

Introducción a R

Sesión 1: Panorama general de R

Jorge de la Vega Góngora

Instituto Tecnológico Autónomo de México

Sábado 21 de enero de 2023

- 1 Introducción
 - ¿Qué es R?
 - Instalación de R y RStudio
 - Instalación de R
 - Instalación de Rstudio
 - Instalación de paquetes
 - Primeros pasos
 - Generar un documento reproducible

- 2 Práctica con R
 - Cómo se usa R
 - Ejemplos de aplicaciones de R

- 3 Estructuras de datos
 - Tipos de objetos en R
 - Vectores
 - Matrices
 - Arreglos
 - Listas
 - Tipos de listas: Dataframes
 - Formas de Indexación

Introducción

- El lenguaje `s` se desarrolló en Laboratorios AT & T-Bell, principalmente por [John M. Chambers](#), en 1976. Participan también Rick Becker y Allan Wilks.
- Su uso se expandió rápidamente después de la publicación del libro de John Tukey: “Exploratory Data Analysis” (EDA).
- `S-Plus` fue una versión comercial de `s` que inició en 1987. Su popularidad incrementó dramáticamente después de 1990, y se mantuvo hasta la versión 8 en 2007.
- `S-Plus` fue atractivo porque contaba con una interfaz de usuario gráfica (GUI), y soportaba muchos formatos para importación y exportación de datos y gráficas.

Stanford | Statistics SCHOOL OF HUMANITIES & SCIENCES

[News & Events](#)[People](#)[Academic Programs](#)[Admissions](#)[Industrial Affiliates](#)[Resources](#)[About](#)

People

[Faculty](#)[Stein Fellows](#)[Research Scientists & Lecturers](#)[Postdocs](#)[Students & Alumni](#)[Staff](#)

John Chambers

Adjunct Professor of Statistics

Mailing Address:

Department of Statistics
Sequoia Hall
390 Serra Mall
Stanford University
Stanford, CA 94305

Email: jmc@stat.stanford.edu

- R comenzó su desarrollo en los 90's, realizado por estadísticos como una alternativa de código abierto a S-Plus, en parte porque entonces no había una versión de S-Plus para Linux.
- Robert Gentleman y Ross Ihaka, de Nueva Zelanda, son los creadores pioneros de R.



- R se basa en el mismo lenguaje S utilizado en la versión 2000 de S-Plus, con algunas excepciones menores. Sin embargo, su arquitectura es un poco diferente.
- Ventajas: bien documentado; fácil de obtener e instalar; fácil de actualizar sus bibliotecas de funciones.
- Algunas desventajas de aquella época:
 - GUI muy básica (hasta la aparición de RStudio)

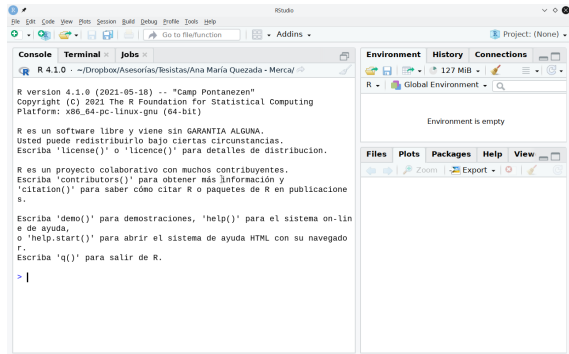
- manejo de memoria RAM ineficiente
- menos capacidades para importar y exportar datos de otros formatos
- no mucha interacción con Office

Estas desventajas se han ido resolviendo con el tiempo.

RStudio es una herramienta que facilita el trabajo con R, sobre todo en los siguientes puntos:

- Trabaja con R y sus gráficas de manera interactiva
- Permite organizar sus programas y tener organizados varios proyectos
- Permite realizar *investigación reproducible*, que esta tomando relevancia en el mundo científico
- Da mantenimiento a los paquetes instalados en el sistema
- Permite crear y compartir reportes
- Permite el intercambio de programas y la colaboración con otros usuarios
- Se puede llevar un control de proyectos en la nube conectando con github

RStudio es tanto una interfaz gráfica (GUI) como un ambiente de desarrollo integrado (IDE). Hay versiones para Windows, Linux y Mac OS X. También permite correr R desde un servidor remoto.



¿Porqué se promueve el uso de R? I

- R es gratuito con un lenguaje poderoso orientado a objetos.
- Permite análisis de datos y gráficas en forma interactiva y reproducible
- Es fácil implementar nuevos métodos y distribuir a otros usuarios
- Su programación y código son abiertos: siempre puedes saber qué estas haciendo
- Investigadores de punta en estadística son programadores de R y cuenta con una muy amplia y extensa comunidad a nivel global
- Millones de recursos (libros, tutoriales, videos, papers, etc.) disponibles de manera gratuita y de fácil acceso. Un Congreso anual de usuarios [UseR!](#).
- R fue planeado para ser extensible
 - Usuarios escriben nuevas funciones, al igual que los desarrolladores.
 - La documentación para agregar funciones es excelente.
 - Las funciones creadas por los usuarios se invocan igual que las internas.
 - Usuarios pueden crear sus propios tipos de datos y agregar atributos, p.ej. comentarios a cada pieza de datos de R.
- R permite trabajar tanto con datos estructurados como no estructurados
- Es un ambiente para análisis estadístico y también un lenguaje de alto nivel: unos cuantos comandos hacen mucho trabajo.

