2

1

Manual de R para usuarios libres

Luis Alonzo
Gabriela Mathieu
Pablo Messina

Versión 1.3

Índice general

1

Preámbulo

Este manual se editó completamente utilizando el programa libre denominado LaTeX[1] y se distribuye bajo licencia Creative Commons[2], en particular, no se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original. En otras palabras:

Reconocimiento –Attribution–: En cualquier explotación de la obra autorizada por la licencia hará falta reconocer la autoría.

Figura 1.1: Reconocimiento

Compartir Igual –Share alike–: La explotación autorizada incluye la creación de obras derivadas siempre que mantengan la misma licencia al ser divulgadas.

Figura 1.2: Compartir igual

No Comercial –Non commercial–: La explotación de la obra queda limitada a usos no comerciales.

Figura 1.3: No comercial

[1] Para saber más sobre LaTeX haz click en: `http://www.latex-project.org/`

[2] A través de Creative Commons se puede licenciar todo tipo de obras intelectuales. Entre otras posibles: fotos, libros, textos académicos, videos, animaciones, música, sitios web, blogs. etc. Sólo existe un tipo de obra para la cual Creative Commons recomienda utilizar otra licencia: en el caso del software. Para ello, Creative Commons recomienda utilizar la Licencia Pública General –GPL– de la Fundación para el Software Libre –FSF–.

2

Introducción

2.1 Conceptos preliminares

2.1.1. ¿Qué es el software libre?

El software libre surge aproximadamente en la década de 1970, cuando investigadores del MIT –Instituto Tecnológico de Massachusetts– comenzaron a tener problemas para desarrollar programas, pues entraban en contradicción con los intereses de algunas empresas y en particular con el modelo vertical y restrictivo que se estaba imponiendo.

En los años ’80 surge un movimiento que trata de impulsar la filosofía del software libre, en torno a Richard Stallman y la creación de la Free Software Foundation. En inglés la palabra free, significa libertad, pero también gratuidad, por ello suele asociarse software libre con software gratis. Sin embargo la gratuidad no es la principal característica de estos programas, sino la garantía de ciertas libertades que lo hacen ser mucho más flexible, seguro y accesible.

En este punto ya podemos estar de acuerdo en que libertad y gratuidad no son sinónimos ni nada parecido, lo cierto es que los programas no libres – privativos – suelen tener un precio y los programas libres suele ser gratuitos, pero existen programas privativos gratuitos y programas libres que pueden comercializarse. Entonces, ¿cuál es la diferencia entre ambos? Lo que dijimos: la libertad, o más concretamente las libertades que que ofrecen los programas libres al usuario. Formalmente estas libertades se expresan en la licencia que acompaña al programa, que es la misma herramienta legal que utilizan los programas privativos para imponer restricciones.

Existen dos niveles para explicar qué es el software libre, un nivel técnico y un nivel filosófico, comenzaremos con el primero para luego dar nuestras razones de por qué el software debe ser libre.

2.1.2. ¿Qué es el software?

Para comprender la diferencia técnica entre software libre y propietario primero debemos establecer claramente qué es el software. Básicamente, el software es lo que hace que el procesador de una computadora realice las tareas que se le indican. Pues el procesador por sí sólo no puede hacer nada y qué tipo de cosas haga depende del software con el cuál se utilice, es en el programa dónde se detallan las instrucciones para la computadora y es lo que le permite realizar una tarea específica. Los programas en general, se escriben en un lenguaje determinado, lo que se denomina código fuente, que luego es traducido al lenguaje máquina, es decir, el lenguaje que entiende el procesador de la computadora, conocido como código binario. Por tanto si queremos estudiar cómo funciona el

programa o bien modificar alguna instrucción debemos contar con el código fuente. El código fuente está escrito en lenguaje de programación, y basta conocer ese lenguaje para entenderlo –y mucho más fácil si está bien programado y documentado–.

Para ser considerado libre, un programa debe ser distribuido de tal modo que el usuario pueda, entre otras cosas, estudiar el modo de funcionamiento del programa, adaptarlo a sus necesidades y distribuir, bajo las mismas condiciones, programas derivados del mismo.

Para que estas libertades sean practicables, no basta con que la licencia del programa las permita. Además, es necesario que el código fuente del programa este a disposición del usuario, ya que de lo contrario las tareas de comprender, adaptar y mejorar el programa se vuelven tan complicadas que es casi lo mismo que si estuvieran prohibidas.

Por eso la definición de Software Libre elaborada por la Free Software Foundation aclara que un programa no puede ser considerado libre si su código fuente, su texto original, no esta disponible. Entonces el Software Libre se define por su tipo de licenciamiento. Podemos decir entonces que Software Libre es un software o programa de computación cuya licencia nos permite ejercer una serie de libertades:

- La libertad de ejecutar el programa con cualquier propósito.
- La libertad de redistribuir copias del programa y de ese modo ayudar a otros.
- La libertad de estudiar cómo funciona el programa y adaptarlo a las necesidades propias.
- La libertad de mejorar el programa y liberar esas mejoras a la comunidad.

Para poder tener estas dos últimas libertades es condición necesaria el acceso al código fuente.

En general, un programa libre solo exige una cosa: que si distribuimos el programa resultante de una modificación, este se distribuya bajo las mismas condiciones del programa original. Las licencias que contienen esta condición son llamadas licencias Copyleft, y su objetivo es evitar que se distribuyan obras derivadas bajo licencias privativas; un ejemplo de estas es la GPL[1].

Como contrapartida podemos decir que software privativo –dado que nos priva de libertades– es el que nos priva de alguna de las libertades antes expuestas.

2.1.3. ¿Por qué elegir el software libre en lugar del privativo?

Son varias las razones que nos deberían conducir al uso del SL en vez del privativo. En primer lugar, porque la concepción de libertad que encierra implica posicionarse contra la propiedad privada en general y contra la propiedad intelectual en particular. Concibiendo las ideas como “productos sociales” resulta inadmisible que haya quienes se apropien individualmente de dichos procesos colectivos.

De hecho, el grueso de la comunidad del SL hace un apoyo explícito a la libre circulación del conocimiento y en contra de las patentes; realidades que no sólo afectan al software sino que atraviesan

[1] GNU General Public Licence –Licencia Pública General,GNU–. Se trata de una licencia creada por Free Software Foundation –Fundación para el software libre–, organización fundada por Richard Matthew Stallmana en el año 1985. El principal propósito de la licencia GNU es declarar la libertad del uso, modificación y distribución del software y protegerlo de intentos de privatización que puedan de algún modo restringir su uso

transversalmente nuestra sociedad en sectores tan sensibles como por ejemplo la medicina.

Por otra parte el SL al ser de código abierto, permite que la comunidad de usuarios y desarrolladores sea quien le vaya imprimiendo mejoras sistemáticamente, lo que lo hace más seguro y adaptable. Cuanto más cooperativo y comunitario sea un programa, cuanto más grande sea su comunidad, cuanto más participativa, entonces será un programa más seguro, estable, y con mejor funcionalidad. Esto último, trae aparejado también que las mejoras y actualizaciones no deben ser necesariamente compradas sino que en su mayoría pueden descargarse y actualizarse desde internet haciendo en definitiva, que los programas sean más acesibles más accesible a todos y todas.

2.1.4. ¿Por qué elegir R en lugar de otro software estadístico?

En nuestro caso, no sólo preferimos R en lugar de otro programa estadístico privativo, sino también entre los programas estadísticos libres.

Preferimos usar R porque permite hacer todo lo que hacen los demás, tiene mayor funcionalidad y es económicamente más rentable. Por otro lado al utilizarlo y encontrar sus posibles problemas estamos apoyando a la propia comunidad de usuarios y desarrolladores.

R es un programa que incluye herramientas para realizar análisis estadístico de los más variados, por lo que cualquier disciplina que se relacione con la estadística puede requerir del uso de R: econometría, biología, psicología, ramas estadísticas de las más variadas, etc.

A su vez, existe una comunidad mundial de usuarios de R muy grande, desde usuarios en sus hogares hasta estudiantes y docentes en muchas universidades que están continuamente controlando el funcionamiento de R, desarrollando paquetes, etc. Esto permite, además de que las funcionalidades de R mejoren versión a versión, que también el programa vaya incorporando aplicaciones nuevas en áreas en que aún no las tenía, lo cual contribuye a incrementar aún más el número de usuarios.

2.1.5. ¿Por qué elegir R en la educación?

El uso de R en la Universidad de la República es de vital importancia por varios motivos.

En primer lugar, porque en la mayoría de los servicios se utiliza algún tipo de programa estadístico, en la mayoría privativos y se pagan licencias por ello o bien se utilizan versiones de estudiantes que imponen limitaciones de uso y no permite que el estudiante cuente con el programa en su casa. Entonces, si logramos hacer con R los análisis que se hacen con otros programas privativos – esos que pagamos entre todos– no hay razón para no utilizarlo; solamente es cuestión de aprender a hacerlo.

En segundo lugar, porque el uso de un software privativo en un centro educativo está condicionando al estudiante a seguir usando ese software y genera de esa manera un círculo de dependencia. Creemos que la UdelaR debe garantizar el uso de herramientas de manera universal y evitar obligar a sus estudiantes hacer grandes desembolsos de dinero o a violar la ley, por un asunto de licencias.

En tercer lugar, este círculo vicioso comienza en la universidad se traslada al ámbito laboral o viceversa. Es decir, en los llamados laborales suelen requerirse que el postulante tenga conocimiento acerca de un cierto software privativo, entonces los estudiantes promueven el uso de dicho software en el ámbito educativo. A su vez, como los docentes en su mayoría han sido formados con software privativo tampoco tienen incentivos a aprender otro programa para poder utilizarlo en la docencia.

En este sentido, la UdelaR es el lugar idóneo para romper con esa lógica y desde el lugar que nos ha tocado queremos brindar al estudiante la posibilidad de elegir por la alternativa libre, dándola a conocer y promoviendo su uso.

2.2 Sesión

R es un programa que se utiliza de manera interactiva, el usuario hace una pregunta y R da una respuesta. Las preguntas y respuestas se hacen a través de comandos propios del lenguaje de R. Para empezar a escribir comandos en R debes invocar el programa: en Windows se debe hacer doble click sobre el ícono de R y en Linux escribir R en la terminal.[2] Una vez abierto el programa aparecerán las siguientes líneas:

```
usuario@maquina:~$ R

R version 2.9.2 (2009-08-24)
Copyright (C) 2009 The R Foundation for Statistical Computing
ISBN 3-900051-07-0

R es un software libre y viene sin GARANTIA ALGUNA.
Usted puede redistribuirlo bajo ciertas circunstancias.
Escriba 'license()' o 'licence()' para detalles de distribucion.

R es un proyecto colaborativo con muchos contribuyentes.
Escriba 'contributors()' para obtener más información y
'citation()' para saber cómo citar R o paquetes de R en publicaciones.

Escriba 'demo()' para demostraciones, 'help()' para el sistema on-line de ayuda,
o 'help.start()' para abrir el sistema de ayuda HTML con su navegador.
Escriba 'q()' para salir de R.

[Previously saved workspace restored]
```

Para evitar el mensaje de bienvenida basta escribir en lugar de lo anterior:

```
usuario@maquina:~$ R --silent
```

Una de las características de R es que, mientras que otros programas estadísticos 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.
Esta característica marca que R es un lenguaje orientado a objetos. 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 sin usar archivos temporales. El usuario puede modificar o manipular estos objetos con operadores –lógicos, aritméticos y comparativos– y funciones –que a su vez son objetos–. En definitiva, bajo este complejo término se esconde la simplicidad y flexibilidad de R.

Un objeto puede ser creado con el operador "asignar" el cual se denota como una flecha: con el signo menos y el símbolo $>$ o $<$ dependiendo de la dirección en que asigna el objeto ($<-$), o bien con el símbolo =. Si el objeto ya existe, su valor anterior es borrado después de la asignación –la modificación afecta sólo objetos en memoria, no a los datos en el disco–.

R es un lenguaje con una sintaxis muy simple, donde las órdenes elementales consisten en expresiones o asignaciones. Si una orden consiste en una expresión, se evalúa, se imprime y su valor se pierde.

```
> 2+3
[1] 5
```

[2] Si aún no tienes instalado R, en el anexo se indica cómo hacerlo en Linux.

Una asignación, por el contrario, evalúa una expresión, no la imprime y guarda su valor en un objeto.

```
> a=2+3
```

Alternativamente puede realizarse una asignación de otras 3 maneras:

```
> a <- 2+3
> 2+3 -> a
> assign("a",2+3)
```

Veamos que efectivamente el objeto "a" se encuentra en la memoria de R. Con la orden objects() podemos obtener los nombres de los objetos almacenados en R:

```
> objects()
[1] "a"
```

Es importante tener en cuenta que R distingue entre mayúsculas y minúsculas, de tal modo que A y a son símbolos distintos y se referirán, por tanto, a objetos distintos.

Las órdenes se separan mediante punto y coma, – ; –, o mediante un cambio de línea.

```
> a*a;a+1
[1] 25
[1] 6
```

Si al terminar la línea, la orden no está completa, R mostrará un signo de + hasta que la orden se complete.

```
> b=a*(1-pi
+

> b=a*(1-pi)
> b
[1] -10.70796
```

2.2.1. Funciones

Las funciones son expresiones no tan elementales como las mencionadas anteriormente, por ejemplo para calcular la media de una variable, se utiliza la función mean(). Las funciones disponibles para usar en R están guardadas en una librería localizada en el directorio donde R está instalado. Este directorio contiene paquetes de funciones, las cuales a su vez están estructuradas en directorios. El paquete denominado *base* constituye el núcleo de R y contiene las funciones básicas del lenguaje para leer y manipular datos, algunas funciones gráficas y algunas funciones estadísticas. Para obtener un detalle de las funciones de este paquete basta escribir en la consola de R:

```
library(help=''base'')
```

El paquete que viene cargado por defecto es *base*, si es necesario utilizar otro, se debe descargar de internet e instalar, como indicaremos más adelante.

Para que una función sea ejecutada en R debe estar siempre acompañada de paréntesis curvos, inclusive en el caso que no haya nada dentro de los mismos –por ejemplo getwd() que indica el directorio de trabajo actual–. Si se escribe el nombre de la función sin los paréntesis, R mostrará el contenido –código– mismo de la función. Una función en R puede carecer totalmente de argumentos, ya sea porque todos están definidos por defecto –y sus valores modificados con opciones–, o porque la función realmente no tiene argumentos.

2.2.2. Trabajar con objetos

Hemos visto que R trabaja con objetos los cuales tienen nombre[3] y contenido, pero también atributos que especifican el tipo de datos representados por el objeto.

Ejemplifiquemos a partir del objeto más simple de R, un vector.

Un vector no es más que un conjunto de elementos que tienen cierto orden: existe un primer elemento, un segundo elemento, hasta un último elemento. Trabajaremos con los datos de los primeros 10 inscriptos al curso, en particular, creamos un vector donde cada elemento indica si el inscripto al curso usa como sistema operativo GNU/Linux –1– o usa el sistema operativo Windows –0–.

```
> usa.linux = c(1,1,0,0,0,0,0,0,0,0)
```

Note que se utilizó para construir el vector usa.linux, la función c(), la cual concatena los diferentes elementos del vector. El vector como todo objeto tiene dos atributos intrínsecos: *tipo* o *modo* y *longitud*. El modo refiere al tipo de elementos del vector: así si los elementos son números –reales y operables–, el modo es númerico, si son letras o números no operables –escritos entre comillas–, el modo es caracter, mientras que si son los valores lógicos FALSE o TRUE[4], el modo es lógico.

A su vez, cada objeto representa una clase, la cual es diferente que el modo, salvo para los vectores. La longitud es simplemente el número de elementos en el objeto. Para ver el tipo, la longitud y la clase de un objeto se pueden usar las funciones mode(), length() y class(), respectivamente:

```
> usa.linux = c(1,1,0,0,0,0,0,0,0,0)
> mode(usa.linux)
[1] "numeric"
> class(usa.linux)
[1] "numeric"

> usa.linux = c("1","1","0","0","0","0","0","0","0","0")
> mode(usa.linux)
[1] "character"

> usa.linux = c("TRUE","TRUE","FALSE","FALSE","FALSE","FALSE",
                "FALSE","FALSE","FALSE","FALSE")
> mode(usa.linux)
[1] "logical"

> length(usa.linux)
[1] 10
```

¿Cuál es el modo correcto de guardar estos datos?

Tener los datos guardados en un vector es muy conveniente, ya que por ejemplo, si queremos modificar alguno de esos elementos, no es necesario que ingresemos todos los datos nuevamente, sino basta indicarle a R cuál se quiere sustituir por uno nuevo.

```
> usa.linux =  c("1","1","0","0","0","0","0","0","0","0")
> usa.linux[3] = "1"
```

[3] El nombre de un objeto debe comenzar con una letra (A-Z and a-z) y puede incluir letras, dígitos (0-9), puntos (.) y guión bajo (_).

[4] Los valores lógicos también se puede escribir T en vez de TRUE y F en lugar de FALSE, pero siempre respetando las mayúsculas

Esto dará como resultado un nuevo vector a, cuyo tercer elemento ahora es un 1 en lugar de un 0. Nótese que para lograr esto se indica entre paréntesis rectos el lugar donde se ubica el elemento que queremos sustituir.

Si por ejemplo, quisieramos modificar también el segundo elemento pero ahora por un 1, ¿cómo modificaría la orden para hacer los dos cambios en simultáneo?

Además un vector puede contener caracteres especiales:NA [5] cuando no hay dato, NaN [6] cuando no es un número, +/-Inf[7] cuando es infinito, en estos casos se debe tener especial cuidado cuando se aplique una función al vector.

¿cómo prescindir de las comillas?

Para ello usamos la función Cs() del paquete Hmisc

```
> Cs(1,1,0,0,0,0,0,0,0,0)
[1] "1" "1" "0" "0" "0" "0" "0" "0" "0" "0"
```

[5] not available, missing data,
[6] not a number
[7] infinity

3

Vectores

3.1 Crear y manipular vectores

Existen muchas maneras de definir un vector, la más simple es la que ya vimos, usando la función "concatenar". Así para obtener un vector que contiene los números 1 al 5 haremos lo siguiente:

```
> c(1,2,3,4,5)
[1] 1 2 3 4 5
```

Pero alternativamente, podemos indicarlo con alguna de las siguientes órdenes:

```
> 1:5
[1] 1 2 3 4 5

> seq(from=1,to=5, by=1)
[1] 1 2 3 4 5

> seq(1, 5, length=5)
[1] 1 2 3 4 5
```

La función seq() genera una secuencia de números ordenados cuya distancia en el eje real se indica con el argumento by. Si en cambio queremos generar una secuencia de números entre 2 valores – los indicados en from y to – de una cantidad determinada de elementos, esto lo indicamos asignando un valor al argumento length.

Si en cambio, queremos que esta secuencia de números se repita una cierta cantidad de veces, podemos recurrir a la función rep() , que repite un número o un conjunto de estos, tantas veces como indica el argumento times:

```
> rep(1:5,times=3)
[1]  1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
```

Para aplicar la función rep() a números no consecutivos es necesario utilizar la función c() para indicar cuáles números se quiere repetir:

```
> rep(c(1,5), times=3)
[1] 1 5 1 5 1 5
```

Finalmente, para repetir cierto conjunto de números cada cierto intervalo, utilizamos el argumento each en lugar de times.

```
> rep(1:5,each=3)
 [1]  1 1 1 2 2 2 3 3 3 4 4 4 5 5 5
```

Función	Argumentos	Descripción
c()	números, letras o valores lógicos	crea un vector
rep()	vector, cantidad de repeticiones	crea un vector repitiendo uno o varios números cierta cantidad de veces
seq()	desde, hasta, cada	crea un vector generando una secuencia numérica

Cuadro 3.1: Comandos de sesión

3.2 Operadores

Los vectores son en definitiva objetos matemáticos. Por tanto se pueden operar utilizando operadores aritméticos, comparativos y/o lógicos, dando lugar a nuevos vectores u otros objetos. Los operadores

Aritméticos		Comparativos		Lógicos	
+	suma	$<$	menor que	!	NO lógico
-	resta	$>$	mayor que	&	Y lógico
$*$	producto	$<=$	menor igual que	&&	id.
/	división	$>=$	mayor igual que	|	O lógico
$\wedge$	potencia	$==$	igual	||	id.
$\%\%$	módulo	$!=$	diferente de	or	O exclusivo
$\%/\%$	división de enteros				

Cuadro 3.2: Tipo de operadores

- **aritméticos**: actúan sobre variables de tipo numérico o complejo, pero también lógico; en este caso los valores lógicos son forzados a valores numéricos[1]. Por ejemplo, tenemos un vector $usaR$ que indica que la persona inscripta al curso tiene algún conocimiento de uso de R, y un vector denominado $usaS$ que indica que la persona tiene algún conocimiento de SPSS. El resultado de sumarlos es el siguiente:

```
> usaR <- c(0, 0, 0, 0, 1, 1, 1, 0, 0, 1)
> usaS <- c(1, 1, 1, 1, 1, 1, 1, 1, 0, 1)

> usaR + usaS
[1]  1 1 1 1 2 2 2 1 0 2
```

  Cada elemento de este nuevo vector resultante, está indicando de cuántos programas –considerando R y SPSS– la persona tiene algún tipo de conocimiento. Así el "1" indica que la persona sólo tiene conocimiento de uno de ellos, "0" de ninguno y "2" de ambos.

