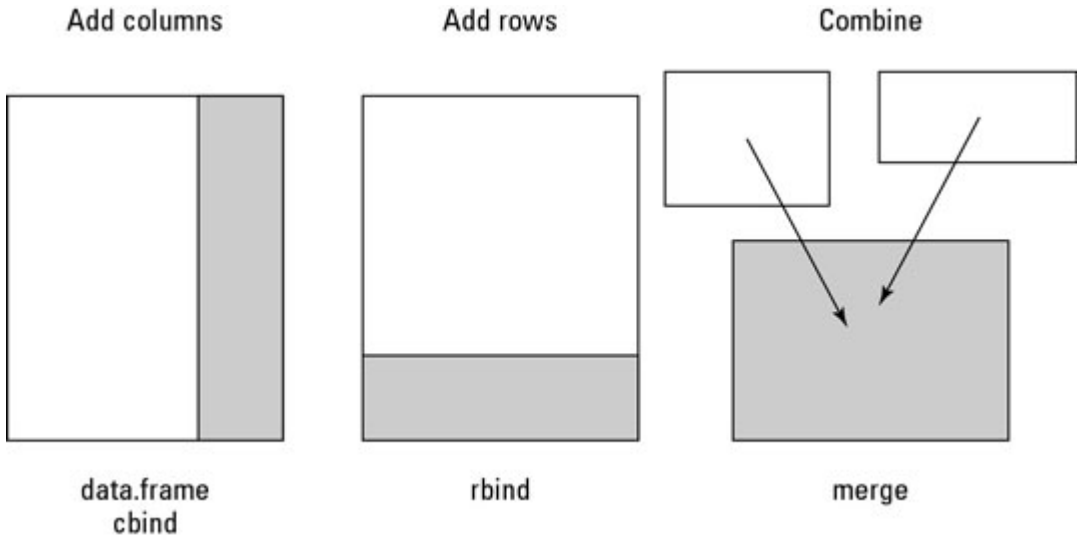


Figura 15. Funciones para combinar conjuntos de datos

Podemos tomar vectores, matrices o *data frames* y combinarlos mediante filas (para igual número de columnas) o columnas (para igual número de filas).

```
rbind(1:5,1:10)
```

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

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

```
cbind(1:5,1:10)
```

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

Podemos utilizar la función para combinar *data frames* basados el columnas o filas comunes (los datos no necesitan tener las mismas dimensiones). Esta función identifica columnas o filas que son comunes entre los dos *data frame* para unirlas. Los argumentos a ingresar son e los *data frame*, los nombres de las columnas comunes a ambos conjuntos de datos, especifican el tipo de unión a realizar (Figura 16).

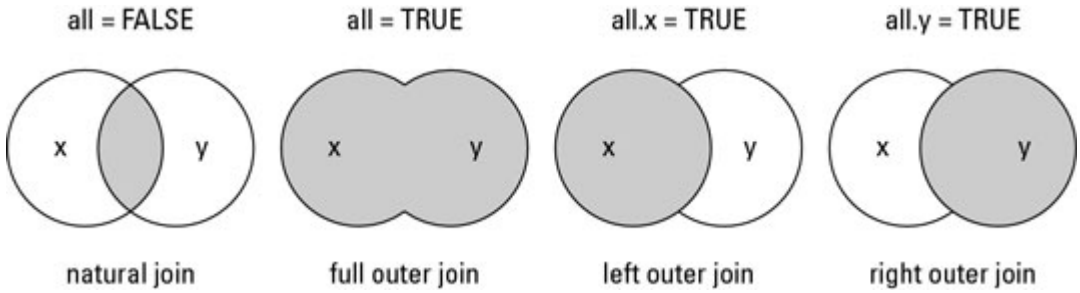


Figura 16. Función texttt{merge}

Crearemos unos datos ficticios para ejemplificar la función. Tenemos una base de datos con los productos que han comprado nuestros clientes y otra base de datos con el estado donde se han comprado dichos productos. Queremos unir las bases de datos.

```
df1 = data.frame(CustomerId = c(1:6),
                 Product = c(rep("Toaster", 3), rep("Radio", 3)))
df1
```

```
##   CustomerId Product
## 1          1 Toaster
## 2          2 Toaster
## 3          3 Toaster
## 4          4   Radio
## 5          5   Radio
## 6          6   Radio
```

```
df2 = data.frame(CustomerId = c(2, 4, 6),
                 State = c(rep("Alabama", 2), rep("Ohio", 1)))
df2
```

```
##   CustomerId   State
## 1          2 Alabama
## 2          4 Alabama
## 3          6    Ohio
```

```
merge(df1, df2)
```

```
##   CustomerId Product   State
## 1          2 Toaster Alabama
## 2          4   Radio Alabama
## 3          6   Radio    Ohio
```

```
merge(x = df1, y = df2, by = "CustomerId", all = TRUE)
```

```
##   CustomerId Product   State
## 1          1 Toaster    <NA>
## 2          2 Toaster Alabama
## 3          3 Toaster    <NA>
## 4          4   Radio Alabama
## 5          5   Radio    <NA>
## 6          6   Radio    Ohio
```

```
merge(x = df1, y = df2, by = "CustomerId", all.x = TRUE)
```

```
##   CustomerId Product   State
## 1          1 Toaster    <NA>
## 2          2 Toaster Alabama
## 3          3 Toaster    <NA>
## 4          4   Radio Alabama
## 5          5   Radio    <NA>
## 6          6   Radio    Ohio
```

```
merge(x = df1, y = df2, by = "CustomerId", all.y = TRUE)
```

```
##   CustomerId Product   State
## 1          2 Toaster Alabama
## 2          4   Radio Alabama
## 3          6   Radio    Ohio
```

```
merge(x = df1, y = df2, by = NULL)
```

```
##    CustomerId.x Product CustomerId.y   State
## 1             1 Toaster            2 Alabama
## 2             2 Toaster            2 Alabama
## 3             3 Toaster            2 Alabama
## 4             4   Radio            2 Alabama
## 5             5   Radio            2 Alabama
## 6             6   Radio            2 Alabama
## 7             1 Toaster            4 Alabama
## 8             2 Toaster            4 Alabama
## 9             3 Toaster            4 Alabama
## 10            4   Radio            4 Alabama
## 11            5   Radio            4 Alabama
## 12            6   Radio            4 Alabama
## 13            1 Toaster            6    Ohio
## 14            2 Toaster            6    Ohio
## 15            3 Toaster            6    Ohio
## 16            4   Radio            6    Ohio
## 17            5   Radio            6    Ohio
## 18            6   Radio            6    Ohio
```

Veamos ahora cómo trabajan con los datos perdidos.

```
#merge
```

```
x <- data.frame(k1 = c(NA,NA,3,4,5), k2 = c(1,NA,NA,4,5),
                data = 1:5)
x
```

```
##   k1 k2 data
## 1 NA  1    1
## 2 NA NA    2
## 3  3 NA    3
## 4  4  4    4
## 5  5  5    5
```

```
y <- data.frame(k1 = c(NA,2,NA,4,5), k2 = c(NA,NA,3,4,5),
                data = 1:5)
y
```

```
##   k1 k2 data
## 1 NA NA    1
## 2  2 NA    2
## 3 NA  3    3
## 4  4  4    4
## 5  5  5    5
```

```
merge(x, y, by = c("k1","k2")) # NA's match
```

```
##   k1 k2 data.x data.y
## 1  4  4      4      4
## 2  5  5      5      5
## 3 NA NA      2      1
```

```
merge(x, y, by = "k1") # NA's match, so 6 rows
```

```
##   k1 k2.x data.x k2.y data.y
## 1  4    4      4    4      4
## 2  5    5      5    5      5
## 3 NA    1      1   NA      1
## 4 NA    1      1    3      3
## 5 NA   NA      2   NA      1
## 6 NA   NA      2    3      3
```

```
merge(x, y, by = "k2", incomparables = NA) # 2 rows
```

```
##   k2 k1.x data.x k1.y data.y
## 1  4    4      4    4      4
## 2  5    5      5    5      5
```

5.3.2 Funciones stack, unstack, split, subset y cut

Dividir en grupos Todas estas funciones nos sirven para dividir una base de datos en función de determinadas condiciones.

