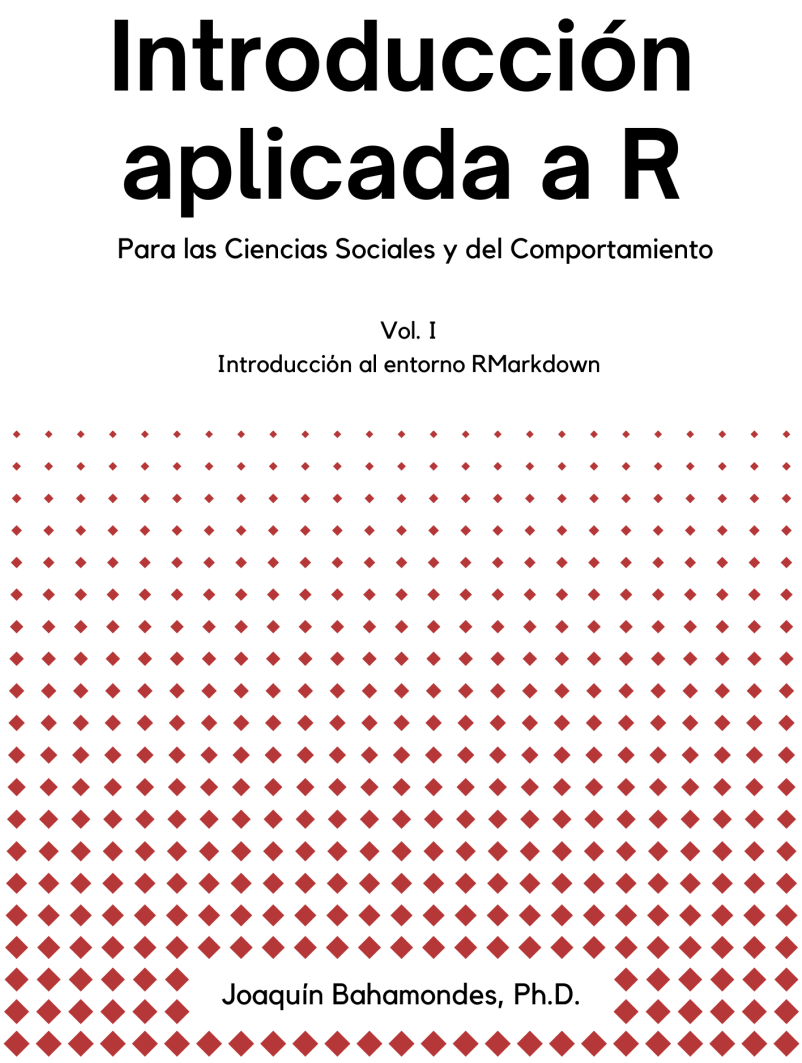
Contents

Sobre este libro



El presente documento tiene como objetivo introducir el uso de R, bajo criterios de **ciencia abierta**, para el ordenamiento y ejecución de código aplicado principalmente a las ciencias sociales y del comportamiento. Por lo tanto, este documento no es un libro de estadística, sino una guía técnica, específica para el uso de R en **RStudio** y la implementación de su código.

Este volumen, en particular, presenta algunos de los fundamentos del uso de R, sugerencias para el ordenamiento del código, uso de notación básica y funciones esenciales.

Cualquier duda, comentario o sugerencia, por favor dirigirla a Joaquín Bahamondes, autor de esta guía, al e-mail jbahamondes@ucn.cl

¿Por qué R?

El programa R es una consola de análisis de operadores lógicos, principalmente diseñado para análisis estadísticos, con su propio lenguaje de programación. Quizás más importante, es un software de código abierto, gratuito, de gran flexibilidad y capacidad para gestionar distintos tipos de información y producir múltiples resultados, desde output de análisis estadístico, hasta imágenes, archivos de texto y sitios web (entre otros).

Índice

Chapter 1

Estructura y flujo de trabajo

Como punto de partida, es preciso comprender la organización del código en el espacio de trabajo de R. En esta sección, nos referiremos a la estructura recomendada para organizar el entorno de trabajo (i.e., las carpetas y el flujo entre ellas), así como algunas recomendaciones generales respecto de la estructura interna de cada documento de trabajo.

1.1 Flujo de trabajo IPO

El modelo input–process–output (IPO) es una forma ampliamente adoptada en análisis de sistemas y procesamiento de datos, con el objetivo de proveer un ordenamiento eficiente de la información y los procesos implicados en algún proyecto. Para nuestros propósitos, el protocolo IPO es una forma de ordenar la estructura y el flujo de las carpetas de un proyecto R (R project file) vinculado a un (o una serie de) estudio(s). Éste es enfáticamente recomendado por el Laboratorio de Ciencia Social Abierta (LISA), dado que es altamente adecuado para la apertura de los materiales en base a criterios de Ciencia Abierta.

La implementación de la reproducibilidad en este tipo de protocolos se basa en generar un conjunto de archivos auto-contenidos, organizados bajo una estructura de proyecto, que cualquier persona pueda acceder y usar. En otras palabras, debe tener todo lo que necesita para ejecutar, y volver a ejecutar el análisis, obteniendo los mismos productos.