- R permite crear las mejores gráficas científicas.
- En R ya hay varios paquetes, contribuidos por diferentes personas, que permiten aplicar eficiente y correctamente muchos métodos estadísticos.
- Permite concentrarse más en la interpretación de los resultados que en la implementación.

Desarrollo de R en los últimos años

- La comunidad de usuarios de R ha crecido significativamente en los últimos años, en parte por la popularidad de la ciencia de datos, y en parte por la difusión que se le ha dado por su fácil acceso.
- Hadley Wickham** ha contribuido de manera importante a su difusión, con la elaboración de paquetes (ggplot2, tidyverse, dplyr, lubridate, ...) que facilitan el manejo de datos.

The screenshot shows the GitHub profile of Hadley Wickham. At the top, there's a navigation bar with links like Product, Team, Enterprise, Explore, Marketplace, and Pricing. Below this, the profile header includes a circular profile picture of Hadley Wickham, his name 'Hadley Wickham', and his handle 'hadley'. To the right of the profile picture, there are tabs for Overview, Repositories (176), Projects, Packages, Stars (156), and Sponsoring (3). The 'Overview' tab is selected. Below the header, there's a 'Pinned' section with four repositories: tidyverse/ggplot2, tidyverse/dplyr, tidyverse/tidyverse, and tidymodels. Each repository card shows the repository name, a brief description, and statistics like stars and forks. At the bottom, there's a section titled '4,490 contributions in the last year' with a calendar heatmap showing activity from August 2019 to August 2021. The heatmap shows a high density of contributions, particularly in the first half of each year.

- También se han creado comunidades que facilitan, que promueven ambientes seguros para los y las programadoras, como **r-ladies** y que organizan seminarios frecuentemente.

R tiene muchos fans, casi fanáticos



¿Porqué NO R?

No todo es maravilloso...

- Para utilizar R a su máximo potencial, la curva de aprendizaje es algo elevada.
- Se requiere, eventualmente, saber programar.
- El uso de paquetes está condicionado al menos parcialmente a los caprichos de su creador: puede decidir cuando cambiar cosas, dejar de dar mantenimiento, etc.
- Es posible que en el software/paquetes haya errores aunque usualmente se detectan, pero **no siempre**.
- Muchas empresas no permiten el uso de *Open Source* para sus procesos. Prefieren pagar por “soporte”, (aunque en la práctica no sirva de nada).
- Es lento para algunas aplicaciones por su estructura, aunque hay maneras de hacerlo más eficiente.

Actualmente, un lenguaje prometedor complementario/sustituto es **Julia**



- ➊ Primero hay que instalar R.
- ➋ Instalar RStudio
- ➌ Instalamos los paquetes complementarios que necesitemos para nuestro trabajo desde RStudio.
- ➍ Creamos un proyecto de trabajo para cada uno de los temas que queramos trabajar en R. Estos se guardarán en sus respectivos directorios.

- Para Windows y Mac: Seguir las instrucciones indicadas en www.r-project.org¹.
- En Linux, cada versión puede tener instrucciones diferentes. Para Ubuntu y sus variedades, se puede consultar [esta liga](#). Para Ubuntu 21.04 en adelante seguir los pasos en una terminal (como root):

```
apt update # actualiza los índices
# instala dos paquetes auxiliares necesarios
apt install --no-install-recommends software-properties-common dirmngr
# añade la firma digital para los repositorios (de Michael Rutter)
# Para verificar llave, correr: gpg --show-keys/etc/apt/trusted.gpg.d/cran_ubuntu_key.asc
# Huella digital: 298A3A825COD65DFD57CBB651716619E084DAB9
wget -q0- https://cloud.r-project.org/bin/linux/ubuntu/marutter_pubkey.asc | sudo tee -a
/etc/apt/trusted.gpg.d/cran_ubuntu_key.asc
# agrega el repo R 4.0 de CRAN -- ajusta 'focal' a 'groovy' o 'bionic' como sea necesario
add-apt-repository "deb https://cloud.r-project.org/bin/linux/ubuntu $(lsb_release -cs)-cran40/"
apt install --no-install-recommends r-base
```

- Puede haber varias versiones de R instaladas en paralelo (cada una tiene su directorio).

¹Alternativamente, se puede instalar una versión de R mejorada por Microsoft de <https://mran.microsoft.com>

- La versión para cualquiera de los sistemas operativos, se obtiene el ejecutable de:
<https://www.rstudio.com/products/rstudio/download/>
- Una vez obtenido el archivo, lo ejecutamos y seguimos las instrucciones que se indiquen.

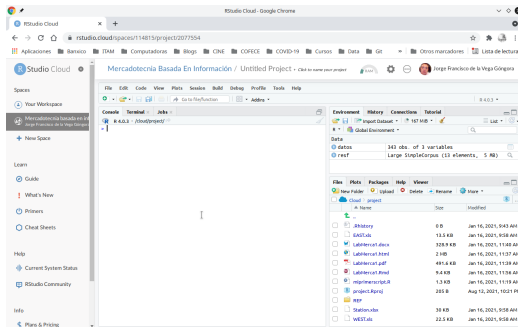
Ejercicio 1

Instalar R y RStudio en sus computadoras personales.

