Notas de R

Roberto Álvarez

2024-08-03

Índice

Capítulo 1

Introducción a R

1.1 Historia de R

R es un lenguaje de programación derivado de S.

¿Qué es S? Es un lenguaje de programación desarrollado por John Chambers (AT&T) en 1976, su función inicial era el análisis estadístico. Fue hasta 1998 que se lanzó la cuarta versión en *"Programming with Data"*, dicho libro documenta una versión muy similar a la que conocemos hoy en día, la cual incluye el análisis funcional estadístio. El lenguaje S tiene como raíz el análisis de datos, sus desarrolladores se enfocaron en contruir un lenguaje que resultara sencillo tanto para los ellos como para los usuarios, ¿Cómo? desarrollando un método de programación basado en líneas de comando.

Volviendo a R, se creó en 1991 por Ross Ihaka y Robert Gentleman. Fue en 1996 que se lanzó oficialmente: *"R: A lenguaje for data anlysis and graphics"*, siendo incialmente bastante similar al lenguaje S. Una gran ventaja de R fue su lanzamiento como software libre (la limitación clave de S fue su única disponibilidad como paquete comercial, S-PLUS), esto permitió que la fuente de código del sistema entero fuera accesible para cualquiera que decidiera emplearlo.

Al día de hoy R se puede emplear en casi cualquier plataforma de cómputo y sistema operativo, esto es gracias a su naturaleza *open source*, es decir, que cualquiera es libre de adaptar el software a la plataforma que desee.

Uno de los más grandes beneficios de R, no relacionado *per se* al lenguaje, es la comunidad de usuarios. Sus característicad de una comunidad muy activa, multidisciplinaria y de distintas partes del mundo ha permitido la construcción de una plataforma que tiene éxito en medida que las personas crean y apoyan el desarrollo de nuevas herramientas, paquetes, aplicaciones, así mismo como el apoyo a nuevos usuarios.

1.2 Ventajas de R

R es un entorno integrado para el manejo de datos, el cálculo, la generación de gráficos y análisis estadísticos. Las principales ventajas del uso de R son:

1. Software libre.
2. Facilidad para el manejo y almacenamiento de datos.
3. Un conjunto de operadores para el cálculo de vectores y matrices.
4. Una colección extensa e integrada de herramientas intermedias para el análisis estadístico de datos.
5. Multitud de facilidades gráficas de altísima calidad.
6. Un lenguaje de programación (muy) poderoso con muchas librerías especializadas disponibles. CRAN tiene aproximadamente 10k paqueterías disponibles, muchas más que el número de funciones de Excel.
7. La mejor herramienta para trabajar con datos genómicos, proteómicos, redes, metabolómica, entre varias más.
8. **Casi todos podemos aprender por nuestra cuenta a usar excel (pero hay que pagar por la licencia, es software privativo...). Sin embargo, aunque es más difícil aprender por nuestra cuenta R, si lo hacemos nos da una ventaja comparativa sobre el resto de estudiantes de licenciaturas afines.**
9. R tiene la capacidad de relacionarse y trabajar de manera paralela con otros software (Microsoft Office, QGIS..). Algunas ventajas de R sobre, por ejemplo, la paquetería Office, son su capacidad de iteración, reproducibilidad, automatización, la generación de reportes dinámicos, múltiples formatos de salida (PDF, HTML, páginas de internet, artículos científicos, diapositivas), conexión directa con buscadores de internet.

1.3 Learning R Hub

En este sitio web se presentan varios recurso adicionales en línea para aprender R

Learning R Hub

1.4 Introducción a R para Estudiantes de Microbiología

R es un lenguaje de programación y un entorno de desarrollo estadístico ampliamente utilizado en la comunidad científica, incluida la microbiología. Este poderoso recurso ofrece una variedad de ventajas y funcionalidades que pueden beneficiar significativamente a los estudiantes de microbiología en su investigación y análisis de datos.

1.4.1 Ventajas de Aprender R para Estudiantes de Microbiología

1. Análisis Estadístico Avanzado

R proporciona una amplia gama de paquetes y herramientas estadísticas que permiten a los estudiantes de microbiología realizar análisis avanzados de datos, desde pruebas de hipótesis básicas hasta modelos de regresión y análisis multivariados.

2. Visualización de Datos

Con paquetes como ggplot2, los estudiantes pueden crear visualizaciones de datos interactivas y de alta calidad que facilitan la comprensión de patrones y tendencias en conjuntos de datos microbiológicos, como datos de secuenciación genómica o datos de ecología microbiana.

3. Reproducibilidad y Documentación

RMarkdown es una herramienta poderosa que permite a los estudiantes escribir documentos reproducibles que combinan código, resultados y narrativa en un solo lugar. Esto promueve la transparencia, la reproducibilidad y una mejor documentación de los análisis microbiológicos.

4. Acceso a una Comunidad Activa

R cuenta con una comunidad activa de usuarios y desarrolladores que comparten código, paquetes y recursos educativos. Los estudiantes pueden aprovechar este recurso para buscar ayuda, colaborar en proyectos y mantenerse al tanto de las últimas tendencias en análisis de datos microbiológicos.

5. Flexibilidad y Personalización

R es un lenguaje altamente flexible que permite a los estudiantes adaptar sus análisis a las necesidades específicas de sus proyectos microbiológicos. Desde la manipulación de datos hasta la creación de modelos personalizados, R ofrece la libertad y la capacidad de personalización necesarias para abordar una amplia variedad de preguntas de investigación en microbiología.

Aprender R puede ser extremadamente beneficioso para los estudiantes de microbiología al proporcionarles las herramientas y habilidades necesarias para realizar análisis de datos sofisticados, visualizar resultados de manera efectiva y promover la reproducibilidad en su investigación científica.

1.5 Comienza con R

1.5.1 Instalación

Para iniciar en R es necesario instalarlo. R está disponible para los sistemas Windows, Mac OS X y Linux. El lenguaje de programación R tiene integrado

un ambiente de desarrollo (IDE, por sus siglas en inglés) llamado RStudio. Este IDE facilita la sintaxis y edición del código, así como la visualización de objetos.

Si requiere ver un tutorial de cómo instalar R y RStudio para Mac o Windows, puede apoyarse de los siguientes videos:

- Instalación para Windows
- Instalación para Mac

1.5.2 Paquetes o bibliotecas

Las funciones especializadas de R se guardan en paquetes *(packages)* que deben ser invocados ANTES de llamar a una función del paquete.

Una manera de instalar paquetes es mediante el repositorio CRAN.

Navega por CRAN y encuentra algunos paquetes que podrían interesarte. Hay miles y cada día aumentan.

Para saber qué paquetes se tienen instalados en tu máquina teclea la función `library()`

```
library()
```

Para cargar un paquete, que se encuentre previamente instalado, se debe teclear `library(nombre_de_paquete)`

Por ejemplo:

```
library(gplots)
```

Para visualizar los paquetes ya cargados, teclea:

```
search()
```

```
##  [1] ".GlobalEnv"             "package:ggpubr"
##  [3] "package:palmerpenguins" "package:ggplot2"
##  [5] "package:readxl"         "package:gplots"
##  [7] "tools:rstudio"          "package:stats"
##  [9] "package:graphics"       "package:grDevices"
## [11] "package:utils"          "package:datasets"
## [13] "package:methods"        "Autoloads"
## [15] "package:base"
```

Para visualizar las funciones dentro de un paquete en particular se utiliza:

```
ls(2)
```

```
##   [1] "%>%"                          "add_summary"
##   [3] "annotate_figure"              "as_ggplot"
##   [5] "as_npc"                       "as_npcx"
##   [7] "as_npcy"                      "background_image"
```