Específicamente, el protocolo **IPO**, para flujo de investigación reproducible, se refiere a Input-Procesamiento-Output. Esta organización permite estructurar los archivos en tres carpetas.

El directorio correspondiente al **input** contiene todo aquello que "alimentaremos" a los procesos. Se trata de archivos de *etrada*, que no modificaremos en su archivo original, pero cuya información resulta relevante para los procedimientos a desarrollar. Algunos ejemplos son archivos de datos en bruto, imágenes, etc.
Una segunda carpeta contiene los archivos de **proceso**, usualmente `.rmd`. Estos corresponden a aquellos donde guardamos el código que ejecuta operaciones sobre los inputs. Por ejemplo, aquí guardaríamos nuestros códigos de gestión de datos o de análisis estadístico.
Finalmente, la última carpeta corresponde al **output**, es decir, las *salidas*. Aquí se alojan los resultados de los procesos, como versiones limpias o modificadas de los datos, o las imágenes (e.g., gráficos) derivados de los archivos de proceso. Si bien estos productos pueden ser utilizados como entradas por algún archivo de proceso, es importante distinguirlos de aquellos que corresponden a archivos en bruto, alojados en la carpeta input, dado que todo output es el resultado de algún proceso llevado a cabo por nosotro/as en el entorno de nuestro proyecto.

El siguiente diagrama grafica la estructura general de estas carpetas:

```
   input: información externa como datos, imágenes, .bib:
      data:
        original : archivos de datos originales y metadatos disponibles
        proc     : archivos de datos procesados
      imagenes
      bib: archivos de bibliografía
      prereg: archivos de pre-registro si están disponibles
|
   procesamiento:
    - preparacion.Rmd
    - analisis.Rmd

   output: tablas, gráficos y otras salidas del procesamiento.
      gráficos
      tablas

- readme.md : archivo general de introducción
- paper.md o paper.Rmd / paper.html / paper.pdf: el artículo/paper
```

Obtenido desde https://lisa-coes.netlify.app/ipo-repro/

1.2 Estructura de un documento Rmarkdown

Las secciones de un documento de R, cualquiera sea (archivo de sólo código, RMarkdown o un proyecto) pueden ser organizadas bajo distintos niveles de

títulos, los que anidan el contenido correspondiente. Como regla general, debe haber sólo un título de nivel uno (`#`) por archivo `.rmd`, y pueden haber múltiples subtítulos de nivel dos (`##`), tres (`###`), o de mayor subordinación (`####...`). En Rmarkdown, bajo cada título o subtítulos, pueden incluirse párrafos de texto como este, lo que no constituye material ejecutable, de modo que éste no será analizado de ninguna forma por el programa, sólo será presentado.

Toda vez que queramos incluir código ejecutable para procesar algún tipo de información en Rmarkdown, debemos usar `chunks`. Los `chunks` son bloques de código ejecutable, los que R entenderá como contenedores de material analizable. En Rmarkdown, se puede crear rápidamente un chunk con el atajo **ctrl + Alt + i**. Verá inmediatamente un bloque de código tal como el que se presenta a continuación:

```
```{r}
# Esto es un comentario dentro de un chunk. Recuerde que esto será omitido.
# La siguiente operación será ejecutada.

5+2

```
```

Chapter 2

Notación y Funciones básicas

2.1 Notación esencial

En R, los operadores básicos son:

Notación/función	símbolo usado
Asignación	<-
Comentario	#
Combinar	c()
Separador de Argumentos	,
Texto	“ ” o ’ ’
Equivalencia	==
Operador O	|
Comparación	>, <, >=, <=
Acceso a columnas de dato, matriz o lista	$
Indexación	[]
Llamada de función o agrupación	()
Pertenencia	%in%
Negación	!
Secuencia	:
Operadores aritméticos	*, /, +, -

2.1.1 <- (Asignación)

Uso: Asigna valores a una variable. También se puede usar = para asignaciones básicas, pero <- es el operador recomendado.

Ejemplo:

```
x <- 10
y <- "texto"
```

Nota: -> también existe, aunque es poco común y asigna en la dirección opuesta (valor -> variable).

2.1.2 # (Comentario)

Uso: Agrega comentarios en el código. R ignora todo el texto a la derecha del # en esa línea.
Ejemplo:

```
x <- 10  # Este es un comentario
```

2.1.3 c (Combinar)

Uso: Agrupa distintos elementos en un listado o grupo que puede utilizarse como bloque para múltiples propósitos.
Ejemplo:

```
sujeto01 <- c(x,y)  # agrupa respuestas de un sujeto
sujeto02 <- c(5, "texto 2") # lo mismo para sujeto 2
```

Nota: Los comentarios son esenciales para documentar y hacer el código más comprensible.

2.1.4 , (Separador de Argumentos)

Uso: Separa elementos en funciones o elementos de un vector/matriz.
Ejemplo:

```
c(1, 2, 3)  # Crea un vector con 3 elementos
```

Nota: También se usa para separar filas y columnas en indexación de matrices y data frames.