- Hay una gran cantidad de paquetes (*libraries*) que son colecciones de funciones empaquetadas que tienen un propósito definido. Los paquetes se publican en un repositorio conocido como **CRAN** (Comprehensive R Archive Network).
- Algunos paquetes en realidad son colecciones de otros paquetes, o bien algunos paquetes dependen de otros.
- Ejemplos de paquetes útiles que usaremos:
 - knitr
 - rmarkdown
 - tidyverse
 - dplyr
 - forecast
- Los paquetes se pueden instalar fácilmente desde RStudio a través de menús, o bien directamente desde la consola de R:

```
install.packages(c('knitr', 'rmarkdown', 'tidyverse', 'dplyr', 'forecast'))
```


- Es posible usar una versión de RStudio en línea, a través de <https://rstudio.cloud/>.



- Se puede usar una versión limitada en memoria y espacio de manera gratuita. Para iniciar el aprendizaje puede ser suficiente, pero en proyectos grandes se requiere más espacio.
- La ventaja es que se puede usar sin problemas de instalación y/o configuración. Todo fluye sin problemas.

Ejercicio 2: Prueba de instalación y primer documento

- Abrir RStudio
- File → New File → R Markdown...
- En la ventana que aparece, Poner título
- Esto crea una plantilla en markdown^a. Guardar con Save as... y poner un nombre con la extensión .Rmd

^aMarkdown es un lenguaje parecido a html pero muchísimo más simple que permite dar formatos marcando el texto y luego usando un intérprete para cambiar el formato

Práctica con R

R se puede utilizar de varias maneras, dependiendo de la experiencia, conocimiento y necesidad de sistematización:

Consola De manera directa, a través de la consola de comandos

Gui A través de una interfaz de usuarios gráfica (GUI). En este tema, hay varias opciones, entre las más importantes están RStudio, Jupyter, RCommander, Rattle, Emacs.

Batch En modo *Batch*, a través de un ícono.

- Comandos pueden introducirse en más de una línea. el prompt que indica la continuación de una línea es el signo más: +
- Comandos múltiples pueden ser introducidos en una misma línea separados por punto y coma (;)
- Los comentarios inician con #.
- Espacios y tabuladores son ignorados excepto cuando están entre comillas.
- R hace distinción entre mayúsculas y minúsculas.
- Se pueden usar las teclas \uparrow o \downarrow para navegar entre los comandos que han sido tecleados previamente.
- Se puede obtener ayuda de una función, por ejemplo `sin`, usando la ayuda HTML o bien tecleando `?sin` o `help(sin)`.
- La combinación de teclas `Ctrl + L` limpia la consola.

Los comandos más elementales consisten de expresiones o asignaciones.

```
2 + 3

[1] 5

sqrt(3/4)/(1/3-2*pi^2)

[1] -0.04462697

# genera datos aleatorios con distribución normal.
x <- rnorm(100,sd=4); y <- runif(100,-2,5)
# Los siguientes comandos devuelven por default los primeros y últimos 6 datos respectivamente
head(x); tail(x)

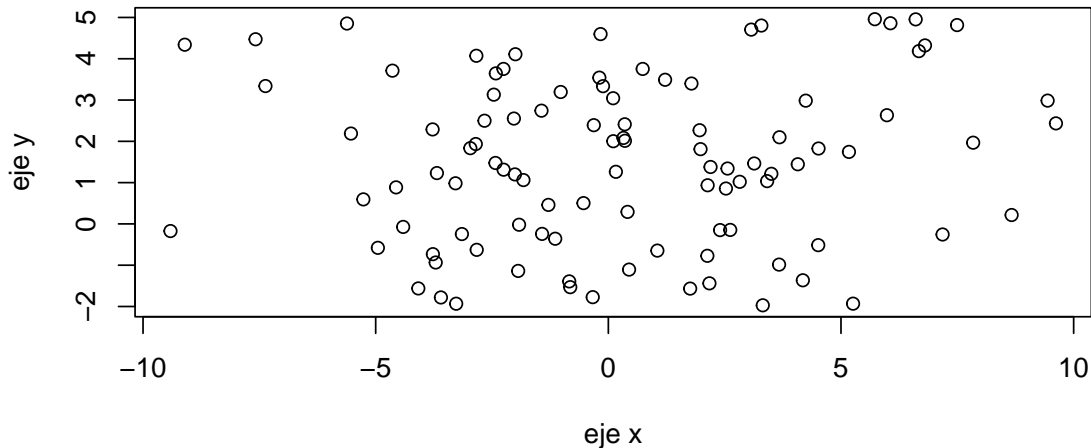
[1] 0.412726 1.981174 -2.250587 1.786312 7.494585 -2.462067
[1] -1.4365817 0.3253272 -2.4176699 2.8237030 0.3556914 4.0714533

head(x,10) # Se puede cambiar el número de datos que queremos ver.

[1] 0.4127260 1.9811737 -2.2505867 1.7863118 7.4945852 -2.4620672
[7] -3.7764199 -0.1675679 -1.1416535 -3.7706505

plot(x,y, main = "Mi primera gráfica\n ya soy expert!", xlab = "eje x", ylab = "eje y")
```

Mi primera gráfica ya soy expert!