La función stack une (o apila) múltiples vectores en un único vector, mientras que la función unstack es invierte esta operación.

Considere el siguiente ejemplo donde tenemos los resultados de un experimento donde se comparan los rendimientos (masa seca por planta) obtenidos bajo 3 condiciones: control y dos diferentes tratamientos. Queremos dar otro formato a los datos.

```
require(stats)
head(PlantGrowth)
```

```
##   weight group
## 1   4.17  ctrl
## 2   5.58  ctrl
## 3   5.18  ctrl
## 4   6.11  ctrl
## 5   4.50  ctrl
## 6   4.61  ctrl
```

```
formula(PlantGrowth) # fórmula por defecto
```

```
## weight ~ group
```

```
levels(PlantGrowth$group) # indica los niveles del factor   group
```

```
## [1] "ctrl" "trt1" "trt2"
```

```
(pg=unstack(PlantGrowth)) # unstack según la fórmula
```

```
##    ctrl trt1 trt2
## 1  4.17 4.81 6.31
## 2  5.58 4.17 5.12
## 3  5.18 4.41 5.54
## 4  6.11 3.59 5.50
## 5  4.50 5.87 5.37
## 6  4.61 3.83 5.29
## 7  5.17 6.03 4.92
## 8  4.53 4.89 6.15
## 9  5.33 4.32 5.80
## 10 5.14 4.69 5.26
```

```
stack(pg) # stack para volver al inicio
```

```
##    values  ind
## 1    4.17 ctrl
## 2    5.58 ctrl
## 3    5.18 ctrl
## 4    6.11 ctrl
## 5    4.50 ctrl
## 6    4.61 ctrl
## 7    5.17 ctrl
## 8    4.53 ctrl
## 9    5.33 ctrl
## 10   5.14 ctrl
## 11   4.81 trt1
## 12   4.17 trt1
## 13   4.41 trt1
## 14   3.59 trt1
## 15   5.87 trt1
## 16   3.83 trt1
```

```
## 17   6.03 trt1
## 18   4.89 trt1
## 19   4.32 trt1
## 20   4.69 trt1
## 21   6.31 trt2
## 22   5.12 trt2
## 23   5.54 trt2
## 24   5.50 trt2
## 25   5.37 trt2
## 26   5.29 trt2
## 27   4.92 trt2
## 28   6.15 trt2
## 29   5.80 trt2
## 30   5.26 trt2
```

```
stack(pg,select=-ctrl) # si queremos omitir un vector
```

```
##    values  ind
## 1    4.81 trt1
## 2    4.17 trt1
## 3    4.41 trt1
## 4    3.59 trt1
## 5    5.87 trt1
## 6    3.83 trt1
## 7    6.03 trt1
## 8    4.89 trt1
## 9    4.32 trt1
## 10   4.69 trt1
## 11   6.31 trt2
## 12   5.12 trt2
## 13   5.54 trt2
## 14   5.50 trt2
## 15   5.37 trt2
## 16   5.29 trt2
## 17   4.92 trt2
## 18   6.15 trt2
## 19   5.80 trt2
## 20   5.26 trt2
```

Cuando queremos tomar un subgrupo de vectores, matrices o *data frame* según una condición establecida, podemos utilizar la función *subset*.

```
subset(PlantGrowth, weight>5 & weight<5.2)
```

```
##    weight group
## 3    5.18  ctrl
## 7    5.17  ctrl
## 10   5.14  ctrl
## 22   5.12  trt2
```

```
subset(PlantGrowth, group=="trt1")
```

```
##    weight group
## 11   4.81  trt1
## 12   4.17  trt1
## 13   4.41  trt1
## 14   3.59  trt1
## 15   5.87  trt1
## 16   3.83  trt1
## 17   6.03  trt1
```

```
## 18   4.89  trt1
## 19   4.32  trt1
## 20   4.69  trt1
```

Cuando deseamos dividir una base de datos según los grupos definidos en , podemos utilizar la función *split*.

```
split(PlantGrowth$weight, f=PlantGrowth$group)
```

```
## $ctrl
##  [1] 4.17 5.58 5.18 6.11 4.50 4.61 5.17 4.53 5.33 5.14
##
## $trt1
##  [1] 4.81 4.17 4.41 3.59 5.87 3.83 6.03 4.89 4.32 4.69
##
## $trt2
##  [1] 6.31 5.12 5.54 5.50 5.37 5.29 4.92 6.15 5.80 5.26
```

Podemos dividir un vector numérico en factores mediante la función *cut*.

```
cut(PlantGrowth$weight,breaks=c(3,6,9))
```

```
##  [1] (3,6] (3,6] (3,6] (6,9] (3,6] (3,6] (3,6] (3,6] (3,6] (3,6] (3,6]
## [12] (3,6] (3,6] (3,6] (3,6] (3,6] (6,9] (3,6] (3,6] (3,6] (6,9] (3,6]
## [23] (3,6] (3,6] (3,6] (3,6] (3,6] (6,9] (3,6] (3,6]
## Levels: (3,6] (6,9]
```

```
# con 3 puntos de corte establecemos 2 categorías.
```

5.3.3 Funciones reshape, melt y cast

Con estas funciones cambiamos el formato de los datos (filas y columnas) seleccionando cómo queremos que estén organizados.

¿Qué significa formato largo (long) y formato ancho (wide)?
Los datos en formato ancho tienen una columna para cada variable, mientras que los datos en formato largo tiene una columna para los nombres de todas las variables y otra para sus valores (Figura 17).

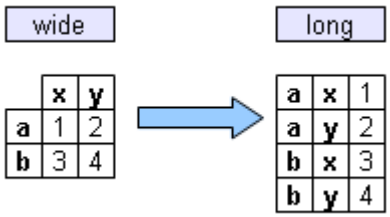


Figura 17. Formatos largo (long) y ancho (wide)

Distintas funciones en R requieren que ingresemos los datos en formatos distintos, por ello aquí veremos algunas funciones que nos permiten *transformar* los datos de manera rápida.

Como ejemplo utlizaremos los datos de farmacocinética (absorción, eliminación y circulación enterohepática) de la indometacina, un antiflamatorio no esteroideo. A 6 sujetos se le inyectó indometacina intravenosa y se registró el

momento en el que se tomó la muestra de sangre y la concentración en plasma de indometacina.

La función intercambia los formatos de *wide* a *long* y viceversa. Hay que especificar , donde es el *data frame* a ingresar, son las variables en el formato *wide* que correspoden a la única variable del formato *long*, son las variables en el formato *long* que corresponden a múltiples variables en el formato *wide*, es la variable en el formato *long* que diferencia múltiples valores para el mismo grupo o individuo e son las variables en el formato *long* que identifican múltiples valores en el mismo grupo/individuo. Pero será más fácil entenderlo con un ejemplo…

```
head(Indometh)
```

```
##   Subject time conc
## 1       1 0.25 1.50
## 2       1 0.50 0.94
## 3       1 0.75 0.78
## 4       1 1.00 0.48
## 5       1 1.25 0.37
## 6       1 2.00 0.19
```