- **comparativos**: pueden actuar cualquiera sea el modo del vector, devolviendo uno o varios valores lógicos. Es necesario tener en cuenta que los operadores comparativos actúan sobre cada elemento de los dos objetos que se están comparando –reciclando los valores de los más pequeños si es necesario–, devolviendo un objeto del mismo tamaño. Podemos obtener información similar a la del ejemplo anterior pero ahora comparando elemento a elemento:

```
> usaR == usaS
 [1] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE
```

[1] El valor TRUE se convierte en 1 y el valor FALSE se convierte en 0

```
 [9] FALSE TRUE
```

En este caso cada vez que el usuario usa ambos programas o no usa ninguno el resultado de la función es “TRUE”, de lo contrario “FALSE”.

Para comparar “totalmente” dos objetos es necesario usar la función identical().

```
> identical(usaR, usaS)
[1] FALSE
```

- Los operadores **lógicos** lógicos proporcionan un resultado a partir de que se cumpla o no una cierta condición. Actuar sobre uno o dos objetos de tipo lógico, y pueden devolver uno o varios valores lógicos. Los operadores $\&$ y or existen en dos formas: uno sencillo donde cada operador actúa sobre cada elemento del objeto y devuelve un número de valores lógicos igual al número de comparaciones realizadas; otro doble donde cada operador actúa solamente sobre el primer elemento del objeto.

  Por ejemplo podemos preguntarnos quiénes de los que usan R también usan SPSS, para ello ejecutamos la siguiente orden en la cual utilizamos un operador comparativo, el igual, y un operador lógico, el Y lógico:

```
usaR==1&usaS==1
[1] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE  TRUE
```

4

Clase de objetos

Como vimos, la forma más simple de almacenar datos es mediante un vector. Sin embargo, no siempre es posible guardar nuestros datos en un vector, en esos casos se hace necesario recurrir a otra clase de objetos, que en particular, son generalizaciones de un vector, pues además de longuitud y modo tienen otros atributos. Analizaremos sólo algunas clases de objetos que serán de interés durante el curso: factor, matriz, data frame y lista.

objeto	modo	clase
vector	numeric o character o complex o logic	numeric o character o complex o logic
factor	numeric o character	factor
matriz	numeric o character o complex o logic	matrix
data.frame	numeric y/o character y/o complex y/o logic	data.frame
lista	numeric y/o character y/o complex y/o logic	list

Cuadro 4.1: Tipos de objetos

Los objetos: vector, factor y matriz no permiten varios modos de elementos en el mismo objeto. Si lo hacen el data.frame y la lista, sin embargo, en el primer caso sólo es posible que todos los vectores o variables que componen el marco de datos tengan la misma longuitud, mientras que para crear una lista, esto no es necesario.

4.1 Matriz

Una matriz es una generalización de un vector y por tanto posee un atributo adicional –dim– que define el número de filas y columnas de la matriz. Una matriz se puede crear con la función matrix():

```
> m <- matrix(c(1,2,3,4), nrow=2)
> m
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

Los argumentos indispensables para construir una matriz son los datos, que en el ejemplo anterior correspondió al vector con 1,2,3,4 y el número de filas o de columnas; en el ejemplo indicamos 2 filas. Genéricamente, los argumentos que definen una matriz son los siguientes:

```
matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)
```

- data son los elementos de la matriz

- nrow indica el número de filas
- ncol indica el número de columnas
- byrow indica si los valores en data deben llenar las columnas sucesivamente –por defecto– o las filas –si es TRUE–.
- dimnames permite asignar nombres a las filas y columnas.

Por defecto, los elementos de una matriz se toman verticalmente, columna a columna:

```
> mat=matrix(1:9, nrow=3, ncol=3)
> mat
      [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

Por lo tanto, si lo que pretendíamos era crear una matriz que leyéndola por filas nos diera la secuencia ordenada del 1 al 9, basta con trasponer la matriz anterior utilizando la función t():

```
  > t(mat)
      [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

O bien, asignar el valor TRUE al argumento byrow.

Tal como podemos acceder a los elementos de un vector, podemos hacerlo con los elementos de una matriz. Para acceder a un elemento en particular debemos indicar su posición por fila y columna, mientras que si queremos acceder a cierta fila o columna basta indicar cuál:

```
> mat[1,2]
[1] 4
> mat[,2]
[1] 4 5 6
> mat[1,]
[1] 1 4 7
```

En el primer caso obtenemos el segundo elemento de la fila 1 del objeto mat. En el segundo caso, obtenemos la columna 2 completa, ya que no se indica ninguna fila en particular; nótese que antes de la coma no se indica nada.

4.2 Factor

El factor, es un tipo de objeto que permite trabajar correctamente con variables cualitativas –nominales u ordinales–, pues es más que un vector de modo character ya que además de los valores correspondientes a dicha variable, incluye sus diferentes niveles posibles. La función factor() tiene esta forma para el objeto conoce.R que contiene el nivel de conocimiento de R por parte de los primeros diez inscriptos al curso:

```
> conoce.R =  c(0,0,0,0,1,1,1,0,0,3)
> conoce.R = factor(conoce.R,
                    levels=c(0,1,2,3),
                    labels = c(``Ninguno", "Bajo", "Medio", ``Avanzado''))
```

El primer argumento de la función factor() son precisamente los datos, indican qué nivel de conocimiento de R tiene la persona. El siguiente argumento son los niveles de la variable los cuales se indican en levels y por último en labels se indica qué significa cada uno de ellos. La función genéricamente es de la siguiente forma:

```
 factor(x = character(),
            levels = sort(unique.default(x),
            labels = levels,
    exclude = NA,
    ordered = is.ordered(x))
```

- x es el vector de datos
- levels especifica los posibles niveles del factor –por defecto los valores no repetidos de los datos–,
- labels define los nombres de los niveles,es decir las etiquetas
- exclude especifica los valores x que se deben excluir de los niveles, y
- ordered es un argumento lógico que especifica si los niveles del factor están ordenados.

4.3 Lista

Una lista se crea con la función list(). Al diferencia de las matrices, no existe ninguna limitación en el tipo de objetos que se pueden incluir, así, las listas permiten incluir elementos de diferentes modos y donde cada elemento puede ser de diferente largo o tamaño. En general, las listas que usaremos se obtendrán como resultado de alguna técnica estadística, por ejemplo, el resultado de una tabla de frecuencias es una lista. Veamos un ejemplo con los datos de los inscriptos al curso:

```
> Lista = list(Curso = c("Introducción a R abril", "Introducción a R mayo"),
   carrera = c("Sociología","Trabajo social","Lic. Economía","Lic.Estadística"),
              sistema.operativo= c("GNU/Linux","GNU/Linux Windows","Windows"))

> Lista
$Curso
[1] "Introducción a R abril" "Introducción a R mayo"

$carreras
[1] "Sociología"       "Trabajo social"   "Lic. Economía"    "Lic.Estadística"

$sistema.operativo
[1] "GNU/Linux"          "GNU/Linux Windows" "Windows"
```

Existen diferentes maneras de acceder a los elementos de una lista:

```
> Lista$Curso
[1] "Introducción a R abril" "Introducción a R mayo"


> Lista[["Curso"]]
 [1] "Introducción a R abril" "Introducción a R mayo"

>Lista[[1]]
 [1] "Introducción a R abril" "Introducción a R mayo"
```

Los procedimientos anteriores devuelven vectores, para extraer una sublista, es decir, otra lista, se usan paréntesis rectos simples.

El siguiente ejemplo extrae una lista con el primer componente de la lista original

```
> Lista[1]
$Curso
 [1] "Introducción a R abril" "Introducción a R mayo"
```

Función	Argumentos	Descripción
factor()	objeto, niveles, etiquetas	crea un factor
matrix()	objeto, número de filas,...	crea una matriz
data.frame()	objeto	crea un marco de datos
table()	objeto	crea una tabla
list()	objeto, objeto, ...	crea una lista
seq()	desde, hasta, salto	genera secuencia numérica
rep()	elementos, repetición	genera repetición de elementos
c()	elementos	crea un vector

Cuadro 4.2: Construir objetos

5

Herramientas generales del trabajo en sesión

5.1 Almacenamiento y eliminación de objetos

Durante una sesión de trabajo con R los objetos que se crean se almacenan por nombre La orden objects() se puede utilizar para obtener los nombres de los objetos almacenados en R.

```
> objects()
```

Esta función es equivalente a la función ls(). La colección de objetos almacenados en cada momento y las órdenes se denomina espacio de trabajo –workspace–, y es posible guardarla para volver a usarla en una nueva sesión de trabajo usando la función save.image(). Esto genera un archivo con extensión .RData que por defecto es guardado en el directorio de trabajo pero se puede modificar el destino.

Para eliminar objetos puede utilizar la orden rm(), por ejemplo:

```
> rm(a,b)
```

Cuando son más los objetos a eliminar que los que nos queremos quedar, podemos usar la función keep() del paquete gdata, debemos especificar los objetos que no queremos eliminar.
Para cerrar el programa basta escribir la orden quit() o de forma abreviada:

```
> q()
```

Luego nos preguntará si queremos guardar el espacio de trabajo, si queremos evitar esta pregunta, en vez de lo anterior debemos escribir:

```
> q(save="no")
```

Para saber en que ubicación del disco de la computadora se encuentra R instalado

```
> R.home()
[1] "/usr/lib/R"
```

Para saber cuál es el directorio de trabajo actual

```
> getwd()
[1] "/home/usuario"
```

Para cambiar de directorio debemos indicar la ruta en la función setwd()

```
> setwd("/home/usuario/nueva carpeta")
```

Para saber cuáles son los archivos contenidos en el directorio de trabajo – no confundir con los objetos en memoria–

```
> dir()
```

Para obtener una descripción de los objetos y un detalle del espacio que ocupan usamos la función ll() y la función object.size, respectivamente; ambas pertenecen al paquete gdata

R permite recuperar y ejecutar órdenes escritas previamente utilizando las flechas verticales del teclado que permiten recorrer el historial de órdenes. A su vez, mediante las flechas horizontales es posible desplazarse sobre dicha orden.

También es posible limpiar la consola, tecleando ctrl + l; esta orden no borra los objetos en memoria.

Es muy útil preguntar a R sobre las características de determinada función o paquete, esto es posible de las siguientes 2 maneras:

```
help(mean)
```

Una forma alternativa es

```
?mean
```

Para consultar acerca de un paquete o una función que no pertenece al paquete base, este paquete debe estar cargado.

Si queremos sólo ver un ejemplo de la función escribimos:

```
example(mean)
```

En caso de no recordar exactamente el nombre de la función podemos usar la función apropos() que nos indicará todas las funciones que contienen el término:

```
apropos("mean")
```

En cualquier caso resulta muy útil la búsqueda web, por ejemplo en: http://rseek.org/, http://wiki.r-project.org

Por defecto, la ventana del help se abre en la propia consola, de manera que para cerrar el help se debe hacer click en la letra q. Esto puede cambiarse indicándolo en options().

5.2 Opciones de sesión

R ofrece mediante la función options() controlar cómo se visualizan los resultados en sesión. Está función define aspectos generales del trabajo en sesión, tales como: el ancho de la salida impresa –width–, el caracter que oficia de prompt –prompt– y el dispositivo usado para graficar por defecto.

Para saber qué opciones están activadas se utiliza la función getOption(). Así podemos consultar acerca de la cantidad de dígitos después de la coma que se visualizan en pantalla (la precisión con que trabaja la máquina son 16 aunque no se muestren todos) y luego modificar este valor por defecto:

```
> getOption("digits")
[1] 7

> options(digits = 4,OutDec=",")
```

Función	Argumentos	Descripción
objects()	vacío	lista los objetos en memoria
ls()	vacío	lista los objetos en memoria
rm()	objeto	elimina objetos en memoria
q()	vacío	cierra R
save.image()	vacío	guarda espacio de trabajo
help()	función o librería	muestra una descripción de la función o librería
getOptions()	varios	indica cómo se visualizan los resultados en sesión
options()	varios	modifica cómo se visualizan los resultados en sesión
getwd()	vacío	indica el directorio de trabajo actual
setwd()	ruta del archivo	cambia el directorio de trabajo

Cuadro 5.1: Comandos de sesión

5.3 Instalar paquetes

Cuando necesitamos usar funciones que no se encuentran por defecto en R, es decir, no componen el paquete Base, debemos instalar el paquete adecuado. En este caso instalaremos la librería *foreign*, la cual permite importar archivos de diferentes extensiones. Para ello utilizamos la función *install.packages()*.

Antes de instalar nuevos paquetes, conviene saber cuáles están disponibles, esto se logra usando la función available.packages() que para dar un resultado requiere la elección de un espejo:

```
> available.packages()
--- Please select a CRAN mirror for use in this session ---
Loading Tcl/Tk interface ... done
```

En nuestro caso, queremos instalar el paquete foreign que nos permitirá cargar datos con extensiones diferentes. Si aún no elegimos, es decir, el país desde donde descargaremos el archivo debemos hacerlo.

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

probando la URL 'http://cran.fiocruz.br/src/contrib/foreign_0.8.36.tar.gz'
Content type 'application/x-tar' length 236140 bytes (230 Kb)
URL abierta
downloaded 230 Kb
package 'foreign' successfully unpacked and MD5 sums checked
```

Ahora bien, para poder usarlo aun resta cargar el paquete, lo cual haremos usando el comando library():

```
library(foreign)
```

En general un paquete está vinculado a otros por lo cual, puede ser necesario instalar otros paquetes para poder usarlo, para que esto se realice en el momento de descargar el paquete, en el

argumento dependences de install.packages() debe escribirse TRUE. De lo contrario, R avisará que es necesario instalar otros paquetes.

La instalación de un paquete se hace una sola vez, sin embargo cada vez que se requiera usarlo en una nueva sesión es necesario cargarlo usando la función library() como vimos.

Función	Argumentos	Descripción
available.packages()	vacío	enumera librerías disponibles
install.packages()	nombre de la librería	instala una librería
library()	nombre de la librería	carga en memoria una librería ya instalada

Cuadro 5.2: Paquetes disponibles e instalación

6

Data Frames –Marco de datos–

Un data frame puede considerarse como una lista de vectores, cada uno del mismo largo que pueden ser –y por lo general lo son– de distintoa clase: numeric para representar variables cuantitativas –contiene números– o character para representar variables cualitativas –contiene caracteres o números que son etiquetas de dichas categorías. En el primer caso estas variables deben tratarse como numéricas y en el segundo como factores. Esto último es sumamente importante pues sino podríamos por ejemplo calcular la media de una variable cualitativa sin que R note el error.

Usualmente, una base de datos contiene una fila para cada individuo y una columna para cada variable medida a dichos individuos; la existencia simultánea de vectores numéricos y caracter impide guardar dicha base como una matriz. Si el archivo con el que se va a trabajar sólo contiene números o sólo caracteres, es mejor guardarlos en una matriz y no en un data frame.

En las próximas secciones trabajaremos con una base pequeña de manera de poder visualizar rápidamente su estructura y los cambios que hagamos en ella.

6.1 Cargar una base de datos

Dependiendo del tipo de archivo que se desee cargar, será la función a utilizar y en algún caso requerirá cargar una librería extra. R puede leer datos guardados como archivos de texto –ASCII–, archivos de Excel, SAS, SPSS, STATA y otros. Para ilustrar cómo se debe proceder en cada caso ejemplificaremos cargando un archivo llamado “base1”, que tendrá en cada caso una extensión diferente.

6.1.1. Importar archivos que no requieren cargar librería alguna

Podemos distinguir las siguientes funciones:

- read.table(): lee un archivo .txt e implícitamente crea un marco de datos o data frame.

  ```
  > base1=read.table("base1.txt", header=TRUE, sep="", dec=",")
  ```

  El primer argumento indica cuál es el archivo que se desea cargar, indicando nombre y extensión. El argumento header si toma el valor TRUE indica que la primer fila del archivo contiene el nombre de las variables. Mientras que el argumento sep indica cuál es el separador que hay entre las variables, en este caso el espacio; si no se elige el adecuado no podrá cargarse la

base o se hará con errores. Finalmente el argumento dec indica que los valores del archivo utilizan la coma como separador de decimales.
Para una lectura correcta del archivo es imprescindible indicar el separador de variables correcto, los separadores más comunes son:

Separador	Símbolo
Coma	,
Punto y coma	;
Dos puntos	:
Tabulador	\t
Espacio	

Cuadro 6.1: Tipo de separadores

Si en cambio el archivo "base1" tiene una extensión xls –archivo de Microsoft Excel– conviene guardarlo con extensión csv –valores separados por comas–, de esta manera se obtiene un tipo de documento en formato abierto, sencillo para representar datos en forma de tabla, en las que las columnas se separan por comas –pudiéndose indicar otro separador– para abrirlo en R usando la función read.csv() [1]

- read.csv(): lee el archivo cuyo separador entre variables es el que debe indicarse en el argumento sep:

```
> base1=read.csv("base1.csv", header=TRUE, sep=",", ...)
```

En caso de querer cargar el archivo de extensión xls sin modificar su extensión, es necesario cargar la librería gdata y aplicarle la función read.xls().

```
> library(gdata)
> base1=read.xls("base1.xls", sheet=1)
```

El argumento sheet toma por defecto el valor 1 que indica que la hoja que se cargará es la 1. También puede ocurrir que el archivo que pretendemos cargar tiene extensión Rdata, es decir, es de la extensión propia de R. En general, cuando se trabaja con una base de datos muy grande, que requiere utilizar varios cientos de megabytes de memoria ram, conviene una vez cargada convertirla a un archivo de Rdata y luego volver a cargarla utilizando la función load()[2]. Ejemplefiquemos nuevamente para una hipotética base1.Rdata:

```
> load("base1.Rdata")
```

Note que, el único argumento que utilizamos es el que indica el nombre del archivo y expresamente no lo guardamos en un objeto, esto lo hace implícitamente. Si preguntamos por el objeto base1, este se encuentra en la memoria activa y es precisamente el correspondiente al archivo "base1.Rdata".

6.1.2. Importar archivos que requieren cargar la librería "foreign"

Mientras que para cargar datos desde un archivo de SPSS con extensión .sav o un archivo de STATA con extensión .dta o un archivo con extensión .dbf –extensión de las ECHs–, es necesario primero cargar el paquete foreign usando la función library():

[1] Este tipo de archivos también puede abrirse utilizando la función read.table()
[2] Al ser un tipo de archivo propio de R, la memoria ram que utiliza para leerlo y trabajar en él, es menor que si fuera cualquier otro tipo de archivo.

```
> library(foreign)
> base1=read.spss("base1.sav", use.value.labels=TRUE, to.data.frame=TRUE)
> base1=read.dbf("base1.dbf", as.is=FALSE)
> base1=read.dta("base1.dta",convert.factors=TRUE)
```

- read.spss(): el argumento use.value.labels permite mantener el nombre de las variables que ya trae el archivo y el argumento to.data.frame permite guardar la base como un data frame.
- read.dbf(): el argumento as.is es FALSE por defecto, esto implica que los vectores caracter no son convertidos a factor.
- read.dta(): el argumento convert.factors es análogo al argumento as.is de la función read.dbf, pero para esta función su valor por defecto es TRUE, por lo tanto los vectores caracter son convertidos a factor.