Ejemplos I

```
z <- cbind(x,y) # 'pega' dos vectores por columnas
head(z)
```

```
      x      y
[1,] 0.412726 0.2912656
[2,] 1.981174 1.8106524
[3,] -2.250587 1.3133048
[4,] 1.786312 3.3997222
[5,] 7.494585 4.8157498
[6,] -2.462067 3.1304338
```

```
t(z) %*% z # producto de matrices
```

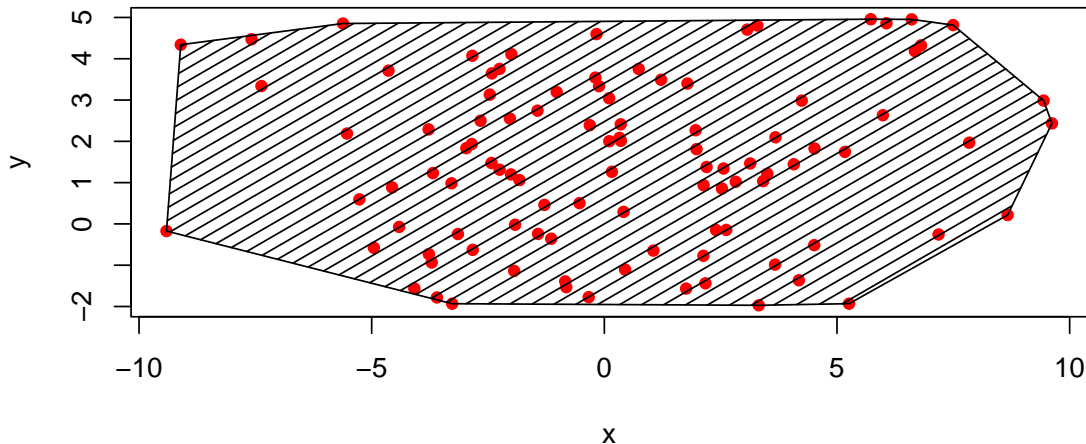
```
      x      y
x 1652.3539 124.4431
y 124.4431 616.0558
```

```
crossprod(z)
```

```
      x      y
x 1652.3539 124.4431
y 124.4431 616.0558
```

```
h <- chull(x,y) # cubierta conveza del conjunto de puntos
plot(x, y, pch = 16, col = "red", main = "Cubierta conveza del conjunto de puntos")
polygon(x[h], y[h], dens = 15, angle = 30)
```


Cubierta convexa del conjunto de puntos



Estructuras de datos

En esta sección se describirán los principales tipos de objetos que se usan comunmente en R. Estos son:

- Vectores
- Matrices
- Arreglos
- Listas
- Factores
- Dataframes
- Funciones
- Indexación

A continuación veremos las características de cada uno de estos tipos de objetos.

Vectores I

Un vector es un conjunto *ordenado* de datos que son **del mismo tipo base o clase** (no se pueden mezclar). Cada elemento de un vector se le llama *componente*. Las clases pueden ser:

- numéricos `numeric`. Estos pueden ser a su vez de subclases: `integer` o `double` o `complex`

```
(x <- c(2,4,6))  
[1] 2 4 6  
class(x)  
[1] "numeric"  
1:5 #El símbolo ":" es para indicar una sucesión  
[1] 1 2 3 4 5
```

- lógicos `logical`

```
(l <- c(TRUE,FALSE,TRUE,TRUE,FALSE))  
[1] TRUE FALSE TRUE TRUE FALSE  
class(l)  
[1] "logical"  
class(l) <- "integer" #Podemos cambiar la clase para representar los valores lógicos como 0's y 1's  
l  
[1] 1 0 1 1 0  
class(l) <- "logical" # y podemos regresarlos a su clase original.  
l  
[1] TRUE FALSE TRUE TRUE FALSE
```

● caracteres character

```
(a <- c("a", "b", 'c')) #Se pueden usar comillas dobles o sencillas, pero tienen que ser consistentes.
[1] "a" "b" "c"

(b <- c('a', 'b', "c"))
[1] "a" "b" "c"

class(b)
[1] "character"
```

Consideremos algunas propiedades y operaciones con los vectores.

- Cada componente de un vector se le puede asignar un nombre. El nombre es un *atributo* del vector.

```
x <- 1:5
colores <- c("rojo", "amarillo", "negro", "azul", "blanco")
names(x) <- colores # modifica el atributo de nombres
x
```

rojo	amarillo	negro	azul	blanco
1	2	3	4	5

```
attr(,"names") #revisa cuál es el atributo de nombres
[1] "rojo"      "amarillo"  "negro"     "azul"      "blanco"
```

- Otro atributo del vector es su longitud (`length`): es decir, el número de elementos que tiene el vector

Propiedades de los vectores II

```
length(x)
[1] 5

length(x) <- 8 # ¿Qué pasa si cambiamos la longitud del vector a una mayor?
x
  rojo amarillo   negro   azul   blanco
    1       2       3       4       5      NA      NA      NA

length(x) <- 3 # ¿a una menor longitud?
x
  rojo amarillo   negro
    1       2       3

length(x) <- 5 # se pierde información si lo regresamos a su tamaño? SI
x
  rojo amarillo   negro
    1       2       3      NA      NA
```

- Los vectores se pueden combinar. Cuando los vectores no son del mismo tipo, R cambia el tipo base al más general posible: esto se llama *coerción*. Esto es necesario para mantener el mismo tipo de datos.

```
x <- 1:5; y <- c("letras", "no números")
z <- c(x,y) # se combinan los vectores con la función de concatenación
z
[1] "1"      "2"      "3"      "4"      "5"
[6] "letras" "no números"
```