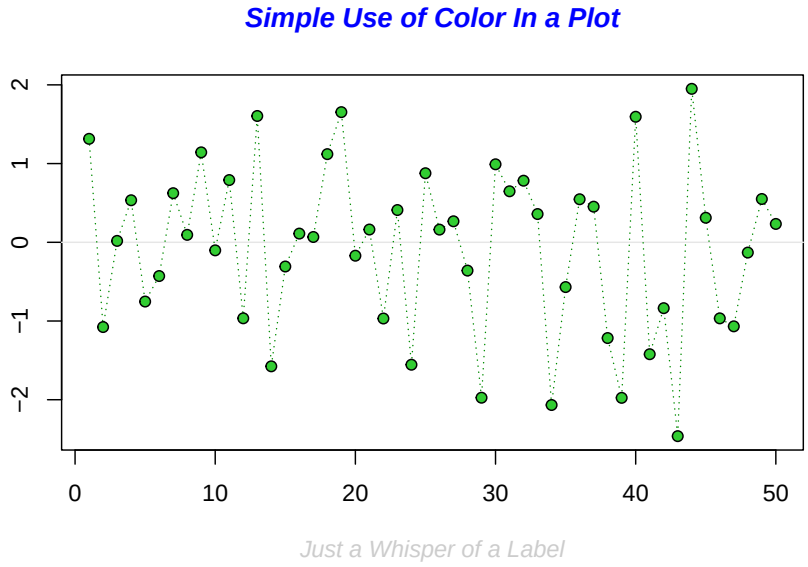
```
##   [9] "bgcolor"                 "border"
##  [11] "change_palette"          "clean_table_theme"
##  [13] "clean_theme"             "colnames_style"
##  [15] "color_palette"           "compare_means"
##  [17] "create_aes"              "desc_statby"
##  [19] "diff_express"            "facet"
##  [21] "fill_palette"            "font"
##  [23] "gene_citation"           "gene_expression"
##  [25] "geom_bracket"            "geom_exec"
##  [27] "geom_pwc"                "geom_signif"
##  [29] "get_breaks"              "get_coord"
##  [31] "get_legend"              "get_palette"
##  [33] "get_summary_stats"       "ggadd"
##  [35] "ggadjust_pvalue"         "ggarrange"
##  [37] "ggballoonplot"           "ggbarplot"
##  [39] "ggboxplot"               "ggdensity"
##  [41] "ggdonutchart"            "ggdotchart"
##  [43] "ggdotplot"               "ggecdf"
##  [45] "ggerrorplot"             "ggexport"
##  [47] "gghistogram"             "ggline"
##  [49] "ggmaplot"                "ggpaired"
##  [51] "ggpar"                   "ggparagraph"
##  [53] "ggpie"                   "ggpubr_options"
##  [55] "ggqqplot"                "ggscatter"
##  [57] "ggscatterhist"           "ggstripchart"
##  [59] "ggsummarystats"          "ggsummarytable"
##  [61] "ggtext"                  "ggtexttable"
##  [63] "ggviolin"                "gradient_color"
##  [65] "gradient_fill"           "grids"
##  [67] "group_by"                "labs_pubr"
##  [69] "mean_ci"                 "mean_range"
##  [71] "mean_sd"                 "mean_se_"
##  [73] "median_hilow_"           "median_iqr"
##  [75] "median_mad"              "median_q1q3"
##  [77] "median_range"            "mutate"
##  [79] "npc_to_data_coord"       "rotate"
##  [81] "rotate_x_text"           "rotate_y_text"
##  [83] "rownames_style"          "rremove"
##  [85] "set_palette"             "show_line_types"
##  [87] "show_point_shapes"       "stat_anova_test"
##  [89] "stat_bracket"            "stat_central_tendency"
##  [91] "stat_chull"              "stat_compare_means"
##  [93] "stat_conf_ellipse"       "stat_cor"
##  [95] "stat_friedman_test"      "stat_kruskal_test"
##  [97] "stat_mean"               "stat_overlay_normal_density"
##  [99] "stat_pvalue_manual"      "stat_pwc"
```

```
## [101] "stat_regline_equation"       "stat_stars"
## [103] "stat_welch_anova_test"       "tab_add_border"
## [105] "tab_add_footnote"            "tab_add_hline"
## [107] "tab_add_title"               "tab_add_vline"
## [109] "tab_cell_crossout"           "tab_ncol"
## [111] "tab_nrow"                    "table_cell_bg"
## [113] "table_cell_font"             "tbody_add_border"
## [115] "tbody_style"                 "text_grob"
## [117] "thead_add_border"            "theme_classic2"
## [119] "theme_cleveland"             "theme_pubclean"
## [121] "theme_pubr"                  "theme_transparent"
## [123] "ttheme"                      "xscale"
## [125] "yscale"
```

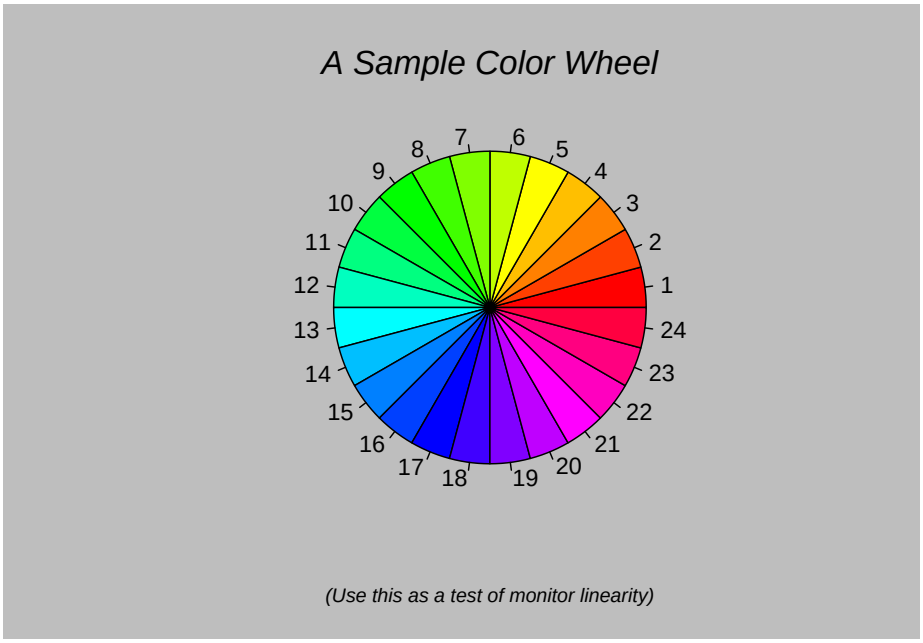
EJEMPLOS DE VISUALIZACIÓN DE GRÁFICOS

```
demo(graphics)
```

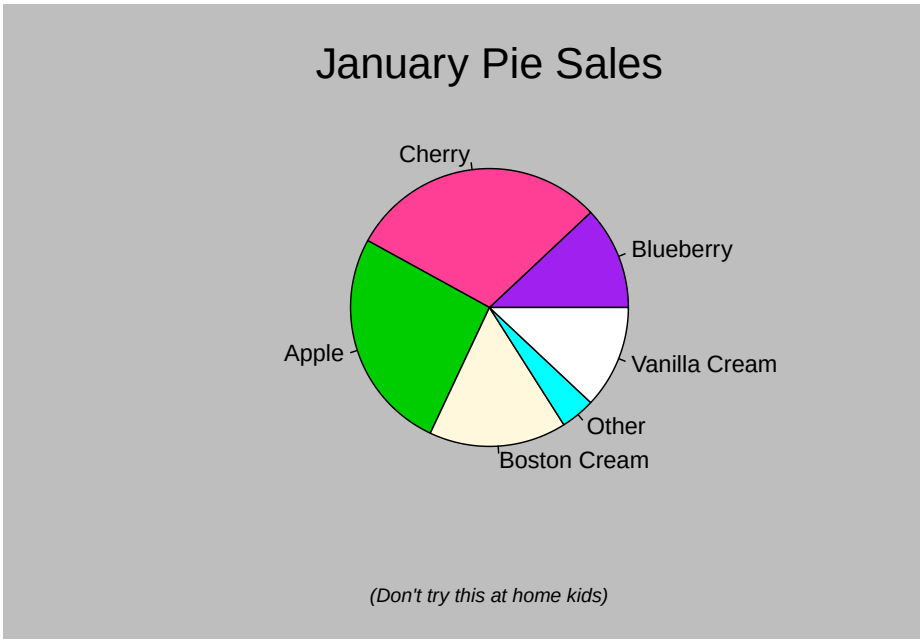
```
##
##
##  demo(graphics)
##  ---- ~~~~~~~~
##
## > #  Copyright (C) 1997-2009 The R Core Team
## >
## > require(datasets)
##
## > require(grDevices); require(graphics)
##
## > ## Here is some code which illustrates some of the differences between
## > ## R and S graphics capabilities.  Note that colors are generally specified
## > ## by a character string name (taken from the X11 rgb.txt file) and that line
## > ## textures are given similarly.  The parameter "bg" sets the background
## > ## parameter for the plot and there is also an "fg" parameter which sets
## > ## the foreground color.
## >
## >
## > x <- stats::rnorm(50)
##
## > opar <- par(bg = "white")
##
## > plot(x, ann = FALSE, type = "n")
```



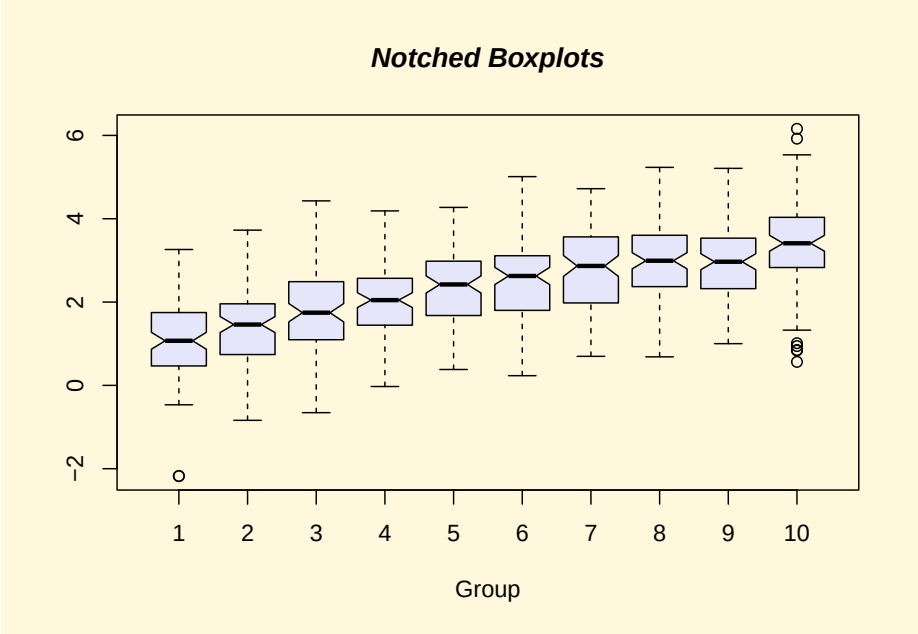

```
##
## > abline(h = 0, col = gray(.90))
##
## > lines(x, col = "green4", lty = "dotted")
##
## > points(x, bg = "limegreen", pch = 21)
##
## > title(main = "Simple Use of Color In a Plot",
## +       xlab = "Just a Whisper of a Label",
## +       col.main = "blue", col.lab = gray(.8),
## +       cex.main = 1.2, cex.lab = 1.0, font.main = 4, font.lab = 3)
##
## > ## A little color wheel.    This code just plots equally spaced hues in
## > ## a pie chart.    If you have a cheap SVGA monitor (like me) you will
## > ## probably find that numerically equispaced does not mean visually
## > ## equispaced.  On my display at home, these colors tend to cluster at
## > ## the RGB primaries.  On the other hand on the SGI Indy at work the
## > ## effect is near perfect.
## >
## > par(bg = "gray")
##
## > pie(rep(1,24), col = rainbow(24), radius = 0.9)
```