  Estas funciones tienen más argumentos que pueden ser útiles según el caso, para consultar el detalle de los mismos recurre al help().

Función	Argumentos	Descripción
read.table()	archivo, encabezado, separador, ...	carga archivo de texto
load()	archivo	carga un archivo Rdta
read.csv()	archivo, encabezado, separador	carga un archivo .csv
read.spss()	archivo, encabezado	carga un archivo de SPSS
read.dta()	archivo, conversión a factor, ...	carga un archivo de STATA
read.dbf()	archivo, conversión a factor, ...	carga un archivo .dbf

Cuadro 6.2: Importar archivos

6.2 Guardar o exportar objetos en archivos

Recordemos que todos los cambios que hagamos en la base de datos no suponen ninguna modificación del archivo, para que esto tenga efecto procedemos a guardar la base en un archivo usando la función write.table(), con el nombre base1. El destino del archivo será el directorio de trabajo pero podemos explicitar otra ruta. La extensión del mismo no está especificada por lo cual puede abrirse en el programa que se desee:

```
write.table(bases.1, file="base.1")
```

Se especifica primero el nombre del objeto y luego el nombre que va a tomar el archivo encerrado entre comillas.
Genéricamente los argumentos de la función y sus valores por defecto son los siguientes:

```
write.table(x, file ="", append=FALSE, sep=" ", col.names=TRUE)
```

- x: el objeto a ser escrito, preferiblemente una matriz o dataframe.
- file: indica el nombre del archivo que será creado
- append logical. Si es TRUE, el output es anexado al archivo existente, si es FALSE, cualquier archivo existente con ese nombre es destruido.

- sep: el separador de string.
- col.names por defecto tiene el valor TRUE, por tanto, conserva el nombre de las variables

Si deseamos guardar objetos en archivos de formato ASCII (.txt) o de valores separados por coma (.csv) debemos utilizar las siguientes funciones [3]:

Para guardar un objeto en un archivo con extensión .txt utilizamos la función write.table():

```
> write.table(base1,file="base1.txt", append=FALSE, sep="")
```

Como se observa, en el primer argumento se indica el nombre del objeto a ser guardado, luego en el argumento file se indica el nombre del archivo y su extensión[4]. Un argumento adicional que se puede utilizar es append, cuando es FALSE –lo es por defecto– implica que si ya existe el archivo que queremos guardar, éste será reemplazado por el nuevo. En cambio, si indicamos pretendemos que el objeto base1 se anexe al archivo existente, debemos indicar append=TRUE.

Es importante resaltar en este punto, que cualquiera sea el objeto lo podemos guardar usando la función write.table(, más adelante veremos que suele ser muy útil guardar de esta forma, por ejemplo, las tablas de frecuencias generadas.

Por otra parte, para guardar un objeto con la extensión .csv utilizamos la función write.table() o la función write.csv():

```
> write.csv(base1,file="base1.csv")
```

Para guardar con el formato propio de R utilizamos la función save():

```
> save(file="objetos.Rdata")
```

De esta manera se guardan todos los objetos que se encuentra en memoria, en un mismo archivo denominado “objetos”. Cuando eso no es lo que se prentende debe indicarse, como primer argumento de la función el nombre del objeto a guardar:

```
> save(base1, file="base1.Rdata")
```

Por defecto, los objetos se ubicarán en el directorio de trabajo, para modificar el destino debe indicarse la ruta junto al nombre que le daremos al archivo.

A esta altura ya queda claro que para cada función de lectura de un archivo –read– existe una función análoga para exportar un archivo de la misma extensión –write–. A su vez, así como para leer archivos de extensión .dta, .sav o .dbf es necesario primer cargar la librería “foreing”, también lo es para exportar un objeto en esas extensiones.

Puede ser necesario tener que enviar a otra persona una base de datos en un archivo de un software privativo –STATA, SPSS, etc.–, para ello tenemos dos opciones. La primera es enviar la base de datos en una extensión .txt que se puede abrir en cualquier programa privativo. La segunda opción es guardarlo directamente en el formato específico de dicho programa.

Para guardar un objeto en un archivo con extensión .dta, utilizamos la función write.dta():

[3] No es necesario invocar ningún paquete de funciones

[4] Eventualmente, si se desea guardar en un lugar diferente al directorio de trabajo debe indicarse la ruta en dónde se ubicará el archivo antes del nombre y extensión.

```
> library(foreign)
> write.dta(base1,file="base1.dta")
```

Para exportar un objeto en formato de extensión.sav, utilizamos la función write.foreign():

```
> write.foreign(base1,"base1.sav",package=c("SPSS"))
```

Esta función también permite guardar un objeto en un archivo de STATA, para ello en el argumento package debemos indicarle STATA en lugar de SPSS.

Función	Argumentos	Descripción
write.table()	objeto, nombre del archivo	crea archivo
write.csv()	objeto, nombre del archivo	crea archivo de extensión csv
save()	objeto, nombre del archivo	crea archivo de extensión Rdata
write.dta()	objeto, nombre del archivo	crea archivo de STATA
write.foreign()	objeto, nombre del archivo, programa	crea archivo de SPPS o STATA

Cuadro 6.3: Exportar archivos

6.3 Ver y editar una base de datos

En lo que sigue trabajaremos con una base de 5 variables y 8 casos, de manera de ilustrar cómo operan las funciones, por tanto todo lo que utilicemos será fácilmente extrapolable para bases de mayores dimensiones.

Procedemos a cargar la base que tiene extensión de archivo d e texto y guardarla en el objeto base1:

```
base1=read.table("base1.txt", header=T)
```

Resulta natural querer ver qué contiene el objeto base1, para ello simplemente escribimos: base1. En este caso la base se visualiza en la consola, lo cual en general no es muy práctico. Cuando se tiene una base de pocas variables y sólo se quieren visualizar algunos casos, se pueden usar las funciones: head() o tail() que muestran los primeros o últimos 6 casos de la base[5], respectivamente:

```
> head(base1)
  idpersona sexo edad idnucleo ingresos
1  11534349    1   53       36      552
2  11534349    1   22       36      509
3  15795957    1   58       79       22
4  38731140    2    5       45       42
5  27387346    2   38       96      958
6  12588307    1   50       11      337

> tail(base1)
   idpersona sexo edad idnucleo ingresos
5   27387346    2   38       96      958
6   12588307    1   50       11      337
7   28131952    1   85       12      494
8   36009937    2   18       42      571
9   12428993    1   19        2      188
10  12794411    1   14       87      966
```

Una mejor opción es usar la función edit(), la misma también permite acceder solamente a algunos casos, indicando las filas y columnas que se quieren ver, pero a diferencia de head() y tail(), abre una ventana para visualizar mejor los datos. Para seguir trabajando, debemos cerrar dicha ventana, como consecuencia se desplegarán los datos en la consola.

```
edit(base1)
edit(base1[1:10,]) #para ver las primeras 10 filas y todas las columnas
edit(base1[,2:5]) #para ver todas las filas y las columnas de la 2 a la 5
```

El inconveniente de utilizar la función edit() es que no permite seguir trabajando manteniendo la base abierta. Una buena alternativa es utilizar la función showData() del paquete “relimp”.

```
library(relimp)
showData(base1)
```

[5]Puede escribirse:head(base1, 15), para obtener los primeros 15 casos de base1, y así sucesivamente.

6.3.1. Inspeccionar la base de datos

Otra forma complementaria de hacerse una idea de qué es lo que contiene dicha base de datos es usando el comando str() que muestra la estructura del objeto y despliega en pantalla parte de los datos que contiene. Este comando permite conocer de que tipo son las variables que componen la base de datos.

```
> str(base1)
'data.frame':   10 obs. of  5 variables:
 $ idpersona: int  11534349 11534349 15795957 38731140 27387346$
 $ sexo     : int  1 1 1 2 2 1 1 2 1 1
 $ edad     : int  53 22 58 5 38 50 85 18 19 14
 $ idnucleo : int  36 36 79 45 96 11 12 42 2 87
 $ ingresos : int  552 509 22 42 958 337 494 571 188 966
```

La salida anterior indica que la base contiene 10 observaciones y 5 variables, las que se detallan abajo, junto con los valores que toman esas variables en cada caso; note que para la variable "idpersona" sólo se muestran los primeros 5 valores. Todas ellas son del tipo integer –entero– lo que indica la palabra int, esto indica el modo de almacenamiento del dato. Luego se explicitan los valores que toma cada una de las variables.

Es importante aplicar esta función antes de empezar a trabajar con la base de datos, porque las variables cualitativas que están etiquetadas con número, como "sexo", "idpersona" e "idnucleo" en este caso, si no se especifica lo contrario son tomadas como numéricas y por tanto podrían calcularse medias y varianzas, por ejemplo, lo cual es incorrecto. En este caso es necesario convertir las variables "idpersona", "sexo" e "idnucleo" a factor, para ello utilizamos la función as.factor(). Óbservese que guardamos la variable convertida en la misma variable.

```
> base1$sexo=as.factor(base1$sexo)
> base1$idpersona=as.factor(base1$idpersona)
> base1$idnucleo=as.factor(base1$idnucleo)
> class(base1$sexo)
[1] "factor"
> class(base1$idpersona)
[1] "factor"
> class(base1$idnucleo)
[1] "factor"
```

Con la función class() verificamos que tipo variable es, en este caso el resultado es factor, como esperábamos.

El comando summary() muestra información resumida de un objeto, por ejemplo, para las variables numéricas muestra un resumen descriptivo, donde se detallan: mínimo –min–, primer cuartil –1st Qu–, mediana –median–, media –mean–, tercer cuartil –3rd Qu– y máximo –max–. El análisis de de este resumen descriptivo puede dar cuenta de que alguna variable no es de la clase que debe ser.

```
> summary(base1)
    idpersona sexo       edad          idnucleo    ingresos
 11534349:2   1:7   Min.   : 5.00   36     :2   Min.   : 22.0
 12428993:1   2:3   1st Qu.:18.25   2      :1   1st Qu.:225.2
 12588307:1         Median :30.00   11     :1   Median :501.5
 12794411:1         Mean   :36.20   12     :1   Mean   :463.9
 15795957:1         3rd Qu.:52.25   42     :1   3rd Qu.:566.2
 27387346:1         Max.   :85.00   45     :1   Max.   :966.0
 (Other) :3                         (Other):3
```

Si quiere obtenerse una medida de resumen adicional, por ejemplo, la varianza[6] de dichas variables debe utilizarse la función var():

[6] La varianza en este caso toma la forma del estimador máximo verosimil: $\sum_{i=1}^{n} x_i \frac{}{n-1}$

```
> var(base1$edad);var(base1$ingresos)
[1] 625.2889
[1] 109323.4
```

Así aplicamos la función var() a cada variable numérica de la base –las órdenes están separadas por punto y coma–, más adelante veremos una forma más simple de hacer esto. Para obtener el desvío estándar, el procedimiento es idéntico pero sustituyendo var() por sd().

Función	Argumentos	Descripción
summary()	objeto	resumen descriptivo
mean()	objeto, exclusión de NA	media aritmética
var()	objeto, exclusión de NA	varianza
sd()	objeto, exclusión de NA	desvío estándar
cov()	objeto, exclusión de NA	covarianza
cor()	objeto, exclusión de NA	correlación lineal
quantile()	objeto, exclusión de NA	cuantiles
IQR()	objeto, exclusión de NA	Recorrido intercuartílico

Cuadro 6.4: Análisis descriptivo de un objeto

Otras características de una base de datos se pueden consultar fácilmente: las dimensiones del objeto a través del comando dim(), los nombres de las variables a través de names() y los nombres de las filas –que por lo general no tienen nombre específico– a través de row.names().

```
> dim(base1)
[1] 10  6
> names(base1)
[1] "idpersona" "sexo"      "edad"      "idnucleo"  "ingresos"
> row.names(base1)
[1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
```

6.3.2. Acceder a variables o casos de un dataframe

Como ya enunciamos, para ver en pantalla los elementos de un objeto basta escribir su nombre. Mientras que para acceder a determinadas filas o columnas existen diferentes maneras: indicando el nombre de la variable, se escribe el nombre del objeto, el símbolo $ y el nombre de la variable:

```
> base1$sexo
 [1] 1 1 1 2 2 1 1 2 1 1
Levels: 1 2
```

o indicando la columna en la cual se encuentra

```
> base1[,2]
 [1] 1 1 1 2 2 1 1 2 1 1
Levels: 1 2
```

También se puede acceder a una variable simplemente escribiendo su nombre, pero ello requiere haber convertido las columnas de un data.frame en variables en memoria usando la función attach(). Conviene evitar usar esto cuando se trabaja simultáneamente con dos bases que tienen algún nombre de columnas iguales.

```
> attach(base1)
> sexo
 [1] 1 1 1 2 2 1 1 2 1 1
Levels: 1 2
```

Para deshacer este cambio se utiliza el comando detach().

```
> detach(base1)
```

6.4 Renombrar

Es posible cambiar los nombres de las columnas, un error frecuente suele ser el siguiente:

```
base1$identificador=base1$idpersona
```

Esto crea una nueva variable denominada identificador que tiene los mismos datos que idpersona. Si queremos en este caso, cambiar el nombre de la variable idpersona que se encuentra en la primer columna de la base:

```
> names(base1)<-c("identificador",names(base1[2:5]))
> names(base1)
[1] "identificador"   "sexo"     "edad"     "idnucleo" "ingresos"
```

Alternativamente puede hacerse de este modo, accediendo al primer elemento de names(base1), un vector de clase character y sustituyéndolo por identificador:

```
names(base1)[1] = "identificador"
```

Función	Argumentos	Descripción
edit()	objeto	abre ventana con datos
head()	objeto	despliega los primeros casos
tail()	objeto	despliega los últimos casos
showData()	objeto	abre ventana de datos
str()	objeto	resumen del tipo de variables
dim()	cantidad de filas, cantidad de columnas,...	indica dimensiones
names()	vector caracter	nombre de variables
row.names()	vector caracter	nombre de filas
rownames()	vector caracter	nombre de filas
colnames()	vector caracter	nombre de columnas
nrow()	objeto	cantidad de filas
ncol()	objeto	cantidad de columnas
length()	objeto	cantidad de elementos
class()	objeto	clase
mode()	objeto	modo
is.na()	objeto	evalúa existencia de NA
complete.cases()	objeto	excluye casos con NA

Cuadro 6.5: Inspección de un objeto

6.5 Etiquetar una variable categórica

Suele ser muy útil a la hora de interpretar los resultados de un análisis, etiquetar los valores de las variables categóricas. Para ello debemos crear un factor usando el comando factor(). Las categorías de un factor se especifican con el argumento levels y sus etiquetas, como ya vimos, con el argumento labels. Ejemplificamos para la variable “sexo”:

```
> base1$sexo = factor(base1$sexo,
                      levels=c(1,2),
                      labels=c("mujer", "hombre"))
> base1$sexo
 [1] mujer  mujer  mujer  hombre hombre mujer  mujer  hombre mujer  mujer
Levels: mujer hombre
```

6.6 Categorizar una variable numérica

Nos proponemos categorizar la variable edad en dos valores, cero y uno, de manera de discriminar entre aquellos individuos que tienen menos de 14 años y el resto. Esto resulta útil cuando se desea agregar los datos de una variable y obtener información, por un lado para los menores de catorce y por otro lado, del resto. Primero creamos el objeto donde guardaremos la nueva variable edadrec, para ello definimos un vector numérico de largo igual a la cantidad de filas del objeto base1:

```
> edadrec <- numeric(nrow(base1))
> edadrec[base1$edad <14] <- 0
> edadrec[base1$edad >=14] <- 1
> edadrec
 [1] 1 1 1 0 1 1 1 1 1 1
```

Luego es necesario convertir la nueva variable, edadrec, en factor:

```
edadrec<-as.factor(edadrec)
```

Alternativamente, se puede categorizarla y convertirla directamente en factor, pero esto implica en lugar de adjudicarle un número, adjudicarle una categoría.

```
> edadrec <- numeric(nrow(base1))
> edadrec[base1$edad <14] <- "menos14"
> edadrec[base1$edad >=14] <- "14ómás"
> edadrec
 [1] "14ómás"  "14ómás"  "14ómás"  "menos14" "14ómás"$
```

Ahora bien, para que tenga sentido lo anterior debo agregar esta variable a la base de datos, para que se mantenga asociada a los individuos de la misma. Para ello utilizo la función cbind() que concatena el vector que contiene a edadrec con el objeto base1, en este caso, la nueva base de datos es guardada en otro objeto.

```
base1.1<-cbind(base1, edadrec)
```

Note que la nueva variable lleva el nombre del objeto.
Alternativamente podemos usar la función recode() del paquete Hmisc.

Función	Argumentos	Descripción
as.numeric()	objeto	convierte a modo numérico
as.character()	objeto	convierte a modo caracter
as.factor()	objeto, niveles, etiquetas	convierte a clase factor
as.matrix()	objeto, número de filas	convierte a clase matriz
as.data.frame()	objeto	convierte a clase data.frame
as.table( )	objeto	convierte a tabla

Cuadro 6.6: Convertir objetos

6.7 Describir un factor

Para resumir la información de un factor mediante una tabla se utiliza la función table():

```
> table(base1.1$sexo)

  mujer hombre
     7      3
```

Si en cambio pretendemos obtener una tabla cruzada o de doble entrada, se deben especificar ambas variables:

```
> tabla=table(base1.1$sexo,base1.1$edadrec)
> tabla

    14ómás menos14
  1      7       0
  2      2       1
```

Para agregar los totales por fila y columna usamos la función addmargins():

```
> tabla.total=addmargins(tabla)
> tabla.total

      14ómás menos14 Sum
  1        7       0   7
  2        2       1   3
  Sum      9       1  10
```

Si agregamos el argumento margin=1, sólo obtenemos los totales por fila, mientras que con margin=2, los obtenemos por columna.

```
> addmargins(tabla,margin=1)

         14ómás menos14
  mujer       7       0
  hombre      2       1
  Sum         9       1

> addmargins(tabla,margin=2)

         14ómás menos14 Sum
  mujer       7       0   7
  hombre      2       1   3
```

Para obtener los valores relativos se aplica la función prop.table(). Esta función permite calcular las frecuencias relativas por fila, columna o total, en cada caso se modifica el argumento margin.

```
> tabla.rel=prop.table(tabla)
> tabla.rel

    14ómás menos14
```

```
  1     0.7      0.0
  2     0.2      0.1
```

Para obtener una tabla de frecuencias relativas con los totales por fila y columna debemos aplicar la función addmargins() a la tabla generada con la función prop.table():

```
> tabla.rel.tot=addmargins(tabla.rel)
> tabla.rel.tot

         14ómás menos14 Sum
  mujer     0.7      0.0 0.7
  hombre    0.2      0.1 0.3
  Sum       0.9      0.1 1.0
```

Si queremos obtener la misma tabla en porcentajes, ¿qué deberíamos hacer?