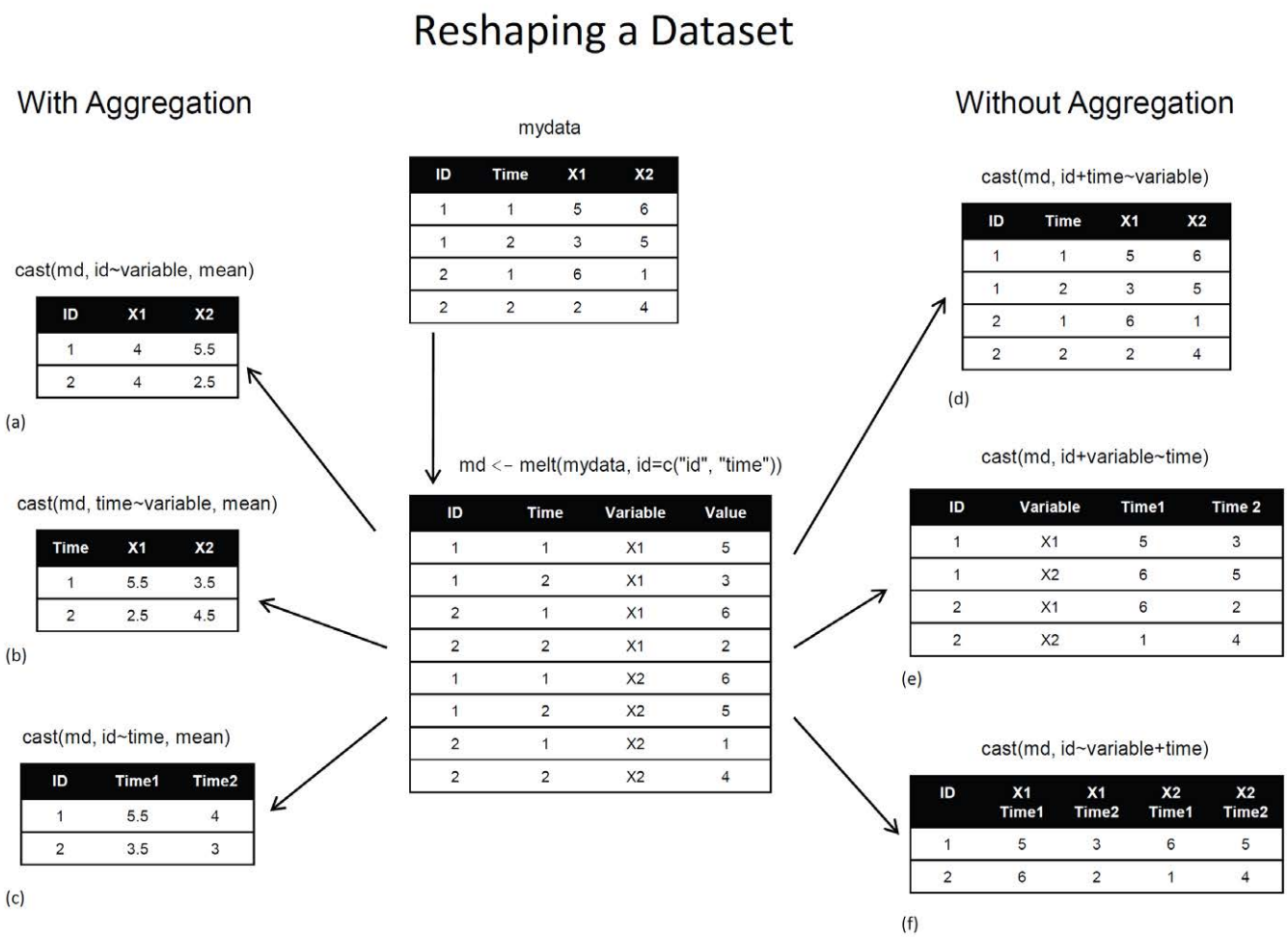
```
wide <- reshape(Indometh, v.names = "conc", idvar = "Subject",
                timevar = "time", direction = "wide")
wide
```

```
##    Subject conc.0.25 conc.0.5 conc.0.75 conc.1 conc.1.25 conc.2 conc.3
## 1        1      1.50     0.94      0.78   0.48      0.37   0.19   0.12
## 12       2      2.03     1.63      0.71   0.70      0.64   0.36   0.32
## 23       3      2.72     1.49      1.16   0.80      0.80   0.39   0.22
## 34       4      1.85     1.39      1.02   0.89      0.59   0.40   0.16
## 45       5      2.05     1.04      0.81   0.39      0.30   0.23   0.13
## 56       6      2.31     1.44      1.03   0.84      0.64   0.42   0.24
##    conc.4 conc.5 conc.6 conc.8
## 1    0.11   0.08   0.07   0.05
## 12   0.20   0.25   0.12   0.08
## 23   0.12   0.11   0.08   0.08
## 34   0.11   0.10   0.07   0.07
## 45   0.11   0.08   0.10   0.06
## 56   0.17   0.13   0.10   0.09
```

```
head(reshape(wide, direction = "long"))
```

```
##        Subject time conc
## 1.0.25       1 0.25 1.50
## 2.0.25       2 0.25 2.03
## 3.0.25       3 0.25 2.72
## 4.0.25       4 0.25 1.85
## 5.0.25       5 0.25 2.05
## 6.0.25       6 0.25 2.31
```

La figura 18 nos indica ejemplos del uso de las funciones *melt* y *cast*.



La función también permite fundir una base de datos de tal modo de obtener una única combinación id-variable por fila. El argumento define la columna de identificación de los sujetos, pero también puede incorporar variables entre-sujetos además de las variables de medidas repetidas, define las variables tratamiento que irán en una sola columna (si se deja en blanco se toman por defecto todas las variables que no sean id).
La función realiza la operación inversa (*long->wide*), donde los datos son agrupados según la función que se especifique.

Utilizaremos los datos de calidad de aire en la ciudad de Nueva York desde mayo a setiembre de 1973.

```
library(reshape)
head(airquality)
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    NA      NA 14.3   56     5   5
## 6    28      NA 14.9   66     5   6
```

```
#melt
head(melt(airquality))
```

```
## Using  as id variables
```

```
##   variable value
## 1    Ozone    41
## 2    Ozone    36
```

```
## 3    Ozone    12
## 4    Ozone    18
## 5    Ozone    NA
## 6    Ozone    28
```

```
#melt
aqm<-melt(airquality, id=c("Month", "Day"))
head(aqm)
```

```
##   Month Day variable value
## 1     5   1    Ozone    41
## 2     5   2    Ozone    36
## 3     5   3    Ozone    12
## 4     5   4    Ozone    18
## 5     5   5    Ozone    NA
## 6     5   6    Ozone    28
```

```
#melt
aql <- melt(airquality, id.vars = c("Month", "Day"),
  variable.name = "climate_variable",
  value.name = "climate_value")
head(aql)
```

```
##   Month Day variable value
## 1     5   1    Ozone    41
## 2     5   2    Ozone    36
## 3     5   3    Ozone    12
## 4     5   4    Ozone    18
## 5     5   5    Ozone    NA
## 6     5   6    Ozone    28
```

```
#cast
cast(aqm, Month ~ variable, mean) #con la función mean
```

```
##   Month Ozone  Solar.R      Wind     Temp
## 1     5    NA       NA 11.622581 65.54839
## 2     6    NA 190.1667 10.266667 79.10000
## 3     7    NA 216.4839  8.941935 83.90323
## 4     8    NA       NA  8.793548 83.96774
## 5     9    NA 167.4333 10.180000 76.90000
```

5.4 Acceder a los elementos de un objeto: attach y detach

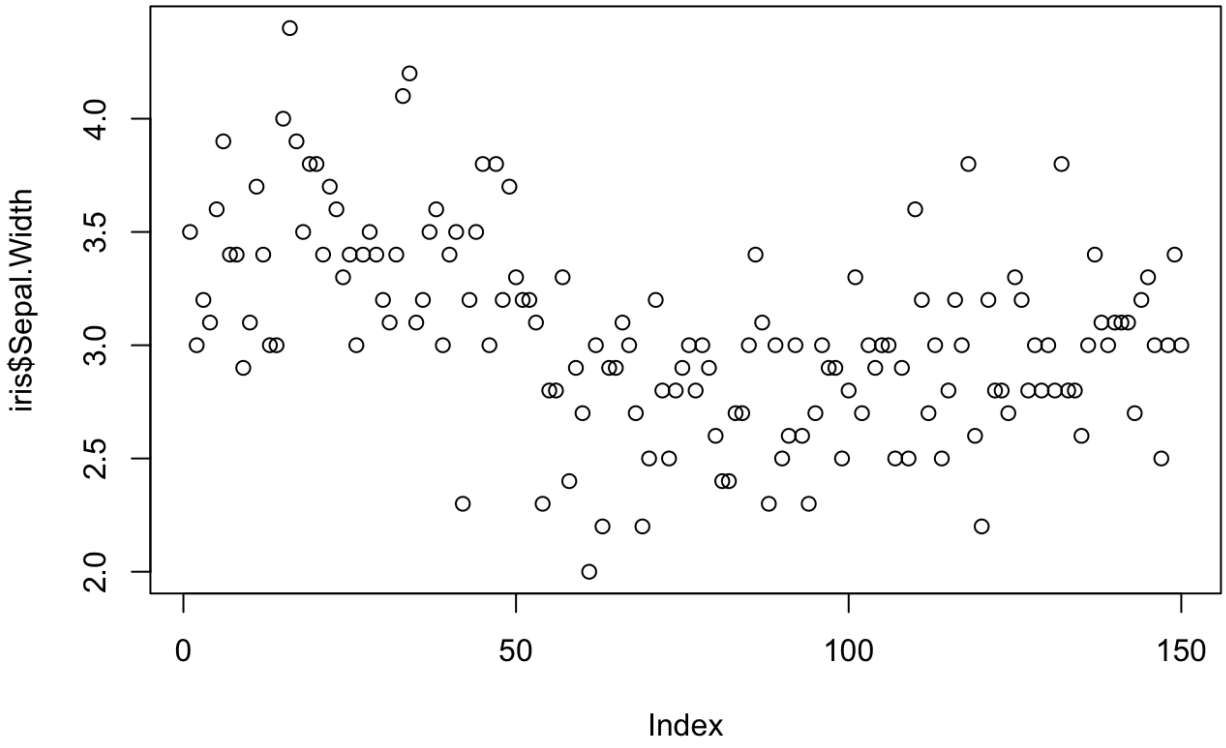
Podemos observar los elementos de un objeto utilizando la función *str*, y con la función o acceder o no acceder a los elementos que contiene dicho objeto. Por ejemplo, utilizaremos los datos *iris* (de Fisher o Anderson), que nos dan las mediciones en centímetros de distintas variables de 50 flores de 3 especies de la familia Iris: Iris setosa, versicolor y virginica.

```
# para acceder a los datos iris
data(iris)
```