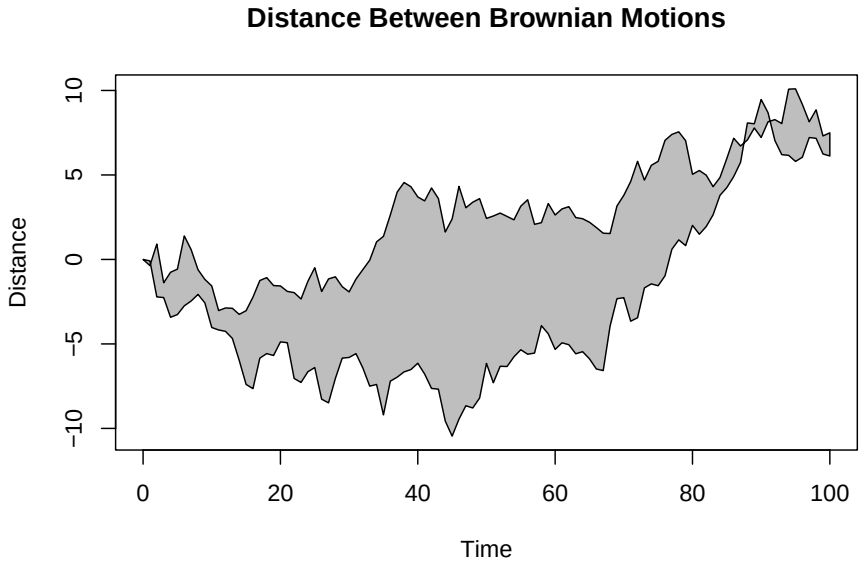

```
##
## > title(main = "A Sample Color Wheel", cex.main = 1.4, font.main = 3)
##
## > title(xlab = "(Use this as a test of monitor linearity)",
## +       cex.lab = 0.8, font.lab = 3)
##
## > ## We have already confessed to having these.  This is just showing off X11
## > ## color names (and the example (from the postscript manual) is pretty "cute".
## >
## > pie.sales <- c(0.12, 0.3, 0.26, 0.16, 0.04, 0.12)
##
## > names(pie.sales) <- c("Blueberry", "Cherry",
## +              "Apple", "Boston Cream", "Other", "Vanilla Cream")
##
## > pie(pie.sales,
## +     col = c("purple","violetred1","green3","cornsilk","cyan","white"))
```



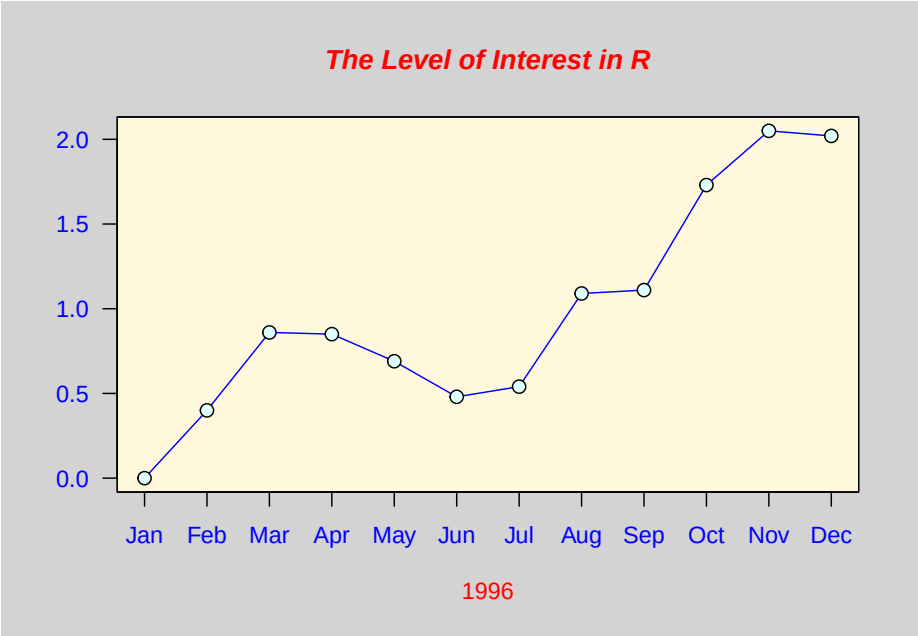

```
##
## > title(main = "January Pie Sales", cex.main = 1.8, font.main = 1)
##
## > title(xlab = "(Don't try this at home kids)", cex.lab = 0.8, font.lab = 3)
##
## > ## Boxplots:  I couldn't resist the capability for filling the "box".
## > ## The use of color seems like a useful addition, it focuses attention
## > ## on the central bulk of the data.
## >
## > par(bg="cornsilk")
##
## > n <- 10
##
## > g <- gl(n, 100, n*100)
##
## > x <- rnorm(n*100) + sqrt(as.numeric(g))
##
## > boxplot(split(x,g), col="lavender", notch=TRUE)
```




```
##
## > title(main="Notched Boxplots", xlab="Group", font.main=4, font.lab=1)
##
## > ## An example showing how to fill between curves.
## >
## > par(bg="white")
##
## > n <- 100
##
## > x <- c(0,cumsum(rnorm(n)))
##
## > y <- c(0,cumsum(rnorm(n)))
##
## > xx <- c(0:n, n:0)
##
## > yy <- c(x, rev(y))
##
## > plot(xx, yy, type="n", xlab="Time", ylab="Distance")
```

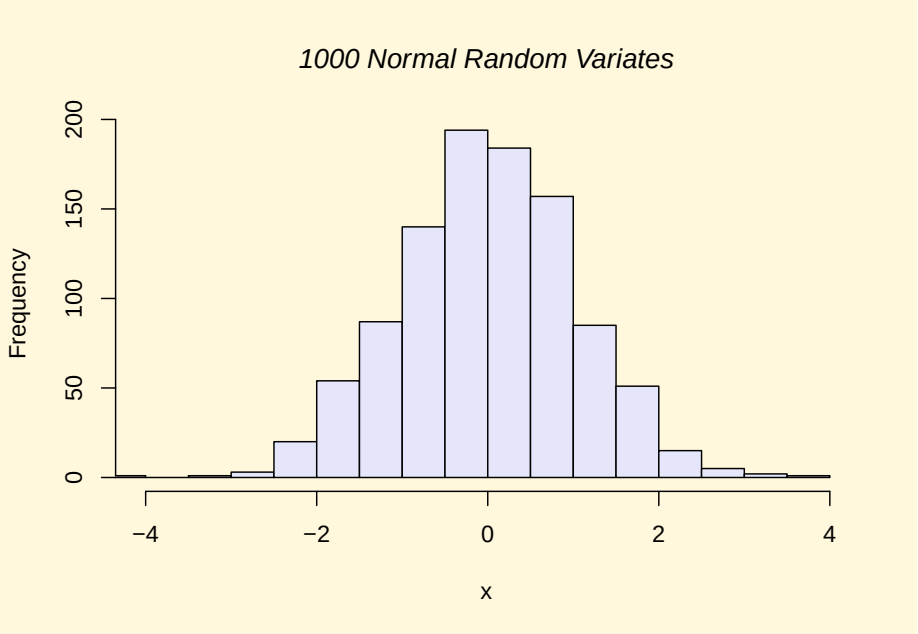



```
##
## > polygon(xx, yy, col="gray")
##
## > title("Distance Between Brownian Motions")
##
## > ## Colored plot margins, axis labels and titles.    You do need to be
## > ## careful with these kinds of effects.    It's easy to go completely
## > ## over the top and you can end up with your lunch all over the keyboard.
## > ## On the other hand, my market research clients love it.
## >
## > x <- c(0.00, 0.40, 0.86, 0.85, 0.69, 0.48, 0.54, 1.09, 1.11, 1.73, 2.05, 2.02)
##
## > par(bg="lightgray")
##
## > plot(x, type="n", axes=FALSE, ann=FALSE)
```

