ÍNDICE GENERAL ERROR! BOOKMARK NOT DEFINED.

PARTE V PRESENTACIONES VISUALES **1**

CAPÍTULO 1 GRÁFICOS **2**

1.1. plot() **2**
1.1.1. Adornos 5
1.1.2. Colores 6

1.2. Gráficos especiales **9**

1.3. ggplot **9**
1.3.1. Histogramas, Diagramas de caja y de barra 12

1.4. levelplot() **13**

1.5. Series temporales **15**

1.6 Técnicas gráficas en el Análisis Exploratorio de Datos (EDA) **15**

PARTE VI ESTRUCTURAS DE CONTROL **18**

1.6. Vectorización **19**

1.7. Iteración **19**
1.7.1. bucle for 19
1.7.2. bucle repeat 21
1.7.3. bucle while 21
1.7.4. familia apply(), lapply(), sapply(), tapply() 22

1.8. Carga automatizada de archivos **24**

1.9. Estructuras de ejecución condicional **25**
1.9.1. Condición sin alternativas 25
1.9.2. Condición con una alternativa 26
1.9.3. Ejemplo C 27
1.9.4. Condición con varias alternativas 27

PARTE VII **30**

1.10. Proyectos **31**

1.11. Scripts **31**
1.11.2. Buenas prácticas en programación 33

Parte V Presentaciones visuales

Capítulo 1 Gráficos

R es reconocido por la diversidad y calidad visual de las rutinas gráficas. Tales recursos son de gran ayuda en la exploración preliminar de conjuntos de datos, antes de iniciar el análisis profundo. De hecho, los recursos gráficos son esenciales en lo que se conoce como Análisis Exploratorio de Datos, metodología que se verá en el Módulo 4 de este curso. Pero también son recursos valiosos al momento de presentar los resultados que hayamos obtenido. En el presente capítulo se exploran varios de los recursos gráficos más usados de R, incluyendo el manejo de series temporales; en los ejemplos de aplicación se ven otros.

Veremos rutinas gráficas que se instalan con R base y otras que se incorporan mediante paquetes externos.

1.1. plot()

La función plot() en una función del tipo genérico, lo que implica que el tipo de gráfico que devuelve depende del tipo de objeto R que le pasemos; opera con datos numéricos y categóricos. La función plot() descarga con el paquete graphics, parte de la instalación base de R. La sintaxis es

> plot(x, y, FUN) donde x e y son vectores con las coordenadas y opcionalmente una función matemática, FUN. Si se ingresa un único vector x, plot() grafica x en función de un índice. x y FUN pueden combinarse.

El caso más sencillo es con un único vector que representa una función matemática

```
> plot(1:100)
> plot(sqrt(1:100))
> plot((1:100)^3)
```

y también puede ser una columna de una matriz o marco de datos.

```
> plot(mtcars$mpg)
```

Emplear una variable para introducir los argumentos

```
> a <- 1:100
> plot(a,sqrt(a))
```

El argumento puede incluir una fórmula (Figura 1.1)

```
> plot((0:20)* pi/10, sin((0:20)*pi/10))
```

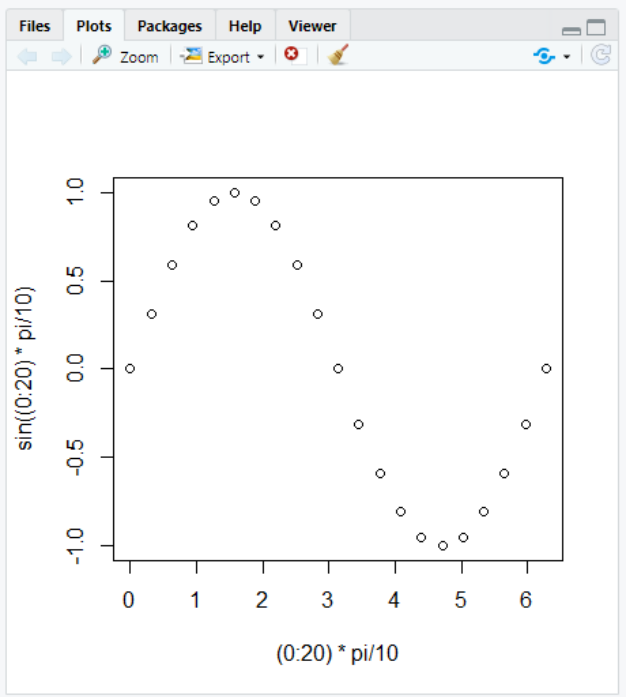


Figura 1.1: *Gráfico con plot.*

Los ejemplos anteriores tienen una única variable, pero se pueden emplear dos variables

```
> x <- seq(1,10,0.5)
> y <- 30*x/(2+x)
> plot(x,y)
```

Si plot() se aplica a un marco de datos devuelve un conjunto de gráficos de dispersión bivariados. (Figura 1.2)

```
> plot(iris)
```

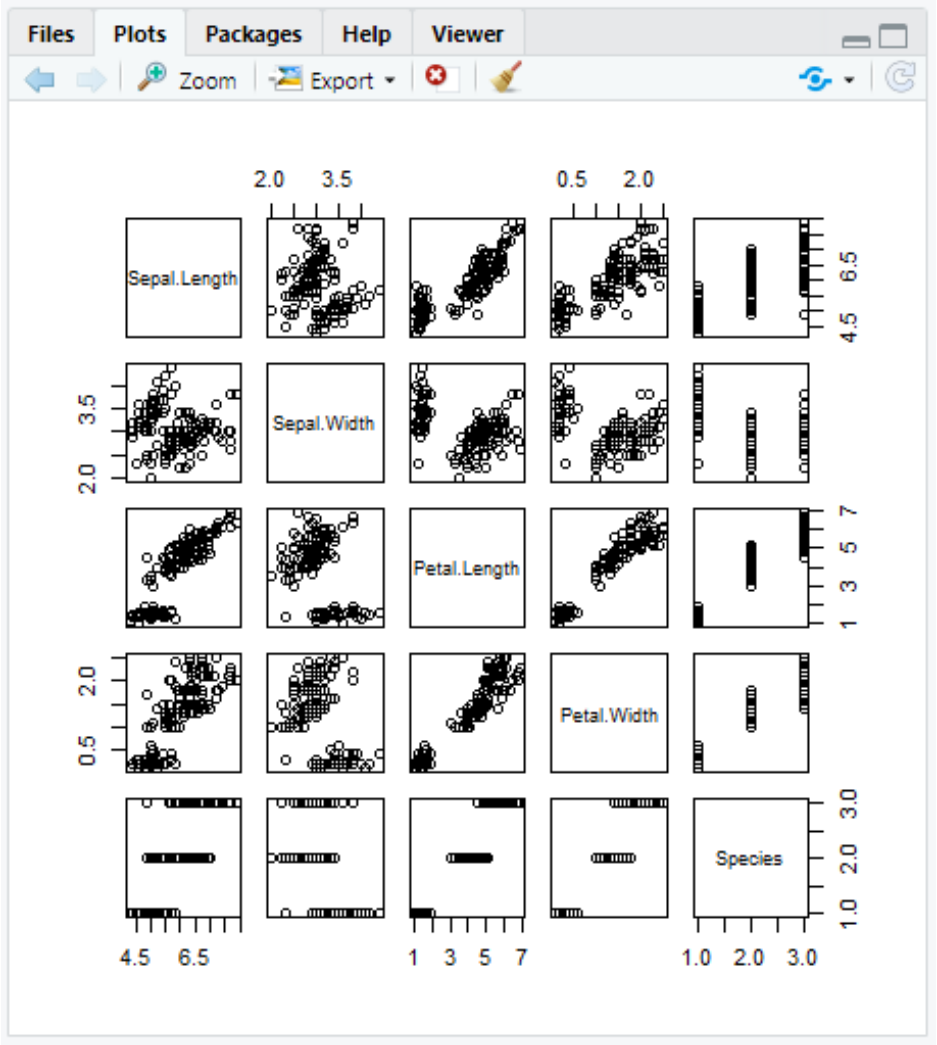


Figura 1.2: *Diagramas de dispersión con plot.*

Lo que hace plot() es llamar la función pairs() (ejemplificando el carácter genérico de plot) que toma cada par de variables en el marco de datos y las muestra separadamente como gráficos de dispersión.