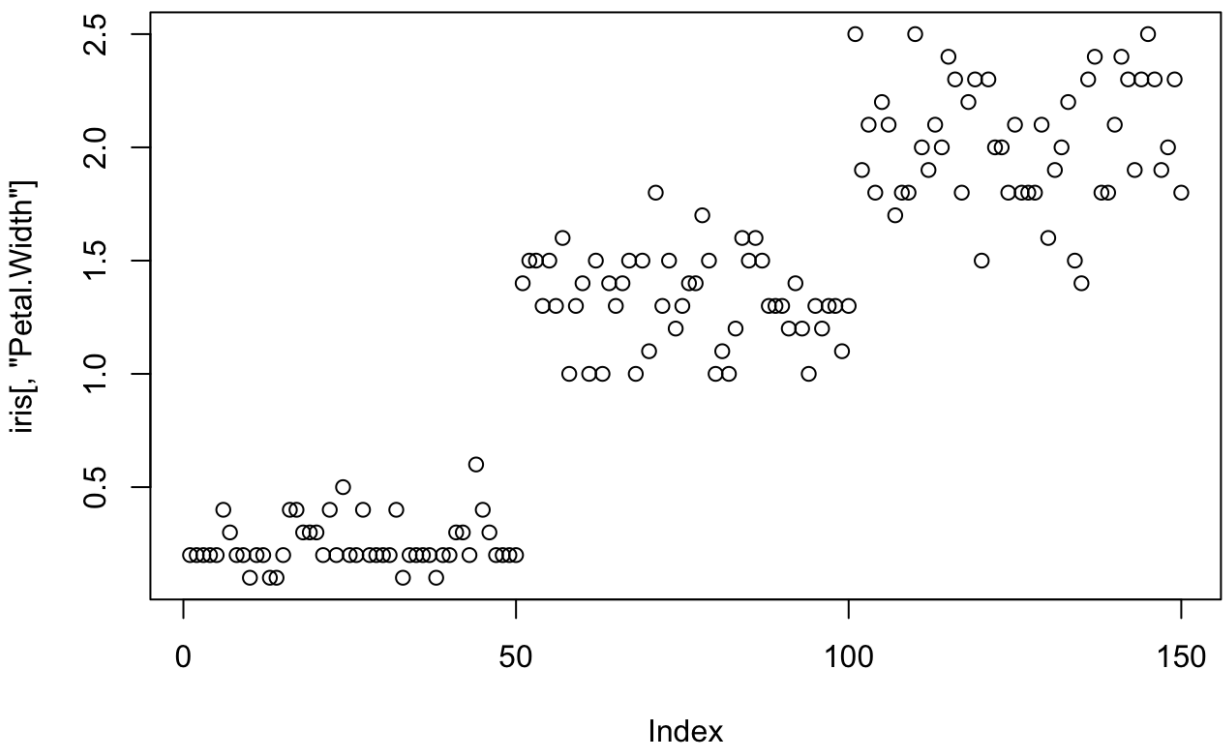
```r
# para ver como esta formado el objeto iris
str(iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

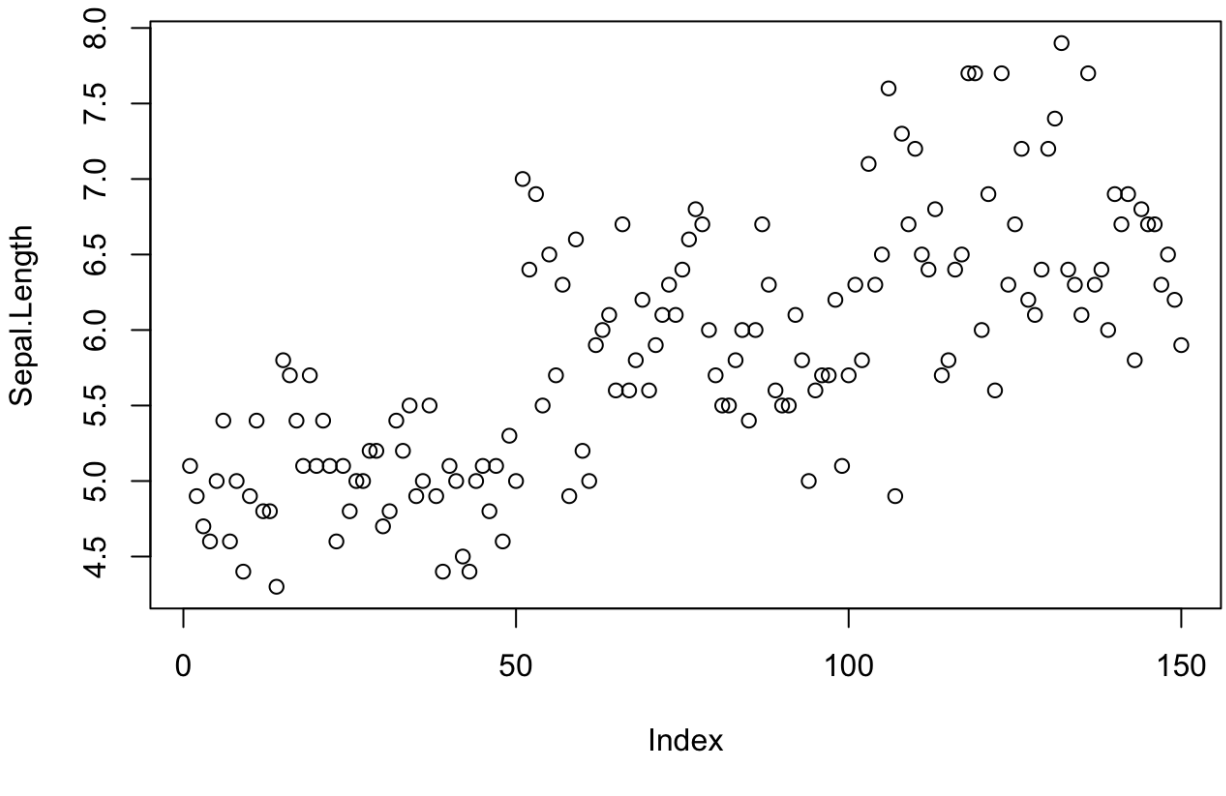
```r
# para trabajar con el elemento Sepal.Width de iris, lo llamamos mediante el operador "$"
plot(iris$Sepal.Width)
```




```r
# o mediante indexación
plot(iris[,"Petal.Width"])
```




```r
# para activar los elementos de iris, y poder trabajar con ellos sin nombrar a iris
attach(iris)

plot(Sepal.Length)
```



```
# para desactivar los elementos de iris
detach(iris)

# Sepal.Length #ya no está disponible
```

5.5 Control de ejecución: funciones condicionales, loops, etc.

Una de las grandes ventajas de R es que podemos generar un **lenguaje de programación propio**. Para ello, utilizamos comandos escritos entre llaves (expresiones).

Ejemplos:

5.5.1 Selección condicionada: funciones if e ifelse

Para realizar una ejecución condicional utilizamos la siguiente sintaxis: *if(expresion1){expresion2}*. Es decir, se utiliza la función *if*, entre paréntesis se incluye un valor lógico o una expresión que conduce a un valor lógico, y entre corchetes se incluye el código que se ejecuta cuando el valor lógico anterior es verdadero (TRUE).

También podemos utilizar un código que se ejecute en caso de que el valor lógico no sea verdadero (FALSE); para lo que cual incluiremos la opción seguido de un código entre corchetes: *if(expresion1){expresion2}else{expresion3}*.