- Uso de índices para acceder componentes del vector

```
x[2]    #podemos tomar un elemento
[1] 2
x[1:3]  # o un subconjunto del vector.
[1] 1 2 3
x[10]   # cuando se usa un índice que no es parte del vector
[1] NA
x[c(1,1,3)] #los índices se pueden duplicar
[1] 1 1 3
```

o por nombre si lo tiene:

```
x["rojo"]
[1] NA
```

Para quitar elementos usamos un signo negativo:

```
x[-1]
[1] 2 3 4 5
x[-c(1,3)] # quita más de uno en diferentes posiciones.
[1] 2 4 5
x          # notar que no se modifica el vector con las operaciones anteriores, porque no están asignadas, sólo calculadas.
[1] 1 2 3 4 5
```


- Otras formas de crear vectores:

```
z <- numeric(10) # crea un vector de longitud 10, numérico. No tiene valores.
z
[1] 0 0 0 0 0 0 0 0 0 0

z <- character(10)
z
[1] "" "" "" "" "" "" "" "" "" ""

z <- logical(10)
z
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

z <- numeric(0) # define un vector sin elementos (vacío) es mejor que z <- NULL. Es mejor definir los objetos del tamaño final que tendr
z
numeric(0)
```

En ocasiones se puede hacer un vector vacío de longitud conocida. El propósito es hacer una 'reserva de memoria' de la computadora cuando estamos programando, y construimos el siguiente ejemplo para aprovechar el momento e introducir el concepto de ciclo `for` en un programa.

Ejercicio

Supongan que quieren distribuir a un número de estudiantes dado los problemas de una tarea al azar.

Para el ejercicio necesitamos extraer tantos números como estudiantes tenemos.

Ejemplo II

```
num_estudiantes <- 10 # podemos considerar a este valor como un parámetro
a <- numeric(num_estudiantes) # En este vector guardaremos el orden obtenido
A <- 1:num_estudiantes      # El conjunto de los números

for (i in 1:10){           # i es una variable auxiliar
  a[i] <- sample(A,1)      # Extraemos un número de A al azar con la función sample
  A <- setdiff(A,a[i])     # Actualizamos el conjunto para no repetir los números
}

a # vemos la solución

[1] 10 9 8 1 7 3 6 4 2 2

#El ejercicio anterior se puede hacer de manera más simple:
sample(A) # extrae una muestra sin reemplazo del mismo tamaño que el conjunto dado.

[1] 4 2 1 3 5
```

- Las operaciones aritméticas de los vectores se llevan componente a componente:

```
x <- c(1,4,5,2,10)
y <- c(3,1,2,5,6)
x + y
[1] 4 5 7 7 16
x - y
[1] -2 3 3 -3 4
x * y
[1] 3 4 10 10 60
x/y
[1] 0.3333333 4.0000000 2.5000000 0.4000000 1.6666667
```

- Si un vector se divide o multiplica por un número, cada componente del vector se multiplica/divide por ese número. Si un número se divide por un vector, se obtiene un vector el número dividido por cada componente

```
w <- 1:5
2*w
[1] 2 4 6 8 10
2/w
[1] 2.0000000 1.0000000 0.6666667 0.5000000 0.4000000
```

- **Regla del reciclaje:** ¿Qué pasa si intentamos hacer operaciones con vectores que tienen diferente longitud? Resulta la siguiente regla:

El vector más corto se reciclará para que tenga el tamaño del vector más grande

```
u <- c(10,20,30)
v <- 1:9
u + v
[1] 11 22 33 14 25 36 17 28 39
```

Hay que tener esta regla presente y tener mucho cuidado al usarla o no darse cuenta de que se está aplicando.

Factores

- Los objetos llamados *factores* son un tipo especial de vector que típicamente se utilizan para guardar variables que *clasifican* cosas. Estas variables tienen un significado especial en estadística, particularmente en el análisis de regresión y en el diseño de encuestas, como veremos más adelante en el diplomado.
- La característica importante de un factor es que etiquetan observaciones a pertenecer a cierto grupo o categoría. Por ejemplo, la religión, el sexo, el estado civil, el estado de la república, etc, son ejemplos de variables que pueden considerarse factores cuando se usan para separar grupos.
- Los *niveles* o etiquetas de los grupos no son relevante en sí, sólo el hecho de que distinguen a las observaciones en diferentes grupos. Se puede usar 'H' y 'M' o bien 0 y 1 o bien 1 y 0, para distinguir las observaciones de hombres y de mujeres.

Un tipo especial de vector: los *factores* II

```
# variable con los tipos de crédito: hip = hipotecario, TC = tarjeta de crédito, per = personal.
credito <- factor(c("hip","TC","per","per","hip","per","TC","TC"))
credito

[1] hip TC  per per hip per TC  TC
Levels: hip per TC
```

En un factor, R ordena Los niveles alfabéticamente. Algunas funciones le dan un valor especial al primer nivel, por lo que a veces se requiere dar explícitamente los niveles.