Pero se puede discriminar

```
> plot(iris$Sepal.Length)
```

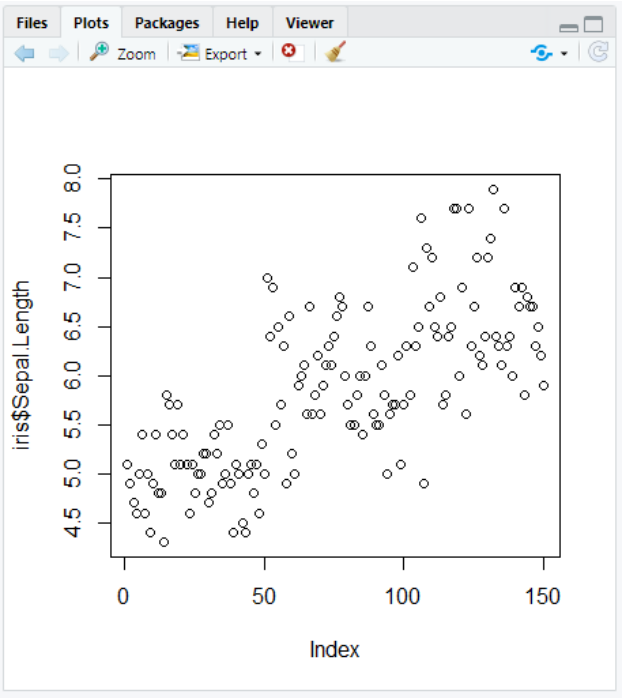


Figura 1.3: Diagrama de dispersión con plot.

1.1.1. Adornos

plot() acepta argumentos para definir color, forma y tamaño de marcadores y líneas. La mayor parte de los argumentos derivan de la función par(), por Graphical Parameters. Con

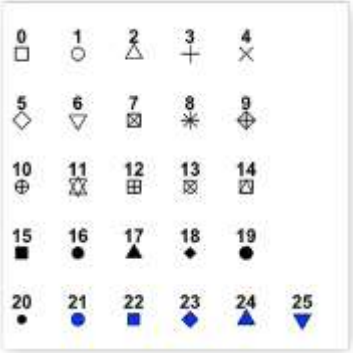
```
> names(par())
```

se obtiene un listado de los argumentos disponibles: cex, col, las, lwd, pch, bg y mfrow son algunos de los más usados.

Con

help("par") se obtiene ayuda para cada argumento en RStudio.

Algunas definiciones:
cex – tamaño de los marcadores relativo al tamaño por defecto, cex =1
lwd – ancho de línea
pch – 25 formas de marcadores
col – color de marcadores



Muestrario de las figuras de pch con los identificadores. Un ejercicio en plot.Rmd da código para imprimir estos símbolos

El siguiente ejemplo grafica 5 figuras geométricas tomadas del muestrario pch, con tamaño definido por cex, grosor de línea con lwd, y color por col.

```
> plot(1:2, cex=1, lwd=3, pch=1:5, col = rainbow(25))
```

El argumento type se combina con par para definir el estilo de la línea: “p” puntos, “l” línea sólida, “b” puntos y líneas.

```
> plot((0:20)*pi/10, sin((0:20)*(pi/10)), type='l', lty='dashed')
```

Al graficar la Figura1.1, R por defecto denominó los ejes con la información provista en el primer término. Para incluir leyendas adecuadas y un título se hace:

```
> plot((0:20)*pi/10, sin((0:20)(pi/10)), main="La función seno", ylab="seno(x)", xlab="x")
```

Los argumentos main, xlab y ylab genera título, y etiquetas para los ejes.

1.1.2. Colores

Los colores son componentes esenciales en la visualización y por ello es importante comprender cómo representarlos correctamente, un tema que trasciende un lenguaje de programación y que excede el propósito de este curso. No obstante, es conveniente decir unas palabras sobre cómo trata los colores R. R tiene almacenados cientos de colores:

```
> > colors()
```

devuelve un listado de los 657 colores almacenados en R. En los ejemplos dados el color se declaró en el mismo código gráfico. Pero es posible declarar el conjunto de colores que se ha de usar en un proyecto al inicio del código, como vector:

```
> colores <- c('chartreuse3', 'cornflowerblue', 'darkgoldenrod1','peachpuff3','mediumorchid2', 'turquoise3', 'wheat4', 'slategray2')
```

Un formato muy utilizado para almacenar colores es el hexadecimal. Cada color es representado por tres pares de caracteres precedidos por #. Los pares corresponden al conocido formato RGB "#rrggbb".

colores <- c("#CC1011", "#665555", "#05a399", "#cfcaca", "#f5e840", "#0683c9", "#e075b0") Los colores puros se representan con: azul - "#0000FF", verde - "#00FF00", y rojo - "#FF0000". El blanco - "#FFFFFF el negro "#000000".

El sitio indicado a continuación da útiles ayudas para diagramar mejor los gráfi- cos: https://www.stat.auckland.ac.nz/~paul/RGraphics/chapter3.html

Ejemplo con plot()

El script plot.Rmd incluye numerosos ejemplos de código para graficar con plot(). Se sugiere que ejecuten el código paso a paso identificando las diferencias y los códigos respectivos.
Ver video Plot.mp4.

Gráficos múltiples

La comparación de dos o más gráficos se facilita si se colocan próximos entre sí. Se vio arriba que cuando plot() se aplica a un marco de datos por defecto genera múltiples gráficos formando un panel. Alternativamente, la función par() permite acomodar varios gráficos en una página empleando el parámetro mfrow. Opera como si se tratara de una matriz, con filas y columnas.

Se trata de acomoda r dos gráficos lado a lado.

```
> par(mfrow <- c(1,2))
```

divide el canvas en filas y columnas virtuales y coloca cada gráfico en la intersección; en este caso dos gráficos en una misma fila

```
> x <- seq(1,10,0.5)
> y <- 30*x/(2+x)
> plot(x, y, pch=20, cex=1, col='red', type='b')
> plot(x, y, pch=20, cex=1, col='blue', type='l')
```

> *par(mfrow<-c(1,1) (Figura1.4). El último comando devuelve el canvas al estado original.*

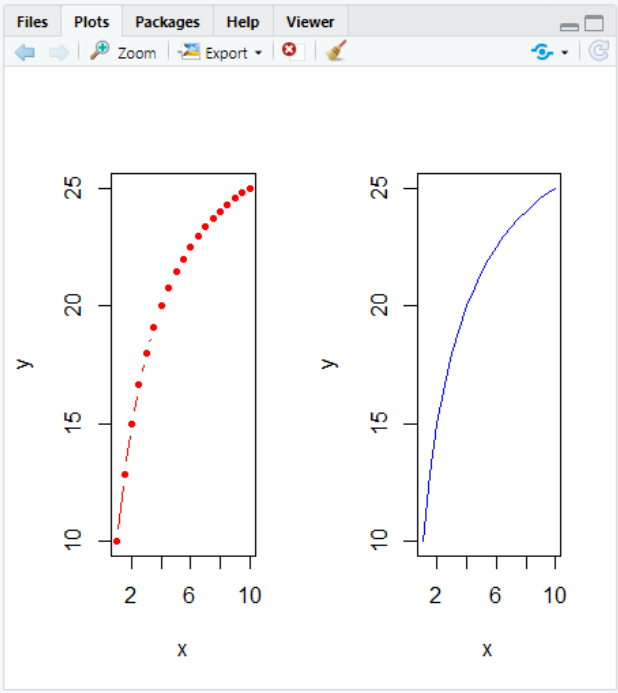


Figura 1.4: *Gráfico lado a lado.*

Cambiar a mfrow <- c(2, 1) y repetir.