```
#supongamos que queremos una función condicional donde
#si el valor analizado es 5, devuelve x+1
 x<-5
 if(x==5){x+1}
```

```
## [1] 6
```

```
#ahora incluimos otra operación, que
#en caso contrario devuelva x+2
 x<-4
 if(x==5){x+1}else{x+2}
```

```
## [1] 6
```

La versión vectorizada de *if* es la función *ifelse(expresion,expresionTRUE,expresionFALSE)*. Aquí,cada elemento del vector ingresado (*i*) será la *expresionTRUE[i]* si expresion es cierta, o *expresionFALSE[i]* si es falsa.

```
# Función "ifelse"
 x
```

```
## [1] 4
```

```
 x<-1:5
```

```
ifelse(x == 1, "Yes", "No") # si x es 1 le adjudica Yes, sino No.
```

```
## [1] "Yes" "No"  "No"  "No"  "No"
```

```
y<--1:1
ifelse(y<=0, "no existe", log(y) )
```

```
## Warning in log(y): NaNs produced
```

```
## [1] "no existe" "no existe" "0"
```

5.5.2 Función switch

Cuando necesitamos aplicar distintas condiciones, en lugar de utilizar varios *if*, es de mayor utilidad la función *switch*. Aquí el primer argumento es un valor o expresión, y en segundo lugar especificamos las opciones que queremos tener disponibles. Estas son las opciones:
1. si escribimos *switch(i, expresion1, expresion2, …, expresionn)*, donde *i* es un número, la función calcula la expresión *i*.
2. si escribimos *switch(nombreq, nombre1=expresion1, nombre2=expresion2, …, nombren=expresion)*, donde es alfanumérico, la función calcula la expresión llamada *nombreq*.
Por ejemplo:

```
switch(1, "lunes", "martes", "miércoles")
```

```
## [1] "lunes"
```

```
switch("martes", lunes="1", martes="2", miercoles="3")
```

```
## [1] "2"
```

```
switch(1, 1:2, 2:3)
```

```
## [1] 1 2
```

```
switch("b", a = 1, b = 2:3)
```

```
## [1] 2 3
```

```
switch("privado", publico=10.5, privado=8.3)
```

```
## [1] 8.3
```

La función *switch* solo utiliza valores, no vectores, por lo que si nuestro caso es más complejo, deberemos crear nuestra propia función con *function*. Por ejemplo, cuando queremos definir una función que nos permita calcular distintos tipos de estadísticos de centralidad según lo solicitemos, podemos aplicar:

```
# si queremos permitir todas las opciones de trim agregamos "..."
centrar <- function(x,type, ...)
       {
       switch(type, media=mean(x),
       mediana = median(x),
       recortada = mean(x, ...),
       mean(x))
      }

# como ejemplo obtenemos 10 números aleatorios con distribución exponencial
x <- rexp(10)
 centrar(x,"recortada")
 centrar(x,"recortada", trim=.2)
# aquí especificamos que queremos la media recortada con trim=0.2
```

5.5.3 Función for

Se trata de un bucle cuya sintaxis es de la forma *for(x in expresion1){expresion2}*.

```
f = factor(sample(letters[1:5], 10, replace=TRUE))
 for( i in unique(f) ){print(i)}
```

```
## [1] "c"
## [1] "a"
## [1] "d"
## [1] "e"
## [1] "b"
```

```
#devuelve los valores únicos del factor

g = 1:10
 for( j in 1:length(f)){print(g[j]*2/3)}
```

```
## [1] 0.6666667
## [1] 1.333333
## [1] 2
## [1] 2.666667
## [1] 3.333333
## [1] 4
## [1] 4.666667
## [1] 5.333333
## [1] 6
## [1] 6.666667
```

```
#observar que dentro de un bucle o loop debemos escribir print
# si queremos que nos devuelva el resultado de la orden que le hemos dado.
```

5.5.4 Función while

También es una función bucle y aplica una expresión hasta que se cumpla una determinada condición *while(condicion){expresion}*:

```
x<-0
while(length(x)<=10) {x<-c(x,1)}
#crea un objeto x hasta un largo de 10, donde va agregando unos.
x
```

```
##  [1] 0 1 1 1 1 1 1 1 1 1 1
```

5.5.5 Función repeat

Es otro tipo de función bucle del tipo *repeat{expresion}*. La expresión debe tener algún comando de parada (*break* o *return*).

```
x <- 1
repeat{
   print(x)
   x = x+1
   if (x == 6){
       break
   }
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

La función *break* se utiliza dentro de un loop para parar las iteraciones y salir del bucle. La función *next* finaliza la iteración actual del bucle y salta a la siguiente sin salir del bucle.

```
x <- 1:5
for(val in x){
    if (val == 3){
        next
    }
    print(val)
}
```

```
## [1] 1
## [1] 2
## [1] 4
## [1] 5
```

5.6 Resumen de un data.frame: funciones by y aggregate

Cuando queremos agrupar nuestros datos reemplazando grupos de observaciones por sus resúmenes estadísticos, utilizamos las funciones *by* y *aggregate*.

```
#Primer visión de los datos
worms=read.table("worms.txt",header=T)
attach(worms)
head(worms)
```

```
##        Field.Name Area Slope Vegetation Soil.pH  Damp Worm.density
## 1     Nashs.Field  3.6    11  Grassland     4.1 FALSE            4
## 2  Silwood.Bottom  5.1     2     Arable     5.2 FALSE            7
## 3   Nursery.Field  2.8     3  Grassland     4.3 FALSE            2
## 4     Rush.Meadow  2.4     5     Meadow     4.9  TRUE            5
## 5 Gunness.Thicket  3.8     0      Scrub     4.2 FALSE            6
## 6        Oak.Mead  3.1     2  Grassland     3.9 FALSE            2
```

```
#summary: resume los contenidos de todas las variables
summary(worms)
```

```
##         Field.Name      Area           Slope           Vegetation
##  Ashurst     : 1   Min.   :0.800   Min.   : 0.00   Arable   :3
##  Cheapside   : 1   1st Qu.:2.175   1st Qu.: 0.75   Grassland:9
##  Church.Field: 1   Median :3.000   Median : 2.00   Meadow   :3
##  Farm.Wood   : 1   Mean   :2.990   Mean   : 3.50   Orchard  :1
##  Garden.Wood : 1   3rd Qu.:3.725   3rd Qu.: 5.25   Scrub    :4
##  Gravel.Pit  : 1   Max.   :5.100   Max.   :11.00
##  (Other)     :14
##     Soil.pH         Damp          Worm.density
##  Min.   :3.500   Mode :logical   Min.   :0.00
##  1st Qu.:4.100   FALSE:14        1st Qu.:2.00
##  Median :4.600   TRUE :6         Median :4.00
##  Mean   :4.555   NA's :0         Mean   :4.35
##  3rd Qu.:5.000                   3rd Qu.:6.25
##  Max.   :5.700                   Max.   :9.00
##
```

```
#aggregate: crea una tabla del tipo tapply
aggregate(worms[,c(2,3,5,7)],by=list(veg=Vegetation),"mean")
```

```
##         veg     Area    Slope  Soil.pH Worm.density
## 1    Arable 3.866667 1.333333 4.833333     5.333333
## 2 Grassland 2.911111 3.666667 4.100000     2.444444
## 3    Meadow 3.466667 1.666667 4.933333     6.333333
## 4   Orchard 1.900000 0.000000 5.700000     9.000000
## 5     Scrub 2.425000 7.000000 4.800000     5.250000
```

```
aggregate(worms[,c(2,3,5,7)],by=list(veg=Vegetation,d=Damp),"mean")
```

```
##         veg     d     Area    Slope  Soil.pH Worm.density
## 1    Arable FALSE 3.866667 1.333333 4.833333     5.333333
## 2 Grassland FALSE 3.087500 3.625000 3.987500     1.875000
## 3   Orchard FALSE 1.900000 0.000000 5.700000     9.000000
## 4     Scrub FALSE 3.350000 5.000000 4.700000     7.000000
## 5 Grassland  TRUE 1.500000 4.000000 5.000000     7.000000
## 6    Meadow  TRUE 3.466667 1.666667 4.933333     6.333333
## 7     Scrub  TRUE 1.500000 9.000000 4.900000     3.500000
```

```r
#by: crea funciones para cada nivel del factor especificado
by(worms$Worm.density, Vegetation, mean)
```

```
## Vegetation: Arable
## [1] 5.333333
## ------------------------------------------------------------
## Vegetation: Grassland
## [1] 2.444444
## ------------------------------------------------------------
## Vegetation: Meadow
## [1] 6.333333
## ------------------------------------------------------------
## Vegetation: Orchard
## [1] 9
## ------------------------------------------------------------
## Vegetation: Scrub
## [1] 5.25
```