En ciertos casos, los grupos pueden tener tener un orden, por ejemplo, si se considera ingreso, se puede pensar en un ingreso bajo, medio o alto. En ese caso se dice que el factor es *ordinal*. Cuando los factores son ordinales, se puede indicar el orden deseado de las etiquetas:

```
ingreso <- ordered(c("M","B","M","A","M","B","B","A"), levels = c("B","M","A"))
ingreso

[1] M B M A M B B A
Levels: B < M < A
```

Una variable que es continua pero que se quiere separar en rangos para definir grupos (por ejemplo, el ingreso bajo puede ir de 0 a 1000 mensuales), se puede *discretizar* usando la función `cut`:

Un tipo especial de vector: los *factores* III

```
z <- rnorm(20)
z

[1]  1.69238662  0.45933701  0.51653447  0.10630044 -0.61102171  0.75661325
[7] -0.60701714  1.16408167 -0.92299776  0.54740665  0.08938035  1.04157043
[13]  2.19281034 -0.63971361  0.82607946  0.67921739 -0.36737674  0.02516756
[19]  0.55050528 -1.93212283

u <- cut(z,breaks = c(-4,-2,0,2,4)) #damos los puntos de corte
u <- ordered(u, levels(u))
u

[1] (0,2] (0,2] (0,2] (0,2] (-2,0] (0,2] (-2,0] (0,2] (-2,0] (0,2]
[11] (0,2] (0,2] (2,4] (-2,0] (0,2] (0,2] (-2,0] (0,2] (0,2] (-2,0]
Levels: (-4,-2] < (-2,0] < (0,2] < (2,4]

levels(u) <- c("B", "MB", "MA", "A") #Podemos reetiquetar los niveles para hacerlos más fáciles de leer o entender
```


Matrices

Una *matriz* es un conjunto de datos arreglados en un rectángulo en dos dimensiones (tiene renglones y columnas). Por ejemplo, la siguiente matriz tiene dos renglones y tres columnas:

$$\mathbf{A} = \begin{bmatrix} 2 & 4 & 3 \\ 1 & 5 & 7 \end{bmatrix}$$

También se puede pensar en una matriz como un vector con atributo de dimensión (`dim`). Entonces, de hecho, las matrices en \mathbb{R} son también vectores.

Veamos unos ejemplos:

Matrices II

```
x <- 1:20
x

[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

dim(x) <- c(2,10) #le asignamos dimensión al vector x
x

      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1    3    5    7    9   11   13   15   17   19
[2,]    2    4    6    8   10   12   14   16   18   20

dim(x) <- NULL #¿qué pasa si le quitamos su dimensión? También se puede usar c(x)
x

[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

Para crear una matriz, se puede usar la siguiente forma

```
A <- matrix(c(2,4,6,7,8,4,5,6,2,1), # los datos
            nrow = 2,                # el número de renglones
            ncol = 5),               # el número de columnas)
A

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

La matriz también se puede llenar por renglones usando índices para sus componentes

```
A <- matrix(  
  c(2,4,6,7,8,4,5,6,2,1),  
  nrow = 2,  
  ncol = 5,  
  byrow = T)           # Se indica que se llene por renglón
```

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

Los elementos de la matriz se pueden acceder usando:

Matrices IV

```
A[2,3] # El elemento en el renglón 2 y columna 3
[1] 6

A[2, ] # Todo el renglón 2, notar que es un vector
[1] 4 5 6 2 1

A[ ,3] # Toda la columna 3, notar que es un vector
[1] 6 6

A[,c(1,3)] # Columnas 1 y 3, es una sub-matriz
      [,1] [,2]
[1,]    2    6
[2,]    4    6
```

El número de renglones y columnas pueden obtenerse con las funciones `nrow` y `ncol`.

```
nrow(A)
[1] 2

ncol(A)
[1] 5
```

Podemos asignar nombres a los renglones y a las columnas de la matriz, y como en el caso de los vectores, podemos acceder a los elementos por los nombres:

```
dimnames(A) = list(           # La función para definir los nombres de las dimensiones, tiene que ser una lista.  
  c("r1", "r2"),             # nombres de renglones  
  c("c1", "c2", "c3", "c4", "c5")) # nombres de columnas
```

A

```
   c1 c2 c3 c4 c5  
r1  2  4  6  7  8  
r2  4  5  6  2  1
```

```
A["r2", "c3"] # elemento en el renglón 2 y columna 3
```

```
[1] 6
```

La *transpuesta* de una matriz intercambia los renglones y columnas. La notación matemática de la transpuesta de una matriz A es A^T o A' o A^t .

```
# No es necesario dar el número de columnas cuando tenemos la longitud  
# y el número de renglones:
```

```
B <- matrix(1:6,nrow=3)  
B
```

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

```
t(B)      # transpuesta de B
```

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

Así como con los vectores, si multiplicamos dos matrices que tengan las mismas dimensiones, la multiplicación se hace entrada a entrada:

B*B

	[,1]	[,2]
[1,]	1	16
[2,]	4	25
[3,]	9	36

Pero ese tipo de producto no es el *producto de matrices* que se utiliza en matemáticas. El producto matricial tiene una notación especial: `%*%` y sólo se puede hacer si el número de renglones de una matriz es igual al número de columnas de la otra matriz, es decir, una matriz es de dimensiones $r \times c$ y la otra debe ser $c \times q$.