2.1.5 “ ” o ’ ’ (Definición de Cadenas de Texto)

Uso: Define valores de tipo carácter (strings). Ejemplo:

```
nombre <- "Joaquín"
apellido <- 'Bahamondes'
```

Nota: En R, ” y ’ son equivalentes. Sin embargo, es más común usar ” para cadenas de texto.

2.1.6 == (Igualdad Lógica)

Uso: Compara si dos valores son iguales. Devuelve TRUE o FALSE.
Ejemplo:

```
x <- 5
y <- 5
x == y  # TRUE
```

Nota: No confundir con = que es un operador de asignación (aunque puede ser usado en argumentos de funciones).

2.1.7 | (OR Lógico)

Uso: Devuelve TRUE si al menos una de las condiciones es TRUE. Ejemplo:

```
(x == 5) | (y == 10)
```

2.1.8 >, <, >=, <= (Comparación)

Uso: Operadores de comparación para verificar si un valor es mayor, menor, mayor o igual, o menor o igual que otro. Ejemplo:

```
x > 3   # TRUE si x es mayor que 3
```

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

```
y <= 10 # TRUE si y es menor o igual a 10
```

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

2.1.9 $ (Acceso a Columnas de Data Frames o Listas)

Uso: Permite acceder a una columna específica en un data frame o a un elemento en una lista. Ejemplo:

```
df <- data.frame(a = 1:3, b = 4:6)
df$a  # Devuelve la columna 'a'
```

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

Nota: Es especialmente útil en análisis de datos cuando necesitas acceder rápidamente a columnas (i.e., variables) específicas.

2.1.10 [] (Indexación)

Uso: Extrae elementos de vectores, listas, matrices o data frames. Ejemplo:

```
vec <- c(10, 20, 30)
vec[1]  # Devuelve el primer elemento
```

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

Nota: [fila, columna] se usa para acceder a elementos en matrices o data frames.

2.1.11 () (Llamada de Funciones o Agrupación)

Uso: Agrupa expresiones o se usa para pasar argumentos en funciones. Hemos estado usándola para aplicar todas las funciones presentadas anteriormente. Ejemplo:

```
suma <- sum(1, 2, 3)  # Llama a la función sum con tres argumentos
```

Nota: También se usa para forzar el orden de operaciones en expresiones matemáticas.

2.1.12 %in% (Pertenencia)

Uso: Verifica si un elemento está presente en un vector. Ejemplo:

```r
x <- 5
x %in% c(1, 2, 3, 5)  # TRUE
```

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

2.1.13 ! (Negación)

Uso: Niega una condición lógica (convierte TRUE a FALSE y viceversa). Ejemplo:

```r
x <- FALSE
!x  # TRUE
```

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

2.1.14 : (Secuencia)

Uso: Genera una secuencia de números enteros. Ejemplo:

```r
1:5  # Genera el vector c(1, 2, 3, 4, 5)
```

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

2.1.15 *, /, +, - (Operadores Aritméticos) {#aritmeticos}

Uso: Realizan operaciones aritméticas básicas de multiplicación, división, suma y resta. Ejemplo: r Copiar código x <- 10 y <- 5 x + y # Suma, devuelve 15 x * y # Multiplicación, devuelve 50

En esta sección, se detallan las funciones básicas más comunes que necesitaremos aplicar al analizar datos en R.
En general, el lenguaje R se trata de aplicar funciones a objetos, de modo que "abrazamos" un objeto x con una función para obtener un resultado deseado, en el formato `función(x)`. El objeto x puede tomar múltiples formas, desde una base de datos, un listado de palabras, o un número único.

Probablemente la función más útil en R es `combine`, la cual se aplica simplemente con `c(x)`. consideremos el siguiente ejemplo:

2.1.16 Combine

Esta función sirve, entre otras cosas, para combinar distintos elementos (e.g., valores) dentro de una misma lista. Por ejemplo, si tenemos 5 sujetos cuyas edades son 28, 37, 29, 43, y 34, podemos crear un objeto (en este caso, un vector) que contenga dicha información, simplemente con el siguiente código

```
edades <- c(28, 37, 29, 43, 34)
```

Al ejecutar ese código, se creará un nuevo objeto, llamado `edades`, que contiene los 5 valores. Por lo tanto, dado que el objeto corresponde a una lista de números, podemos aplicar funciones algebráicas, tal como calcular la media `mean(x)`, o la desviación típica `sd(x)`, para lo cual obtendremos:

`mean(edades)` = 34.2
`sd(edades)` = 6.14

Chapter 3

Instalación y uso de paquetes

En R, los paquetes son colecciones organizadas de funciones, datos y/o documentación diseñadas para facilitar tareas específicas en programación y análisis. Lo principal es que extienden las capacidades base de R, permitiendo a los usuarios acceder a funcionalidades adicionales sin tener que programarlas desde cero.

3.1 Instalación:

Los paquetes deben instalarse una vez antes de su primer uso en el entorno de R. Esto se hace con la función

```
install.packages("nombre-del-paquete")
```

la cual descarga el paquete y lo guarda en la biblioteca de R local.

3.2 Carga:

Después de instalar un paquete, se debe cargar en cada nueva sesión de R para acceder a sus funciones, lo cual se hace usando

```
library(nombre_del_paquete)
```

Esto permite que R reconozca las funciones y datos contenidos en el paquete.