5.7 Familia Apply

Las funciones *apply*, *tapply*, *sapply*, *lapply*, *rapply*, *mapply*, se utilizan para aplicar una función específica a cada columna o fila de un objeto en R. Son generalmente mucho más eficientes que un bucle, más claras y directas.
La orden *apply* se utiliza para matrices y nos devuelve un vector, array o lista. necesita 3 argumentos *apply(x, margin, function)*, donde *x* es la matriz a ingresar, *margin* indica si aplicaremos la función por filas (1) o columnas (2), y *function* es la función a aplicar (e.g.).

La función *sapply* simplifica el resultado a un vector o una matriz (“s” de “simplify”; *sapply(list,fun)*).

La función *lapply* siempre devuelve una lista (*lapply(list,fun)*), la función *tapply* una tabla (*tapply(x,factor,fun)* ingresamos un vector, un factor de agrupación y la función a aplicar), la función *vapply* trabaja como sapply pero necesitamos especificar el tipo de valor que queremos obtener (*vapply(x,fun,fun.value)*) y *mapply* es la versión multivariada de *sapply* (*mapply(FUN,…)*).

```r
require(stats)

## apply ----------------------------
## creamos una matriz con la que trabajar
x=matrix(1:6,2,3)
x
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```r
sum(x)
```

```
## [1] 21
```

```r
#aplicar la función suma por filas (1)
apply(x,1,sum)
```

```
## [1]  9 12
```

```
#aplicar la función suma por columnas (2)
apply(x,2,sum)
```

```
## [1]  3  7 11
```

```
## sapply -----------------------------
sapply(1:5,sqrt)
```

```
## [1] 1.000000 1.414214 1.732051 2.000000 2.236068
```

```
list<-list(a=c(1,1), b=c(2,2), c=c(3,3))
sapply(list,sum) #devuelve un vector suma de a, b y c
```

```
## a b c
## 2 4 6
```

```
list<-list(a=c(1,2), b=c(1,2,3), c=c(1,2,3,4))
sapply(list,range) #devuelve una matriz con el rango de a, b y c
```

```
##      a b c
## [1,] 1 1 1
## [2,] 2 3 4
```

```
## lapply -----------------------------
list<-list(a=c(1,1), b=c(2,2), c=c(3,3))
lapply(list, sum)
```

```
## $a
## [1] 2
##
## $b
## [1] 4
##
## $c
## [1] 6
```

```
## tapply ----------------------
head(warpbreaks)
```

```
##   breaks wool tension
## 1     26    A       L
## 2     30    A       L
## 3     54    A       L
## 4     25    A       L
## 5     70    A       L
## 6     52    A       L
```

```
tapply(warpbreaks$breaks, warpbreaks[,-1], sum)
```

```
##     tension
## wool   L   M   H
##    A 401 216 221
##    B 254 259 169
```

```r
## vapply -----------------------
vapply(list(i=1:25,j=5,k=5),fivenum,
       c("Min."=0,"1stQu."=0,"Median"=0,"3rdQu."=0,"Max."=0))
```

```
##         i j k
## Min.    1 5 5
## 1stQu.  7 5 5
## Median 13 5 5
## 3rdQu. 19 5 5
## Max.   25 5 5
```

```r
## mapply -----------------------
list(rep(1,4), rep(2,3),rep(3,2), rep(4,1))
```

```
## [[1]]
## [1] 1 1 1 1
##
## [[2]]
## [1] 2 2 2
##
## [[3]]
## [1] 3 3
##
## [[4]]
## [1] 4
```

```r
# en lugar de repetir la función rep, podemos usar
mapply(rep, 1:4, 4:1)
```

```
## [[1]]
## [1] 1 1 1 1
##
## [[2]]
## [1] 2 2 2
##
## [[3]]
## [1] 3 3
##
## [[4]]
## [1] 4
```

5.8 Gráficos

Para conocer el potencial de R en la realización de gráficos de alto nivel, puedes escribir la siguiente orden en la consola:

```r
demo(graphics)
# y también mira
demo(lattice, package = "lattice", ask = TRUE) #debes tener instalado el paquete lattice
```

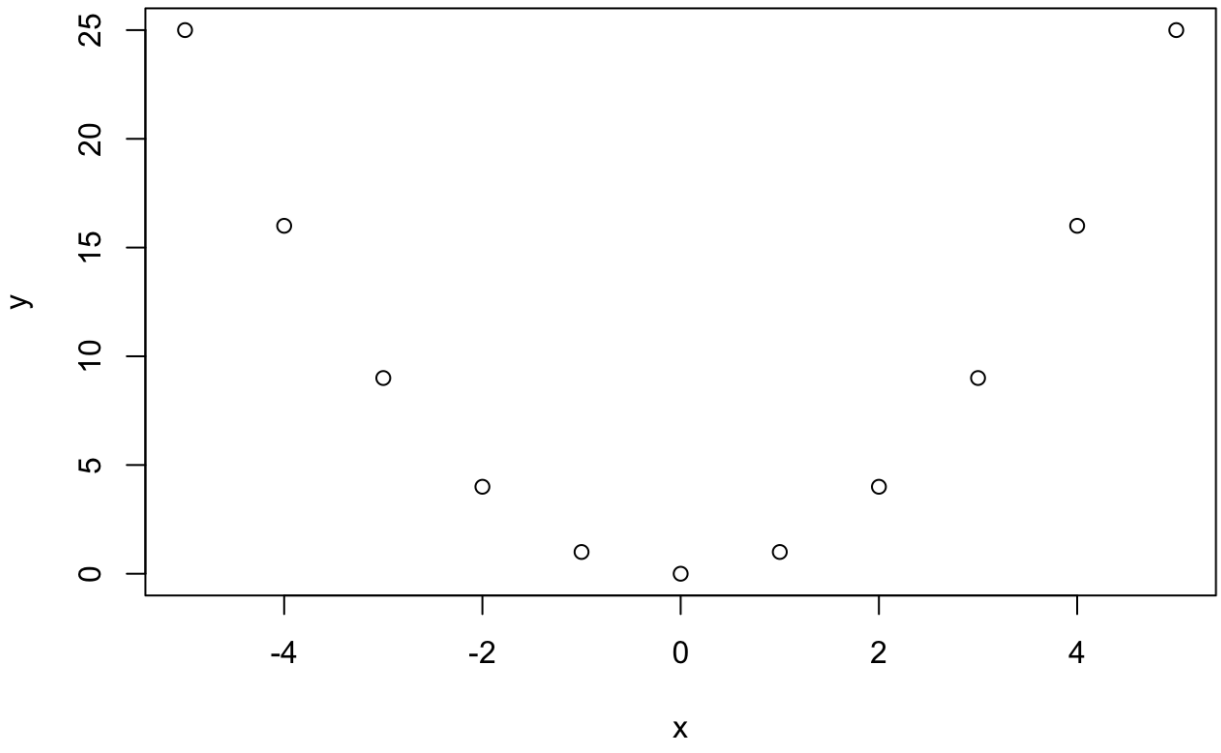
Aquí tienes una breve lista de referencia, para que consultes en R.