1.2. Gráficos especiales

Aparte de los gráficos de dispersión vistos arriba, el paquete graphics permite generar gráficos de barra, de frecuencia e histogramas.

El vector hab <- c(20,30,10,15) representa la proporción de Hombres, Mujeres, Niños, Niñas, en una ciudad. Se desea representar estos datos con un gráfico de barras. Como primera acción se asignan nombres a las columnas con names(hab)<-c('Hombre','Mujer','Niño','Niña'). Ahora se grafica

```
> barplot(hab)
```

Si la barras quedan demasiado juntas se la espacia con

```
> barplot(hab,space=2)
```

Y se colorean con

```
> barplot(hab,space = 2,col = "dodgerblue3")
```

S incorpora una tercer variable, la edad. Se tienen los vectores con porcentajes de Hombres y Mujeres de determinadas edades a <- c(11.7, 8.7) b <- c(18.1, 11.7) c <- c(26.9, 20.3) donde a, b y c representan las franjas etáreas: 20-30, 30-40 y 40-50. Crear la matriz y completarla

```
> mat<-rbind(a,b,c)
> colnames(mat)<- c('Hombre','Mujer')
> rownames(mat)<-c('20-30','30-40','40-50')
```

Graficar

```
> barplot(mat2, legend= rownames(mat))
```

O en este formato

```
> barplot(mat2, legend= rownames(mat2),beside = T)
```

O en arreglo horizontal

```
> barplot(mat2, horiz=T,legend= rownames(mat2), beside = T)
```

1.3. ggplot

Ver video ggplot.mp4.

La función ggplot() se carga con el paquete tidyverse o individualmente con el paquete ggplot2, del repositorio CRAN.

El paquete HistData reune conjuntos de datos que se han hecho famosos en estadística. Una descripción de este paquete se obtiene con

```
> ?HistData
```

Uno de los conjuntos de datos en HistData es GaltonFamilies, que fue compilado por Galton en 1886 para estudiar la relación entre las alturas de los padres y las de los hijos. La compilación comprende 934 hijos de 205 familias. Instalar y activar HistData

Revisar los primeros registros

```
> head(GaltonFamilies)
```

Revisar la estructura

> str(GaltonFamilies)

Se reconocen como factores family y gender; family da el identificador de familia de 001 a 205, o sea que tiene 205 levels; gender sólo tiene 2 levels. childNum también está representado por variables categóricas, aunque R no lo reconozca así.

> names(GaltonFamilies)

devuelve los encabezamientos

En este ejercicio se utilizan las variables father, mother, gender y childrenHeight. Retener estas y eliminar las demás

> gf <- GaltonFamilies[, c(2,3,7,8)]

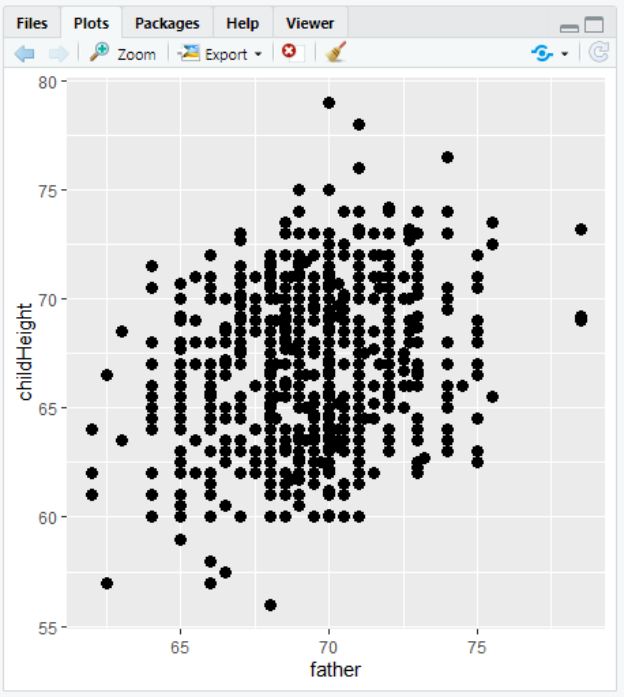


Figura 1.5: *Gráfico de dispersión.*

Se trata de graficar las alturas de los hijos en función de las alturas de los padres. Escribir

> ggplot(gf, aes(x = father, y = childHeight))

El argumento aes (aesthetic mapping) describe cómo se representa visualmente el dato. Los valores x e y dan las coordenadas y, por defecto, también los identificadores de ejes; aes puede incluir ecuaciones; otro valor posible en aes es color <- 'blue'. El comando escrito arriba prepara el canvas y dibuja los ejes pero carece de instrucciones para graficar los datos. Las instrucciones se agregan a continuación con el comando geom_point(), que se suma al anterior.

> ggplot(gf, aes(x = father, y = childHeight)) + geom_point(size = 3)

(Figura 1.5).

Se puede aprovechar que se distingue entre hijos e hijas para controlar aun más el gráfico. Hacer

> p+geom_point(size<-3,aes(col<-gender)) En el comando aes se agregó

gender como control del color. Probar agregar shape<-gender.

geom_point() comanda la impresión de puntos cuyas coordenadas están dadas por las alturas de los padres y de los hijos/hijas. Notar que ambas instrucciones se suman. Esta es la base gramatical de ggplot2. Los comandos

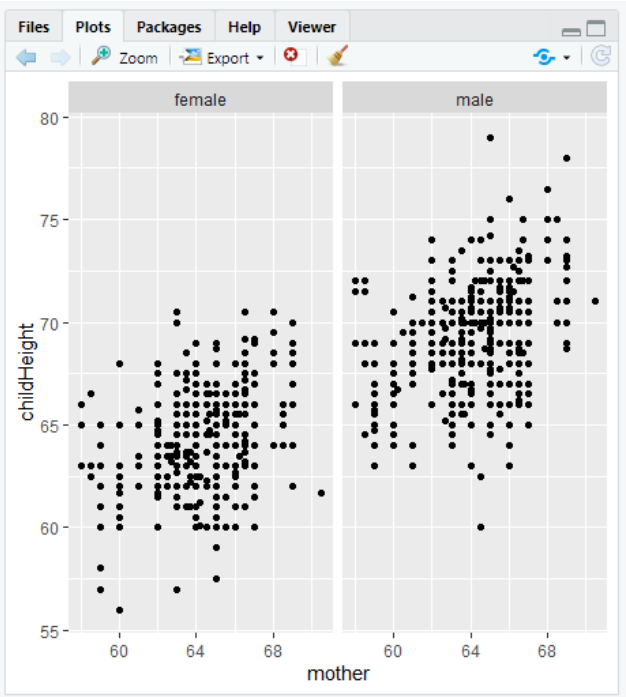


Figura 1.6: *Gráficos lado a lado.*

se acumulan. Cambiar size <- 1; este es el tamaño por defecto.

Incorporar etiquetado de ejes y leyendas implica sumar más capas a ggplot(). Se enfatiza la estructura aditiva de ggplot(). Se asigna el gráfico base a un objeto p, y a este objeto se suman las capas

> p <- ggplot(gf, aes(x = father, y = childHeight)) > p+geom_point()

> p+geom_point()+xlab(."lturas Padres biologicos")+ylab(."lturas Hijos/Hijas")+ggtitle(."so de alturas padres-vastagos")

Seguidamente repetir con las madres

> ggplot(gf, aes(x = mother, y = childHeight)) + geom_point()

En plot() se usó la función par() para acomodar gráficos en un panel. En ggplot() se utiliza la función facet_wrap()

> ggplot(gf, aes(x = mother, y = childHeight)) + geom_point() + facet_wrap(~gender) (Figura1.6).