Puede ser útil expresar la proporción de una variable en función de las categorías de otra variable. Por ejemplo, para obtener la proporción de menores a 14 años dentro de la proporción de mujeres y lo mismo para la proporción de hombres, podemos utilizar el argumento margin:

```
> tabla.rel.fila=addmargins(prop.table(tabla,margin=1), margin=2)
> tabla.rel.fila

            14ómás   menos14       Sum
  mujer  1.0000000 0.0000000 1.0000000
  hombre 0.6666667 0.3333333 1.0000000
```

La presentación de una tabla es muy importante, en este caso por ejemplo conviene reducir la cantidad de decimales de los valores de la tabla, esto lo hacemos usando la función round():

```
> tabla.rel.fila=addmargins(prop.table(tabla,margin=1), margin=2)
> round(tabla.rel.fila,2)

         14ómás menos14  Sum
  mujer    1.00    0.00 1.00
  hombre   0.67    0.33 1.00
```

Para incluir el objeto tabla.rel.tot en un documento de texto, debemos guardarlo en un archivo con extensión .csv, luego abrirlo en una hoja de cálculo para finalmente pegarlo en un documento de texto.

```
> write.table(tabla.rel.tot,"tabla1.csv")
```

Función	Argumentos	Descripción
table()	objeto	crea tabla
addmargins()	objeto	crea totales de tabla por fila y columna
prop.table()	objeto	crea tabla en valores relativos

Cuadro 6.7: Tablas descriptivas de un objeto

6.8 Seleccionar sub-bases

Muchas veces debemos reducir la cantidad de variables de una base de datos, esto resulta sumamente sencillo de realizar usando lo que aprendimos de seleccionar elementos de una matriz. Trabajemos con la base1:

```
> base1
   idpersona sexo edad idnucleo aniosed ingresos
1   27387346    2   38       96       0      958
2   11534349    1   53       36       0      552
3   12588307    1   50       11       0      337
4   28131952    1   85       12       0      494
5   38731140    2    5       45       0       42
6   12428993    1   19        2       0      188
7   11887350    1   22       36       0      509
8   36009937    2   18       42       0      571
9   15795957    1   58       79       0       22
10  12794411    1   14       87       0      966
```

Así, si por ejemplo queremos seleccionar las variables 1,3, 4 y 6 de la base1 escribimos la siguiente orden:

```
> base1.1=base1[,-c(2,5)]
> base1.1
   idpersona edad idnucleo ingresos
1   27387346   38       96      958
2   11534349   53       36      552
3   12588307   50       11      337
4   28131952   85       12      494
5   38731140    5       45       42
6   12428993   19        2      188
7   11887350   22       36      509
8   36009937   18       42      571
9   15795957   58       79       22
10  12794411   14       87      966
```

Esto indica que dejamos de lado las variables ubicadas en las columnas 2 y 5. Otra manera, análoga a la anterior, es indicar cuáles variables queremos retener:

```
> base1.1=base1[,c(1,3:4,6)]
> base1.1
   idpersona edad idnucleo ingresos
1   27387346   38       96      958
2   11534349   53       36      552
3   12588307   50       11      337
4   28131952   85       12      494
5   38731140    5       45       42
6   12428993   19        2      188
```

```
7   11887350   22        36       509
8   36009937   18        42       571
9   15795957   58        79        22
10  12794411   14        87       966

#Alternativamente:
# base1.1=base1[,c("idpersona","edad","idnucleo", "ingresos")]
```

Cuál de estas opciones es mejor, claramente depende de la cantidad de variables que debe especificarse en uno u otro caso.

Otra opción es utilizar la función subset() de la siguiente forma:

```
> base1.1=subset(base1, select=c(1,3,4,6))
> base1.1

   idpersona edad idnucleo ingresos
1   27387346   38        96       958
2   11534349   53        36       552
3   12588307   50        11       337
4   28131952   85        12       494
5   38731140    5        45        42
6   12428993   19         2       188
7   11887350   22        36       509
8   36009937   18        42       571
9   15795957   58        79        22
10  12794411   14        87       966
```

Veamos los argumentos de la función subset():

```
subset(x, subset, select)
```

- x: objeto
- subset: expresión lógica que indica las observaciones a seleccionar
- select: expresión que indica las columnas a seleccionar

Note que en el ejemplo anterior no establecimos ninguna condición lógica para seleccionar algunos casos de la base pues pretendíamos quedarnos con todos.
Alternativamente sin necesidad de crear otra base podemos utilizar la función frameApply() del paquete gdata que permite hacer análisis para un subcojunto de datos

6.9 Seleccionar algunos casos

Si en cambio, pretendemos reducir la dimensión de la base pero en relación a las filas, es decir, buscamos seleccionar algunos casos de acuerdo a cierta condición, podemos recurrir a la función which() o subset(). Por ejemplo, construímos un vector con los individuos que son mayores de 40 años, en el objeto edad40 se guarda el número de fila correspondiente a dichos individuos.

```
> edad40=which(base1[,3]>40)
> edad40

[1] 2 3 4 9

> base1.edad40

  idpersona sexo edad idnucleo aniosed ingresos
2  11534349    1   53       36       0      552
3  12588307    1   50       11       0      337
4  28131952    1   85       12       0      494
9  15795957    1   58       79       0       22

#alternativamente
# base1.edad40=base1[which(base1[,3]>40),]
```

Alternativamente, usando la función subset()

```
> edad40=subset(base1, edad>40)
> edad40

  idpersona sexo edad idnucleo aniosed ingresos
2  11534349    1   53       36       0      552
3  12588307    1   50       11       0      337
4  28131952    1   85       12       0      494
9  15795957    1   58       79       0       22
```

6.10 Crear nuevas variables

Es sencillo crear nuevas variables en función de otras, para esto resulta muy útil en muchos casos la función ifelse() la cual evalúa si se cumple cierta condición y asigna cierto valor a la nueva variable y en caso contrario asigna otro valor. La función ifelse() tiene la siguiente forma:

```
ifelse(test, yes, no)
```

- test es un objeto en modo lógico: una condición.
- yes retorna el o los valores indicados cuando se cumple la condición
- no retorna el o los valores indicados cuando no se cumple la condición

Por ejemplo, construímos una variable que imputa el valor de la variable "ingresos" a las personas mayores de 13 años, de lo contrario le asigna un valor "0" pues no correspondería que un menor de 14 años tuviera ingresos positivos.

```
> ingresos.trabajo=ifelse(base1$edad>13, base1$ingresos,0)
> ingresos.trabajo
 [1] 958 552 337 494   0 188 509 571  22 966
```

Observe que en este caso asignamos un valor si se cumplía una condición y si no se cumplía asignamos otro[7]. Muchas veces es necesario, a partir de una variable generar una nueva pero según más de una condición. Esto puede realizarse utilizando la función ifelse() de forma anidada, esto es:

```
ifelse(test1, yes1, ifelse(test2, yes2, ifelse(test3, yes3, no)))
```

En este ejemplo utilizamos 3 condiciones, si la primer condición se cumple la nueva variable vale lo especificado en yes1, de lo contario se continúa evaluando la segunda condición, test2. Si la segunda condición se cumple, la nueva variable vale lo especificado en yes2, de lo contrario se evalúa la tercer condición, test3. Finalmente, si la tercera condición se cumple, la nueva variable valdrá lo especificado en yes3, en caso contrario valdrá lo especificado en no. Así se establecieron 3 condiciones y una por defecto –si no se cumplen ninguna de las anteriores–, para dar lugar a 4 posibles valores de la nueva variable.

[7] Naturalemente una condición puede componerse de varias condiciones a la vez, es decir, podemos exigir que el individuo sea mayor a 13 años y también que sea hombre, sin embargo, estas 2 condiciones como se evalúan a la vez, representan una única condición o test.

6.11 Aplicar una función a varias variables

Para aplicar una función a varias variables en muchos programas es necesario recurrir a un bucle[8] –loop– o en el peor de los casos a aplicar el comando cada vez, pero en R podemos utilizar la función apply() o lapply(), dependiendo del objeto y el resultado que queramos obtener. La función apply() se puede aplicar a una matriz, array o data frame y el resultado será un objeto de tipo matriz, array o lista –en caso que no pueda devolver una matriz–. Mientras que la función lapply() se puede aplicar a un data frame o a una lista, devolviendo siempre una lista.

Genericamente la función apply() tiene esta forma:

```
> apply(x,margin,function)
```

- x es una matriz o data.frame
- margin indica si la operación se realiza por fila –1– o por columna –2–.
- function indica la función a utilizar, puede ser propia de R o creada por el usuario

Para ejemplificar el uso de esta función cargamos el archivo “apply.csv” y lo guardamos en el objeto base1.

```
> base1=read.csv("apply.csv", sep=";", header=T)
```

Si queremos calcular la media de las variables “edad”, “aniosed” e “ingresos” podemos aplicar la función mean() tres veces, es decir, una vez para cada variable. Alternativamente, resulta mucho más práctico utilizar la función apply().

```
> apply(base1[ , c("edad", "aniosed", "ingresos") ] , 2 , mean)
    edad  aniosed  ingresos
    42.0    11.1     463.9
```

El primer argumento está indicando que del objeto base1, se utilizan las 3 variables mencionadas. Para aplicar la función por columnas es necesario agregar el argumento margin igual a 2, si se desea calcular por filas –fíjese que en este caso sería incorrecto–, se debe indicar con un 1. Mientras que la función que se aplica se indica escribiendo su nombre, en este caso, mean().

Observe que el objeto al cual aplicamos la función es un data frame, mientras que el resultado del mismo es una matriz. Es importante notar que siempre que usemos la función apply() obtendremos como resultado una matriz o un array, esto es, una matriz de mayor dimensión. Mientras que la función lapply() siempre devuelve una lista como resultado, por ello su nombre l(ist)apply. Por lo tanto, dependerá de la clase de objeto que queramos como resultado cuál de las 2 variables utilizar.

La función lapply() a su vez, se diferencia de la anterior pues no tiene el argumento margin:

```
> lapply(x, function)
```

[8] Un bucle en programación, es una sentencia que se realiza repetidas veces a una parte de código, hasta que la condición asignada a dicho bucle deje de cumplirse. Generalmente, un bucle es utilizado para hacer una acción repetida sin tener que escribir varias veces el mismo código, lo que ahorra tiempo, deja el código más claro y facilita su modificación en el futuro.

Si esta función es aplicada a un data frame, el data frame es visto como un caso particular de una lista donde cada componente –las variables– tienen la misma cantidad de elementos. Así la función lapply() a cada componente de la lista le aplica la función especificada. Si esa lista es un data frame, lo anterior implica que considera cada variable del data frame para aplicarle la función especificada; es por esto que no tiene el argumento margin.

Si aplicamos esta función a las mismas variables que en el caso anterior para calcular la media de éstas, obtenemos lo siguiente:

```
> lapply(base1[ , c("edad","aniosed","ingresos")] , mean)
$edad
[1] 42
$aniosed
[1] 11.1
$ingresos
[1] 463.9
```

6.11.1. Utilzar loops

Dijimos que otra manera de realizar lo mismo que hace un apply() o lapply() es usar loops. Veamos cómo puede obtenerse la media de esas tres variables mediante la repeteción de una misma orden tres veces. Para ello es necesario utilizar una estructura de programación denominada for, la cual permite repetir un código –una o más sentencias de programación– un cierto número de veces[9].

```
> base2=base1[,c(3,5,6)]
> medias=numeric(ncol(base2))
> for (i in 1:ncol(base2)){
  medias[i]=mean(base2[i])
  }
```

Primero como sólo vamos a calcular la media de algunas variable del objeto base1, creamos uno nuevo que sólo contiene las variables de interés y le llamamos base2. En cualquier caso siempre creamos antes de realizar el loop la variable dónde se guardarán los resultados del mismo, en este caso, el objeto medias es un vector numérico que a priori contiene tantos ceros como número de columnas tiene el objeto base2. Luego se ejecuta la estructura for iniciando de 1 a 3 –. y dentro de ella se ejecuta el bloque, es decir, aquello encerrado entre llaves –{ }–.

Veamos detenidamente qué es lo que hace esta estructura. La primer línea está indicando que se repetirá la estructura ncol(base2) veces, es decir tres veces, empezando de uno, por esto el indicador de la repetición, i, varía de 1 a ncol(base2). Lo que se encuentra entre llaves es la estructura o el bloque de código a repetir. Para la repetición i, se calcula la media de la variable i del objeto base2, esto se guarda en el elemento i del vector medias. Esta orden se ejecuta como dijimos 3 veces, primero para la variable 1, luego para la 2 y finalmente para la 3. Para ver el resultado observamos qué contiene el vector medias:

```
> medias
[1]  42.0  11.1 463.9
```

[9] Existen varias estructuras de este tipo cuando el número de repeticiones está determinado por un número dado –for, repeat–, o hasta que deje de cumplirse o se cumpla una condición –while–.

Es bueno aclarar que escribimos el for en tres líneas para hacer más claro el código –así también suele escribirse en otros lenguajes de programación–, pero sin inconvenientes se puede escribir en una sóla línea. A su vez, cada vez que escribimos ncol(base2) podríamos haber puesto 3 directamente, sin embargo, al hacer referencia al número de columnas de base2, la orden es genérica y si luego agregamos una nueva variable a la base2 para calcular la media, la orden for no es necesario modificarla.

Si bien, en este ejemplo, resulta más sencillo usar la función apply() en relación a la estructura for, en otras ocasiones es imprescindible realizar un for, pues éste último permite no sólo aplicar una función a varias variables sino repetir una orden cualquiera sea un cierto número de veces. Al igual que como vimos con la función ifelse(), puede anidarse un for dentro de otro, tantas veces como sea necesario.

6.12 Aplicar una función que relacione varias variables

En este caso no pretendemos aplicar una función a varias variables sino aplicar una función que relacione dichas variables. Para ello, una manera sencilla es usar la función tapply(), que genera una tabla para la variable especificada según los factores de otra u otras variables. Por ello su nombre t(able)apply.

Si queremos calcular como antes la media de la variable “edad”, pero ya no para el conjunto de individuos sino para cada uno de los hogares, usamos tapply() y aplicamos la función mean() a la variable “edad” para cada nivel del factor “idnucleo”.

```
tapply(base1$edad , base1$idnucleo , mean)
     136     142     145     179     186     196     961
   53.00   18.00   24.67   58.00   44.00   38.00   67.50
```

El resultado es un objeto del mismo largo que la cantidad de niveles del factor. Es decir, primero agrupa los individuos por “idnucleo”, luego dentro de cada conjunto de individuos con “idnucleo” común, calcula la media de la variable “edad”. Así obtenemos que el promedio de edad de los integrantes del hogar número 136 es de 53 años, para el hogar número 142 es de 18 años y así sucesivamente.

El resultado de un tapply() en este caso es un array[10] pero en otros casos resulta ser una lista, para poder incluírla en un documento de texto como una tabla debemos exportarla usando la función write.table(). Mientras que para poder incluirla en un data frame es necesario convertirla en matriz y luego agregarla a la base utilizando la función cbind() o merge() según corresponda.

Veamos genéricamente como se formula la función tapply():

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

- X un vector
- INDEX una lista de factores del mismo largo que X

[10] Un array es un tipo de objeto de R que generaliza a la matriz pues además de filas y columnas tiene por lo menos una dimensión adicional, que en este caso está indicando a qué variable refiere la tabla.

- FUN la función a aplicar
- simplify si es FALSE el resultado del tapply será un array de modo list, si su valor es TRUE el modo del array será del tipo de objeto que retorne la función

Es importante resaltar que el segundo argumento de la función debe ser un factor, si es una variable numérica que toma pocos valores, R la convierte en factor para poder realizar la operación, pero si se trata de una variable numérica que toma muchos valores, no se podrá utilizar la función. También es posible indicar más de un factor para realizar la subdivisión de los individuos. Veamos un ejemplo en que la función tapply() aplica la función mean() a la variable "edad" para cada nivel del factor "idnucleo" y cada nivel del factor "sexo". es decir, conforma grupos de individuos según "idnucleo" y dentro de ellos separa por *hombre* y *mujer*, en cada uno de estos subgrupos, calcula la media de la variable "edad".

```
> tapply(base1$edad , list(base1$idnucleo,base1$sexo) , mean)
     hombre   mujer
136      NA    53.0
142      18      NA
145      33    20.5
179      NA    58.0
186      44      NA
196      38      NA
961      NA    67.5
```

Así tenemos que en el hogar 136, no hay dato para la media de la variable "edad" de hombres, lo que implica que en ese hogar no hay hombres, mientras que la media de "edad" de la mujeres es 53 años.

Alternativamente, en lugar de usar la función tapply(), podemos utilizar la función by() o aggregate(), a diferencia de la primera, éstas dos no devuelven el resultado como una tabla.

La función by(), aplica una función a un conjunto de datos agrupados según cierto factor. Genéricamente, tiene la siguiente forma:

```
by(data, INDICES, FUN)
```

- data: data frame o matriz
- INDICES: factor o lista de factores según los cuales se agruparán los datos
- FUN: la función a aplicar

En este caso agrupo a los individuos de base1 según la variable "idnucleo" –segundo argumento–, y calcula la media –tercer argumento– de la variable "edad" –primer argumento–. El resultado es una lista, donde cada elemento de la misma indica el hogar que corresponde y la media de la variable edad para dicho hogar.

```
> by(base1$edad,base1$idnucleo,mean)
base1$idnucleo: 136
[1] 53
------------------------------------------------------------
base1$idnucleo: 142
```

```
[1] 18
------------------------------------------------------------
base1$idnucleo: 145
[1] 24.66667
------------------------------------------------------------
base1$idnucleo: 179
[1] 58
------------------------------------------------------------
base1$idnucleo: 186
[1] 44
------------------------------------------------------------
base1$idnucleo: 196
[1] 38
------------------------------------------------------------
base1$idnucleo: 961
[1] 67.5
```

Ahora, para aplicar la función mean() por *hogar* y dentro de estos a *hombres* y *mujeres* por separado, podemos volver a utilizar la función by():

```
> by(base1$edad,list(base1$idnucleo,base1$sexo),mean)
```

El resultado de lo anterior no lo mostramos porque es una lista más extensa que la anterior. Sin embargo, podemos obtener un resultado más manejable usando la función aggregate(), según se muestra a continuación:

```
> aggregate(base1$edad, list(base1$idnucleo,base1$sexo),mean)
  Group.1 Group.2    x
1     142  hombre 18.0
2     145  hombre 33.0
3     186  hombre 44.0
4     196  hombre 38.0
5     136   mujer 53.0
6     145   mujer 20.5
7     179   mujer 58.0
8     961   mujer 67.5
```

Note que el resultado de aplicar esta función es un data frame y ya no una lista, aunque puede serlo en algunos casos. Sin embargo, los nombres de las variables del data frame son genéricos, así “Group.1” corresponde a la variable “idnucleo”, “Group.2” corresponde a la variable “sexo” mientras que “x”, corresponde a la variable “edad”.

Esta función lo que hace es agrupar los individuos del data frame según la o las variables que se indican en by –segundo argumento– y calcula para cada uno de ellos la función especificada en FUN –tercer argumento– para la variable o las variables indicadas en x –primer argumento–. Devuelve como resultado un objeto en la forma más conveniente según sea el objeto x.

Genéricamente tiene esta forma:

```
aggregate(x, by, FUN)
```

- x: matriz, data frame, serie de tiempo[11]
- by: lista de factores para agrupar los elementos de x
- FUN: la función a aplicar

Función[12]	Argumentos	Descripción
apply()	objeto, margen, función	se aplica a una matriz o data frame, se obtiene un objeto del tipo ingresado
lapply()	objeto, función	se aplica a un data frame o lista, se obtiene una lista
tapply()	objeto, factores, función	se aplica a un vector, data frame o lista, se obtiene un array, data frame o una lista
sapply()	objeto, función	se aplica a una matriz, data frame o lista, se obtiene el resultado más simple dentro de los posibles

Cuadro 6.8: Funciones de tipo “apply”

[11] En este caso conviene utilizar otros argumentos de la función aggregate().

7

Unir data frames

7.1 Buscar casos coincidentes entre dos bases

El primer paso para unir dos bases de datos es compararlas, para ello utilizamos la función match(), la cual busca coincidencias en los elementos del primer argumento, en el ejemplo la variable "idpersona" de base2 –$base2[,1]$–, con los elementos del segundo argumento, la variable "idpersona" de base2 –$base1[,1]$– y devuelve un vector que indica las posiciones de los casos coincidentes en la variable "idpersona". En principio, tiene sentido unir la base1 con la base2 –que contiene datos relativo a la educación y a la ocupación– siempre y cuando algunas o todas las observaciones de base2 refieran a los mismos individuos que están en base1. Para el ejemplo que sigue consideramos los archivos basem1.txt y basem2.txt.

```
> base1=read.table("basem1.txt", header=T)
> base1
   idpersona sexo edad idnucleo aniosed ingresos
1   27387346    2   38       96       0      958
2   11534349    1   53       36       0      552
3   12588307    1   50       11       0      337
4   28131952    1   85       12       0      494
5   38731140    2    5       45       0       42
6   12428993    1   19        2       0      188
7   11887350    1   22       36       0      509
8   36009937    2   18       42       0      571
9   15795957    1   58       79       0       22
10  12794411    1   14       87       0      966

> base2=read.table("basem2.txt", header=T)
> base2
  idpersona anios_educ ocupacion
1  11534349          1         1
2  27387346          2         1
3  12588307          1         3
4  28131952          1         2
5  36009937          2         2
6  12428993          1         1
7  12794411          1         3
8  12222221          2         1

> comun<-match(basem2[,1], basem1[,1], nomatch = NA_integer_, incomparables = FALSE)
> comun
[1]  2  1  3  4  8  6 10 NA
```

Así el objeto comun indica los individuos de base2 que están en base1, de acuerdo al identificador,

en este caso la variable “idpersona”. Para obtener los elementos de base1 que están en base2 debemos intercambiar el orden de los objetos en la función match(). El valor NA, indica que el último elemento de base2 no está en base1, a su vez, el valor 4 indica que el primer elemento de base2, corresponde al elemento 4 de base1.