Función	Descripción
demo(graphics)	demostración de gráficos
?plot	gráfico de dispersión; x e y son vectores
?par	selección de parámetros gráficos
?layout	configuración del gráfico
example("pch")	lista de estilos de puntos
colours()	lista de colores
?plotmatch	lista de símbolos matemáticos

Figura 17. Funciones gráficas para consultar

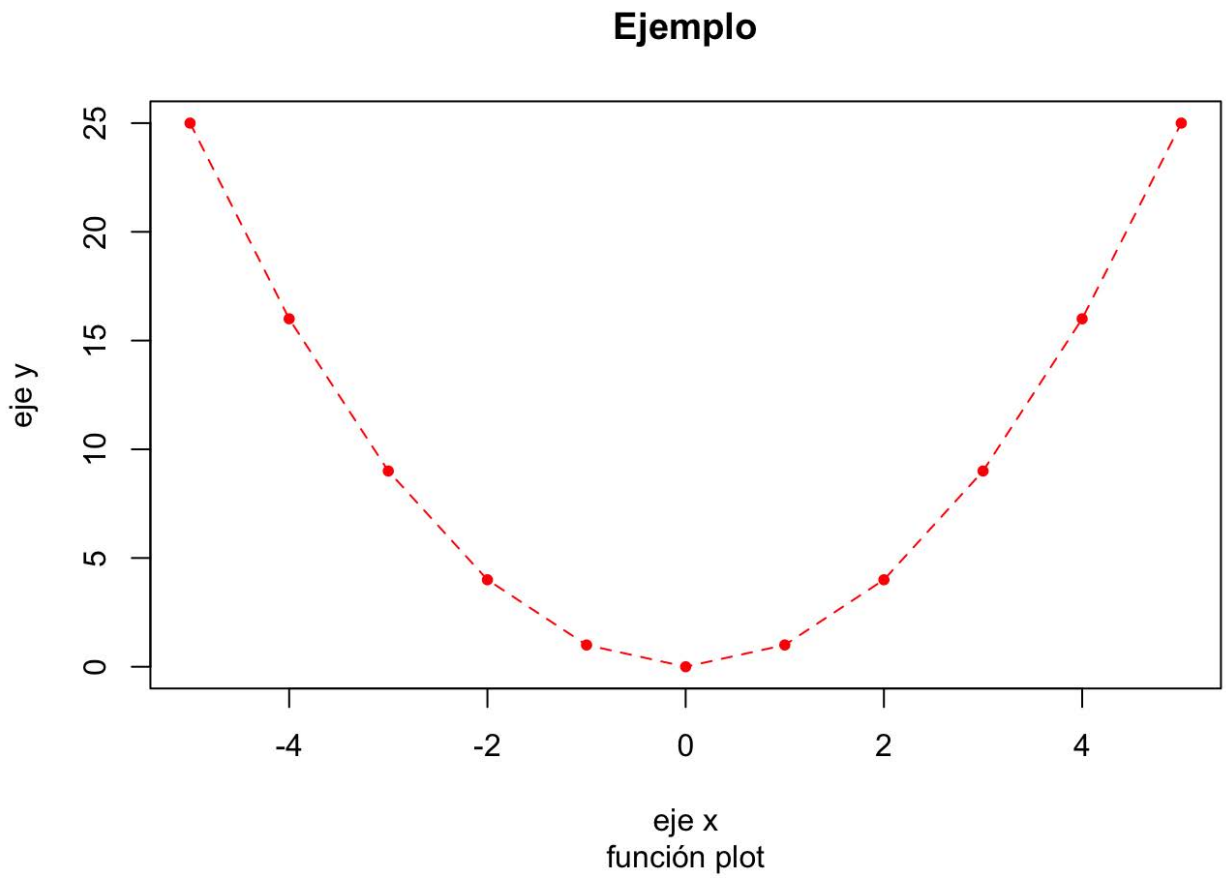
El comando más sencillo de R para graficar es la función *plot*. Los argumentos que podemos seleccionar son: *type: "l" para representar líneas, "b" para representar puntos unidos por líneas , "s" tipo escalera, "h" tipo histograma, "n" sin linea y "p" para representar puntos (por defecto).*
pch: para seleccionar el tipo de punto
lty: selecciona el tipo de línea
lwd: cambia el ancho de la línea
col: color de las líneas
font: fuente del texto

```
x<-c(-5:5)
y<-x^2
plot(x,y)
```




```
# podemos mejorar el gráfico mediante sus argumentos
```

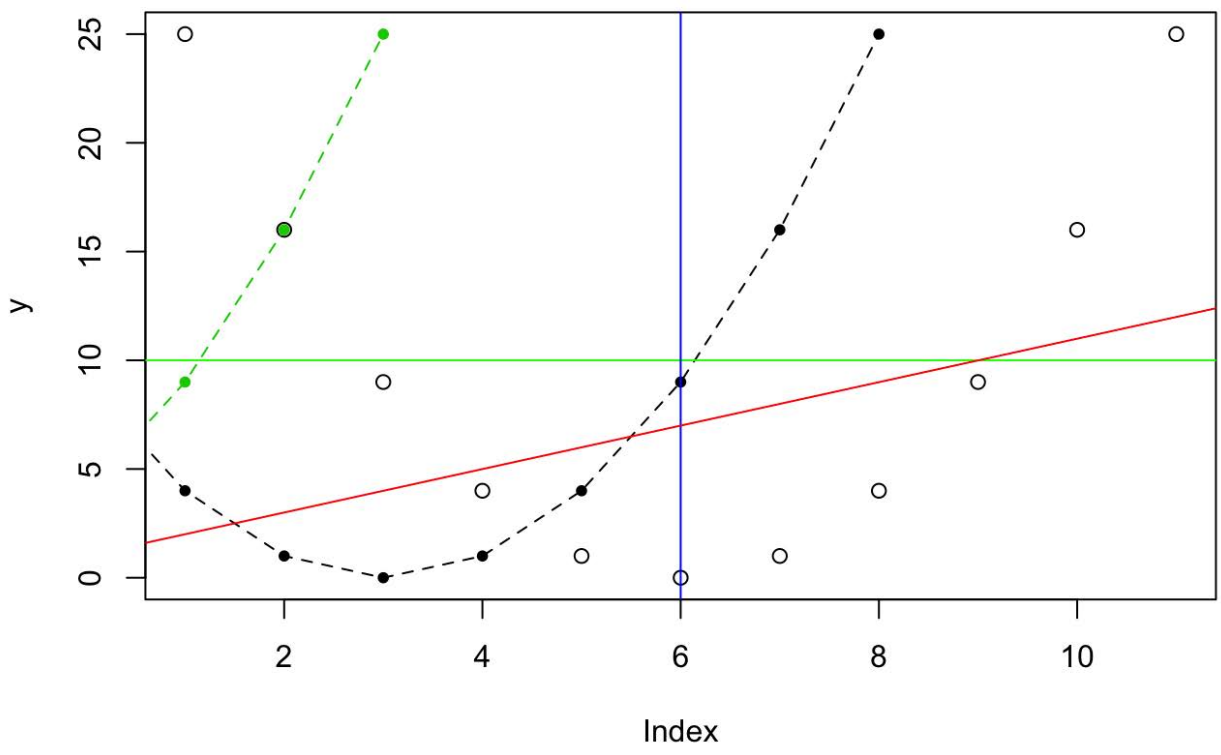
```
plot(x,y, main="Ejemplo",sub="función plot",
     type="o",pch=20,lty=2,col=2,
     xlab="eje x", ylab="eje y")
```




```
# aquí seleccionamos el tipo de línea "both", es decir, con punto y línea (type)
# seleccionamos el tipo de punto 20, que es un círculo relleno (pch)
# la línea es quebrada (lty)
# el color=2 corresponde al rojo
# elegimos un título (main) y un subtítulo (sub)
# también seleccionamos el nombre de los ejes x e y (xlab e ylab)
```