El comando se suma a los anteriores. El operador ~ indica usar el factor gender como variable de control, o de agrupamiento. Ambos sexos muestran tendencias similares y un desplazamiento en altura.
Si se desea distinguir los conjuntos con colores hay que modificar el argumento aes.

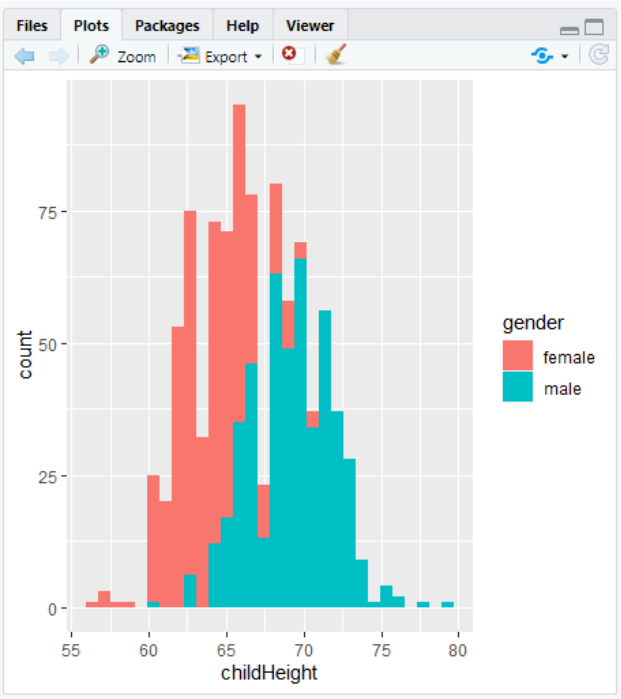


Figura 1.7: *Histograma.*

> ggplot(gf, aes(mother, childHeight, color= gender)) + geom_point() + facet_wrap(~gender)

1.3.1. Histogramas, Diagramas de caja y de barra

Empleando datos de GaltonFamilies

> ggplot(gf, aes(childHeight, fill = gender)) + geom_histogram() (Figura 1.7).

El argumento fill da color pleno al histograma controlado por género. Representar los mismos datos con una curva uniendo los puntosmedios

> ggplot(gf, aes(childHeight, color = gender)) + geom_freqpoly(binwidth = 1, center = 70,size = 2).

Y también crear un diagrama de barras con las alturas de los niños usando la opción geom_col()

> p+geom_col(mapping = aes(x=gender,y=childHeight)

Y repetir pero con la opción geom_bar

> ggplot(gf, aes(x=gender)) + geom_bar()

Inspeccionar los resultados. geom_col sumó los valores de las alturas. Verificar con sum(gf$childHeight) que da 62340.7. Mientras que geom_bar cuenta los individuos, como se verifica haciendo summary(gf) que da 481 varones y 453 mujeres.

El diagrama de cajas (boxplot) es muy útil para explorar la dispersión de valores. El alto del rectángulo está determinado por los cuartiles, los extremos de la recta vertical indica el rango y los puntos aislados los valores extremos (outliers). La función es geom_boxplot().

> ggplot(airquality, aes(y=Ozone, x=1)) + geom_boxplot(fill<-'green') (Figura1.8).

Un mensaje advierte de datos faltantes.

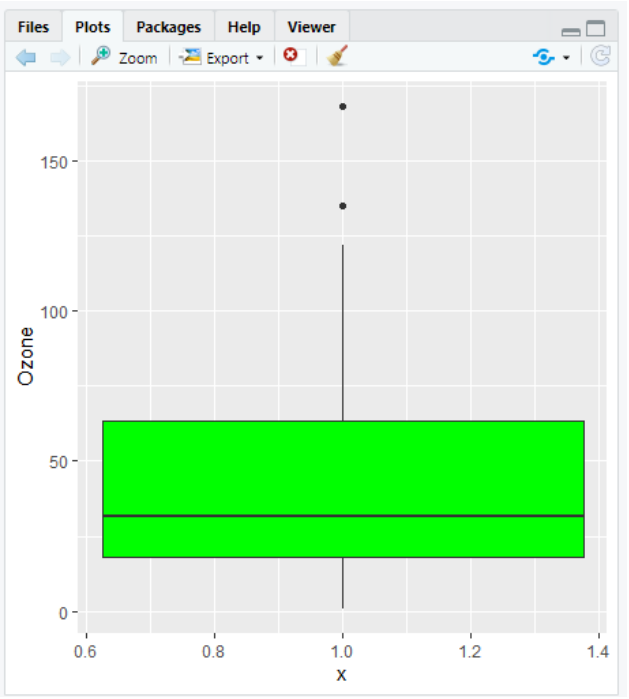


Figura 1.8: *Gráfico de cajas.*

La línea horizontal marca la mediana. Valores mayores que la mediana mues- tran una mayor dispersión. Los dos puntos aislados arriba son valores extremos.

Ejemplo con ggplot()

Ver script ggplot.Rmd

Se sugiere que ejecuten el código paso a paso identificando las diferencias y los códigos respectivos.

1.4. levelplot()

La función levelplot() representa una superficie 3D en un plano discriminando las variaciones topográficas, o equivalentes, con colores. levelplot() es parte del paquete lattice, que implementa mejoras respecto de las rutinas gráficas en el paquete base. Una de las mejoras es que permite expresar relaciones condicionales entre grupos de variables: y ~ x | z, y versus x dada la condici'on z. levelplot() es una de las funciones en lattice. La sintaxis elemental por defecto es

levelplot(formula | variables, data = un dataframe)
Donde fórmula establece una relación: z ˜ x * y, con z la variable respuesta y x y son variables numéricas evaluadas sobre una grilla rectangular.
Adicionalmente, el operador pipe, |, permite incluir factores adicionales. Por ejemplo, z ~ x | A indica mostrar la realción entre z y x separadamente para cada nivel del factor A.

Ejemplo

Tomado de RDocumentation

```
> library(lattice)
> x <- seq(pi/4, 5 * pi, length = 100)
> y <- seq(pi/4, 5 * pi, length = 100)
> r <- as.vector(sqrt(outer(xˆ2, yˆ2, '+')))
> grid <- expand.grid(x=x, y=y)
```

Crear un marco de datos como el producto cartesiano de x e y

```
> grid$z <- cos(rˆ2) * exp(-r/(piˆ3))
> levelplot(z ~ x*y, grid, cuts = 50, scales=list(log='e'), xlab='', ylab='',
main<-'Vistoso diseño', sub = 'con escalas logs', colorkey = FALSE, region
= TRUE)
```

Ver

```
> ?levelplot para más detalles
```

1.4.1. Exportar gráficos

Los gráficos se exportan como imágenes bitmap: jpeg, png, bmp y tiff, o como imágenes vectoriales: pdf, ps.

Ejemplo

Primero se llama a la función de exportación especificando el nombre que se da a la salida

```
> jpeg(file<-'Mi_grafico.jpg')
```

Luego se ejecuta el dibujo

```
> par(mfrow<-c(1,2))        > x <- seq(1,10,0.5)
> y <- 30*x/(2+x)
> plot(x, y, pch=20, cex=1, col='red', type='b')
> plot(x, y, pch=20, cex=1, col='blue', type='l')
> dev.off()
```

Finalmente se cierra el canvas o no se guardará la imagen.