Genéricamente la función tiene la siguiente expresión:

```
match(x, table, nomatch = NA_integer_, incomparables = NULL)
```

Los argumentos de la función match() indican lo siguiente:

- x: los valores que se buscan matchear
- table: los valores que serán comparados con los de x
- nomatch: el valor devuelto cuando no hay coincidencias en el identificador
- incomparables: un vector de los valores que no se encontraron coincidencias.

7.2 Ordenar data frames

Cuando las bases que se utilizan para hacer un merge –unir las bases– son grandes, conviene antes de realizar el merge, ordenar cada base por la variable que servirá de identificador para unirlas; esto hará más rápido la operación. En este caso ambas bases son chicas pero para acostumbrarnos vamos a ordenar cada base por la variable [U+0093]idnucleo[U+0093] de forma creciente:

```
> orden=order(base1$idpersona,decreasing=FALSE)
> orden
 [1]  2  7  6  3 10  9  1  4  8  5

#alternativamente:
# orden=order(base1[,1],decreasing=FALSE)
```

La aplicación de la función order(), al objeto base1 de acuerdo a la variable “idpersona”, devuelve un vector con los números de fila de la base1 ordenada en forma creciente, esto último se indica haciendo que el argumento decreasing sea FALSE.

Por lo tanto, para ordenar la base1 según la variable “idpersona” en forma creciente, debemos acceder a las filas de base1 de acuerdo a los índices de este vector que generamos.

```
> base1=base1[orden,]
> base1

   idpersona sexo edad idnucleo aniosed ingresos
2   11534349    1   53       36       0      552
7   11887350    1   22       36       0      509
6   12428993    1   19        2       0      188
3   12588307    1   50       11       0      337
10  12794411    1   14       87       0      966
9   15795957    1   58       79       0       22
1   27387346    2   38       96       0      958
```

```
4   28131952    1   85       12       0      494
8   36009937    2   18       42       0      571
5   38731140    2    5       45       0       42
```

Esto da como resultado una ordenación de las filas del objeto base1 de acuerdo al valor de la variable “idpersona” puestos en orden creciente, de manera que cuando indicamos entre paréntesis rectos, el objeto orden, estamos simplemente estableciendo un orden para las filas del objeto base1. Como no especificamos ninguna columna estamos incluyendo todas.

Esto mismo podríamos haberlo hecho en un sólo paso:

```
>base1=base1[order(base1$idpersona,decreasing=FALSE),]
```

Hacemos lo mismo para el objeto base2.

```
>base2=base2[order(base2$idpersona,decreasing=FALSE),]
```

7.3 Unir data-frames

Ahora que hemos ordenado ambas bases de datos y sabemos que hay elementos comunes entre ambas, tiene sentido unirlas a través de ese identificador, para ello utilizamos la función merge():

```
merge(x, y, by = intersect(names(x), names(y)),
      by.x = by, by.y = by, all = FALSE,
      all.x = all, all.y = all,
      sort = TRUE, suffixes = c(".x",".y"), ...)
```

Es importante entender que significan sus argumentos para realizar la operación buscada.

- x, y: son data frames u objetos que serán colapsados en uno solo.
- by, by.x, by.y: especifica las columnas comunes por las cuales se hará el merge. Si la variable en cuestión tiene el mismo nombre en ambas bases, este nombre se especifica en by. Si por el contrario, tienen nombre diferentes, debe especificarse en by.x el nombre de la variable para la base x y en by.y el nombre que corresponde en la base y.
- all: argumento lógico, si es TRUE la nueva base tendrá las observaciones de los 2 objetos originales, tanto las comunes como las que no lo son. Por defecto, este argumento es FALSE, por lo tanto, la nueva base sólo contiene las observaciones comunes a ambas bases.
- all.x: argumento lógico, si es TRUE entonces el nuevo objeto contendrá todas las observaciones de x.
- all.y: argumento lógico, si es TRUE entonces el nuevo objeto contendrá todas las observaciones de y.
- sort: argumento lógico, el objeto resultante puede ser ordenado de acuerdo a las columnas indicadas en by
- suffixes: agrega un identificador a aquellas variables que tienen el mismo nombre en ambas bases de manera que luego de unirlas se pueda distinguir entre unas y otras.

Antes de proceder a unir las bases de datos es necesario analizar si cada una de las bases contiene casos duplicados y si es correcto que los tenga, pues si existen observaciones duplicadas en una de las bases, el resultado de la función merge() será diferente a que éstas no estuvieran. En el próximo capítulo indicaremos cómo analizar la duplicación de observaciones, pero en este ejemplo, sabemos por construcción de las bases que no tienen duplicados según la variable "idpersona". Ahora veamos un ejemplo de la utilización de esta función.

Por defecto la unión se hace sobre todas las columnas comunes presentes en ambas bases pero puede restringirse a un sub conjunto de columnas usando el argumento by; esto es lo que haremos.

Unimos las bases por la variable "idpersona", y nos quedamos con todas las observaciones –comunes y no–

```
> bases<-merge(base1.sindup,base2,all=TRUE,by="idpersona")
> bases
```

```
   idpersona sexo edad idnucleo aniosed ingresos anios_educ ocupacion
1   11534349    1   53       36       0      552          1         1
2   11887350    1   22       36       0      509         NA        NA
3   12222221   NA   NA       NA      NA       NA          2         1
4   12428993    1   19        2       0      188          1         1
5   12588307    1   50       11       0      337          1         3
6   12794411    1   14       87       0      966          1         3
7   15795957    1   58       79       0       22         NA        NA
8   27387346    2   38       96       0      958          2         1
9   28131952    1   85       12       0      494          1         2
10  36009937    2   18       42       0      571          2         2
11  38731140    2    5       45       0       42         NA        NA
```

También podemos quedarnos sólo con las observaciones comunes a ambas bases, es decir aquellas que tienen datos en todas las variables de la nueva base; observe que este nuevo objeto no tienen ningún elemento NA. Esto se logra no indicando ningún valor a los argumentos all, all.x y all.y, o bien, solamente indicando el valor FALSE al argumento all.

```
> nueva.bases<-merge(base1,base2,all=FALSE,by="idpersona")
> nueva.bases
  idpersona sexo edad idnucleo aniosed ingresos anios_educ ocupacion
1  11534349    1   53       36       0      552          1         1
2  12428993    1   19        2       0      188          1         1
3  12588307    1   50       11       0      337          1         3
4  12794411    1   14       87       0      966          1         3
5  27387346    2   38       96       0      958          2         1
6  28131952    1   85       12       0      494          1         2
7  36009937    2   18       42       0      571          2         2
```

Alternativamente, podemos generar un nuevo objeto que contenga los datos de las bases unificadas pero solamente de los individuos de la base1, para ello hacemos que el argumento all.x sea TRUE:

```
> nueva.base1<-merge(base1,base2,all.x=TRUE,by="idpersona")
> nueva.base1
   idpersona sexo edad idnucleo aniosed ingresos anios_educ ocupacion
```

```
1   11534349    1   53       36       0       552          1          1
2   11887350    1   22       36       0       509         NA         NA
3   12428993    1   19        2       0       188          1          1
4   12588307    1   50       11       0       337          1          3
5   12794411    1   14       87       0       966          1          3
6   15795957    1   58       79       0        22         NA         NA
7   27387346    2   38       96       0       958          2          1
8   28131952    1   85       12       0       494          1          2
9   36009937    2   18       42       0       571          2          2
10  38731140    2    5       45       0        42         NA         NA
```

A su vez, podemos obtener un nuevo objeto que contenga los datos de la base unificada pero que contenga ahora únicamente los individuos de la base2, para ello hacemos que el argumento all.y sea TRUE:

```
> nueva.base2<-merge(base1,base2,all.y=TRUE,by="idpersona")
> nueva.base2
  idpersona sexo edad idnucleo aniosed ingresos anios_educ ocupacion
1  11534349    1   53       36       0      552          1          1
2  12222221   NA   NA       NA      NA       NA          2          1
3  12428993    1   19        2       0      188          1          1
4  12588307    1   50       11       0      337          1          3
5  12794411    1   14       87       0      966          1          3
6  27387346    2   38       96       0      958          2          1
7  28131952    1   85       12       0      494          1          2
8  36009937    2   18       42       0      571          2          2
```

Para quedarnos solamente con los casos de la base1 que no están en la base2 podemos generar un nuevo objeto que sólo incluya a los individuos que tienen dato en “sexo” –variable exclusiva del objeto base1– y no tienen dato en “anios_educ” –variable exclusiva del objeto base2–.

```
> base1.1<-subset(bases,!is.na(sexo)&is.na(anios_educ))
> base1.1

   idpersona sexo edad idnucleo aniosed ingresos anios_educ ocupacion
2   11887350    1   22       36       0      509         NA         NA
7   15795957    1   58       79       0       22         NA         NA
11  38731140    2    5       45       0       42         NA         NA
```

Como se desprende de lo anterior, una forma de obtener un nuevo objeto que sólo contenga los casos de la base2 que no están en la base1, es estableciendo una condición que incluya a los individuos que tienen dato en “anios_educ” –variable exclusiva del objeto base2– y no lo tienen en la variable “sexo” –variable exclusiva del objeto base1–.

7.3.1. Combinar bases

Muchas veces necesitamos combinar bases que están ordenadas pero contienen distinto número de filas, con la función cbind, esto no se puede hacer, pero sin necesidad de hacer un merge podemos usar la función cbindX() del paquete gdata.

En otros veces no necesitamos hacer un merge pues sólo pretendemos seleccionar a los individuos de base1 que están en base2, pero no trabajar con las variables de base2.

```
> en1=base1$idpersona %in% base2$idpersona
> base1[en1,]
   idpersona sexo edad idnucleo aniosed ingresos
1   27387346    2   38       96       0      958
2   11534349    1   53       36       0      552
3   12588307    1   50       11       0      337
4   28131952    1   85       12       0      494
6   12428993    1   19        2       0      188
8   36009937    2   18       42       0      571
10  12794411    1   14       87       0      966
```

Función	Argumentos	Descripción
cbind()	vector y/o matriz y/o dataframe	concatena por columnas
rbind()	vector y/o matriz y/o dataframe	concatena por filas
subset()	matriz o dataframe, condición, selección	selecciona sub base
sort()	vector, orden	devuelve el vector ordenado
order()	vector o dataframe, variable/s de ordenación	devuelve el objeto ordenado de acuerdo a una o varias variables
match()	objeto, objeto, variable de coincidencia	devuelve los casos que coinciden
merge()	objeto, objeto, variable de unión	une los objetos
duplicated()	objeto	devuelve vector lógico

Cuadro 7.1: Manipulación de objetos

8

Manipular una base de datos

8.1 Casos duplicados

El trabajo con observaciones duplicadas en las bases de datos es bastante frecuente tanto con bases administrativas como con encuestas, pudiéndose tratar de una duplicación de individuos por error o porque a los individuos de un mismo hogar se los identifica con un indicador común, así la variable que identifica al hogar está duplicada, triplicada, etc. según cuántos integrantes tenga cada hogar respectivamente.

Para encontrar los casos duplicados en función de una o varias variables usamos la función duplicated().

```
duplicated(x, incomparables = FALSE, MARGIN = 1, ...)
```

Veamos los argumentos de la función duplicated(), la cual es muy flexible como se desprende de analizar su estructura:

- x: es un vector o un data frame
- incomparables: es un vector de valores que no será comparado. Por defecto toma el valor FALSE, lo cual significa que todos los valores serán comparados.
- MARGIN: si MARGIN=1 busca duplicados por filas, mientras que si MARGIN=2, los busca por columnas.

En este caso vamos a trabajar con el archivo “buenpagador.csv”, el cual contiene los datos de compradores de cierto artículo y detalla las cuotas que han pagado. Como un mismo cliente puede haber pagado más de una cuota, la base de datos puede contar con casos duplicados, por lo tanto buscaremos la existencia de casos duplicados por la variable ïdpersona”que es la que identifica al cliente:

```
> buenpagador=read.csv("buenpagador.csv", header=T, sep=";", dec=",")
> buenpagador
   idpersona sexo edad cuota  monto
1   11534349    1   53     1 775.80
2   11534349    1   53     2 775.80
3   15795957    1   58     1 330.00
4   38731140    2   33     1 892.77
5   27387346    2   38     1 551.00
6   12588307    1   50     1 122.00
7   28131952    1   85     1 795.00
```

```
8   27387346    2   38     2 551.00
9   12428993    1   19     1 330.00
10  12794411    1   14     1 750.00
11  11534349    1   53     3 775.80

> duplicados.idpersona=duplicated(buenpagador$idpersona)
> duplicados.idpersona
 [1] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE

> table(duplicados.idpersona)
duplicados.idpersona
FALSE  TRUE
    8     3
```

El resultado de utilizar la función duplicated() es un vector lógico, dónde cada elemento indica si la observación se encuentra duplicada –TRUE– o no –FALSE–. Veamos detenidamente lo que hace esta función.

Primero verifica si el elemento de la fila 1 correspondiente a la variable "idpersona", está duplicado con algún elemento de las filas anteriores. En este caso resulta claro que no hay ninguna fila antes de la primera, por lo tanto, el primer elemento sobre el que opera la función duplicated() siempre da como resultado FALSE.

El segundo paso es verificar si el elemento de la fila 2 correspondiente a la variable "idpersona" se encuentra duplicado, para ello se compara este elemento con los correspondientes de la variable "idpersona" de las filas anteriores. En este caso sólo compara el elemento 1 de la fila 2 –11534539– con el elemento 1 de la fila 1 –11534539–.Esto da como resultado TRUE –segundo elemento del objeto duplicados.idpersona–, esto implica que el individuo 11534539 se encuentra duplicado.

El tercer paso es comparar el elemento de la fila 3 correspondiente a la variable "idpersona" –1579557– con los elementos correspondientes a la variable "idpersona" de las filas anteriores. Como 1579557 es diferente de 11534349 y 11534349, el resultado de la función duplicated() es FALSE –tercer elemento del objeto duplicados.idpersona–.

Los pasos siguientes de la función duplicated() son análogos a los que ya mencionamos, ésta toma un elemento de la variable "idpersona" y lo compara con cada uno de los anteriores: si este elemento es igual a algún otro entonces la función devuelve un TRUE, que indica que ese elemento se encuentra duplicado, de lo contrario, si este elemento es diferente a todos los anteriores la función devuelve un FALSE.

En total obtenemos 8 FALSE y 3 TRUE, esto no implica que hay 3 elementos duplicados, ya que un mismo elemento puede repetirse esas 3 veces y en ese caso, el resultado de esta función también serían 3 TRUE. Si observamos la base [1] podemos ver que sólo hay 2 valores de "idpersona" duplicados: el 11534539 –en 2 oportunidades– y el 27387346 –en una oportunidad–. Con esto queremos hacer énfasis en que es clave, hacer un análisis detallado de los casos duplicados para saber con certeza cuáles individuos se repiten y sí es correcto que así sea.

Habitualmente, los análisis que se hacen con una base requieren no trabajar con casos duplicados, pero saber con cual de las observaciones duplicadas conviene quedarse es un problema cuya solución es necesariamente adhoc a cada situación. En este caso, vamos a armar una nueva base dónde no haya individuos duplicados, de manera que cada fila corresponda a un individuo diferente,

[1] Esto podemos hacerlo porque es una base con pocos casos.

pero donde cada individuo tenga un registro de cada una de las cuotas que pagó. Para ello, primero vamos a anexar el objeto duplicados.idpersona a la base original usando la función cbind().

```
> buenpagador1=cbind(buenpagador, duplicados.idpersona)
```

Luego, utilizamos la función subset para sólo retener de la base las observaciones cuya variable "duplicados.idpersona" es FALSE y generamos otro objeto buenpagador2 que contiene los restantes casos.

```
> buenpagador1.sindup=subset(buenpagador1,duplicados.idpersona==FALSE)
> buenpagador2=subset(buenpagador1,duplicados.idpersona==TRUE)
> dim(buenpagador1.sindup)
[1] 8 6
> dim(buenpagador2)
[1] 3 6
```

Ahora seguiremos trabajando con el objeto buenpagador2, revisaremos que esta base contenga duplicados, si los tuviera estos serán guardados en otro objeto; el cometido final es generar tantas nuevas bases como sean necesarias para que en ellas no se encuentren casos duplicados –siempre para la variable "idpersona"– de manera de luego unir esas bases y obtener una sin duplicados.

```
duplicados.idpersona2=duplicated(buenpagador2$idpersona)
> table(duplicados.idpersona2)
duplicados.idpersona2
FALSE  TRUE
    2     1
```

No es sorprendente que volvamos a encontra un individuo duplicado ya que miramos la base y detectamos que el idpersona 11534349 se encontraba 3 veces en la base original. Si se tratara de una base con muchos más casos la inspección visual no hubiera podido hacerse, llegado a este punto aún no sabríamos qué elementos se encuentran duplicados. Procedemos a crear un objeto que contenga el objeto buenpagador2 y duplicados.idpersona2.

```
> buenpagador2=cbind(buenpagador2, duplicados.idpersona2)
```

Nuevamente, creamos un objeto que contenga los casos en que duplicados.idpersona resultó FALSE, y otro objeto que contenga los TRUE.

```
> buenpagador2.sindup=subset(buenpagador2, duplicados.idpersona2==FALSE)
> buenpagador3=subset(buenpagador2, duplicados.idpersona2==TRUE)

> buenpagador2.sindup
  idpersona sexo edad cuota monto duplicados.idpersona duplicados.idpersona2
2  11534349    1   53     2 775.8                 TRUE                  FALSE
8  27387346    2   38     2 551.0                 TRUE                  FALSE

> buenpagador3
   idpersona sexo edad cuota monto duplicados.idpersona duplicados.idpersona2
11  11534349    1   53     3 775.8                 TRUE                   TRUE
```

A esta altura tenemos 3 objetos finales: buenpagador1.sindup, buenpagador2.sindup y buenpagador3, este último al tener un sólo caso, no tiene duplicados por definición. Por lo tanto, estamos en condiciones de unir estas bases de datos, en primer lugar haremos un merge entre buenpagador2.sindup y buenpagador3 y al resultado de esto, le agregaremos a la base buenpagador1.sindup.

```
> buenpagador.aux=merge(buenpagador2.sindup, buenpagador3, by="idpersona", all=T)
> buenpagador.def=merge(buenpagador1.sindup, buenpagador.aux, by="idpersona", all=T)
```

Finalmente, de la base de datos final nos quedamos con algunas variables, todas las de la base original y las correspondientes a la cuota y el monto de las nuevas bases.

```
> buenpagador.def=buenpagador.def[c(1:5,9:10,15:16)]
> buenpagador.def
  idpersona sexo edad cuota  monto cuota.x monto.x cuota.y monto.y
1  11534349    1   53     1 775.80       2   775.8       3   775.8
2  12428993    1   19     1 330.00      NA      NA      NA      NA
3  12588307    1   50     1 122.00      NA      NA      NA      NA
4  12794411    1   14     1 750.00      NA      NA      NA      NA
5  15795957    1   58     1 330.00      NA      NA      NA      NA
6  27387346    2   38     1 551.00       2   551.0      NA      NA
7  28131952    1   85     1 795.00      NA      NA      NA      NA
8  38731140    2   33     1 892.77      NA      NA      NA      NA
```

9

Datos faltantes

Además de los casos duplicados, otra cuestión a tener en cuenta antes de empezar a analizar resultados de una base de datos es revisar la existencia de valores faltantes. En R los valores perdidos se representan con el símbolo *NA* que siginfica que el valor está no disponible[1]. Conviene no confundir estos valores con los valores imposibles que se representan con el símbolo *NaN* que significa que no es un número, por ejemplo el resultado de dividir un número entre 0 da como resultado un valor imposible[2].

```
> x <- c(1,5,9,NA,2)
> x
[1]  1  5  9 NA  2
```

El comportamiento por defecto de muchas funciones es rechazar datos que contienen datos perdidos.

```
> mean(x)
[1] NA
```

Si calculamos la media aritmética nos dará como resultado un valor missing

```
> x <- c(1,2,NA,3)
> mean(x)
[1] NA
```

Por tanto lo adecuado es tratar por separado a los valores perdidos o descartarlos del análisis. Usando el argumento na.rm igual TRUE, se calcula la media del objeto sin considerar los valores perdidos.

```
> mean(x, na.rm=T)
[1] 4.25
```

Tampoco es adecuado usar valores perdidos al realizar comparaciones lógicas. Por ejemplo, para comparar si 2 números son iguales, y uno de ellos es missing, el resultado será *NA*.

```
> x
[1]  1  5  9 NA  2

> x == 5
[1] FALSE  TRUE FALSE    NA FALSE
```

La forma de identificar un *NA* no es comparar un número o vector con un *NA*, pues el resultado también será un *NA*

[1] Su nombre viene por el término inglés Not Avaible

[2] Not a number

```
> x == NA

[1] NA NA NA NA NA
```

Para verificar la existencia de valores missing se utiliza la función is.na() que retorna el valor lógico TRUE para cada valor perdido existente:

```
> y <- c(1,2,3,NA)
> is.na(y)
[1] FALSE FALSE FALSE  TRUE
```

Una manera sencilla de omitir los valores missings es a través de la función na.omit(). Mediante la función attr() se puede acceder a la posición del valor perdido.

```
> x
[1]  1  5  9 NA  2

> na.omit(x)
[1] 1 5 9 2
attr(,"na.action")
[1] 4
attr(,"class")
[1] "omit"
```

Lo anterior es válido tanto para un vector como para una data.frame, así si utlizamos la función na.omit() el resultado es que se descartan las filas con valores perdidos. Retornemos al ejemplo del objeto bases.

```
> bases.sin.NA = na.omit(bases)
> bases.sin.NA
  idpersona sexo edad idnucleo ingresos duplicados.idpersona anios_educ ocupacion
1  11534349    1   53       36      552                FALSE          1         1
3  12428993    1   19        2      188                FALSE          1         1
4  12588307    1   50       11      337                FALSE          1         3
5  12794411    1   14       87      966                FALSE          1         3
7  27387346    2   38       96      958                FALSE          2         1
8  28131952    1   85       12      494                FALSE          1         2
9  36009937    2   18       42      571                FALSE          2         2
```

En este caso usando la función na.omit(), omitimos aquellos casos donde almenos una de sus variable contiene un valor NA. Esto da como resultado un nuevo objeto sin los individuos 2, 6 y 10 del objeto bases.