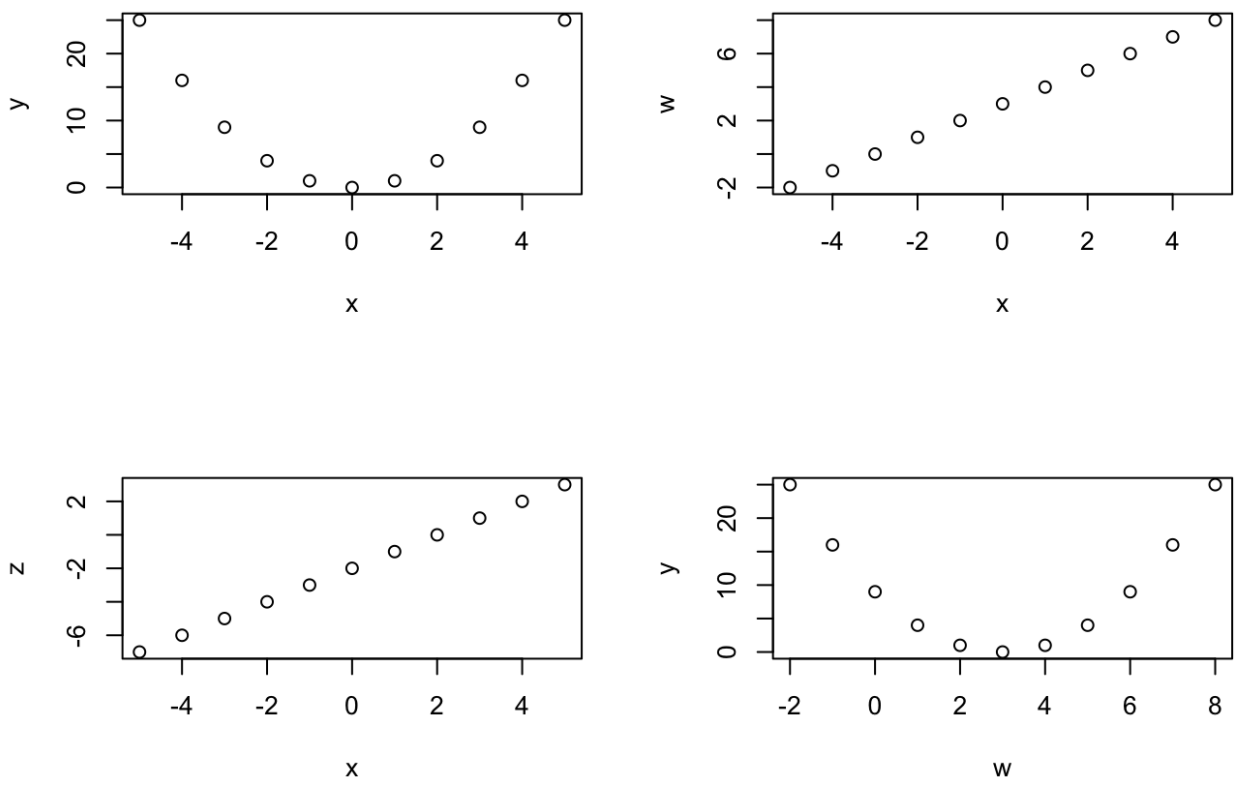
Otros comandos básicos para agregar características a un gráfico ya existente:
points: añade puntos
lines: superpone funciones
text: añade un texto
legend: añade una leyenda

```
w<-x+3
plot(y)
points(w,y,type="o",pch=20,lty=2,col=1)

z<-x-2
lines(z,y,type="o",pch=20,lty=2,col=3)

text(0,20,"Texto")

abline(h=10, col="green")
abline(v=6, col="blue")
abline(a=1,b=1,col="red")
```



Si queremos generar múltiples gráficos en una misma ventana, utilizaremos la función :

```
# mfrow divide la ventana en c(filas, columnas)
par(mfrow=c(2,2))
plot(x,y)
plot(x,w)
plot(x,z)
plot(w,y)
```



Para guardar un gráfico podemos utilizar:

Función	Descripción
pdf("mygraph.pdf")	genera un archivo pdf
png("mygraph.png")	genera un archivo png
jpeg("mygraph.jpg")	genera un archivo jpeg
bmp("mygraph.bmp")	genera un archivo bmp
postscript("mygraph.ps")	genera un archivo windows metafile
win.metafile("mygraph.wmf")	genera un archivo postscript
dev.off()	cierra un dispositivo gráfico

Figura 18. Funciones útiles para guardar gráficos.

Por ejemplo:

```
x<-1:10
y<-2:11

pdf("grafico_prueba.pdf")
plot(y~x)
dev.off()
```

```
## quartz_off_screen
##                 2
```

En el próximo tema veremos más opciones gráficas y especificaremos en qué caso utilizar una u otra. Sin embargo, es interesante que le des un vistazo al paquete , ya que es una herramienta gráfica muy potente. La función puede utilizarse para la mayoría de los gráficos habituales.

5.9 Trabajar con texto

Una colección de letras o palabras se llama . Si queremos concatenar texto debemos utilizar la función , mientras que si queremos separarlo utilizaremos . A continuación veremos un ejemplo:

```
y<-c("Hello","word")
paste(y,collapse=" ")
```

```
## [1] "Hello word"
```

```
(x<-paste("Hello","world",sep=" ")) #utilizamos sep para especificar la separación
```

```
## [1] "Hello world"
```

```
strsplit(x," ")
```

```
## [[1]]
## [1] "Hello" "world"
```

```
#cuidado! nos devuelve una lista
```

El argumento especifica cómo se unen (e.g. con un espacio en blanco , sin espacio , mediante una coma).

Si queremos cambiar de mayúsculas a minúsculas usaremos:

```
tolower(x)
```

```
## [1] "hello world"
```

```
toupper(x)
```

```
## [1] "HELLO WORLD"
```

Para operar con texto podemos utilizar las siguientes funciones:

```
# para extraer un sub-texto desde la posición start hasta la posición stop
substr("Hello world",start=3, stop=5)
```

```
## [1] "llo"
```

```
# encontrar un patrón en un vector: grep(pattern, x)
z<-c("comandos","directorio","grafico","consola")
grep("co", z)
```

```
## [1] 1 3 4
```

```
grepl("co", z) # ídem, pero lógico
```

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

```
# sustituir un patrón por un reemplazo: sub(pattern, replacement, x)
sub("e",".", c("El","lunes"))
```

```
## [1] "El"    "lun.s"
```

```
gsub("e",".", c("El","lunes"))
```

```
## [1] "El"    "lun.s"
```

```
# también puedes investigar otras funciones como:
# regexpr("ma", z)
# gregexpr("ma", z)
# regexec("ma", z)
```

Si quieres obtener funciones más avanzadas para trabajar con texto, puedes mirar el paquete .

5.10 Control de errores: traceback, browser, debug

Cuando queremos investigar por qué y dónde se produce un error en la programación, debemos utilizar las funciones:
* , cuando no comprendemos el mensaje de error, esta función nos indica la secuencia de llamadas antes del problema.
* , cuando queremos parar la ejecución en el punto de error, para continuar examinando la programación. La opción “n” nos permite continuar el código paso a paso y la opción “Q” es para salir.
* , similar a la función anterior, pero desde el comienzo del código. La opción “Q” se utiliza para salir.

Ejemplo:

```
# traceback -------------------
#error por el mal uso de una función automática
 foo<-function(x){ print(1); bar(2) }
 bar<-function(x){x+a.variable}
 foo(2) # da un error extraño
 traceback() #nos dice cuál es el error

# debug -----------------------------
#revisar paso a paso una función
require(stats)
centre <- function(x, type) { switch(type, mean = mean(x),  median = median(x), trimmed = mean(x, trim =
.1)) }
x <- rcauchy(10)
centre(x, "mean")
debug(centre)
centre(x, "mean")
undebug(centre)
```

5.11 Control de la duración de la operación:

Ejemplo:

```
#  unix.time(centre(rnorm(10),mean)) #unix.time, Rprof
 require(stats)
 system.time(for(i in 1:10) mad(runif(10)))
```

```
##    user  system elapsed
##   0.001   0.000   0.001
```