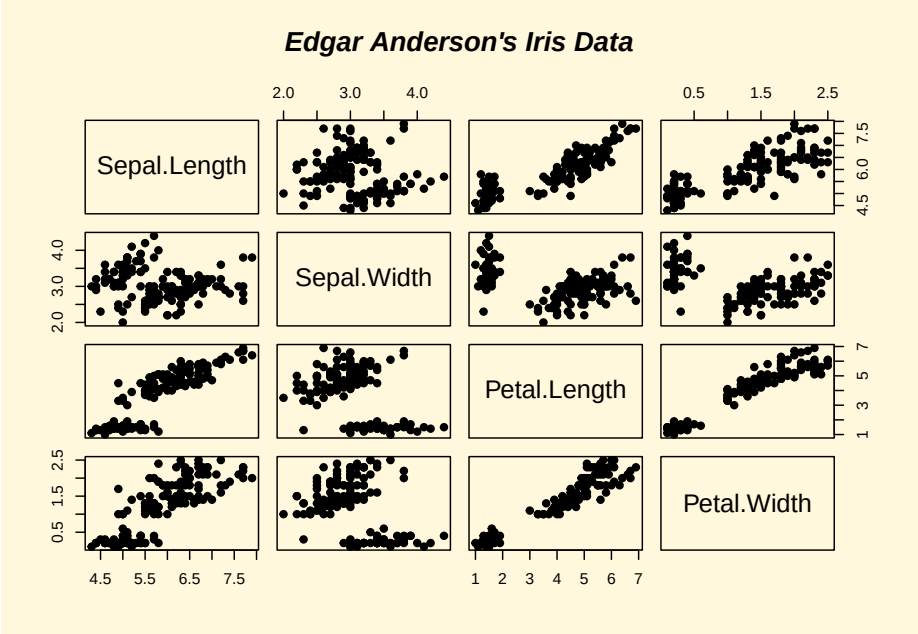



```
##
## > usr <- par("usr")
##
## > rect(usr[1], usr[3], usr[2], usr[4], col="cornsilk", border="black")
##
## > lines(x, col="blue")
##
## > points(x, pch=21, bg="lightcyan", cex=1.25)
##
## > axis(2, col.axis="blue", las=1)
##
## > axis(1, at=1:12, lab=month.abb, col.axis="blue")
##
## > box()
##
## > title(main= "The Level of Interest in R", font.main=4, col.main="red")
##
## > title(xlab= "1996", col.lab="red")
##
## > ## A filled histogram, showing how to change the font used for the
## > ## main title without changing the other annotation.
## >
## > par(bg="cornsilk")
##
## > x <- rnorm(1000)
```

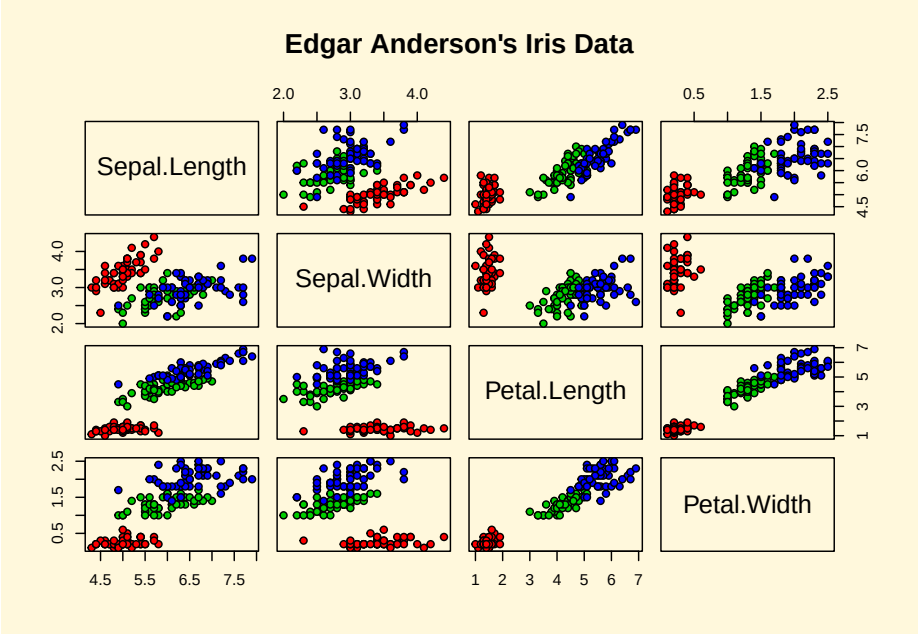
```
##
## > hist(x, xlim=range(-4, 4, x), col="lavender", main="")
```




```
##
## > title(main="1000 Normal Random Variates", font.main=3)
##
## > ## A scatterplot matrix
## > ## The good old Iris data (yet again)
## >
## > pairs(iris[1:4], main="Edgar Anderson's Iris Data", font.main=4, pch=19)
```

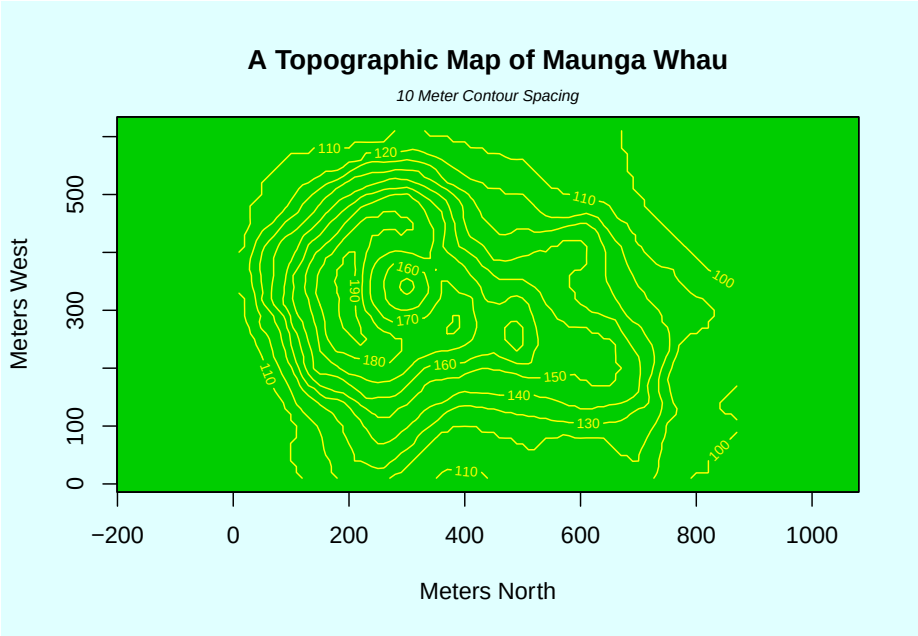



```
##
## > pairs(iris[1:4], main="Edgar Anderson's Iris Data", pch=21,
## +       bg = c("red", "green3", "blue")[unclass(iris$Species)])
```

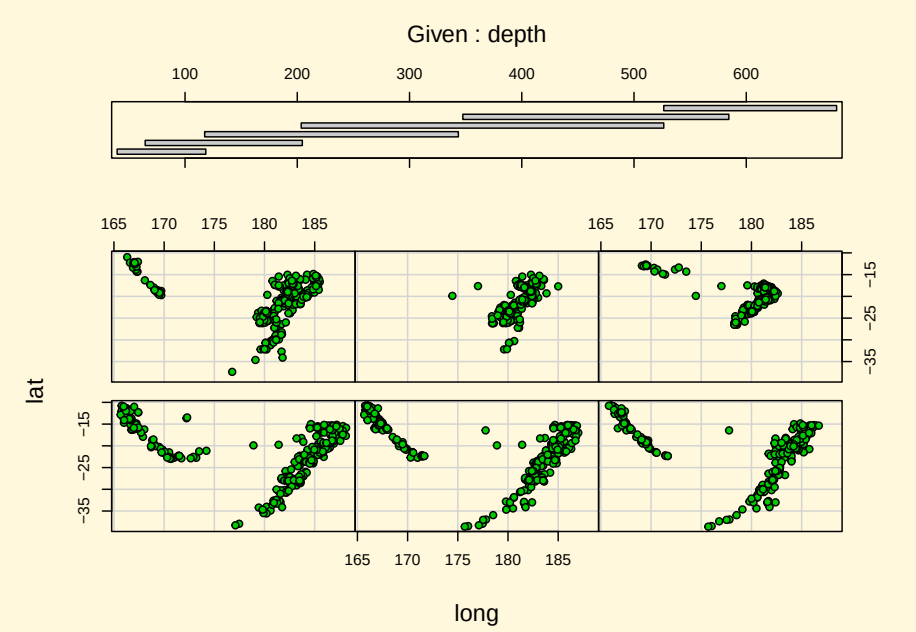



```
##
```

```
## > ## Contour plotting
## > ## This produces a topographic map of one of Auckland's many volcanic "peaks".
## >
## > x <- 10*1:nrow(volcano)
##
## > y <- 10*1:ncol(volcano)
##
## > lev <- pretty(range(volcano), 10)
##
## > par(bg = "lightcyan")
##
## > pin <- par("pin")
##
## > xdelta <- diff(range(x))
##
## > ydelta <- diff(range(y))
##
## > xscale <- pin[1]/xdelta
##
## > yscale <- pin[2]/ydelta
##
## > scale <- min(xscale, yscale)
##
## > xadd <- 0.5*(pin[1]/scale - xdelta)
##
## > yadd <- 0.5*(pin[2]/scale - ydelta)
##
## > plot(numeric(0), numeric(0),
## +      xlim = range(x)+c(-1,1)*xadd, ylim = range(y)+c(-1,1)*yadd,
## +      type = "n", ann = FALSE)
```




```
##
## > usr <- par("usr")
##
## > rect(usr[1], usr[3], usr[2], usr[4], col="green3")
##
## > contour(x, y, volcano, levels = lev, col="yellow", lty="solid", add=TRUE)
##
## > box()
##
## > title("A Topographic Map of Maunga Whau", font= 4)
##
## > title(xlab = "Meters North", ylab = "Meters West", font= 3)
##
## > mtext("10 Meter Contour Spacing", side=3, line=0.35, outer=FALSE,
## +       at = mean(par("usr")[1:2]), cex=0.7, font=3)
##
## > ## Conditioning plots
## >
## > par(bg="cornsilk")
##
## > coplot(lat ~ long | depth, data = quakes, pch = 21, bg = "green3")
```




```
##
## > par(opar)
```

Ejercicios

1. Instala las siguientes librerías que te serviran durante todo el curso
   - markdown
   - ggplot2

INFORMACIÓN ADICIONAL

Existen repositorios adicionales a CRAN, uno de ellos es Bioconductor, en él puedes buscar e instalar paquetes como `ggtree`.