Alternativamente es posible indicar cuales son los casos completos de la base usando la función complete.cases() que retorna un vector lógico indicando que casos están completos:

```
bases[complete.cases(bases),]
```

Por el contrario, cuando lo que pretendemos es quedarnos con los casos que tienen algún dato faltante para trabajar aparte con ellos nos quedamos con los casos que no están completos:

```
bases[!complete.cases(bases),]
```

10

Encuesta Continua de Hogares

La Encuesta Continua de Hogares -ECH- es realizada a lo largo de cada año de forma continua por el Instituto Nacional de Estadística –INE–. Tiene como cometido relevar información socio-económica de la población residente en hogares particulares, actualmente de todo Uruguay. El área de cobertura ha ido cambiando, pero desde 2006 hasta ahora releva todo el territorio nacional; antes estaban excluídas localidades rurales y urbanas pequeñas.

Como dijimos, las unidades de análisis de la encuesta son los hogares particulares y las personas que residen en dichos hogares. Por tanto, se excluyen miembros de hogares colectivos o residentes particulares en hogares colectivos como hospitales, conventos,etc. Mientras que las unidades de muestreo de la ECH 2009 las constituyen los 59 estratos que subdividen todo el territorio nacional: 4 estratos en Montevideo, un estrato para la zona periférica a la capital, y 3 estratos en cada uno de los departamentos del Interior. La muestra se selecciona en 3 etapas: localidad, zona censal y vivienda particular de forma independiente mes a mes y año a año[1].

La ECH tiene por objetivo principal estimar la tasa de actividad económica, de desocupación y el ingreso total medio de los hogares en los estratos mencionados.
Así, los productos más frecuentes resultados de un análisis de la ECH son:

- Indicadores sobre el Mercado de Trabajo: Tasa de Empleo, Tasa de Actividad y Tasa de Desempleo.
- Indicadores socio-económicos: Ingresos de los Hogares e Indicadores sobre Distribución del Ingreso.

En las secciones siguientes ejemplificaremos usando la ECH del año 2009. Si bien vamos a trabajar específicamente con la ECH 2009 hay algunos aspectos que son relevantes para el trabajo con las ECH de años anteriores:

- Recién desde 2006 se releva para el total del país.
- Desde el año 1991 al 1997 las Encuestas eran autoponderadas, a partir de 1998 pasan a tener ponderadores –las variables *pesoano*, *pesosem*, *pesotri* –.
- Hasta el 2009 el identificador del hogar era la variable *correlat* no obstante, para algunos años no alcanzaba con dicha variable sino que debía generarse un identificador que comprendiera *correlat* y *departamento*.

[1] Para un mayor detalle, vea el anexo

Procedemos a cargar la ECH 2009 que obtuvimos descargando el archivo correspondiente en la página del INE: http://www.ine.gub.uy/microdatos/ech2009/ech2009%20spss.zip Este archivo contiene, a su vez, 3 archivos, uno correspondiente a los hogares, otro a las personas y un último que corresponde a la fusión de los 2 anteriores. Por motivos didácticos, sólo utilizaremos los 2 primeros. Cuando cargamos una base que contiene tildes y ñ, como es el caso de la ECH, debemos hacerlo diferente. Cargamos la base de personas de la ECH con las etiquetas de las variables factor. Sin embargo, las etiquetas que tienen tilde o ñ no las lee bien.

```
library(foreign)
p2009=read.spss("PER_2009.sav",to.data.frame=T)
```

Realizamos un loop para cambiar la codificación de las variables de tipo factor, de manera que lea las etiquetas correctamente. Dado que vamos a convertir la codificación de las variables, el bloque de sintaxis (lo que aparece entre llaves) que vamos a repetir con el for lo hacemos por columnas

```
for (i in 1:ncol(p2009))
{
  if(is.factor(p2009[,i])==T)
    p2009[,i]=iconv(p2009[,i],"ISO_8859-1","UTF-8")
}
```

Para conocer las dimensiones de cada objeto procedemos como en secciones anteriores usando la función dim()[2]:

```
> dim(p2010)
[1] 132599    354

> dim(hog2010)
[1] 46936   128
```

Por lo tanto, la base de personas contiene 132.599 personas y 354 variables, mientras que la base de hogares contiene 46.936 hogares y 128 variables. Debido a la cantidad de variables en cuestión conviene leer detenidamente el diccionario de variables que puede descargarse desde: http://www.ine.gub.uy/microdatos/Microdatos%20ECH%202010/DICCIONARIO%20ECH%202010.xls.

Algunas de las variables que utilizaremos en esta sección son:

- numero: número de indentificación del hogares. Se encuentra en base de hogares y de personas
- dpto: código del departamento del 1 al 19, comenzando en Montevideo y luego alfabéticamente. Se encuentra en base de hogares y de personas
- pesoano: expansor del año. Se encuentra en base de hogares y de personas
- HT11: ingreso total del hogar con valor locativo sin incluir el servicio doméstico –en pesos uruguayos–. Se encuentra en la base de hogares
- ht19: cantidad de personas del hogar sin incluir al servicio doméstico. Se encuentra en la base de hogares

[2] Son datos de 2009

[2] Note que los nombres de las variables exclusivas de la base de hogares comienzan con la letra h o H.

Resulta importante agregar a la base de personas algunas variables de la base de hogares, en particular, la cantidad de integrantes del hogar –"ht19"–y el ingreso de dicho hogar –"HT11". Por lo tanto, a la base personas le agregaremos usando la función merge() estas variables, lo cual implica de antemano reducir la base de hogares a 3 variables; si usáramos la base completa haríamos un consumo enorme de recursos en memoria:

```
> var.hog=subset(hog2009,select=c(numero,ht19,HT11))
> p2009=merge(p2009,var.hog,by="numero")
```

Note que la variable que identifica a los hogares es *numero* y ésta es la variable común entre las 2 bases a través de la cual se hace el merge. A su vez, note que hemos reemplazado el objeto p2009 original por uno al cual se le agregan las variables *ht19* y *HT11*. También podríamos haber elegido un nombre diferente para este nuevo objeto, por ejemplo, p2009bis, si este fuera el caso convendría eliminar de la memoria el objeto p2009, de manera de liberar espacio en memoria para los cálculos que haremos más adelante. Hay que tener en cuenta que estamos trabajando con bases de un gran tamaño y que podemos quitar de la memoria las que no estemos utilizando, guardarlas en formato Rdata, para luego poder volver a cargarlas en caso de que volvamos a utilizarlas más adelante, o como simple respaldo.

10.1 Calcular ingresos

La medición de los ingresos tiene un rezago de 1 mes, pues si la encuesta se realizó, por ejemplo en marzo, en este mes se preguntó por los ingresos del mes pasado, es decir que en marzo se midieron los ingresos del hogar correspondientes a febrero. A su vez, estos ingresos están expresados a precios corrientes, por lo cual es necesario llevarlos todos a una medida común. Tomamos como base diciembre de 2009 y construímos un "inflactor" en base a los datos del IPC que se encuentran en: `http://www.ine.gub.uy/banco%20de%20datos/ipc/IPC%205%20gral%20rubagsubarfa%20M.xls`. Así construímos un objeto incorporando los datos de la siguiente tabla:

Mes-Año	IPC
dic-09	282,43
nov-09	281,11
oct-09	280,95
set-09	280,98
ago-09	280,33
jul-09	276,92
jun-09	274,21
may-09	271,13
abr-09	270,03
mar-09	270,14
feb-09	268,08
ene-09	268,80
dic-08	266,69

Cuadro 10.1: Valores del IPC mensual de diciembre de 2008 a diciembre de 2009, según INE.

```
> ipc=c(282.43, 281.11, 280.95, 280.98, 280.33, 276.92, 274.21,
271.13, 270.03,270.14, 268.08, 268.80, 266.69)
```

Para construir el actualizador de cada mes, el valor del IPC base –correspondiente a diciembre de 2009– es dividido por el valor del ipc de cada uno de los meses anteriores: noviembre de 2009 a diciembre de 2008.

```
> actualizador=ipc[1]/ipc[-1]
> actualizador
 [1] 1.004696 1.005268 1.005161 1.007491 1.019897 1.029977 1.041677 1.045921
 [9] 1.045495 1.053529 1.050707 1.059020
```

Debemos relacionar cada uno de los valores del actualizador, con los meses a los cuales corresponden. Por lo tanto, generamos un data frame que contenga al vector actualizador y otro vector que indica los meses del 12 al 1.

```
> mes=12:1
> actualizador=cbind(actualizador,mes)
> actualizador=as.data.frame(actualizador)
```

Ahora estamos en condiciones de relacionar este objeto actualizador, con el objeto p2009. De manera que a cada observación de la base de personas le adicionaremos el valor correspondiente (de acuerdo al mes de la entrevista) del IPC actualizado. Para ello utilizamos la función merge():

```
> p2009.ingreso=merge(p2009, actualizador, by="mes")
```

Calculamos, el ingreso del hogar combinando la variable “HT11” con la variable “actualizador” recién agregada a la base.

```
> p2009$yestacional=p2009$HT11*p2009$actualizador
```

Ahora estamos en condiciones de calcular algunas medidas descriptivas del ingreso de los hogares, para lo cual creamos la variable “yestacional”. Así el ingreso medio de los hogares de la muestra a diciembre de 2009–note que aún no expandimos los datos– es la media de dicha variable:

```
> media.ingresoh.muestra=mean(p2009$yestacional)
> media.ingresoh.muestra
[1] 33899.83
```

Para obtener el ingreso medio por hogar a diciembre de 2009, dividimos la variable “yestacional” por la cantidad de integrantes de cada hogar –“ht19”–, esto lo guardamos en la propia base en la variable “ycapita” y luego calculamos la media de dicha variable:

```
> p2009$ycapita=p2009$yestacional/p2009$ht19
> media.ingresop.muestra=mean(p2009$ycapita)
> media.ingresop.muestra
[1] 10971.72
```

A su vez, podemos calcular este promedio por departamento utilizando la función tapply():

```
> media.ycapita.depto=tapply(p2009$ycapita,p2009$dpto,mean)
> media.ycapita.depto
     MONTEVIDEO          ARTIGAS        CANELONES      CERRO LARGO
       14192.637         6283.181         9272.188         7378.489
        COLONIA          DURAZNO           FLORES          FLORIDA
        9912.026         8142.991         9544.721         8831.643
      LAVALLEJA        MALDONADO         PAYSANDU        RIO NEGRO
        8712.539        11521.710         8378.106         8085.142
         RIVERA            ROCHA            SALTO         SAN JOSE
        7428.956         8302.184         8127.467         8818.061
        SORIANO       TACUAREMBO   TREINTA Y TRES
        8369.557         7628.701         7683.470
```

10.2 Expansión de datos

Las ECHs como muchas otras de las encuestas públicas –de Gastos e Ingresos, Intenciones de Voto, etc.– no son un relevamiento del universo a investigar sino que son "muestras representativas" de dicho universo.

Es por ello que para poder inferir datos poblaciones de los datos muestrales es imprescindible expandir los datos, es decir, generar un mecanismo para pasar de los datos muestrales a los poblacionales. El expansor se calcula en base a las probabilidades de selección de los elementos en la muestra, siendo el factor de expansión el inverso de la probabilidad de selección de los elementos.

10.2.1. Precauciones

La expansión de datos de la ECH es algo que debe hacerse con muchísima precaución, pues puede suceder que en pos de querer extraer la mayor información posible de los datos, extraemos información equívoca, que nos lleva a realizar lecturas erróneas de la realidad que pretendemos describir y/o analizar. Para evitar -o al menos minimizar- dichos errores es necesario que nos adentremos al estudio de la metodología de cada una de las encuestas con las que vayamos a trabajar. No obstante, hay algunas cuestiones que son más o menos generales a todas ellas:

1. **En las ECHs no es correcto a la hora de hacer expansiones hablar de cantidades.** Cuando hacemos esto, damos una falsa noción de exactitud y estamos desconociendo que el diseño de la encuesta no prevé calcular exactamente cantidades sino tasas.

2. **Existen niveles de desagregación de las estimaciones que no se pueden alcanzar sin incurrir en errores de estimación importantes.** Por ejemplo, si quisiéramos estimar el clima educativo poblacional por barrio obtendríamos estimaciones con un margen de error –que no podemos calcular con los datos que poseemos– que puede ser tan alto que invalide cualquier lectura que pretendamos hacer de los resultados.

3. **Al comparar datos de diferentes ECHs, hay que tener en cuenta posibles cambios metodológicos que hacen que no sean directamente comparables resultados de distintas encuestas.**
   Estos cambios obedecen a cambio en el diseño muestral, la cantidad de visitas a un mismo hogar, entre otros. A modo de ejemplo hagamos el siguiente ejercicio de imaginación:
   Supongamos que en el momento A se realiza una encuesta en la cual el número de re-visitas es igual a 2, mientras que en el momento B se realiza la misma encuesta pero con un número de re-visitas es igual a 4. Parece sensato suponer que la probabilidad de ser encontrado en cada visita es inversamente proporcional al hecho de estar ocupado – y que ocurre a la inversa si uno es inactivo o está desocupado–. Asimismo podemos suponer razonablemente que la probabilidad de ser encontrado aumenta cuanto más revisitas se hagan. Este cambio metodológico, ante una realidad "idéntica" en A y B hace por sí sólo que la tasa de actividad, la tasa de ocupación sean más altas en B que A, a la vez que, la tasa de desocupación es más baja en B que en A. Es por ello que para poder inferir que en un momento del tiempo la tasa "X" es mayor o menor no implica sólamente mirar los resultados de cada encuesta sino contemplar si hubo o no algún cambio metodológico así como también las distintos márgenes de error, etc.

[2] Una revisita se amerita hacerla cuando no se encuentra a nadie del hogar a quien realizarle la encuesta.

10.2.2. Expansión de datos con el paquete survey

Alternativamente, podemos expandir los datos utilizando el paquete "survey" y algunas funciones allí contenidas, en primer lugar utilizamos la función svydesign() para crear un objeto a partir del cual obtener las estimaciones para la población.

```
svydesign(ids, data = NULL, weights=NULL, fpc, strata)
```

- ids: Fórmula que especifica la cantidad de etapas del muestreo. Dicha fórmula debe estar precedida por el símbolo $\sim$.
- weights: Fórmula o vector que especifica los ponderadores de la muestra. Dicha fórmula debe estar precedida por el símbolo $\sim$.
- data: data frame
- fpc: factor de corrección de poblaciones finitas. Dicha fórmula debe estar precedida por el símbolo $\sim$.
- strata: variable que indica el estrato al cual pertenece la observación.Dicha fórmula debe estar precedida por el símbolo $\sim$.

Aplicamos esta función al objeto p2009, indicando que se realizó una sola etapa de muestreo –ya que no contamos con ponderadores de las primeras etapas–, y utilizamos el ponderador "pesoano":

```
> expandida=svydesign(id=~1,weights=~pesoano, data=p2009)
```

El objeto resultante de aplicar la función svydesign() es de clase survey.design[3], por lo tanto, para obtener estimaciones de las variables de la población, por ejemplo totales, medias o varianzas, se deben utilizar las funciones svytotal(), svymean() y svyvar() respectivamente.

```
svytotal(x, design, na.rm=FALSE,...)
svymean(x, design, na.rm=FALSE,...)
svyvar(x, design, na.rm=FALSE,...)
```

- x: una fórmula, vector o matriz.
- design: un objeto de clase survey.design o svyrep.design
- na.rm: si na.rm=TRUE, se quitan todos los casos con datos faltantes

Calculemos la estimación del total población para los 2 niveles de la variable "e26", indicando en la función svytotal(), la variable que queremos expandir luego del símbolo $\sim$ y como segundo argumento el objeto de clase survey.design que contiene las características del diseño de muestreo.

```
> total.sexo.poblacion=svytotal(~e26, expandida)
> total.sexo.poblacion
              total     SE
e26 HOMBRE  1431990 5081.4
e26 MUJER   1591666 5128.4
```

[3]Es decir, no es una base de datos con las variables expandidas sino un objeto que contiene como información las características del diseño de muestreo.

La tabla anterior, indica para cada nivel de la variable “e26”, la estimación del total poblacional y la estimación del desvío estandar del estimador. Como en la ECH sólo contamos con el ponderador de la última etapa de muestreo esta estimación del desvío estándar no es la correcta pues es necesario contar con los ponderadores de las primeras etapas de la muestra. Además podemos mejorar esta estimación incluyendo el estrato y el factor de corrección de poblaciones finitas en la función svydesign()

Para obtener sólo las estimaciones usamos la función coef(), mientras que para obtener sólo las estimaciones de los desvíos usamos la función SE().

```
> coef(total.sexo.poblacion)
e26 HOMBRE   e26 MUJER
    1431990     1591666

> SE(total.sexo.poblacion)
e26 HOMBRE   e26 MUJER
   5081.369    5128.389
```

A su vez, para obtener una estimación de la media poblacional de la variable edad –“e27”–, usamos la función svymean() indicando sus argumentos como en el caso anterior, pero ahora la variable en cuestión es “e27”:

```
> total.edad.poblacion=svymean(~e27, expandida)
> total.edad.poblacion
      mean     SE
e27 36.374 0.0717
```

Para mejorar la estimación de la varianza de los estimadores, incorporamos información: la variable estrato y el tamaño del estrato en la población que nos permitirá calcular el factor de corrección de poblaciones finitas: $fpc = 1 - n_h/N_h$. Donde n_h es el tamaño de muestra en el estrato h y N_h es el tamaño de la población en ese estrato.

```
Nh=tapply(p2009$pesoano,p2009$estrato,sum)
Nh=as.matrix(Nh)
p2009=merge(p2009,Nh,by.x="estrato",by.y="row.names")
names(p2009.1)[357]="Nh"
expandida.mejor=svydesign(id=~1,strata=~estrato,
                  weights=~pesoano, fpc=~Nh,data=p2009)
```

Así por ejemplo, si recalculamos la estimación del total poblacional por sexo, encontramos que la estimación puntual no se ve afectada (ya que no depende de la información incorporada) pero se reduce la estimación de la varianza de estos estimadores.

```
> sexo.pob.m=svytotal(~e26, expandida.mejor)
> sexo.pob.m
            total     SE
e26hombre 1431990 4952.4
e26mujer  1591666 4997.0
```

10.2.3. Estimación por subgrupos

Para expandir los datos de una variable en función de otra utilizamos la función svyby(), que genéricamente es:

```
svyby(formula, by ,design,vartype, FUN)
```

- formula: es la variable a aplicarle la función de tipo svy.
- by: es la variable de referencia que debe ser una variable categórica
- design: es el objeto con la información del diseño de muestreo y la muestra
- FUN: la función a aplicar de tipo svy, puede ser por ejemplo svytotal(), svymean, etc.
- vartype: devuelve la estimación del desvío estándar, el intervalo de confianza, la estimación del coeficiente de variación y/o la estimación de la varianza.