3.3 Uso:

Una vez cargado, el usuario puede llamar a las funciones y utilizar los datos incluidos en el paquete de manera inmediata. Las funciones dentro del paquete están diseñadas para integrarse con el entorno R y suelen seguir la misma sintaxis y estilo.

3.4 Actualización y Gestión:

Los paquetes deben actualizarse periódicamente para aprovechar mejoras, correcciones y nuevas funcionalidades. Esto se hace con

```
update.packages()
```

3.5 Uso de Pacman

Pacman es un paquete destinado a facilitar la instalación y carga de paquetes en R.
En esta guía, recomendamos hacer uso rutinario de la función `p_load`, dentro de la cual se puede definir un conjunto de paquetes (seprados por ,) que deban cargarse, según las funciones necesarias para la sesión. Una de las principales utilidades de esta función es que instala aquellos paquetes que no estén instalados en el entorno local.

```
pacman::p_load()
```

Chapter 4

Simulación

Simular datos en R es una técnica útil para pruebas, experimentos o validaciones estadísticas sin necesidad de datos reales. Con funciones como `rnorm()`, `runif()`, `sample()`, y `rbinom()`, R permite generar datos que cumplen con ciertas distribuciones y características básicas, como la media, la desviación estándar, la cantidad de observaciones, entre otras.

4.1 Generación de Datos

4.1.1 Datos Normales:

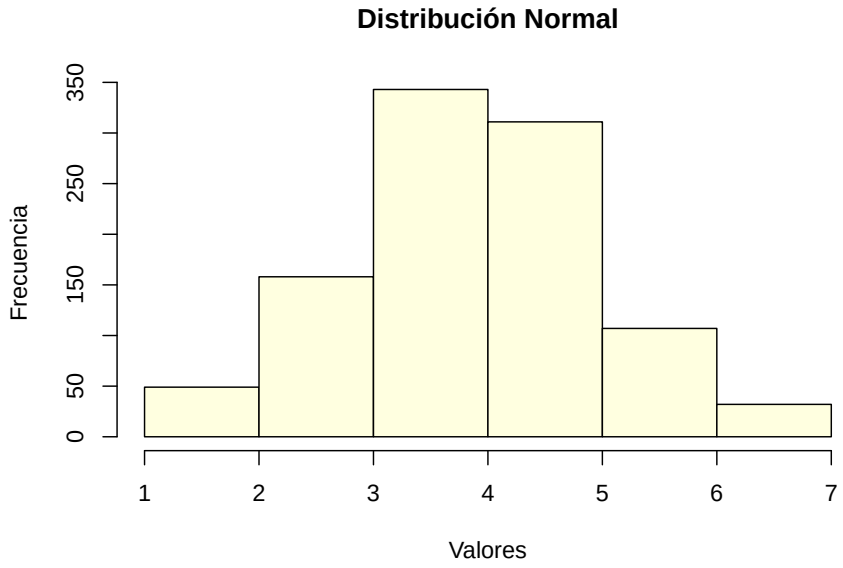
Para generar datos con una distribución normal, se usa `rnorm(n, mean, sd)`, donde:

- `n` es el número de observaciones,
- `mean` es la media,
- `sd` es la desviación estándar.

```
datos_normales <- rnorm(1000, mean = 3.8, sd = 1.1)
# Imponer límites mínimo y máximo
minimo <- 1
maximo <- 7
datos_normales_lim <- ifelse(datos_normales < minimo, minimo,
                        ifelse(datos_normales > maximo, maximo, datos_normales))

hist(datos_normales_lim,
```

```
     breaks = 7,
     col = "lightyellow",
     main = "Distribución Normal",
     xlab = "Valores",
     ylab = "Frecuencia")
```

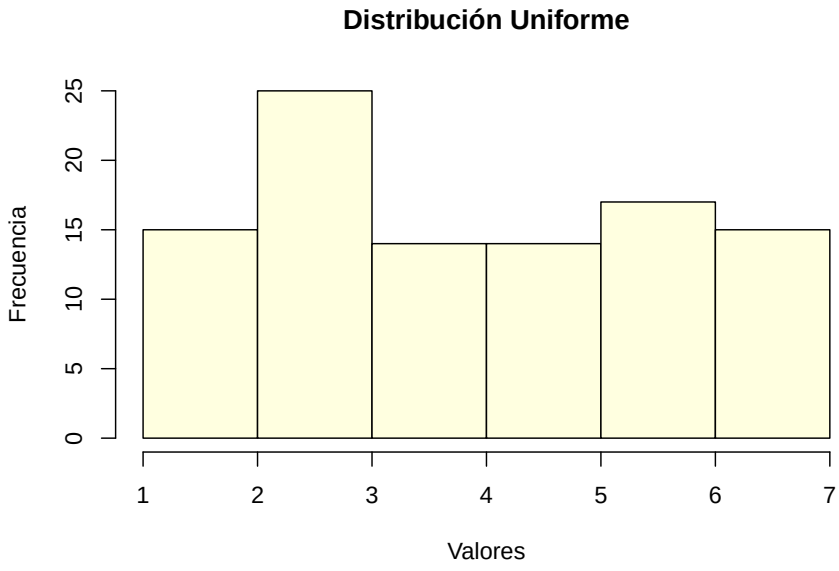


4.1.2 Datos Uniformes:

Para una distribución uniforme, que genera valores entre un rango específico con la misma probabilidad, se usa `runif(n, min, max)`:

- `n` es el número de observaciones,
- `min` y `max` definen el rango.

```
datos_uniformes <- runif(100, min = 1, max = 7)
hist(datos_uniformes,
     breaks = 7,
     col = "lightyellow",
     main = "Distribución Uniforme",
     xlab = "Valores",
     ylab = "Frecuencia")
```

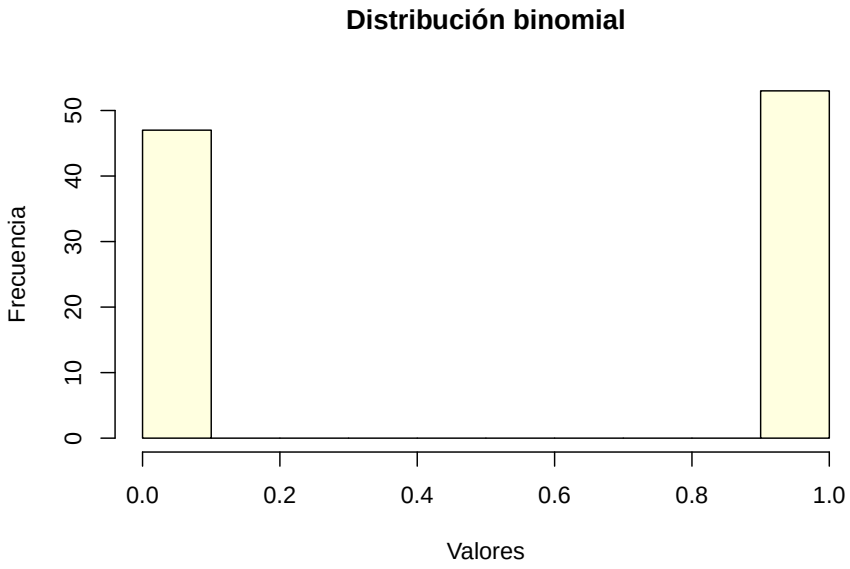


4.1.3 Datos Binomiales:

Para datos binarios o con un número fijo de éxitos (p.ej., tiradas de moneda), se usa `rbinom(n, size, prob)`, donde:

- `n` es el número de observaciones,
- `size` es el número de ensayos,
- `prob` es la probabilidad de éxito en cada ensayo.

```
datos_binomiales <- rbinom(100, size = 1, prob = 0.5)
hist(datos_binomiales,
     breaks = 12,
     col = "lightyellow",
     main = "Distribución binomial",
     xlab = "Valores",
     ylab = "Frecuencia")
```



4.2 Manipulación de Características de los Datos

4.2.1 Cambio de la media y desviación estándar:

La media y la desviación estándar se pueden ajustar en funciones como `rnorm()` para controlar la distribución y dispersión de los datos.

4.2.2 Tamaño de la muestra:

Ajustando el parámetro `n`, se modifica la cantidad de datos generados, lo cual es útil para realizar pruebas de análisis en conjuntos grandes o pequeños.

4.2.3 Rango de valores:

Para obtener valores en rangos específicos (p.ej., entre 0 y 100), `runif()` permite definir estos límites con `min` y `max`.

Estos métodos permiten generar datos con propiedades controladas, ayudando a diseñar experimentos o pruebas que se asemejan a escenarios reales y asegurando variabilidad y control en las simulaciones.

4.3 Creación de una base de datos en R

A continuación, crearemos una base de dato desde cero en este mismo entorno, practicando la creación de distintos objetos, vectores, matrices, y terminando por guardar una base de datos exportable. Recuerda que las características de cualquiera de estas variables puede ser modificada a nuestro antojo, y poder ajustarlas a nuestros propósitos particulares.
Para no repetir sistemáticamente el mismo número en caso que queramos modificarlo, definiremos el número de casos. Para esto, simplemente guardamos en un objeto el n deseado.

```
n_sim <- 200
```

Partiremos por crear un vector identificador para un total de 200 casos. Para esto, utilizamos la función `paste()`, de modo que podamos asignar un código alfanumérico a cada fila que registre un caso.

```
# Vector de caracteres
id_vec <- paste("id", 1:200)
```

Ahora, inventaremos que esta muestra de 200 personas tiene edades entre los 20 a 60 años de edad. Entonces, usamos la función `sample()` para crear un listado de 200 valores aleatorios entre 20 y 60, con posibilidad de reemplazo. Como se puede observar en el código, se utilizará la función `set.seed()` para fijar la semilla que produce la aleatorización, de modo tal que no variará cada vez que ejecutemos el código, lo que permitirá la reproducibilidad.

```
set.seed(123)  # Para reproducibilidad

# Vector numérico
edades <- sample(20:60, 200, replace = TRUE)
```