Otra plataforma que resulta de gran apoyo es GitHub, permite crear, almacenar, administrar y compartir códigos de distintos lenguajes de programación. Una de sus ventajas es la consulta de repositorios, por ejemplo mixOmics, el cual contiene una amplia variedad de métodos para la exploración e integración de datos biológicos. El paquete `mixOmics` contiene una gran cantidad de técnicas multivariadas que se han desarrollado y validado en múltiples estudios biológicos, esto mediante la implementación simultánea de distintas "ómicas" para obtener una mejor compresión del sistema.

Ejercicio 1. Explora la página de Bioconductor, apóyate de su buscador e instala el paquete `ggtree`.

1.6 Ayuda en R

En la mayoría de las ocasiones desconocemos el alcance de alguna paquetería, los criterios de alguna función o en general, sabemos lo que queremos hacer pero no tenemos ni idea de qué paqueteria usar.

Los comandos `help()` y `?` son equivalentes, ambos van a permitir encontrar información sobre paqueterias, comandos o funciones generales de R. Se debe teclear `help(nombre_comando)` o `?nombre_comando`

Por ejemplo, para buscar información detallada del comando solve:

```
help(solve)
```

```
?solve
```

Para buscar ayuda de funciones o palabra reservadas se utilizan comillas:

```
help("for")
```

También existen opciones como `help.start()` y `help.search()` para obtener una versión extendida de la ayuda general desplegada en un navegador. Para ello se requiere tener la ayuda en HTML instalada y conexión a la red.

`help.search()` es una función que escanea documentación para paquetes previamente instalados.

Ejemplo:

```
help.search("clustering")
```

`help.start()` es una función que despliega información basada en documentos en línea de la versión actual de R, además de brindar links a manuales y la lista de las paqueterías instaladas, entre otras cosas.

Ejemplo:

```
help.start()
```

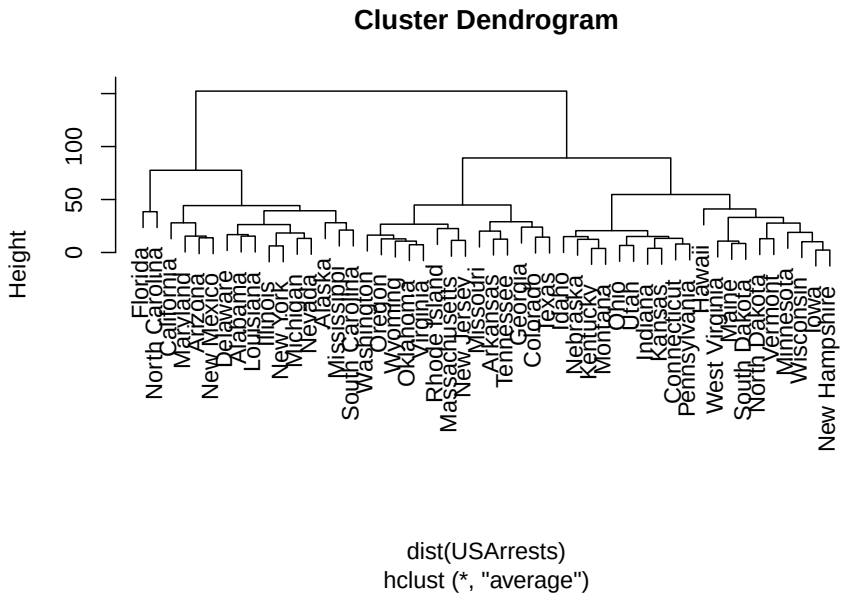
Cuando queremos ver ejemplos del uso de los comandos usamos la función `example()`

Ejemplo:

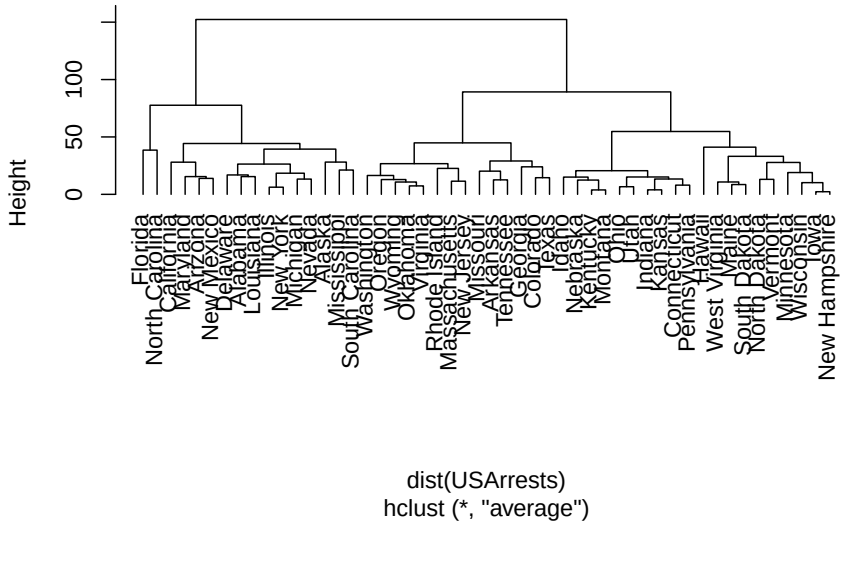
```
example("hclust")
```

```
##
## hclust> require(graphics)
##
## hclust> ### Example 1: Violent crime rates by US state
## hclust>
## hclust> hc <- hclust(dist(USArrests), "ave")
##
```

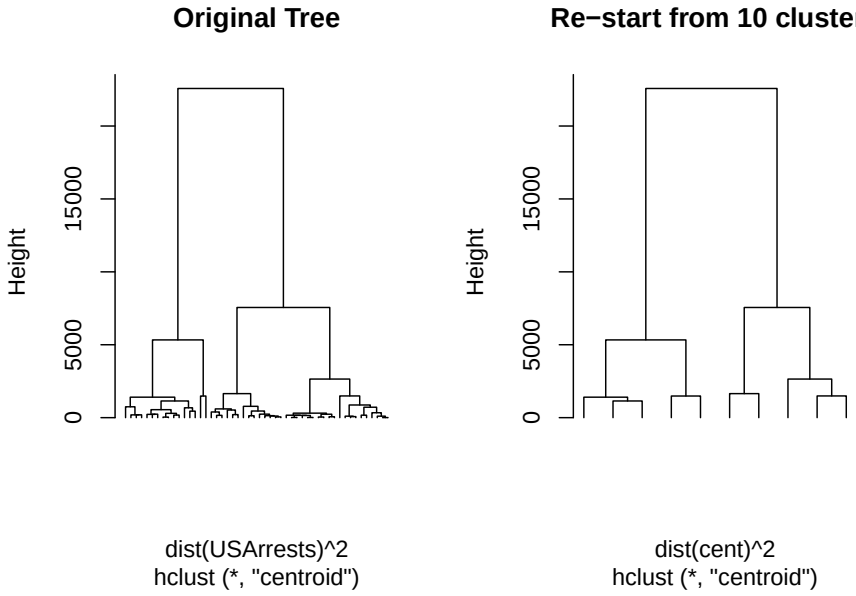
```
## hclust> plot(hc)
```




```
##
## hclust> plot(hc, hang = -1)
```