El dibujo se guarda en el directorio de trabajo. Con jpeg() por defecto la resolución es 480x480 píxeles. Con png() la resolución se puede ajustar. Para exportar como PDF cambiar la primera línea a: pdf(file<-'Mi_grafico.pdf').

1.5. Series temporales

Cargar la biblioteca de ggplot2

> library(ggplot2)

Descargar los datos del sitio Figshare y dirigirlos a nuestro directorio de trabajo creado previamente.

> download.file('https://ndownloader.figshare.com/files/9282364', destfile<- 'boulder-precip.csv')

Luego importar los datos a R

> boulder_precip <- read.csv(file <- 'boulder-precip.csv')

Revisar la estructura con head() y str(). Verificar que las columnas DATE y PRECIP se leen correctamente escribiendo

> boulder_precip$DATE y boulder_precip$PRECIP. Graficar usando una rutina de ggplot2, es decir, no de R nativo.

> qplot(x = boulder_precip$DATE, y = boulder_precip$PRECIP) (Figura1.9).

Otros procedimientos gráficos se verán a lo largo del curso.

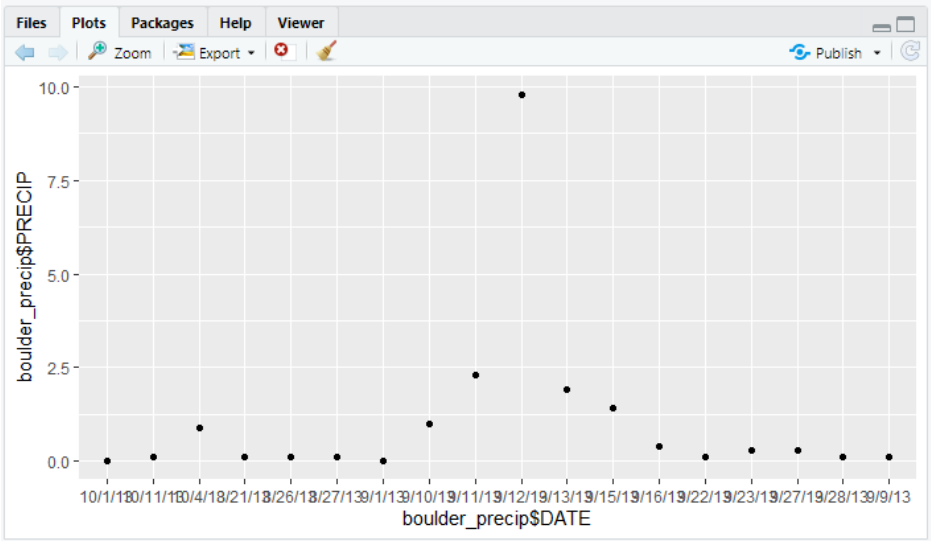


Figura 1.9: *Serie temporal con qplot.*

1.6 Técnicas gráficas en el Análisis Exploratorio de Datos (EDA)

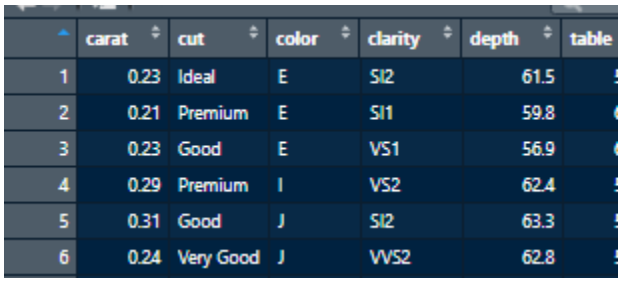
El Análisis Exploratorio de Datos reúne procedimientos para explorar un conjunto de datos antes de emplearlo en un análisis profundo. El EDA se basa fuertemente en la representación gráfica de los datos, tanto de datos categóricos como cuantitativos. Recordamos que un dato categórico, también llamado cualitativo, define la pertenencia de un objeto estadístico a una categoría o clase de acuerdo a alguno de sus atributos. Por ejemplo, color, sexo, club, tamaño. El valor puede ser un número pero generalmente porque se lo

asigna a la categoría. En cambio una variable, o dato, cuantitativo está representado por un número. El valor que toma puede ser continuo o discreto. En el primer caso la separación entre dos valores consecutivos es infinitésima, y en el segundo es finita.

No existen reglas fijas en el proceso de EDA. Pero en general nos interesa conocer dos asuntos: la variabilidad que muestran las variables en ese conjunto de datos, y si existe covariación entre las variables y de qué tipo.

Cuando se trabaja con variables categóricas un diagrama de barras es apropiado.
Usamos el conjunto diamonds que se instala con ggplot2.

```
library(ggplot2)
View(diamonds)
```



	carat	cut	color	clarity	depth	table
1	0.23	Ideal	E	SI2	61.5	
2	0.21	Premium	E	SI1	59.8	
3	0.23	Good	E	VS1	56.9	
4	0.29	Premium	I	VS2	62.4	
5	0.31	Good	J	SI2	63.3	
6	0.24	Very Good	J	VVS2	62.8	

La columna cut indica el tipo de corte, una variable categórica.
Creamos un diagrama de barras.

```
library(ggplot2)

ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = corte))
```

Cada categoría, es decir, cada corte, es asignado a una columna, y la altura de la columna corresponde al número de veces que se repite la categoría.
Puedo obtener un conteo de incidencias haciendo:

```
library(tidyverse)

diamonds %>% count(cut)
```

count es función del paquete dplyr que se carga con tidyverse.

Si la variable es cuantitativa continua, la distribución, la variabilidad, de valores se obtiene con un histograma. Como ejemplo graficamos la distribución del número de carates.

```
library(ggplot2)

ggplot(data = diamonds) +
  geom_histogram(mapping = aes(x = carat), binwidth = 0.5)
```

binwidth da el ancho de cada compartimiento.

El número de instancias por compartimiento se obtiene empleando la función cut_width(), que se instala con ggplot2.

```
library(ggplot2)
library(tidyverse)

diamonds %>% count(cut_width(carat, 0.5))
```

cut_width() requiere la variable y el intervalo de corte.

Una representación de covariación se obtiene con los diagramas de caja (box plots).
Usamos el conjunto de datos mtcars para comprobar la covarianza entre consumo y cilindrada.

```
library(ggplot2)

boxplot(mpg~cyl,data=mtcars,
      xlab="Numero de cilindros", ylab="Millas por galon")
```

Otra gráfico que muestra la covariación es el de dispersión.

```
library(ggplot2)

ggplot(data=diamonds) + geom_point(mapping = aes(x = carat, y = price))
```

El precio claramente aumenta con los carates pero con mucha variabilidad, posiblemente dependiendo del corte.

Parte VI Estructuras de control

En este capítulo se resumen funcionalidades de R útiles en programación. Se comienza por resaltar la propiedad de vectorización, que permite acortar mucho el código. Se sigue con un repaso de estructuras iterativas y condicionales, comunes en otros lenguajes y otras exclusivas de R. Este tipo de estructuras se suelen denominar de control debido a que gobiernan la distribución de acciones en el tiempo de ejecución.

1.6. Vectorización

R es un lenguaje vectorizado. Esto implica que puede ejecutar operaciones en paralelo entre vectores y que una determinada función, como mean() o sqrt(), se ejecuta automáticamente sobre cada uno de los elementos de un vector. Esta propiedad ya fue aplicada a lo largo del curso pero sin hacerla manifiesta. Por ejemplo, dados los vectores a <- 1:4 y b <- 6:9, la suma de ambos se obtiene con a + b, mientras que en otros lenguajes requeriría un bucle for.

Otro ejemplo. Dado a <- 1:4, a > 2 devuelve FALSE FALSE TRUE TRUE. Es decir, la orden de comparación se aplicó sucesivamente a cada elemento del vector a. La propiedad se extiende a matrices. Dadas m <- matrix(1:4, 2, 2) y n <- matrix(6:9), el producto m * n devuelve el resultado buscado.