Aceleremos algo el proceso, creando una matriz `matrix()` en lugar de un sólo vector. Para esto, creemos aleatoriamente 200 valores tanto para altura (en cms) como para peso (kgs).

```
set.seed(123)

# Matriz de altura (cm) y peso (kg)
mx_altura_peso <- matrix(c(
  sample(150:200, 200, replace = TRUE),  # Altura en cm
  sample(50:100, 200, replace = TRUE)    # Peso en kg
), nrow = 200, byrow = FALSE)
```

Demos nombres a las columnas (i.e., variables) usando la función `colnames()`, sobre la cual se asigna el listado de nombres, tal como si estuviésemos creando un objeto a través de una asignación `<-`. Usaremos la función `head()` para chequear que la matriz se haya dispuesto correctamente.

```
# Asignar nombres a las columnas
colnames(mx_altura_peso) <- c("Altura_cm", "Peso_kg")

# Inspeccionar los primeros casos de la matriz
head(mx_altura_peso)
```

```
##      Altura_cm Peso_kg
## [1,]       180      65
## [2,]       164      72
## [3,]       200      82
## [4,]       163      89
## [5,]       152      89
## [6,]       191      59
```

Creemos una variable sin usar aleatorización. Por ejemplo, asignemos deliberadamente las distribuciones de la variable educación.

```
n_med  <- 90
n_univ <- 70
n_post <- 40
```

Entonces, asignaremos los primeros 90 casos a educación media completa, los segundos 70 a graduados universitarios, y los últimos 40 serán asignados con título de postgrado. Esto lo logramos haciendo uso de la función `rep()`, para repetir un valor de forma continua por un número determinado de veces.

```
# Factor de nivel educativo
nivel_educativo <- factor(c(rep("Media", 90), rep("Universitaria", 70), rep("Postgrado
```

Como cada variable está por su lado, crearemos una base de datos (i.e., dataframe) para unirlas todas en un mismo conjunto, a través de la función `data.frame()`. Esto, claro, asumiendo que cada una de las respuestas a todas las variables no sólo tienen la misma cantidad total de sujetos, sino que también cada sujeto ocupa la misma posición en el listado de datos.

```
# Crear el data frame
datos_sim01 <- data.frame(
               id = id_vec,
               edad = edades,
```

```
                altura_cm = mx_altura_peso[, "Altura_cm"],
                peso_kg = mx_altura_peso[, "Peso_kg"],
                educacion = nivel_educativo
)

# Imprimir el data frame
head(datos_sim01)
```

```
##     id edad altura_cm peso_kg educacion
## 1 id 1   50       180      65     Media
## 2 id 2   34       164      72     Media
## 3 id 3   33       200      82     Media
## 4 id 4   22       163      89     Media
## 5 id 5   56       152      89     Media
## 6 id 6   33       191      59     Media
```

4.3.1 Simulación de Variables Correlacionadas

En este ejemplo, generaremos ocho variables simuladas divididas en dos bloques. Las variables dentro de cada bloque tendrán correlaciones más altas entre sí, mientras que las correlaciones entre bloques serán más bajas. Esto debiese facilitar (si no asegurar) la emergencia de dos factores en un análisis factorial. Posteriormente, las incluiremos en la base de datos creada.

Definimos las dimensiones y las matrices de correlación.

```
pacman::p_load(MASS, psych, misty)

# Parámetros generales
n <- n_sim  # Número de casos
num_bloque <- 4  # Número de variables por bloque

# Correlaciones dentro de los bloques
cor_alta <- 0.75

# Correlación baja entre elementos de distintos bloques
cor_baja <- 0.25
```

Creamos la matriz de correlaciones, y la observamos para asegurarnos de que hemos creado los datos correctamente.

```
# Crear matriz base de correlación
cor_matrix <- matrix(cor_baja, nrow = 8, ncol = 8)
```

```r
# Ajustar las correlaciones dentro de cada bloque
cor_matrix[1:4, 1:4] <- cor_alta
cor_matrix[5:8, 5:8] <- cor_alta

# Asegurar que la diagonal sea 1 (varianza de cada variable)
diag(cor_matrix) <- 1

# Imprimir la matriz de correlación
cor_matrix
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
## [1,] 1.00 0.75 0.75 0.75 0.25 0.25 0.25 0.25
## [2,] 0.75 1.00 0.75 0.75 0.25 0.25 0.25 0.25
## [3,] 0.75 0.75 1.00 0.75 0.25 0.25 0.25 0.25
## [4,] 0.75 0.75 0.75 1.00 0.25 0.25 0.25 0.25
## [5,] 0.25 0.25 0.25 0.25 1.00 0.75 0.75 0.75
## [6,] 0.25 0.25 0.25 0.25 0.75 1.00 0.75 0.75
## [7,] 0.25 0.25 0.25 0.25 0.75 0.75 1.00 0.75
## [8,] 0.25 0.25 0.25 0.25 0.75 0.75 0.75 1.00
```

Ahora definiremos arbitrareamente parámetros para simular datos que reproduzcan una matriz de correlaciones similar a la establecida previamente, como unas medias para las ocho variables, y un rango (aquí, 0 a 5).

```r
# Medias y matriz de covarianza
medias <- rep(2.5, 8)  # Medias para las 8 variables
mini <- 0
maxi <- 5
```