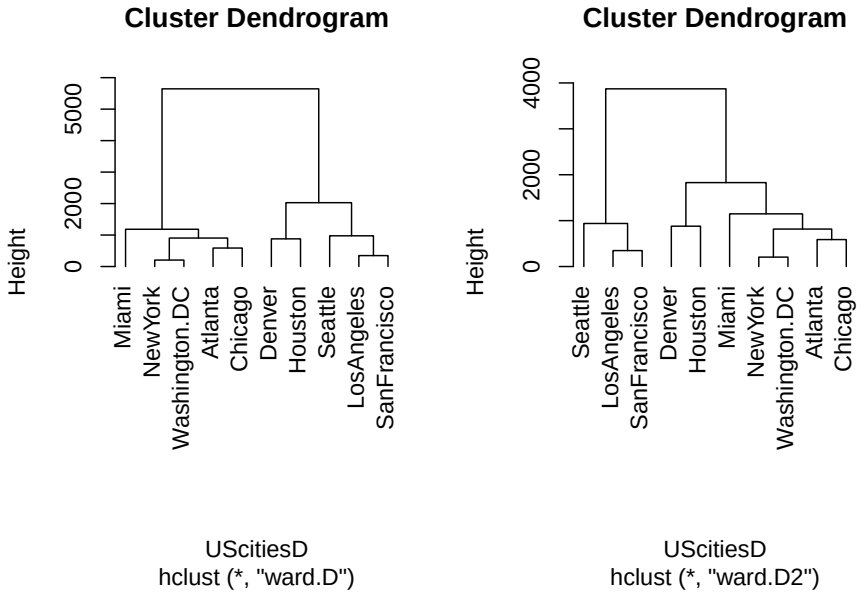
```
##
## hclust> ## Do the same with centroid clustering and *squared* Euclidean distance,
## hclust> ## cut the tree into ten clusters and reconstruct the upper part of the
## hclust> ## tree from the cluster centers.
## hclust> hc <- hclust(dist(USArrests)^2, "cen")
##
## hclust> memb <- cutree(hc, k = 10)
##
## hclust> cent <- NULL
##
## hclust> for(k in 1:10){
## hclust+   cent <- rbind(cent, colMeans(USArrests[memb == k, , drop = FALSE]))
## hclust+ }
##
## hclust> hc1 <- hclust(dist(cent)^2, method = "cen", members = table(memb))
##
## hclust> opar <- par(mfrow = c(1, 2))
##
## hclust> plot(hc,  labels = FALSE, hang = -1, main = "Original Tree")
##
## hclust> plot(hc1, labels = FALSE, hang = -1, main = "Re-start from 10 clusters")
```




```
##
## hclust> par(opar)
##
```

```
## hclust> ### Example 2: Straight-line distances among 10 US cities
## hclust> ##  Compare the results of algorithms "ward.D" and "ward.D2"
## hclust>
## hclust> mds2 <- -cmdscale(UScitiesD)
##
## hclust> plot(mds2, type="n", axes=FALSE, ann=FALSE)
```


```
##
## hclust> text(mds2, labels=rownames(mds2), xpd = NA)
##
## hclust> hcity.D  <- hclust(UScitiesD, "ward.D") # "wrong"
##
## hclust> hcity.D2 <- hclust(UScitiesD, "ward.D2")
##
## hclust> opar <- par(mfrow = c(1, 2))
##
## hclust> plot(hcity.D,  hang=-1)


##
## hclust> plot(hcity.D2, hang=-1)
```




```
##
## hclust> par(opar)
```

Todo lo anterior requiere que conozcamos el nombre correcto del comando, pero ¿qué pasa si no lo sabemos?, ¿lloramos? no. Podemos utilizar el comando `apropos()` para encontrar todo lo relacionado con algún término.

Ejemplo:

```
apropos("solve")
```

```
##  [1] ".rs.markdown.resolveCompletionRoot"
##  [2] ".rs.resolveAliasedPath"
##  [3] ".rs.resolveAliasedSymbol"
##  [4] ".rs.resolveContextSourceRefs"
##  [5] ".rs.resolveEnvironment"
##  [6] ".rs.resolveFormals"
##  [7] ".rs.resolveFormalsImpl"
##  [8] ".rs.resolveFormalsImplS3Dispatch"
##  [9] ".rs.resolveObjectFromFunctionCall"
## [10] ".rs.resolveObjectSource"
## [11] ".rs.reticulate.resolveModule"
## [12] ".rs.rnb.resolveActiveChunkId"
## [13] "backsolve"
## [14] "forwardsolve"
## [15] "qr.solve"
## [16] "solve"
```

```
## [17] "solve.default"
## [18] "solve.qr"
```

Ahora, ¿qué pasa cuando tengo la idea de lo que quiero hacer pero no se qué paqueteria usar, ni cuál comando? puedo usar `??` seguido de una palabra clave. Esto nos arrojará sugerencias sobre lo que deseamos hacer.

Ejemplo:

```
??DNA
```

NOTA Se recomienda el uso del autocompletado, de esta manera reducirás errores de dedo.

Capítulo 2

Bases prácticas en R

2.1 Expresiones y asignaciones

Las **expresiones** y **asignaciones** son los dos tipos de resultados que arroja R.

Las **expresiones** sólo se muestran en la salida estándar y NO se guardan en alguna variable, es decir, cada que se corra la línea se obtendran valores distintos.

Ejemplo:

```
rnorm(10)
```

```
##  [1] -0.038998018  0.758203514 -1.349242559  0.001171862  0.504689087
##  [6]  0.827538781  1.196159067 -0.502982027  1.386914012 -0.708304493
```

```
rnorm(10)
```

```
##  [1]  0.5854544 -1.3837363 -1.9708028 -0.9765416  1.1359097
##  [6]  1.0721026  1.4859182  0.9697558 -2.3765914  0.6074945
```

Las **asignaciones**, como su nombre lo indica, se guardan los valores al ser asignados a una variable. Esto se puede lograr mediante el uso de `<-`

Ejemplo:

```
x <- rnorm(10)
```

```
x
```

```
##  [1] -0.59587707  0.68101309  1.58648056 -0.53781906 -0.59819972
##  [6]  1.19153644 -0.01946832 -0.13359182 -0.57294842  0.88481736
```

NOTA El símbolo `<-` es equivalente en función a `=` pero puede llevar a confusiones importantes con el operador `==`.

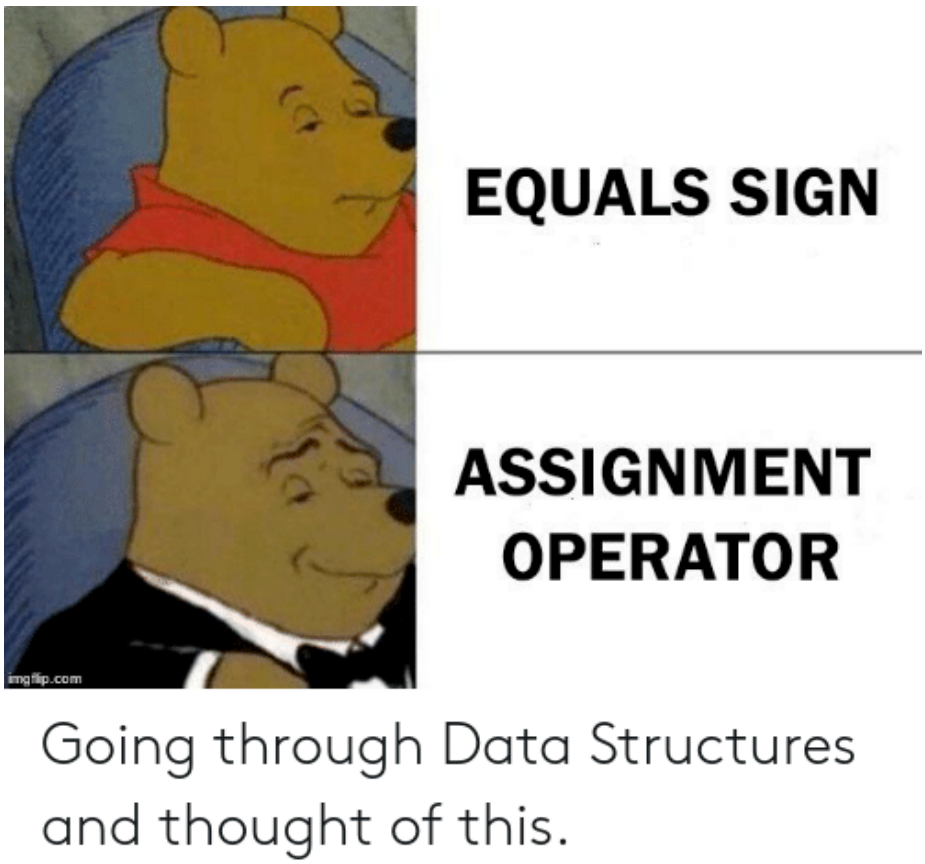


Figure 2.1: Operador de asignación. Así que evita el uso del igual

Otro punto a considerar respecto a las **asignaciones** es que R es capaz de distinguir entre mayúsculas y minúsculas, por lo que la misma letra puede contener valores distintos.

Ejemplo:

```
a <- 3
A <- 6
```

```
a
```

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

```
A
```

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

Una opción es definirlas en más de una línea, ejemplo:

```
a <-
  pi + 12
```

NOTA Ten mucha precaución con el nombre que asignas a un valor, ya que podrías sobreescribirlo y se le quedará asignado el último valor.

Ejemplo:

```
b <- 3
b
```

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

```
b <- 8
b
```

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

La separación de comandos puede darse de dos formas:

Empleando ;, ejemplo:

```
a <- 3; b <-5
```

o usando un salto de línea, esta es una mejor opción, ejemplo:

```
a <- 3
b <-5
```

2.2 Movimiento entre directorios

Otra de las ventajas que ofrece R es que permite ubicar algún archivo o saber la dirección del directorio en la que nos encontramos actualmente, todo ello sin necesidad de salir de la interfaz.

Para saber en qué directorio estamos, se teclea:

```
getwd()
```

```
## [1] "/Users/robertoalvarez/Documents/GitHub/Bravo_Garcia_Maria_Fernanda_Bioinfo_2024
```

Para cambiar de directorio utilizamos `setwd("direccion_a_la_que_quieres_ir")`

```
setwd("~")
```

2.3 Bash en R

También se pueden usar los comandos de la terminal de bash dentro de R, utilizando la función `system()`

Para listar archivos de una carpeta usamos `ls`

```
system("ls -la")
```

Para saber en qué directorio estamos usamos la función análoga a `getwd()`, que es `pwd`

```
system("pwd")
```

Importante: Como regla general todos los nombres van entre comillas: nombre de carpetas, archivos, de columnas, de renglones,etc.