B %*% t(B)

	[,1]	[,2]	[,3]
[1,]	17	22	27
[2,]	22	29	36
[3,]	27	36	45

Podemos *combinar* matrices si tienen el mismo número de renglones, o el mismo número de columnas para crear una matriz más grande:

```
C <- matrix(c(7,4,1), nrow = 3) #C tiene dimensiones 3x1
# Combinamos B y C por columnas (porque ambas tienen el mismo número de renglones:
cbind(C,B)
```

	[,1]	[,2]	[,3]
[1,]	7	1	4
[2,]	4	2	5
[3,]	1	3	6

```
cbind(B,C)
```

	[,1]	[,2]	[,3]
[1,]	1	4	7
[2,]	2	5	4
[3,]	3	6	1

Para combinar por renglones, usamos la función `rbind`

factores como matrices

En realidad, los objetos que llamamos factores, en realidad son matrices.

Arreglos

Los *Arreglos* son una extensión de las matrices, en donde se pueden agregar más dimensiones (dim) de longitud 3 o más.

Por ejemplo:

```
x <- 1:30
dim(x) <- c(5,2,3) # es como un 'cubo'
dimnames(x) <- list(color = colores, tipo = c("gordo", "flaco"), var = c("H", "M", "N"))
```

Los elementos se pueden acceder como en el caso de vectores.

```
x[3,2,1]
[1] 8
x[, ,1]  # caras del arreglo

      tipo
color   gordo flaco
rojo      1      6
amarillo  2      7
negro     3      8
azul      4      9
blanco    5     10

x[2, ,3]

gordo flaco
  22    27
```

La convención en R para *llenar* un arreglo es la siguiente: el primer índice es el que se ‘mueve’ más rápido y el último es el que se ‘mueve’ más lento.

Un arreglo sigue siendo un vector, pero con más dimensiones. Acomodar las cosas en dimensiones puede ser útil para *organizar la información*.

listas

Una *lista* es un objeto que sirve para contener otros objetos que pueden ser todos de diferentes tipos.

- Esta puede ser la estructura de datos más interesante para almacenar datos que se conocen como *no-estructurados*. El manejo de las listas puede ser un poco más difícil de entender, por la manera en que tenemos que acceder a sus elementos. Lo veremos a través de un ejemplo concreto
- Consideremos un crédito a una persona. El crédito tiene varias características importantes que se tienen que definir:
 - El tipo: puede ser hipotecario, tarjeta de crédito, personal, automotriz.
 - El monto del préstamo
 - La tasa de interés que tiene el crédito
 - Su plazo (en meses)
 - su fecha de contratación
 - datos demográficos de la persona que recibe el crédito (edad, sexo, estado civil).

Podemos definir una lista con todas estas características:

```
Credito <- list(tipo = "Hipotecario",  
               plazo = 28,  
               fecha = "01/04/20",  
               tasa = 7.58,  
               demograficos = c(35,1,0))
```

```
Credito
```

```
$tipo
```

```
[1] "Hipotecario"
```

```
$plazo
```

```
[1] 28
```

```
$fecha
```

```
[1] "01/04/20"
```

```
$tasa
```

```
[1] 7.58
```

```
$demograficos
```

```
[1] 35 1 0
```

- A los elementos de la lista se puede acceder ya sea por posición o por nombre:

```
Credito[[3]] #tercer objeto en la lista
[1] "01/04/20"

Credito[3] # lista con un componente
$fecha
[1] "01/04/20"

Credito$fecha #componente con nombre fecha
[1] "01/04/20"

Credito[3:4] #se seleccionan varios elementos como vector
$fecha
[1] "01/04/20"

$tasa
[1] 7.58
```

Se nos olvidó agregar el monto:

```
Credito <- c(Credito, monto = 1.2) #se agrega el monto del crédito, en millones
Credito

$tipo
[1] "Hipotecario"

$plazo
[1] 28

$fecha
[1] "01/04/20"

$tasa
[1] 7.58

$demograficos
[1] 35 1 0

$monto
[1] 1.2
```

Elimina el componente monto de la lista

```
Credito$monto <- NULL #o Credito[["monto"]] <- NULL
Credito

$tipo
[1] "Hipotecario"

$plazo
[1] 28

$fecha
[1] "01/04/20"

$tasa
[1] 7.58

$demograficos
[1] 35 1 0
```

- La función `unlist` convierte una lista en un vector, pero se aplica coerción para que todos los objetos tengan el mismo tipo, y cuando un objeto tiene una dimensión mayor, agrega cada componente numerado secuencialmente.

```
unlist(Credito)

      tipo      plazo      fecha      tasa demograficos1
"Hipotecario"    "28"    "01/04/20"    "7.58"          "35"
demograficos2 demograficos3
      "1"      "0"
```

- Un *dataframe* es una lista hecha de vectores de la misma longitud, pero pueden ser de diferente tipo. Es la estructura más adecuada para guardar datos estructurados y lo más cercano a lo que se conoce como una *base de datos*.
- Un dataframe tiene atributos que las listas no tienen. La función `data.frame` genera un dataframe. Se puede usar la función `cbind` o `rbind` como si los datos fueran una matriz, pero regresa un dataframe:

```
z <- data.frame(inst = factor(c("Cete", "Cete", "M0", "M1", "M1")),  
               precio = c(9.91, 9.93, 9.988, 9.87, 9.67),  
               ven = rep("01/10/04", 5))
```

z

	inst	precio	ven
1	Cete	9.910	01/10/04
2	Cete	9.930	01/10/04
3	M0	9.988	01/10/04
4	M1	9.870	01/10/04
5	M1	9.670	01/10/04

```
(z2 <- cbind(z, z))
```

	inst	precio	ven	inst	precio	ven
1	Cete	9.910	01/10/04	Cete	9.910	01/10/04
2	Cete	9.930	01/10/04	Cete	9.930	01/10/04
3	M0	9.988	01/10/04	M0	9.988	01/10/04
4	M1	9.870	01/10/04	M1	9.870	01/10/04
5	M1	9.670	01/10/04	M1	9.670	01/10/04


```
attributes(z)

$names
[1] "inst"  "precio" "ven"

$class
[1] "data.frame"

$row.names
[1] 1 2 3 4 5
```