Para estimar la media poblacional de la variable edad –"e27"– según sexo –"e26", utilizamos esta función e indicamos en el argumento vartype que queremos obtener el intervalo de confianza aproximado, la estimación del desvío estándar y el coeficiente de variación.

```
> edad.sexo.pob=svyby(~e27,~e26, expandida.mejor,
                      vartype=c("ci","se","cv"), svymean)
> edad.sexo.pob
    e26      e27     se.e27     ci_l     ci_u      cv.e27
1  hombre 34.67274 0.09965531 34.47741 34.86806 0.002874169
2   mujer 37.90387 0.09771852 37.71235 38.09540 0.002578062
```

Así, obtenemos un intervalo de confianza aproximado para la media de edad de los hombres cuyo extremo izquierdo –ci-l– es 34.47741, mientras que el extremo derecho –ci-u– es 34.86806. Mientras que el coeficiente de variación estimado resulta de hacer el cociente entre la estimación del desvío estándar y la estimación de la media: (0.9965531/34.67274)= 0.02874169.

10.2.4. Estimación por intervalo: Intervalo de confianza

La interpretación de un intervalo de confianza no refiere a un intervalo particular sino a todos los posibles intervalos que se pueden construir para cierto nivel de confianza y tamaño de muestra. Una correcta interpretación de un intervalo de confianza al 5 % para la media poblacional μ sería decir: que si se eligieran muchas muestras, se puede esperar que el 95 % de los intervalos aleatorios generados por esas respectivas muestras contengan a la media poblacional.

Para calcular un intervalo de confianza usamos la función confint() que tiene la siguiente forma:

```
confint(object,  parm, level = 0.95)
```

Donde sus argumentos son:

- parm: especificación del parámetro
- level: el nivel de confianza. Por defecto es 95 %
- Se obtiene una matriz o vector cuyas columnas indican el extremo izquierdo y derecho del intervalo de confianza para la estimación puntual

Aplicamos esta función para calcular un intervalo de confianza aproximado para la media de la variable edad, usando el objeto expandida y luego el objeto expandida.mejor. Si comparamos la amplitud de ambos intervalos, se observa que el segundo intervalo es de menor amplitud que el primero ya que la estimación de la varianza del estimador es menor al incorporar información adicional.

```
> confint(svymean(~e27, expandida))
       2.5 %   97.5 %
e27 36.23303 36.51421

> confint(svymean(~e27, expandida.mejor))
      2.5 %   97.5 %
e27 36.2368 36.51044
```

Lo mismo ocurre si comparamos los intervalos del total de hombres (mujeres) según se utilizó en su construcción el objeto expandida o el objeto expandida.mejor.

```
> confint(svytotal(~e26, expandida))
       2.5 %  97.5 %
e26 1 1422031 1441949
e26 2 1581615 1601717


> confint(svytotal(~e26, expandida.mejor))
       2.5 %  97.5 %
e26 1 1422284 1441696
e26 2 1581872 1601460
```

11

Gráficos descriptivos

Si bien R está orientado a objetos, el resultado de una función gráfica no puede ser asignado a un objeto –salvo algunas excepciones – ya que es enviado a un dispositivo gráfico, es decir a una ventana o a un archivo. Las funciones gráficas de R las podemos dividir en 3 tipos:

- de alto nivel: son funciones que crean un nuevo gráfico, con ejes, etiquetas, títulos y otros elementos. Veremos sólo algunas, en particular, plot(), barplot(), hist(), pie().
- de bajo nivel: son funciones que añaden información a un gráfico existente, tales como: puntos adicionales y líneas.
- interactivas: son funciones que permiten interactuar al usuario con un gráfico, añadiendo o eliminando información, utilizando el ratón.

Otra característica que distingue a las funciones de alto nivel es que automáticamente – salvo que se indique lo contario – abren la ventana gráfica. Mientras que si queremos ejecutar una función gráfica de bajo nivel sin usar previamente una función gráfica de alto nivel, debemos abrir el dispositivo gráfico usando la función X11() en GNU/Linux, con windows() en Windows y con macintsh() bajo Mac.

11.1 Argumentos gráficos generales

Antes de realizar algunos gráficos vamos a enumerar los argumentos y parámetros gráficos que permiten adaptar la creación de nuestros gráficos a nuestro gusto.

Los argumentos generales que se pueden agregar a un gráfico de alto nivel son los siguientes:

Argumento	Descripción
main	título superior
new	adiciona el gráfico sobre el anterior
sub	título inferior
type	tipo de línea y/0 punto
xlab	etiqueta eje de abcisas
ylab	etiqueta eje de ordenadas
xlim	límites del eje de las abcisas
ylim	límites del eje de ordenadas
mtext	texto a los márgenes de los ejes
add=TRUE	el nuevo gráfico se superpondrá al actual
axes=FALSE	suprime la generación de ejes
log	valores del eje x en logarítmos

Cuadro 11.1: Información adicional

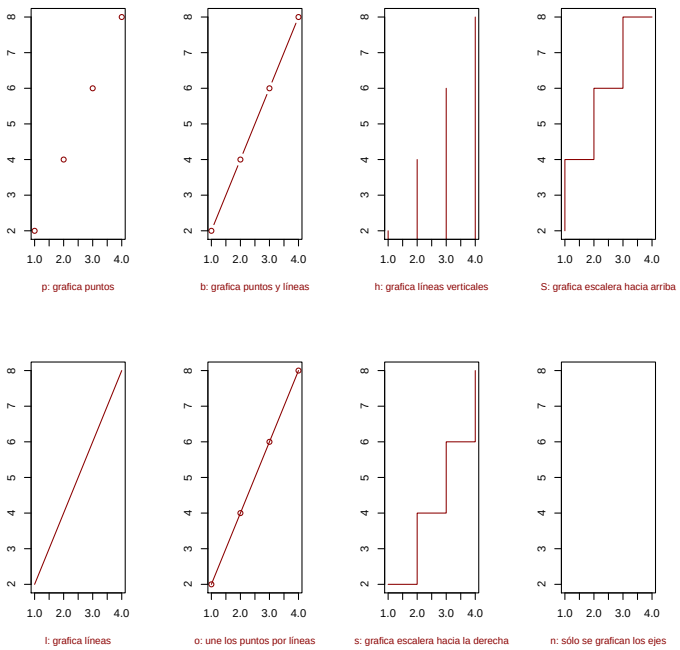

Figura 11.1: Opciones del argumento type

11.2 Funciones gráficas de bajo nivel

Por otra parte a las funciones gráficas de alto nivel podemos adicionarles funciones gráficas de bajo nivel, detallamos algunas de ellas:

Función	Descripción
points(x, y)	puntos
lines(x, y)	líneas
text(x, y, etiquetas, ...)	texto en las coordenadas x, y
abline(a, b)	recta de pendiente b y ordenada en el origen a
abline(h=y)	recta horizontal de altura y
abline(v=x)	recta vertical de altura x
polygon(x, y, ...)	polígono cuyos vértices son los elementos de (x,y)
legend(x, y, legend, ...)	leyenda, en la posición (x,y)
title(main, sub)	título y subtítulo
axis(side, ...)	eje en el lado indicado por el primer argumento, de 1 a 4

Cuadro 11.2: Funciones gráficas de bajo nivel

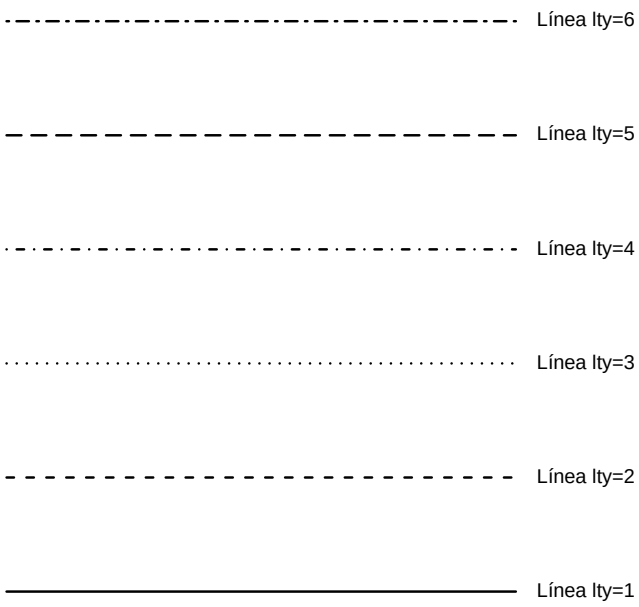

Figura 11.2: Tipos de líneas

11.3 Parámetros gráficos

Los parámetros gráficos junto a los comandos de bajo nivel mejoran la representación gráfica. Estos se puede usar como opciones de funciones gráficas – no en todas funciona–, o definiéndolos de forma general a través de la función par(). La totalidad de los parámetros gráficos existentes se describen bajo la orden ?par(), aquí sólo enumeramos los siguientes.

Argumento	Descripción	Valores
adj	justificación del texto	0 izquierda, 0.5 centrado, 1 derecha
cex	tamaño del texto y símbolos	0 y más
cex.axis	tamaño del texto de números en los ejes	0 y más
cex.lab	tamaño del texto de letras en los ejes	0 y más
cex.main	tamaño del texto del título	0 y más
cex.sub	tamaño del texto del subtítulo	0 y más
font	estilo del texto	1 normal,2 cursiva,3 negrita,4 negrita cursiva

Cuadro 11.3: Argumentos de texto

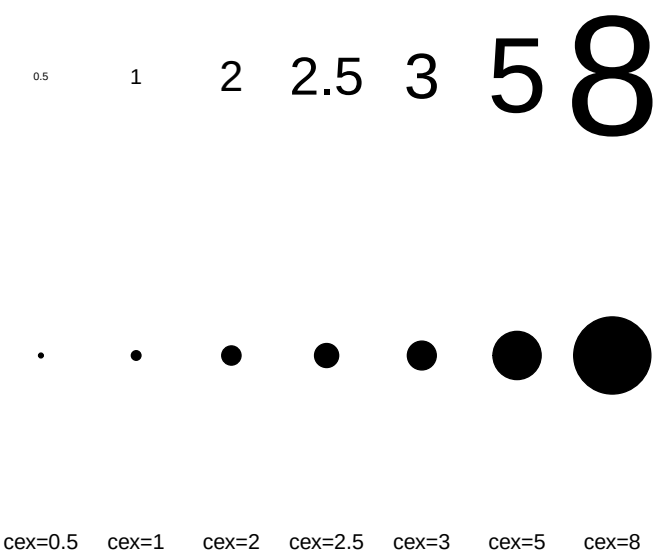

Figura 11.3: cex: tamaño de texto

Argumento	Descripción	Valores
bg	color de fondo	enteros a partir de 1
col	color de los símbolos	enteros a partir de 1
col.axis	color de los ejes	enteros a partir de 1
col.lab	color de etiquetas	enteros a partir de 1
col.main	color del título	enteros a partir de 1
col.sub	color del subtítulo	enteros a partir de 1

Cuadro 11.4: Colores

El siguiente gráfico es una muestra de los colores que se pueden obtener en R[1], se detalla el nombre y número de cada uno, amplíe la imagen tanto como necesite para observar claramente nombre y número.

Figura 11.4: Colores

Argumento	Descripción	Valores
bty	tipo de caja dibujada alrededor del gráfico	o,l,c,u
lty	tipo de línea	1 a 6
lwd	ancho de las líneas	0 y más
pch	tipo de símbolo 1 a 25	

Cuadro 11.5: Líneas y símbolos

[1] Para ver la lista de colores se debe ejecutar: colors()

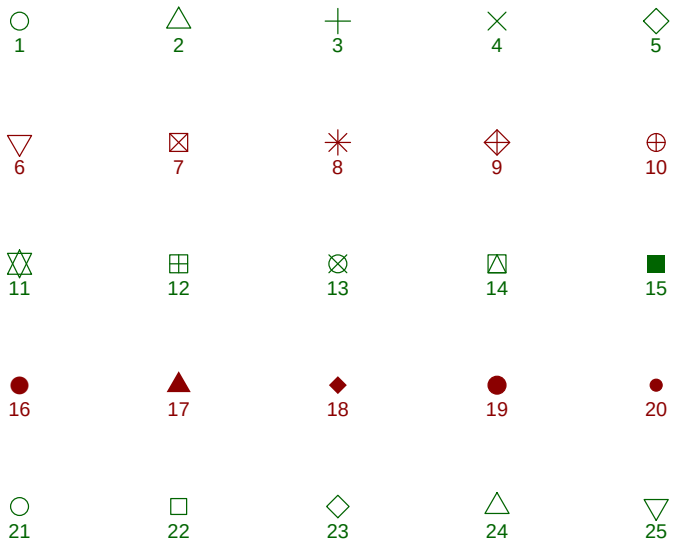

Figura 11.5: Símbolos gráficos

Argumento	Descripción	Valores
mfcol	divide la ventana gráfica en r filas y c columnas	vector
mfrow	divide la ventana gráfica en r filas y c columnas	vector

Cuadro 11.6: Ventana gráfica

11.4 Guardar gráficos

Un gráfico puede guardarse en varias extensiones: pdf, postscript, bmp, jpeg, png y tiff usando las funciones que se describen a continuación:

```
pdf(filename = "gráfico.pdf")

postscript(filename = "gráfico.ps", horizontal=F, width, height)

bmp(filename = "gráfico.bmp",
         width = 480, height = 480, units = "px",
         pointsize = 12, bg = "white")

jpeg(filename = "gráfico.jpeg",
          width = 480, height = 480, units = "px",
          pointsize = 12, quality = 75, bg = "white")

png(filename = "gráfico.png",
         width = 480, height = 480, units = "px",
         pointsize = 12, bg = "white")

tiff(filename = "gráfico.tiff",
          width = 480, height = 480, units = "px")
```

En todos los casos basta incluir el argumento que indica el nombre del archivo en el cual se guardará el gráfico. La secuencia de órdenes para guardar un gráfico correctamente es: indicar con que extensión se guardará, crear el gráfico y cerrar el dispositivo gráfico. Ejemplificaremos lo anterior construyendo algunos gráficos muy útiles y a la vez mostraremos cómo se usan algunos argumentos gráficos.

11.5 Diagrama de dispersión

Para ejemplificar cómo se hacen algunos gráficos de alto nivel vamos a trabajar con la base de datos de participantes del Curso, a la cual le agregaremos 2 variables: edad e ingresos que nos permitirán hacer un diagrama de dispersión. Como de dichas variables no tenemos información generaremos aleatoriamente sus valores[2].

```
> curso1r=read.csv("curso1R.csv",header=T, sep=";")
#para hacer un diagrama de dispersión creo 2 variables: edad e ingresos
> curso1r$edad=(rbinom(nrow(curso1r),9,0.1)+22)
> curso1r$edad
```

[2]La generación de números aleatorios implica crear números o símbolos a través de un software, en una forma que carezca de un patrón evidente. Pero esta generación parte de un valor inicial, denominado semilla, a partir del cual se van generando los valores que ya no son aleatorios sino pseudoaleatorios. Siempre que se parta de la misma semilla, se obtendrá la misma secuencia de valores. En R se puede fijar la semilla usando la función set.seed().

```
[1] 24 26 22 24 24 25 25 26 25 24 25 23 24 24 23 23 23 25
> curso1r$ingresos=round(runif(nrow(curso1r),0,50000),0)
> curso1r$ingresos
[1] 38717  9280  5378 15977 20997 47436 44186 10171 11604
     19551 25276 14021 4971 15579  8445 19623 32231 41666
> attach(curso1r)
```

De aquí en más trabajaremos con las variables de la base como objetos de manera de simplificar la sintaxis, por ello ejecutamos la función attach().

La función plot(), es una función genérica, esto es, el tipo de gráfico producido es dependiente de la clase del primer argumento que se defina. Por ejemplo, podemos realizar un diagrama de dispersión si el primer y segundo argumento de la función son vectores. Es decir, en el eje de las abscisas se miden los valores de la primer variable –primer argumento– y en el eje de las ordenadas se miden los valores de la segunda variable –segundo componente–.

```
pdf("diagrama.pdf")
plot(edad, ingresos,
      col="green",pch=16,font.axis=1,
      xlab="edad", ylab="ingesos", cex=1.5)
grid()
abline(h=median(ingresos), col="red",lwd=1.2)
dev.off()
```

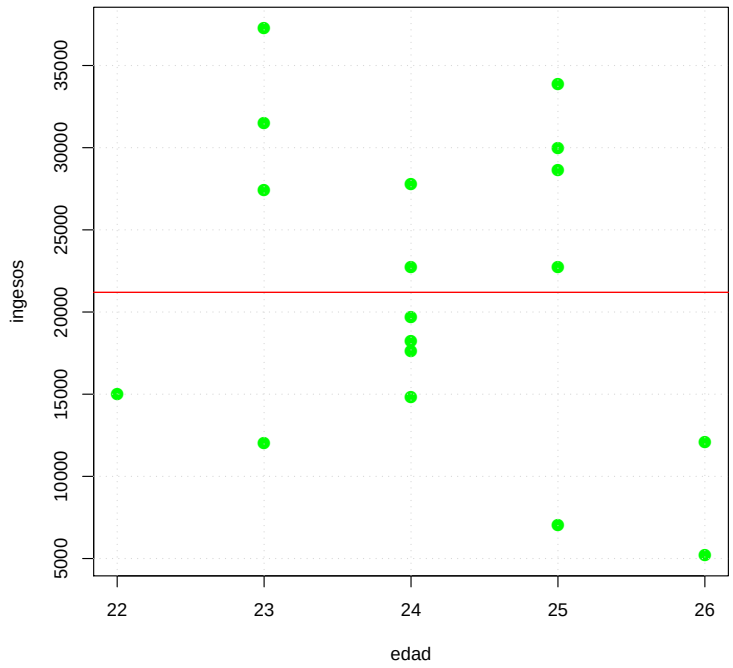

Figura 11.6: Diagrama de dispersión Edad e Ingresos

Con la primer orden, usando la función pdf() guardamos el gráfico con el nombre “diagrama” en formato pdf. Esta orden deben indicarse siempre antes de realizar el gráfico y para que tenga efecto debe incluirse la orden dev.off() luego de realizar el gráfico. La función dev.off() cierra la ventana

gráfica y permite que el gráfico se imprima en el formato especificado antes.

En la función plot() se especifican como primeros dos argumentos las variables a graficar –“edad” e “ingresos”–, con esto alcanza para obtener un diagrama de dispersión pero incorporamos otra información: color verde para los símbolos –col–, elegimos el símbolo “círculo” para los puntos–pch–, un tamaño más grande que el que viene por defecto para los símbolos –cex– y por último definimos los nombres de los ejes x –xlab– e y –ylab–.

Usamos también la función grid() que permite incorporar una grilla detrás del gráfico, en este caso de líneas entre-cortadas de color gris claro, como estos son sus valores por defecto, no es necesario indicar el valor de ningún argumento.

Por último, usando la función abline() trazamos una línea horizontal –h– a la altura de la mediana de la variable “ingresos”, de color rojo –col– y de un ancho –lwd– mayor al que viene por defecto.