2.4 Operaciones aritméticas

R también puede ser usado como calculadora. Se puede sumar, restar, multiplicar,dividir, "exponenciar" y calcular la raíz cuadrada.

SUMA con el operador `+`

```
a + b
```

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

RESTA con el operador `-`

```
a - b
```

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

MULTIPLICACIÓN con el operador `*`

```
a * b
```

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

DIVISIÓN con el operador `/`

```
a / b
```

```
## [1] 0.6
```

EXPONENTE con los operadoradores `**` o `^`

```
a ** b
```

```
## [1] 243
```

```
a ^ b
```

```
## [1] 243
```

RAÍZ CUADRADA con la función `sqrt()`

```
sqrt(a)
```

```
## [1] 1.732051
```

LOGARITMO con la función `log()`

```
log(a)
```

```
## [1] 1.098612
```

2.4.1 Prioridad en las operaciones

Las operaciones se efectuan en el siguiente orden:

1. izquierda a derecha
2. `sqrt()` y `**` `^`
3. `*` y `/`
4. `+` y `-`
5. `<-`

IMPORTANTE Este orden se altera si se presenta un paréntesis. En ese caso la operación dentro del paréntesis es la que se realiza primero.

Ejemplos:

```
4 + 2 * 3 = 4 + 6 = 10
```

```
4 - 15/3 + 3^2 + sqrt(9) = 4 - 15/3 + 9 + 3 = 4 - 5 + 12 = 13
```

```
4 - (3+7)^2 + (2+3)/5 = 4 - (10)^2 + 5/5 = 4 - 100 + 1 = -95
```

Ejercicios

Resuelve en un pedazo de papel primero para saber cuál sería el resultado de las siguientes operaciones aritméticas. Después comprueba tu resultado tecleandolas en R.

1. 1 + 2*3 + 3 + 15/3
2. 4 - 15/3 + 3^2 + 3*sqrt(81)
3. 40 - (4+3)^2 + (10-5)/3
4. 32^5 - (3-5)*2 + 32/sqrt(64)
5. 1/(3^-3) + (8-10^2) - (25/sqrt(25))^2

2.5 Tipos de valores en R

2.5.1 Valores booleanos

También conocidos como *Datos lógicos*. Este tipo de datos **sólo** contienen información `TRUE` o `FALSE`, lo cual sirve para evaluar si los elementos de un vector cumplen con los criterios deseados. Para ello se utilizan los operadores de comparación:

- igual `==`
- no es igual a `!=`
- menor que `<`
- mayor que `>`
- menor o igual que `<=`
- mayor o igual que `>=`

Ejemplo:

```
1 < 5
```

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

```
10 == 0
```

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

```
10 != 0
```

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

```
10 <= 0
```

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

NOTA Dentro de R los valores lógicos `TRUE` y `FALSE` tienen un valor numérico. `TRUE`equivale a 1 y `FALSE` es equivalente a 0. Esto permite cuantificar el número de elementos que cumplen con los criterios, ¿cómo? mediante la suma de los `TRUE`s.

Ejercicios Demuestra si: 1. El logaritmo base 10 de 20 es menor que la raíz cuadrada de 4. Desarrollalo en una sola línea. 2. 1/3^-1 es igual a 3/1^-1 3. (-2)^2 no es igual a (2)^2

2.5.2 Caracter

Son *strings* de texto y se caracterizan porque cada uno de los elementos va entre comillas. Los elementos pueden ser desde sólo un caracter hasta oraciones. Podría parecer que la variable a la cual lo asignamos contiene números, pero las comillas indican que serán tratados como texto. Podemos subsetearlos por su índice o buscando literalmente el texto.

Ejemplo:

```
x<- "La candente mañana de febrero en que Beatriz Viterbo murió, después de una imperiosa agonía
```

2.5.3 Enteros y números (numeric)

Existen dos formas diferentes en que las computadoras pueden guardar los números y hacer operaciones matemáticas con ellos: `numeric` e `integer`. Por lo común no importa esta diferencia, pero puede ser relevante para algunas funciones de Bioconductor. En R se representan los números como `numeric` y el tamaño máximo que maneja para un `integer` es ligeramente más chico que el tamaño del genoma humano.

¿Cómo revisar si un objeto es numeric o entero? Con la función `class()`

```
x <- 1
class(x)
```

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

```
x <- 1:3
class(x)
```

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

Capítulo 3

Vectores

R permite manejar datos dentro de estructuras para poder trabajarlos, estas estructuras pueden ser: - Vector: Es de una sola dimensión y solo permite almacenar datos del mismo tipo. - Matriz: Es un arreglo en dos dimensiones y permite ingresar datos del mismo tipo. - Data Frame: Similar a la matriz por ser también de dos dimensiones, solo que este arreglo permite distintas clases de datos. - Lista: Es de una sola dimensión como el vector, la diferencia que es que una lista permite incorporar diferentes tipos de datos.

3.1 Contenido

Para conocer el contenido de una variable, sólo es necesario poner la variable y presionar *enter* (sesión interactiva). En el caso de estar en un *script* es necesario usar la función `print()`

Ejemplo:

```
x<-3
print(x)
```

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

3.2 Vectores en R

Un *vector* es una colección de datos del mismo tipo, siempre del **mismo tipo**, no es posible mezclarlos. Los elementos contenidos dentro de un vector son conocidos como *componentes* y pueden ser del tipo lógico, caracteres, numéricos o integer.

3.3 Definición

Para definir un vector se utiliza la función `c()`, que significa *combine*. Existen dos formas de definir un vector: **Extensiva** y **Secuencia**.

3.3.1 Definición extensiva

La **forma extensiva** para definir un vector considera que los elementos situados dentro del paréntesis tienen que estar separados por comas.

Ejemplo:

```
vector_numerico <- c(1, 3, 5, 7)      # vector numérico
vector_texto <- c("a", "b", "c", "d") # vector de texto
vector_logico <- c(TRUE, FALSE, TRUE) # vector lógico / booleano
```

3.3.2 Definición en secuencia

3.3.2.1 Uso de c()

En algunas ocasiones definir de manera extensiva puede resultar muy poco eficiente, sobre todo para vectores que contengan una gran cantidad de elementos en secuencia, para ello se definen en **forma de secuencia** empleando el operador `:`.

Ejemplo: Un vector que tenga los primeros 100 números enteros

```
x <- c(1:100)

x
```

```
##   [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16
##  [17]  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32
##  [33]  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48
##  [49]  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64
##  [65]  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80
##  [81]  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96
##  [97]  97  98  99 100
```

3.3.2.2 Uso de seq()

Otra alternativa es la función `seq()` que significa *sequence* y es una generalización del operador `:`.

Ejemplo:

```
x <- seq(1,100)

x
```

```
##   [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16
##  [17]  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32
##  [33]  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48
##  [49]  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64
##  [65]  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80
##  [81]  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96
##  [97]  97  98  99 100
```

Esta función permite generar secuencias numéricas de distintas clase y por diferentes rangos.

Ejemplo: Una secuencia que vaya desde -12 hasta 30 en un rango de 3, es decir, -12, -9, -6, ..., 27, 30:

```
x <- seq(from=-12,to=30,by=3)

x
```

```
##  [1] -12  -9  -6  -3   0   3   6   9  12  15  18  21  24  27  30
```

NOTA Se puede omitir *from*, *to* y *by* mientras se sigan colocando los valores en el mismo orden.

Ejemplo:

```
y <- seq(0,1,0.1)

y
```

```
##  [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

NOTA Si queremos cambiar el orden, debemos necesariamente ponerlos. Ejemplo:

```
z <- seq(by=0.1, to =1, from=0.5)

z
```

```
## [1] 0.5 0.6 0.7 0.8 0.9 1.0
```

3.4 Longitud de un vector

La *longitud* de un vector se refiere al número de elementos que contiene. Para conocer ese dato se emplea la función `length()`

Ejemplo:

```
s <- (1:5)

length(s)
```

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

3.5 Elementos de un vector

Para acceder a elementos de un objeto con *indices* (componentes que tienen una posición asignada), debemos usar *corchetes* `[]`. Los corchetes pueden contener posiciones consecutivas o no consecutivas.

3.5.1 Elementos consecutivos

Se definen el rango de las posiciones a seleccionar, para ello se emplea el operador `:`

Ejemplo:

```
#Se define el vector con la variable "x"
x <- c("Muchos", "años" ,"después" ,",", "frente", "al" ,"pelotón")

#Elegir desde el primer hasta el cuarto elemento del objeto "x"
x[1:4]
```

```
## [1] "Muchos"  "años"    "después" ","
```

Ejemplo:

```
x <- c(1,2,3,5,8,13,21)

x[3:6]
```

```
## [1]  3  5  8 13
```

3.5.2 Elementos no consecutivos de un vector

Las posiciones a seleccionar se colocan en un vector separadas por `,`, a su vez este vector se coloca dentro de los corchetes.

Ejemplo:

```
#Se define el vector "x"
x <- c("Hola", "Bien", "cómo", "!", "estás", ":(", "?")