1.7. Iteración

Quienes provienen de otros lenguages de progamación asocian el término iteración, o bucles, con ciclos for ..., repeat ... y while R también ofrece estos recursos.

1.7.1. bucle for

Un bucle for tiene la siguiente estructura for
(valor en una secuencia)
{
código
}

Se usa el bucle for cuando se conoce de antemano cuantas iteraciones habrá. El bucle requiere una variable, cuyo valor inicial se asigna por fuera del bucle y antes de ingresar en él, y un código para ejecutar. Una vez ejecutado el código el valor de la variable cambia al siguiente en la secuencia. Y así hasta completar la secuencia. Excepciones al funcionamiento descripto pueden darse si se intercala break o next. (Figura1.10)

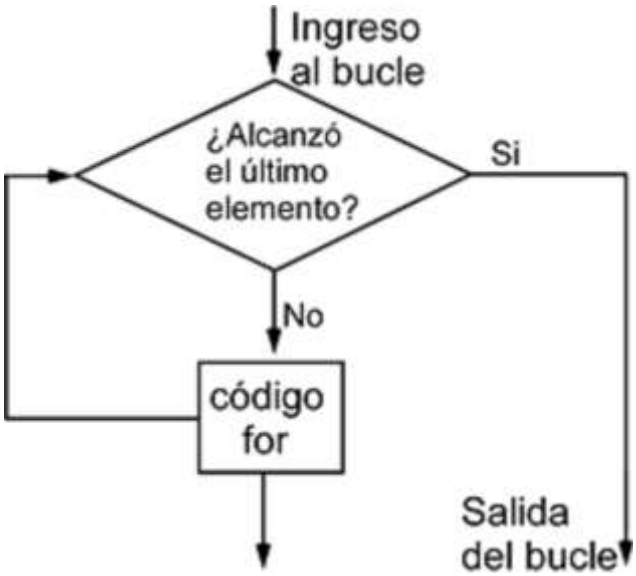


Figura 1.10: *Bucle for.*

En este y los demás diagramas de flujo el rombo representa tomar un decisión.

Ejemplo A

```
>  for(i in 1:5) print(1:i)
```

Predecir las respuestas en los tres ejercicios que siguen

```
a) a <- 0
for (j in 3:5) a <- j+a b)
a<- 10
for (j in 3:5) a <- j+a c) a
<- 10
for (j in 3:5) a <- j*a
```

Ejemplo B

```
Crear una matriz vacía y llenarla con valores mi_mat <-
matrix(nrow=3, ncol=5)
for(i in 1:dim(mi_mat)[1]) {
      for(j in 1:dim(mi_mat)[2]) {
        mi_mat[i,j] <- i*j
      }
}
Mostrar la matriz
mi_mat
```

1.7.2. bucle repeat

Un bucle repeat ejecuta una secuencia de comandos hasta que se le indique que pare. Se usa el bucle repeat cuando no se sabe de antemano cuantas iteraciones serán necesarias para completar la operación pero sí con cual resultado debe detenerse.

Ejemplo A

La instrucción inicial se repite hasta que el contador supera a 5, entonces entra a operar break. Notar que el bucle repeat incluye una sentencia condicional.

```
v <- c(Repite")
contador <- 2
repeat {
      print(v)
      contador <- contador+1 )
      if(contador > 5) {
      break
      }
}
```

1.7.3. bucle while

El bucle con while tiene la estructura i
<- número entero
variable a ingresar en la prueba
while (prueba condicional)
{
código
}
Se usa el bucle while cuando no se sabe de antemano cuantas iteraciones serán necesarias para completar la operación. En cada iteración while verifica

el resultado de la prueba, mientras sea verdad ejecuta el código, cuando se hace falso sale inmediatamente del bucle (Figura1.11).

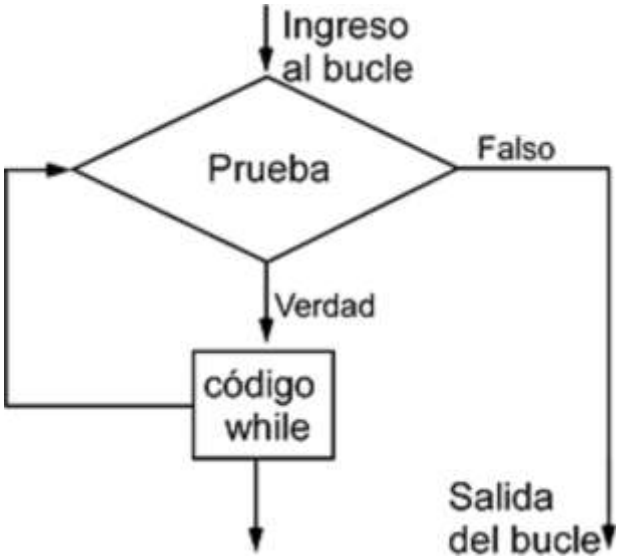


Figura 1.11: *Bucle while.*

Ejemplo A

```
i <- 1
while (i < valor) {
    print(i)
    i <- i+1
}
```

Ejemplo B

```
c<-0;n<-1
while(c «- 1000) {
     n <- n + 1
     c <- nˆ2
}
```

Verificar escribiendo

```
> c
```

1.7.4. familia apply(), lapply(), sapply(), tapply()

Las estructuras de código para iterar una operación vistas arriba son comunes a muchos lenguajes de programación y, por cierto, dan buenos resultados. El lenguaje R incorpora varios otros algoritmos que, en muchos casos, son

preferibles al tradicional bucle for.

En el acápite Funciones, en el Módulo 1 - Estructuras, se introdujo brevemente la función apply(), útil para reiterar una operación sobre objetos diferentes. Aquí la presentamos formalmente, y agregamos tres funciones más con similares prestaciones: lapply(), sapply() y tapply(), que en conjunto confirman la familia apply, que se encuentra en el paquete base.

La función apply()

apply() toma tres argumentos: X - nombre del array sobre el que se aplica, MARGIN – un número entero indicando si la operación se ha de ejecutar por filas, 1, por columnas, 2, o en ambas direcciones, c(1,2), y FUN - la función que se ha de aplicar a X; FUN puede ser una función creada por el analista. Es decir, apply() es una función que llama a otra función y devuelve unvector, una lista o un array. Si X es un marco de datos apply() lo coerciona a array.

Veamos un ejemplo. Dada la matriz 5 x 6 m1,obtener la suma de las columnas y de las filas empleando la función predefinida sum(). Notar el control MARGIN.

```
m1 <- matrix(c(1:10),nrow=5, ncol=6)
m1_suma_cols <- apply(m1, MARGIN = 2, FUN = sum)
m1_suma_filas <- apply(m1, MARGIN = 1, FUN = sum)
```

La función lapply()

lapply() opera sobre listas y vectores, y devuelve una lista cuyos valores resultan de aplicar una función a cada elemento ingresado; la 'l' inicial representa lista. lapply() toma sólo dos argumentos, X y FUN. Convertimos la matriz m1 a lista y aplicamos lapply().

```
m1_suma_cols <- lapply(list(m1), FUN = sum)
```

Devuelve 165, la suma de todos los elementos de m1, en formato de lista. Para obtener el resultado como vector aplicar unlist()

```
m1_suma_cols <- unlist(lapply(list(m1), FUN = sum))
```

Otro ejemplo. Convertir vars a minúsculas

```
> vars <- c('UNO', 'DOS', 'TRES', 'CUATRO')
> vars_min <- lapply(vars, tolower)
> vars_min
```

La función sapply()

sapply() opera sobre listas, vectores o marcos de datos y devuelve un vector o una matriz; toma dos argumentos: X y FUN. Aplicarlo sobre vars

```
> vars_min <- sapply(vars, tolower)
```

devuelve un vector con los nombres convertidos a minúsculas.