La función `mvrnorm()` nos permitirá crear una distribución normal multivariada, bajo ciertos parámetros que hemos definido.

```r
set.seed(123)

# Generar datos
datos_sim02 <- mvrnorm(n = n, mu = medias, Sigma = cor_matrix)
## aseguramos que ningún dato baje del mínimo
datos_sim02 <- ifelse(datos_sim02 < 0, 0, datos_sim02)
## aseguramos que ningún dato supere el máximo
datos_sim02 <- ifelse(datos_sim02 > 5, 5, datos_sim02)
## transformamos todos los valores creados a números enteros
datos_sim02 <- round(datos_sim02, 0)
## Chequeamos valores creados
table(datos_sim02)
```

```
## datos_sim02
##   0   1   2   3   4   5
##  33 185 578 559 220  25
```

```
# Convertir a un data.frame
datos_sim02 <- as.data.frame(datos_sim02)

# Nombrar las variables
names(datos_sim02) <- paste0("var", 1:8)

# Imprimir primeros casos para chequear que todo ande ok
head(datos_sim02)
```

```
##   var1 var2 var3 var4 var5 var6 var7 var8
## 1    4    5    4    4    2    2    1    1
## 2    3    3    4    3    2    2    2    1
## 3    2    1    1    1    2    1    2    1
## 4    3    3    3    2    2    2    2    2
## 5    2    2    2    2    3    3    2    2
## 6    1    1    1    1    1    1    2    1
```

Prosigamos a estimar la matriz de correlación observada para estos datos, a través de la función `cor()`. Dado que es esperable que los resultados produzcan índices con múltiples decimales, haremos la matriz más simple visualmente redondeando los decimales a dos, a través del uso de `round()`.

```
# Matriz de correlación de los datos generados
round(cor(datos_sim02),2)
```

```
##      var1 var2 var3 var4 var5 var6 var7 var8
## var1 1.00 0.70 0.64 0.68 0.22 0.24 0.18 0.15
## var2 0.70 1.00 0.65 0.69 0.21 0.23 0.16 0.11
## var3 0.64 0.65 1.00 0.66 0.21 0.23 0.16 0.11
## var4 0.68 0.69 0.66 1.00 0.15 0.23 0.18 0.16
## var5 0.22 0.21 0.21 0.15 1.00 0.65 0.67 0.65
## var6 0.24 0.23 0.23 0.23 0.65 1.00 0.62 0.66
## var7 0.18 0.16 0.16 0.18 0.67 0.62 1.00 0.69
## var8 0.15 0.11 0.11 0.16 0.65 0.66 0.69 1.00
```

4.4 Organizar y exportar base de datos

Una vez ya tenemos todos estos valores, juntaremos las dos bases de datos que tenemos hasta el momento, `datos_sim01` y `datos_sim02`, en una misma base

`datos_sim`. La función `cbind()` permite precisamente esto, al unir columnas para dos conjuntos de datos cuyas filas son exactamente las mismas.

```
datos_sim <- cbind(datos_sim01, datos_sim02)
head(datos_sim)
```

```
##     id edad altura_cm peso_kg educacion var1 var2 var3 var4 var5 var6 var7 var8
## 1 id 1   50       180      65     Media    4    5    4    4    2    2    1    1
## 2 id 2   34       164      72     Media    3    3    4    3    2    2    2    1
## 3 id 3   33       200      82     Media    2    1    1    1    2    1    2    1
## 4 id 4   22       163      89     Media    3    3    3    2    2    2    2    2
## 5 id 5   56       152      89     Media    2    2    2    2    3    3    2    2
## 6 id 6   33       191      59     Media    1    1    1    1    1    1    2    1
```

Ahora, antes de proseguir a crear nuestro libro de código y exportar los datos, asignaremos una descripción breve a cada variable, de modo que podamos comunicar de qué se trata cada variable, más allá de los códigos que las identifican en la base de datos, los que usualmente no resultan demasiado informativos.

```
# Etiquetas variables

var.labels  <- c(
id="identificador unico",
edad ="edad en años",
altura_cm ="altura en centrimetros",
peso_kg="peso en kilos",
educacion="nivel educativo",
var1="actitud 1",
var2="actitud 2",
var3="actitud 3",
var4="actitud 4",
var5="actitud 5",
var6="actitud 6",
var7="actitud 7",
var8="actitud 8")

# Incluir etiquetas a la base de datos

Hmisc::label(datos_sim) <- as.list(var.labels[match(names(datos_sim), names(var.labels)

rm(var.labels) # elimina el objeto
```

4.4.1 Libro de código

Para esta sección, debemos asegurarnos de haber cargado los siguientes paquetes.

```
pacman::p_load(codebookr, tibble, sjlabelled, dplyr, xtable, gridExtra, openxlsx)
```

Una alternativa relativamente completa es a través de la función `codebook()`, del paquete `codebookr`. Esta produce un documento al cual se le puede añadir bastante información, incluyendo una descripción de los datos.