- Los dataframes pueden ser indexados como matrices o como listas.

```
z[2] #es un dataframe
  precio
1  9.910
2  9.930
3  9.988
4  9.870
5  9.670

z[[2]] #es un vector, que es el segundo elemento de la lista
[1] 9.910 9.930 9.988 9.870 9.670

z[z$inst == 'Cete',] # selecciona sólo los instrumentos que son Cetes.
  inst precio    ven
1 Cete  9.91 01/10/04
2 Cete  9.93 01/10/04
```

- Matrices y listas pueden ser anexadas a un dataframe:

```
y <- matrix(rnorm(10),nrow=5)
y
```

```
      [,1]      [,2]
[1,]  2.1964428  0.96541438
[2,]  0.6804533 -1.19048827
[3,] -0.4286199 -0.47276874
[4,] -2.9890746 -0.01402013
[5,] -0.8803263 -0.64307828
```

```
z <- data.frame(z,y)
z
```

```
  inst precio    ven      X1      X2
1 Cete  9.910 01/10/04  2.1964428  0.96541438
2 Cete  9.930 01/10/04  0.6804533 -1.19048827
3  M0  9.988 01/10/04 -0.4286199 -0.47276874
4  M1  9.870 01/10/04 -2.9890746 -0.01402013
5  M1  9.670 01/10/04 -0.8803263 -0.64307828
```

Qué es la indexación?

La indexación consiste en extraer partes de un objeto en R, típicamente de alguna de las estructuras de datos.

Para vectores, los vectores índices pueden ser de 5 tipos:

- 1 Una proposición lógica que se puede valorar como verdadera o falsa para cada elemento del vector:

```
y
      [,1]      [,2]
[1,]  2.1964428  0.96541438
[2,]  0.6804533 -1.19048827
[3,] -0.4286199 -0.47276874
[4,] -2.9890746 -0.01402013
[5,] -0.8803263 -0.64307828

y < 0

      [,1] [,2]
[1,] FALSE FALSE
[2,] FALSE  TRUE
[3,]  TRUE  TRUE
[4,]  TRUE  TRUE
[5,]  TRUE  TRUE

y[y < 0]
[1] -0.42861990 -2.98907463 -0.88032630 -1.19048827 -0.47276874 -0.01402013
[7] -0.64307828
```

- 2 Un vector de enteros positivos o un factor:

```
z[z$inst == "M0", ]  
  
  inst precio      ven      X1      X2  
3   M0  9.988 01/10/04 -0.4286199 -0.4727687
```

- 3 Un vector de enteros negativos: sirve para *excluír* términos:

```
y[-c(2,5,8)]  
  
[1]  2.19644285 -0.42861990 -2.98907463  0.96541438 -1.19048827 -0.01402013  
[7] -0.64307828
```

- 4 Un vector de cadenas de caracteres: esto sólo aplica cuando el objeto tiene nombres. De otra forma, devuelve NA.
- 5 La posición de índice es vacía. `y[]`. Se comporta como si se reemplazara el índice por `1:length(y)`.
- 6 Un arreglo puede ser indexado por una matriz: si el arreglo tiene k índices, la matriz índice debe ser de dimensiones $m \times k$ y cada renglón de la matriz es usado como un conjunto de índices especificando un elemento del arreglo.

- 7 Índices ceros no caen en ningún caso anterior: un índice 0 en un vector ya creado pasa nada y un índice 0 en un vector al que se le asigna un valor no lo acepta:

```
a <- 1:4
a[0]

integer(0)

a[0] <- 10
a

[1] 1 2 3 4
```

Ejercicio 3

- R tiene algunos archivos de datos precargados para mostrar cómo se realizan algunos análisis de datos. En este caso, carguen el conjunto de datos que se llama `state.x77` que contiene algunas estadísticas de cada estado de los EUA, para el año 1977:

```
head(state.x77)
```

	Population	Income	Illiteracy	Life Exp	Murder	HS Grad	Frost	Area
Alabama	3615	3624	2.1	69.05	15.1	41.3	20	50708
Alaska	365	6315	1.5	69.31	11.3	66.7	152	566432
Arizona	2212	4530	1.8	70.55	7.8	58.1	15	113417
Arkansas	2110	3378	1.9	70.66	10.1	39.9	65	51945
California	21198	5114	1.1	71.71	10.3	62.6	20	156361
Colorado	2541	4884	0.7	72.06	6.8	63.9	166	103766

- Con los datos anteriores, hacer lo siguiente:
 - Calculen la media de analfabetismo y de asesinatos de ese año.
 - Ordenar los valores de menor a mayor de acuerdo al número de asesinatos, usando la función `sort`
 - Hagan una gráfica que muestre los datos de manera útil.
 - Identifiquen los estados que tienen una tasa de asesinatos mayor al 12 %.
 - Grafica los datos para tasa de analfabetismo y número de asesinatos.
 - Obten el subconjunto de datos que empiezan con "M".