La función tapply()

tapply() opera sobre arrays cuyas filas varían en longitud ('ragged array')

tomando únicamente las celdas con datos. La sintaxis es: tapply(X, INDEX una lista de factores, cada uno de igual largo que X; y FUN la función a aplicar. La función

Ejemplo. Se opera sobre el marco de datos mtcars. Se desea calcular el consumo promedio, mpg, según la cilindrada, cyl.

```
> tapply(mtcars$mpg,mtcars$cyl, mean)
```

donde X<-mtcars$mpg, INDEX<-mtcars$cyl, FUN<-mean.

En resumen, las funciones del grupo apply() toman el primer elemento del vector X y aplican sobre él la función FUN, luego pasan al segundo elemento, y así hasta procesarlos a todos. Es decir, se comporta como un bucle for pero con menos código y evitando la preocupación de llevar bien la contabilidad de iteraciones. Los bucles for ... son útiles donde la salida de una iteración depende del resultado de la iteración previa. En otras situaciones una de las funciones de la familia apply() probablemente sea más eficiente.

apply() versus bucle for

Un último ejemplo compara un bucle for con apply(). Dada la matriz

```
> m1 <- matrix(C = (1:10),nrow=5, ncol=6) El
```

bucle for calcula los promedios de cada columna for(i in 1:6) {

```
 print(mean(m1[,i]))
}
```

El mismo cálculo con apply()

```
> apply(m1,2,mean)
```

La sección siguiente aplica iteración en una operación que se presenta con frecuencia en el trabajo con conjuntos de datos, cual es el procesamiento de numerosos archivos simultáneamente.

1.8. Carga automatizada de archivos

Ver video ArchivosMultiples.mp4.

Si los archivos a cargar son pocos, cargarlos uno a uno, como se hizo en el Módulo 2, no es tarea dura. Si hay que cargar decenas de archivos es conveniente disponer de otro recurso. La iteración brinda una solución.

Los archivos a cargar deben residir en un mismo directorio y sus nombres deben tener alguna similaridad distintiva, como por ejemplo la extensión. En este ejemplo se trata de cargar veinte archivos Excel que se encuentran en la carpeta Listados_xls. Se ofrece el script Listado_xls.Rmd, en formato RMarkdown (ver la carpeta Listados_xls).

Tener en cuenta que los 20 archivos Excel suman 389 filas de registros, excluyendo los encabezamientos. Y también que el código de carga suprime los encabezamientos. Usando la función colnames() (ver sección 1.5) reinstauren los nombres de columnas.

1.9. Estructuras de ejecución condicional

Se presentan tres estructuras que permiten procesar datos con condiciones: **if**, **if else** y **ifelse**
Hay tres maneras de establecer condiciones en bucles if: Sin alternativas, con una alternativa y con varias alternativas.

1.9.1. Condición sin alternativas

if(condicion booleana) código el código se ejecuta si la condición resulta verdad (Figura1.12).
La condición puede ser lógica o numérica. Ejecutar los siguientes ejemplos.

Ejemplo A

```
x <- valor if(y
> valor){
    print('y mayor que x')
}
```

Ejemplo B

```
x <- 'Gustavo'
if(is.numeric(x)) {
    print('x es un número')
}
```

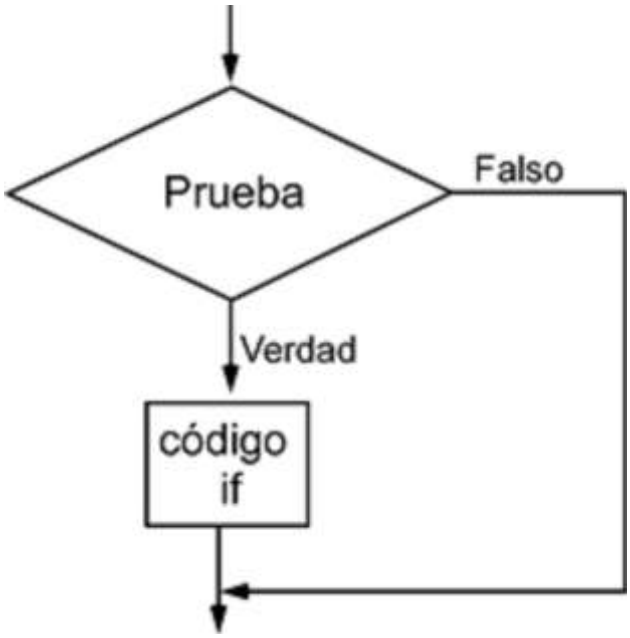


Figura 1.12: *Bucle if sin alternativa.*

1.9.2. Condición con una alternativa

if (prueba) {código 1} else {código 2}
Ejemplo
x <- 'Gustavo'
if(is.numeric(x)) {
    print('x es un número')
} else {
    print('x no es un número')
} (Figura1.13).

Ejemplo A

if (prueba) { declaración
    1
    } else {
    declaración 2
}

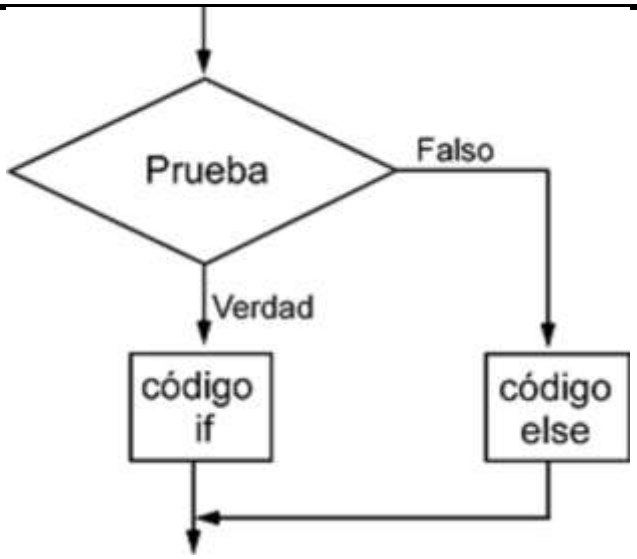


Figura 1.13: *Bucle if else.*

Ejemplo B

Dado un vector con dos elementos x <- c(10, 1) escribir un bucle condicional que detecte el elemento de mayor valor.

```
if(x[1]      > x[2]) {
print(paste('El valor mayor es ', x[1]))
} else {print(paste('El valor mayor es ', x[2])) }
```

1.9.3. Ejemplo C

Bucle if anidado en una función

Ejecutar con raiz(5, -32); devuelve 2.

```
raiz <- function(n,x){
   if(n%%2 == 1 || x $>$ <- 0 ){
     sign(x)*abs(x)^(1/n)
   } else{ NaN }
```

1.9.4. Condición con varias alternativas

if (prueba) {código 1} else if(prueba){código 2} else {código 3}

Ejemplo A

```
x <- ' Gustavo'
if(is.numeric(x)) {
  print(' x es un número' )
  } else if(is.complex(x))
  {
  print(' x es complejo' )
  } else {
  print(' x no es número o complejo' )
}
```

Este bucle acepta más opciones anidadas.

La estructura ifelse corresponde en este acápite pues ofrece dosalternativas. En general ocupa una línea de comando y puede operar en vectores. La sintaxis es: ifelse(condicion, opcion verdad, opcion falso) O sea, si la condición evalúa a verdadero, se ejecuta la primera opción, de lo contrario se ejecuta la segunda opción. Dado

> a <- 8

> ifelse(a > 7,200,0) devuelve 200

Operando con caracteres. Un código continúa ejecutando si la respuesta es SI y se detiene si es NO

> b <- "SI"

> ifelse(b == "SI",Çontinuar","Parar")

También opera con vectores, evaluando uno por uno cada elemento

> x <-c(4,7,9,44)

> ifelse(x > 7,."lto","bajo"))

Pero ¿qué pasa si se agregan comandointernos?