Primero se crea un objeto codebook

```
codebook01 <- codebookr::codebook(
  df = datos_sim,
  title = "Datos sim",
  subtitle = "Datos simulados para el taller de introducción en R 2025",
  description = "La base de datos simulados DATOS SIMULADOS es un producto de los procesos realiz
)
```

Una vez contamos con el objeto codebook en nuestro entorno, podemos "imprimirlo" en un documento. Por ejemplo, el siguiente código lo exportará a un documento word.

```
print(codebook01, "../03_output/codebook01.docx")
```

Es posible que los libros de código creados por la función `codebook()` contengan más información de la que, para nuestros usos, consideramos necesaria. En R, es simple hacer libros de código más sencillos a través de la creación de tablas ad-hoc. Para esto es necesario haber incluido etiquetas (labels) a las variables; Cosa que ya hicimos con la función `label()`.

```
# extract labels from dataframe and store as new object
codebook02 <- tibble::enframe(get_label(datos_sim))
# use more informative column names
colnames(codebook02) <- c("Variable", "Contenido")
# Show the new data frame
codebook02
```

```
## # A tibble: 13 x 2
##    Variable  Contenido
##    <chr>     <chr>
##  1 id        identificador unico
##  2 edad      edad en años
```

```
##  3 altura_cm altura en centrimetros
##  4 peso_kg   peso en kilos
##  5 educacion nivel educativo
##  6 var1      actitud 1
##  7 var2      actitud 2
##  8 var3      actitud 3
##  9 var4      actitud 4
## 10 var5      actitud 5
## 11 var6      actitud 6
## 12 var7      actitud 7
## 13 var8      actitud 8
```

```
descriptives <- datos_sim %>% psych::describe() %>% as_tibble() %>% select("n","min","r

# add stats to codebook
codebook02 <- cbind(codebook02,descriptives)
codebook02$min <- round(codebook02$min, 2)
codebook02$max <- round(codebook02$max, 2)
codebook02$mean <- round(codebook02$mean, 2)
```

```
# HTML codebook
colnames(codebook02) <- c("ID variable", "Contenido", "n", "min", "max", "media")
print(xtable(codebook02[-1, ]), type="html", file="../03_output/codebook_html.html")

# PDF codebook
pdf("../03_output/codebook_pdf.pdf", height=12, width=7)
grid.table(codebook02)
dev.off()

# excel codebook
write.xlsx(codebook02, file = "../03_output/codebook_xl.xlsx", colNames = TRUE)
```

```
## pdf
##   2
```

4.4.2 Exportar datos

Para guardar los datos que hemos creado, utilizaremos la función `write_sav()`, que producirá un archivo .sav (de SPSS). Para establecer el directorio, usamos `../` para "retroceder" una carpeta. Es decir, dado que este código está (o debiese estar) ubicado en la carpeta `02_process`, retornaremos a la carpeta general del proyecto, y luego entraremos a la carpeta `01_input`. Luego de ejecutado el código, debiesemos poder observar la aparición de un archivo llamado `datos_sim.sav` en dicha carpeta. La dejaremos ahí para futuros ejemplos y ejercicios.

```r
## datos de SPSS
haven::write_sav(datos_sim, "../03_input/datos_sim.sav")
## datos de R
saveRDS(datos_sim, file = "../03_input/datos_sim.rds")
## datos de excel
openxlsx::write.xlsx(codebook02, file = "../03_input/datos_sim.xlsx", colNames = TRUE)
```

Chapter 5

Cargar datos

R facilita la importación de datos desde una variedad de formatos, como **CSV**, **Excel**, **TXT** y **RDS**. Existen funciones específicas para cada tipo de archivo, que permiten cargar los datos y manipularlos de manera directa en el entorno de trabajo de R.

5.1 Formato CSV

El formato **CSV** (valores separados por comas) es uno de los más comunes para almacenar datos tabulares. Para cargar un archivo CSV, se usa la función `read.csv()` o `read_csv()` del paquete `readr` (que es más rápida para archivos grandes):

5.2 Formato Excel

Para archivos Excel (.xls o .xlsx), R requiere el paquete readxl. La función `read_excel()` permite especificar la hoja que se quiere importar:

5.3 Formato TXT

Los archivos TXT suelen estar delimitados por tabulaciones o espacios. Se utiliza `read.table()` o `read_delim()` (de `readr`), donde se especifica el delimitador:

5.4 Formato RDS

El formato RDS es específico de R y se usa para almacenar objetos R de manera directa. Para cargar archivos `.rds`, se usa `readRDS()`:

5.5 Formato SPSS

Para cargar archivos SPSS (`.sav`), R usa el paquete `haven`. La función `read_sav()` permite importar estos archivos de manera directa, manteniendo las etiquetas de los valores si están disponibles:

Opciones Comunes de Importación:
- `header:` En `read.csv()` y `read.table()`, `header = TRUE` indica que la primera fila es un encabezado de columna.
- `sep` y `delim`: Permiten especificar el delimitador en archivos de texto (`sep = ","` para CSV y `sep = "\t"` para TXT).
- `sheet:` En `read_excel()`, permite seleccionar una hoja específica del archivo Excel.

Estas funciones de carga de datos permiten trabajar con archivos de los formatos más comunes, simplificando el proceso de análisis y manipulación en R.