11.6 Diagrama de caja

Para realizar un diagrama de caja usamos la función boxplot(), e indicamos como primer argumento la variable que queremos graficar. En este caso, además definimos el color de la caja –col– y la etiqueta del eje de las ordenadas –xlab–.

```
boxplot(ingresos, col="green", xlab="Ingresos")
```

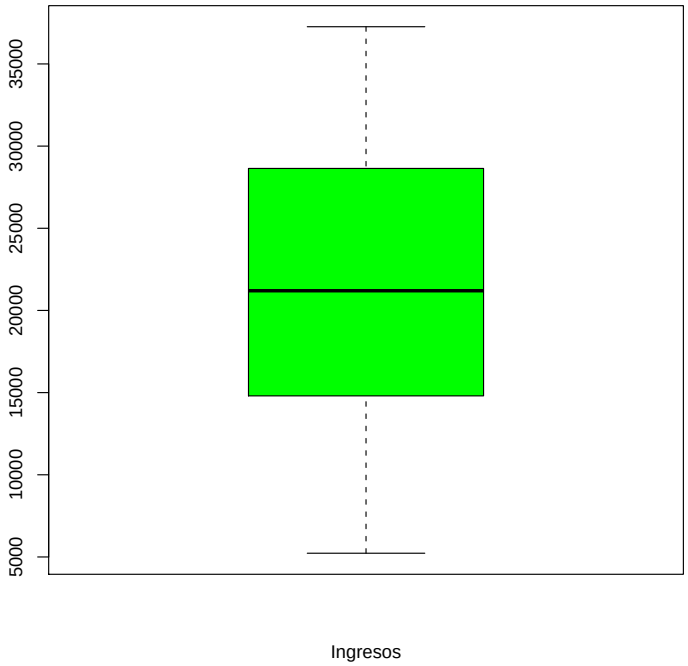

Figura 11.7: Diagrama de caja de Edad

Para construir un diagrama de caja de una variable –x– de acuerdo a los valores de otra variable –y– debemos incluir como primer argumento la fórmula $x \sim y$. En este caso agregamos el argumento

pch para elegir el símbolo en que se grafiquen los valores atípicos, es decir aquellos que están por fuera de los bigotes.

```
boxplot(ingresos~edad, col="green", xlab="Ingresos según Edad",pch=16)
```

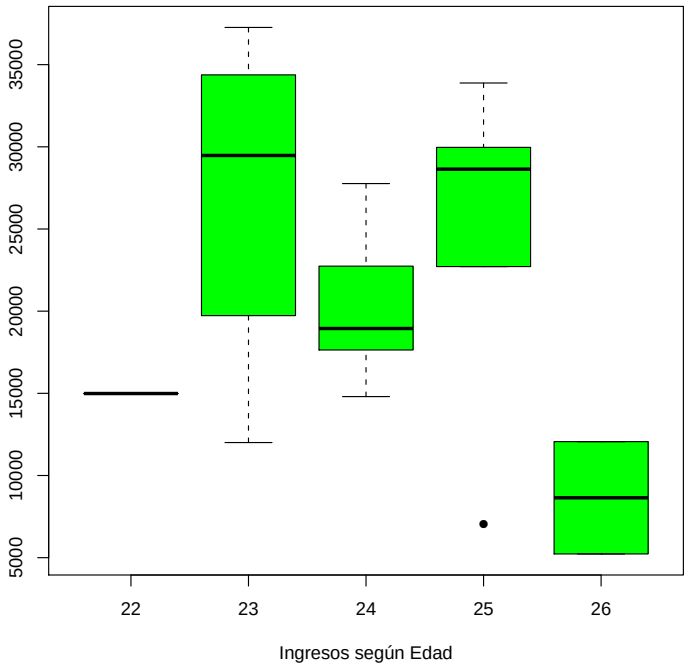

Figura 11.8: Diagrama de caja de Edad

11.7 Histograma de frecuencias

La orden hist() produce un histograma del vector numérico que indica los datos. El número de clases se calcula por defecto pero puede cambiarse con el argumento nclass, o bien especificar los puntos de corte con el argumento breaks. En este ejemplo, realizamos 2 histogramas de la variable edad, en el primero no especificamos el número de clases y en el segundo indicamos que sean 4 clases –nclass–. Para representar las frecuencias relativas en vez de las absolutas asignamos el valor TRUE al argumento probability. A su vez, elegimos el color del borde –border– por defecto, es decir, negro.
Genéricamente la función es:

```
par(mfrow=c(1,2))

hist(edad, probability=T, border="black",col="green",xlab="edad",ylab="frecuencias")

hist(edad, probability=T, nclass=4,border="black",col="green",
      xlab="edad",ylab="frecuencias")
```

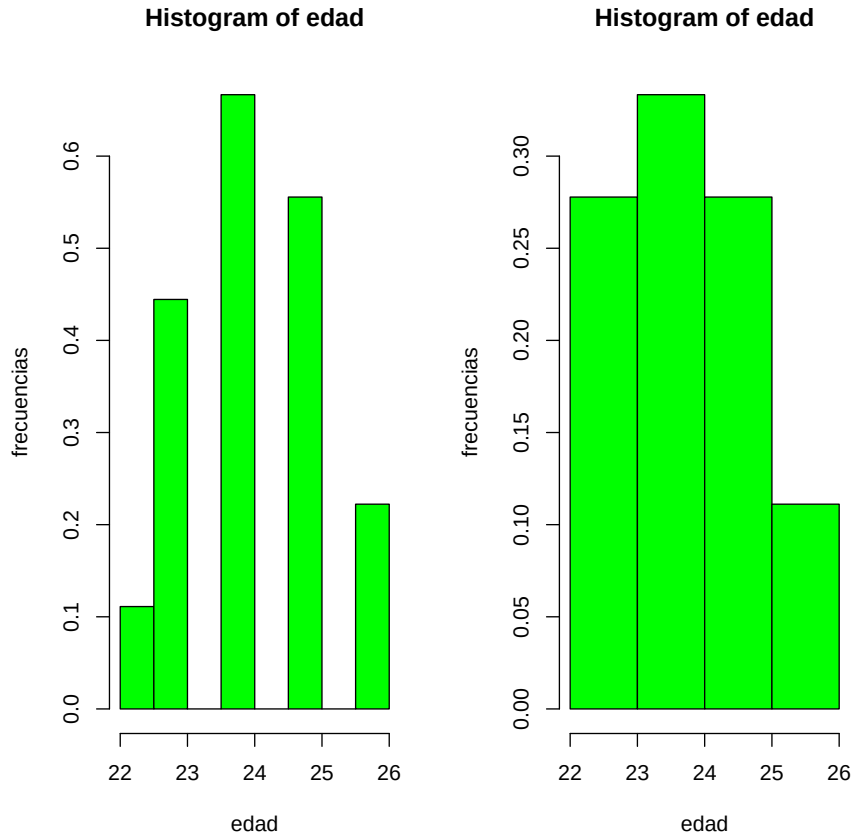

Figura 11.9: Histograma de frecuencias

11.8 Diagrama de barras

Suele ser muy útil realizar un diagrama de barras, esto se obtiene usando la función barplot() cuyo argumento principal es una tabla con los datos:

```
tabla=table(Carrera,Conocimiento.R)
tabla=as.table(round(tabla,1))
tablaf=round(prop.table(tabla),2)

barra=barplot(tabla,beside=T, col=c("seagreen4","seagreen3","seagreen1", "green"),
      legend =T, col.lab="darkgreen")
mtext(side =1, at = barra, text = tabla, line = 0, cex=1.0,font=1, col="darkgreen")
mtext(side =1, at = barra, text = tablaf, line = 3, cex=1.0, col="darkred")
```

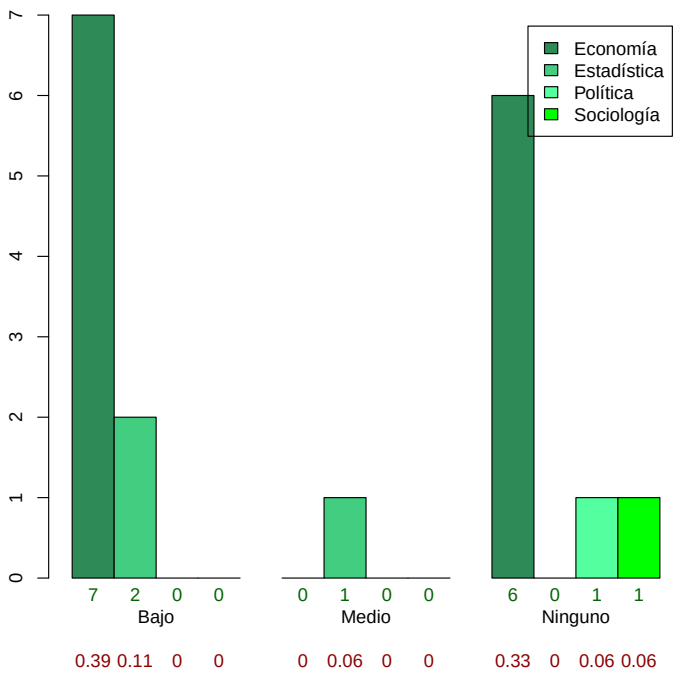

Figura 11.10: Diagrama de barras: Conocimiento de R según Carrera

11.9 Diagrama circular

```
tabla=table(Sistema.operativo)
a=as.numeric(tabla)
pie(tabla,col=c("green","seagreen3","seagreen1"),clockwise=T)
text(0.5,0.9,a[1], col="green")
text(0.5,-0.9,a[2],col="seagreen3")
text(-0.6,0.9,a[3],col="seagreen1")
```

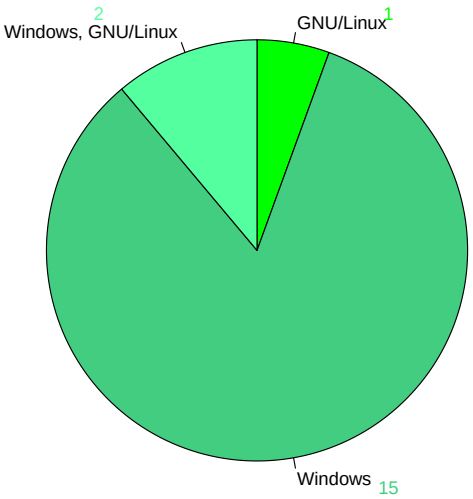

Figura 11.11: Diagrama de barras: Conocimiento de R según Carrera

12

Anexo

12.1 Instalar R en Linux

Para cualquier versión de Ubuntu o Debian, puede instalarse R utilizando el gestor de paquetes Synaptic o directamente la terminal. Con el gestor de paquetes solo es necesario buscar e instalar los paquetes que se listan para el caso de terminal. En terminal basta escribir lo siguiente:

1. Actualizamos la lista de paquetes

   ```
   sudo aptitude update
   ```

2. Instalamos r-base y algunas dependencias

   ```
   sudo aptitude install r-base r-recommended r-base-latex
   ```

3. Podemos opcionalmente instalar R Comander

   ```
   sudo aptitude install r-cran-rcmdr r-cran-rodbc r-cran-rgl
   ```

   Para ejecutar R Comander, hay que ejecutar R desde consola y luego escribir:

   ```
   > library(Rcmdr)
   ```

4. Instalar algunas librerías interesantes a. Para poder importar/exportar datos de y hacia otros programas –por ejemplo SPSS–, utilizamos la librería foreign:

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

5. Cambiar el editor de texto de R por defecto Para que quede el cambio fijo, ejecutar:

   ```
   > .First<-function() {options(editor="/usr/bin/gedit")}
   ```

6. Instalar un editor externo avanzado. Ejemplo: Kate
   En las versiones últimas Ubuntu –9.10 por ejemplo– se instala Kate y luego en plugins se activa la consola. Finalmente se le puede agregar una tecla rápida para enviar lo que hay en pantalla a consola.

   ```
   sudo aptitude install kate konsole
   ```

12.2 Conceptos de muestreo probabilístico

El muestreo probabilístico es un procedimiento de investigación estadística que a partir de información sobre una parte de los elementos que componen el universo o población objetivo –las unidades de muestreo– pretende hacer inferencias sobre todos los elementos que componen el universo.

Realizar una encuesta por muestreo implica seleccionar una parte de los elementos de la población, en el caso de la ECH, se seleccionan los hogares particulares de todo el país, con el objetivo de obtener información de dichos elementos para extrapolarlos a toda la población. Pero las unidades muestrales no se seleccionan directamente de la población sino del marco muestral.

12.2.1. Marco muestral

El marco muestral puede ser un listado de los elementos de la población –"marco de lista"– o bien un listado de subconjuntos de elementos de la población –"marco agrupado"–. Denominamos a los componentes del marco "unidades", para diferenciarlos de los componentes de la población que denominamos "elementos", esto pues no necesariamente una unidad del marco se corresponde a un elemento de la población. A su vez, si el muestro se realiza en varias etapas como es el caso de la ECH, debemos distinguir entre unidades de muestreo, aquellas que corresponden a las etapas 1 y 2 –localidades y zonas censales– de las unidades de análisis que son las correspondientes a la tercer etapa –las viviendas particulares.

Debido a que la muestra se realiza directamente desde el marco, la inclusión en éste de elementos ajenos a la población hace que se pueda incurrir en errores "no de muestreo". El caso ideal de un marco muestral es cuando existe una relación uno a uno entre las unidades del marco y los elementos de la población, pero esto no es lo que suele ocurrir en la práctica. Es decir, en caso de la ECH, la información del último censo permitiría tener un marco perfecto, pero como la información al día de hoy está desactualizada, tendríamos un marco que ya no es perfecto.

Por lo tanto, primero debe contarse con un listado de localidades, luego un listado de zonas censales –al menos para las localidades seleccionadas– y por último un listado de viviendas particulares –al menos para las viviendas particulares seleccionadas–. Concretamente, el marco de la ECH 2009 "está basado en los listados por zona censal del Censo 2004 [U+0096] Fase 1, a partir de los cuales se realizó la estratificación. Las zonas censales seleccionadas cada año que no han sido listadas durante el año anterior, son utilizadas para proceder a la actualización del número de viviendas particulares. A partir del listado actualizado se procede a la selección de las viviendas/hogares a entrevistar"[1].

[1] "Ficha técnica, Encuesta Continua de Hogares 2009", INE, 2009.

12.2.2. Errores en las encuestas por muestreo

A diferencia de lo que intutivamente, puede pensarse, de una muestra pueden obtenerse estimaciones de las variables de interés más precisas que los valores de dichas variables obtenidas de un censo. Porque a pesar de que en una muestra se cometen "errores de muestreo" –debido a que la información de toda la población es una estimación que utiliza como base los datos de una parte representativa de esa población– y "errores no de muestreo", estos pueden ser más reducidos que los "errores no de muestreo" de un censo.

Los errores no muestrales no se pueden medir fácilmente y aumentan a medida que aumenta el tamaño de la muestra. Lo que hacen los errores no muestrales es sesgar los resultados, en una dirección y magnitud desconocidas.

Errores	Descripción
Muestrales	causado por la variabilidad muestra a muestra de las observaciones
No muestrales	de Subcobertura: el marco no cubre todos los elementos de la población de No respuesta: alguna o todas las variables de interés no son observadas en las Observaciones: errores de medida y procesamiento de los datos

Cuadro 12.1: Tipo de errores de una muestra

12.2.3. Muestra

Una muestra es un subconjunto de la población. Para que el muestreo sea probabilístico la selección de la muestra debe cumplir con lo siguiente:

- El conjunto de muestras posibles es conocido de antemano.
- Cada muestra posible tiene asociada una probabilidad de selección, $p(s)$. Así el procedimiento de selección asegura para todo elemento de la población, una probabilidad no nula de inclusión en la muestra.
- Una muestra es seleccionada por el mecanismo aleatorio que asegura que cada una de las posibles muestras, tiene una probabilidad de ser seleccionada exactamente igual a $p(s)$.

Entonces, una muestra obtenida mediante el procedimiento anterior será una "muestra aleatoria".

12.2.4. Diseño muestral

Existen distintas formas de seleccionar una muestra, el diseño muestral define el procedimiento de selección de la muestra. Un diseño muestral $p(s)$ es una distribución de probabilidad sobre todas las muestras posibles.

Definir el diseño de muestreo implica definir una serie de cuestiones:

- Si se realiza un muestreo directo de elementos –una sola etapa– o para llegar a las viviendas particulares es necesario varias etapas de muestreo.

- Si las viviendas particulares se eligen con equiprobabilidad o con probabilidades desiguales.
- El tamaño de la muestra.
- El muestreo a utilizar, estratificado, con reposición, sin reposición, etc.

En el caso de la ECH 2009 se realiza un muestreo estratificado, el cual consiste en particionar a la población en cierto número de subpoblaciones –llamadas estratos– y tomar una muestra aleatoria de manera independiente en cada una de ellas. Así el territorio nacional, es subdividido en 59 estratos, según los siguientes criterios:

- Montevideo. El departamento está dividido en cuatro estratos socioeconómicos: bajo, medio bajo, medio alto y alto.
- Anillo periférico. Incluye Canelones y San José y se extiende hasta un límite aproximado a 30 km desde el centro de la capital. Constituye un estrato.
- Departamentos del Interior (exceptuando anillo periférico). Cada departamento es dividido en tres estratos: el constituido por localidades de 5000 o más habitantes, el constituído por localidades de menos de 5000 habitantes, y las zonas rurales. Totalizan 54 estratos.

Las variables utilizadas para estratificar son:

- el ingreso medio per cápita real de los hogares a nivel de segmento censal en Montevideo. “Esta variable explica más del 80 % de la variabilidad total entre las unidades de muestreo y produce la mejor segmentación geográfica por características socio-económicas de la población”.
- En los restantes departamentos se estratifica según el tamaño de las localidades.

Así en el caso de la ECH 2009 se seleccionó una “muestra probabilística, estratificada con afijación óptima[2] para las variables ingreso per cápita de los hogares y tasa de desempleo para las subpoblaciones de referencia. La muestra se selecciona en 3 etapas: localidad, zona censal y vivienda particular y es independiente mes a mes y año a año”.

12.2.5. Selección de las unidades de muestreo

A la hora de determinar el tamaño de muestra en cada estrato es necesario considerar las varianzas de las variables de interés, la precisión con la que se desean obtener las estimaciones y la confianza requerida. La información de las varianzas de las variables de interés puede venir de un censo previo o a partir de encuestas anteriormente realizadas.

En la primera etapa se procede a seleccionar las localidades –en el caso de los departamentos del Interior– y luego las zonas censales –manzanas o territorio identificable– a través de un método de selección con probabilidades proporcionales al tamaño –PPT–. La medida de tamaño a ser considerada en la selección de las zonas con PPT será el número total de viviendas particulares.

[2] La asignación o afijación óptima del tamaño de muestra, surge de resolver un problema de minimización de la varianza del estimador de cada variable de interés sujeto a los costos en que se incurre al realizar las encuestas, diseñar el muestreo, etc.

12.2.6. Tamaño de muestra: selección de las unidades de análisis

Finalmente dentro de cada zona se seleccionan las unidades de muestreo, esto es, las viviendas particulares. Las viviendas son seleccionadas al azar en número de 3 titulares y 2 suplentes. Si una zona censal no contiene cinco viviendas particulares es consolidada con una o más zonas vecinas hasta alcanzar el tamaño mínimo.

Así en el año 2009, el plan de muestreo estableció un tamaño de muestra de 46.936 viviendas (aproximadamente 3900 casos por mes) distribuido en un 43.9 % en Montevideo, 8.3 % en la Periferia, 35.4 % en el Interior urbano residente en localidades de 5.000 habitantes o más, un 5.3 % en localidades de menos de 5.000 y un 7.1 % en zonas rurales. Esta muestra comprende aproximadamente 132.599 personas.

12.2.7. Glosario

Diseño Muestral: es una distribución de probabilidad sobre todas las muestras posibles.

Error de muestreo: se produce debido a que la información de toda la población es una estimación que utiliza como base los datos de una parte representativa de esa población.

Marco Muestral: es un conjunto de unidades, procedimientos y mecanismos que identifican, distinguen y permiten acceder a los elementos de la población.

Muestreo: es un procedimiento de investigación estadística que pretende estudiar el universo de interés con base en la información que se obtiene de una parte de las unidades que componen dicho universo.

Población: conjunto de elementos de los cuales se desea conocer el valor de ciertas variables.

Muestra: subconjunto de la población y representativo de ésta en las variables de interés

Unidades de muestreo: subconjunto de unidades del marco muestral, seleccionadas aleatoriamente en etapas previas a la selección de las unidades de análisis

Unidades de muestreo: subconjunto de unidades del marco muestral que son seleccionadas en la última etapa, sobre éstas se miden ciertas variables.

Para un detalle sobre las diferentes técnicas de muestreo probabilístico existentes se recomienda leer ”Model Assisted Survey Sampling”,1991. de Särndal C.E., Swensson B. y Wretman J.