> ifelse(x > 7, print(."lto"),print('bajo'))

ifelse ejecuta primero los comandos y luego evalúa la condición sobre el vector.

ifelse tiene ciertas limitaciones. Por ejemplo, dado

> a <- 8; aa<-0

> ifelse(a > 7,aa<-200,aa<-0) genera error. Para ejecutar la asignación hay que usar

> if(a > 7) aa <- 200 else aa <- 0

Ejemplos de bucles

Ejemplo while, if

El número Armstrong, también llamado número narcisista, es un número

tal que la suma de sus dígitos elevados al cubo devuelve el mismo número. El bucle while determina si un número ingresado por teclado es un número Armstrong. Abrir el Editor de RStudio con File>New File>R Script. Escribir los siguientes comandos en el editor de RStudio, sin ingresar >.

```
num <- as.integer(readline(prompt<-'Ingresar un número sin decimales: ')) sum
<- 0
Inicializa la suma a 0
temp <- num
Guarda el número en una variable temporaria
while(temp > 0) {
Cuando temp <- 0 termina el bucle
digit <- temp%% 10 # Halla el módulo de dividir por 10; lo guarda en digit sum <-
sum + (digit ^3 ) # Eleva el resto al cubo y lo suma
temp <- floor(temp / 10) # Divide por 10 el número original y lo trunca
}
if(num == sum) { # Si el número inicial es igual a la suma el número es
Armstrong
print(paste(num, 'es un número Armstrong')) } else{
print(paste(num, 'no es un número Armstrong'))
}
```

Para ejecutar posicionar el cursor en la primera línea. Pulsar Run en la cinta superior. Al pedido ingresar un número, por ejemplo 371. Pulsar en Run sucesivamente hasta completar el proceso. Probar con otro número.

Parte VII

Fundamentos de programación en R

R es un lenguaje de programación funcional, es decir, un lenguaje basado en el uso de funciones matemáticas y en el cual las funciones pueden ser asignadas a variables. "Los lenguajes funcionales priorizan el uso de recursividad y aplicación de funciones de orden superior para resolver problemas que en otros lenguajes se resolverían mediante estructuras de control (por ejemplo, ciclos)."(Wikipedia Programación funcional). La consiguiente compacidad de los comandos en R ha hecho que no se lo vea como un lenguaje de programación al estilo de Python, con sus ubícuos bucles. Pero igualmente R brinda recursos para crear códigos estructurados que permiten la ejecución concatenada de comandos, como se vio en Estructuras de control.

Al iniciarse en programación es conveniente aprender a trabajar con proyectos.

1.10. Proyectos

Mientras se trate de escribir unas pocas líneas de código, y ya habrán apreciado cuanto se puede lograr con unas pocas líneas de código en R, no es necesario complicarse con la creación y administración de proyectos. Pero con tareas más complejas la organización en proyectos es conveniente.

En RStudio se puede crear un proyecto desde File>New Project, o pulsando en Project>New Project en el ángulo superior derecho. En ambos casos una pantalla Create Project invita a crear un proyecto en un directorio nuevo o uno existente. Pulsae en New Directory y seleccionar New Project. Pulsar en Browse y apuntar a un directorio raíz. En el rengló arriba indicar el nombre del directorio a crear. Aceptar y se crea el proyecto, cuyo nombre aparece en la consola y en el ángulo superior derecho. En el directorio se crea un archivo con el nombre del proyecto y la extensión *.Rproj.

Un proyecto puede tener propiedades definidas por el usuario. Pulsando en el nombre del proyecto en la cinta superior seleccionar Project Options. En esta etapa de la instrucción no interesa modificar nada.

1.11. Scripts

Un script es un texto en formato ASCII con instrucciones codificadas en el lenguaje R que se ejecutan al llamarlo desde RStudio. Los scripts permiten almacenar numerosos comandos en una correcta secuencia de ejecución, resumiendo una larga tarea de preparación, y facilitan tanto el retorno a un proyecto temporariamente abandonado como el intercambio de proyectos entre colegas. El conjunto de comandos escritos para determinar si un número es un número Armstrong constituyen un script. Se da otro ejemplo. 1 – Escribir los siguientes comandos en un procesador de texto

```
y<-c(12,15,28,17,18)
x<-c(22,39,50,25,18)
print(mean(y))
print(mean(x))
plot(x,y)
```

y guardarlos con extensión TXT en el directorio de trabajo, por ejemplo, script.txt.

2 – En la consola de RStudio escribir

> source(‘script.txt’)

y se ejecutan los comandos imprimiendo los promedios en la consola y el gráfico en el panel Plots. scripts.txt debe residir en el directorio actual, o completar el enlace.

Alternativamente

1 - Ir a File>New File>R script

2 - Copiar el script a la pantalla del editor.

3 – Ir a File>Save As y guardar el script con extensión*.R. 4 –
Se lo puede cargar con File>Open File.

El Módulo 4 incluye numerosos scripts aplicando R a diversos problemas.

Instalar y activar paquetes en scripts

Son muy pocos los proyectos en R que se ejecuten empleando solamente los paquetes base. A poco andar verán la necesidad recurrente de instalar y activar otros paquetes. Una vez que hayan definido los recursos programáticos necesarios para un proyecto en particular, y el protocolo de ejecucion, probablemente convenga consolidar todo en un script. Mientra el script se ejecute en vuestra computadora no será necesario instalar los paquetes, si no sólo activarlos con library() o require(). Pero si desearan distribuir el script a terceros podrán preferir incluir los comandos de instalación en el propio script. Hay más de una manera de proceder.

Crear un vector con los nombres de los paquetes a activar, por ejemplo,

> mis_paquetes <- c(’dplyr’, ’mice’, ’stringr’)

Y usar lapply() para repartir la acción require en cada uno

> lapply(mis_paquetes, require, character.only <- TRUE)

Alternativamente recurrir al paquete easypackages, que ofrece las funciones packages() y libraries(), las cuales se aplican así:
packages(’dplyr’, ’ggplot2’, ’RMySQL’, ’data.table’)
libraries(’dplyr’, ’ggplot2’, ’RMySQL’, ’data.table’)

1.11.1. attach()

Si se desea referir a una determinada variable en un marco de datos se puede emplear $; marco de datos$variable. Aunque práctica y transparente, este modo de referir a una variable puede resultar tedioso. La función attach() brinda otra opción. Con

> attach(marco_de_datos)

incorpora el nombre de un marco de datos a la ruta de búsqueda en R. Hecho esto el analista puede referirse a una variable escribiendo solamente el nombre y omitiendo el nombre del marco de datos que la contiene.

> detach() anula la asociación.

1.11.2. Buenas prácticas en programación

Es un error muy común creer que uno recordará por siempre con claridad un script elaborado con mucho esfuerzo. Eso no ocurre. Por lo tanto es recomendable implementar algunas prácticas que faciliten retomar un script pasado cierto tiempo, y también facilitar la tarea de terceros que pudieren interesarse en él. Algunas de estas prácticas son:

- Incluir abundantes comentarios.
- Al inicio indicar autores, fechas de inicio y actualización, enumeración de los paquetes requeridos, y propósito del script.
- Insertar en el código explicaciones de cada función y otras aclaraciones.
- No incluir URLs y enlaces en el código, más bien inicializarlos al comienzo. Por ejemplo
- Reunir las funciones que se hayan creado al principio del código o en un archivo separado.
- Mantener una nomenclatura consistente, vectores v_, matrices m_, por ejemplo.

Fin del Módulo 3