#Se indica dentro de un nuevo vector que se seleccionen las posiciones 1, 3, 5 y 7 del
x[c(1,3,5,7)]
```

```
## [1] "Hola"  "cómo"  "estás" "?"
```

NOTA: No es necesario que estén en orden

Ejemplo:

```
x<-c(1,2,3,5,8,13,21)

x[c(2, 7, 4)]
```

```
## [1]  2 21  5
```

3.5.3 Excluir elementos de un vector

Para omitir algun o un conjunto de elementos de un vector, se emplea el signo menos dentro de los corchetes `[- ]`

Ejemplo:

```
#Se define el vector "x"
x <- c(1,2,3,5,8,13,21)

#Dentro del corchete indicamos la posición que se quiere omitir
x[-4]
```

```
## [1]  1  2  3  8 13 21
```

```
#Cuando se quiere omitir un conjunto de elementos, se definen las posiciones dentro de un vector
x[-c(2, 7, 4)]   # Todos menos el segundo , séptimo y cuarto elemento
```

```
## [1]  1  3  8 13
```

¿Esto qué hace?

```
x[-length(x)]
```

```
## [1]  1  2  3  5  8 13
```

NOTA Este comando `[- ]` **no elimina** elementos de un vector sólo los selecciona y omite. Sin embargo, el vector original continua intacto.

```
x <- c(1,2,3,5,8,13,21)
x[-6]
```

```
## [1]  1  2  3  5  8 21
```

```
x     # Estoy intacto
```

```
## [1]  1  2  3  5  8 13 21
```

3.6 Reasignar elementos de un vector

Se pueden asignar nuevas posiciones y valores a un vector previamiente definido

Ejemplo:

```
#Se define el vector "x"
x <- c(88,5,12,13)

#Agregamos el valor "168" en la posición 4. Intenta explicar paso a paso la siguiente
x <- c(x[1:3],168,x[4])
x
```

```
## [1]  88   5  12 168  13
```

Se puede definir un vector vacío y luego "llenarlo" asigando una posición a los componentes.

```
x<-c()
x                 # Soy un vector vacío :(
```

```
## NULL
```

```
x[1]<- 2
x[2:5]<-c(56,78,90,12)
x                 # Ahora ya no :)
```

```
## [1]  2 56 78 90 12
```

3.7 Repetición de elementos de un vector

La función `rep()`, que viene del inglés *repeat*, nos permite repetir elementos en un vector dado. El comando `rep()` sigue el siguiente formato `rep(valor, n veces)`.

Ejemplo:

```
#Repite 5 veces el valor "3" y asignalo a la variable "x"
x <- rep(3,5)
x
```

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

```
#También se pueden repetir un conjunto de valores una vez que sean colocados dentro de
y <- rep(c(1,2,3,5),3) #Repite 3 veces los valores 1, 2, 3 y 5.
y
```

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

Ejemplo:

```
primos <- c(1,2,3,5,7,11)
z <- rep(primos,4)
z
```

```
##  [1]  1  2  3  5  7 11  1  2  3  5  7 11  1  2  3  5  7 11  1  2  3
## [22]  5  7 11
```

Dentro del comando `rep()` se encuentra la opción `each`, la cual permite definir la frecuencia de repetición.

Ejemplo:

```
x<-c(1,2,3,4)

y<-rep(x,each=2)
y
```

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

3.8 Uso de funciones `any()` y `all()`

Las funciones `any()` *algún* y `all()` *todos* permiten conocer si alguno o todos los elementos de un vector cumplen cierta condición. El resultado obtenido siempre será un valor booleano: **TRUE** o **FALSE**

Ejemplo:

```
x <- 1:15
any(x > 7.5)
```

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

```
any(x > 19.76)
```

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

```
any(x >= 15)
```

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

```
all(x > sqrt(100))
```

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

```
all(x>0)
```

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

Ejercicios

1. Dado un vector x, escribe un código que determine si todos los elementos del vector son iguales a cero utilizando la función all().
2. Escribe un código que tome un vector x y devuelva TRUE si hay algún elemento repetido en el vector, utilizando la función any().
3. Dado un vector x, escribe una función que determine si todos los elementos del vector son iguales entre sí utilizando la función all().
4. Escribe una función que tome dos vectores ("x" y "y") y devuelva TRUE si ambos vectores tienen algún elemento en común, utilizando la función any().

5. Escribe una función que tome dos vectores ("x" y "y") y devuelva TRUE si todos los elementos del vector x son mayores que los elementos correspondientes en el vector y, utilizando la función all().
6. Dado un vector x, escribe una función que determine si todos los elementos del vector son menores que cero utilizando la función all().
7. Escribe una función que tome dos vectores ("x" y "y") y devuelva TRUE si al menos un elemento del vector x es mayor que los elementos correspondientes en el vector y, utilizando la función any().
8. Dado un vector x, escribe una función que determine si todos los elementos del vector son iguales a un valor específico a utilizando la función all().
9. Escribe una función que tome dos vectores ("x" y "y") y devuelva TRUE si al menos un elemento del vector x es menor que los elementos correspondientes en el vector y, utilizando la función any().

3.9 Operaciones con vectores

3.9.1 Operaciones aritméticas

Al igual que en álgebra podemos definir varias operaciones que nos dejan siempre otro vector. Las operaciones se pueden realizar vector/vector o vector/escalar.

Se definen los vectores

```
x<-c(1,2,3)
y<-c(4,5,6)
```

SUMA con el operador `+`

```
x + y
```

```
## [1] 5 7 9
```

```
x + 2
```

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

RESTA con el operador `-`

```
x - y
```

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

```
x - 1
```

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

MULTIPLICACIÓN con el operador `*`

```
x * x
```

```
## [1] 1 4 9
```

```
x * y
```

```
## [1]  4 10 18
```

```
y * 3
```

```
## [1] 12 15 18
```

DIVISIÓN con el operador `/`

```
x / y
```

```
## [1] 0.25 0.40 0.50
```

```
y / 5
```

```
## [1] 0.8 1.0 1.2
```

EXPONENTE con los operadoradores `**` o `^`

```
x ** y
```

```
## [1]   1  32 729
```

```
y ^ 2
```

```
## [1] 16 25 36
```

RAÍZ CUADRADA con la función `sqrt()`

```
sqrt(y)
```

```
## [1] 2.000000 2.236068 2.449490
```

LOGARITMO con la función `log()`

```
log(x)
```

```
## [1] 0.0000000 0.6931472 1.0986123
```

NOTA CUIDADO con el tamaño de los vectores al momento de realizar operaciones entre ellos.

3.10 Operaciones con un comando

También podemos aplicar funciones para calcular con una sola instrucción varias operaciones útiles, esto nos ahorra tiempo.

Ejercicio Calcula el promedio de los números del 1 al 10.

```
#Respuesta muy larga
(1+2+3+4+5+6+7+8+9+10)/10
```

```
## [1] 5.5
```

¿Qué alternativas se tienen para realizar operaciones con una cantidad mucho mayor de datos? Se pueden emplear los siguientes comandos `min()`, `max()`, `range()`, `sum()`, `mean()`, `median()`, `sd()`,`quantile()`,`unique()`,`sort()`

```
#Se define el vector que incluye mil datos
x<-rnorm(1000)

min(x) #Se obtiene el valor mínimo
```

```
## [1] -2.97576
```

```
max(x) #Se obtiene el valor máximo
```

```
## [1] 3.896923
```

```
range(x) #Da a conocer el rango en el cual se encuentran los valores, es decir el valo
```

```
## [1] -2.975760  3.896923
```

```
sum(x) #Realiza la suma de todos los valores contenidos en el vector
```

```
## [1] -4.918641
```

```
mean(x) #Calcula el promedio del conjunto de valores
```

```
## [1] -0.004918641
```

```
median(x) #Se obtiene la mediana
```

```
## [1] -0.004371671
```

Ejercicio ¿Qué función tienen los siguientes comandos `sd()` y `quantile()`?

Para `unique()` y `sort()` conviene tener elementos discretos más que continuos.

```
x <- c(rep(3,5),1:15,rep(c(1,2,3),5))
unique(x)
```

```
##  [1]  3  1  2  4  5  6  7  8  9 10 11 12 13 14 15
```

```
x <- sample(10,10)
x
```

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

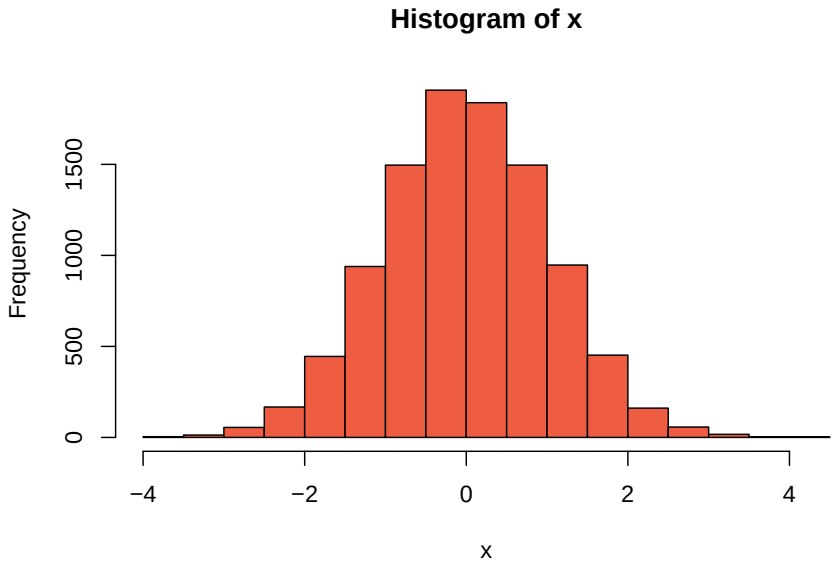
```
sort(x)
```

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

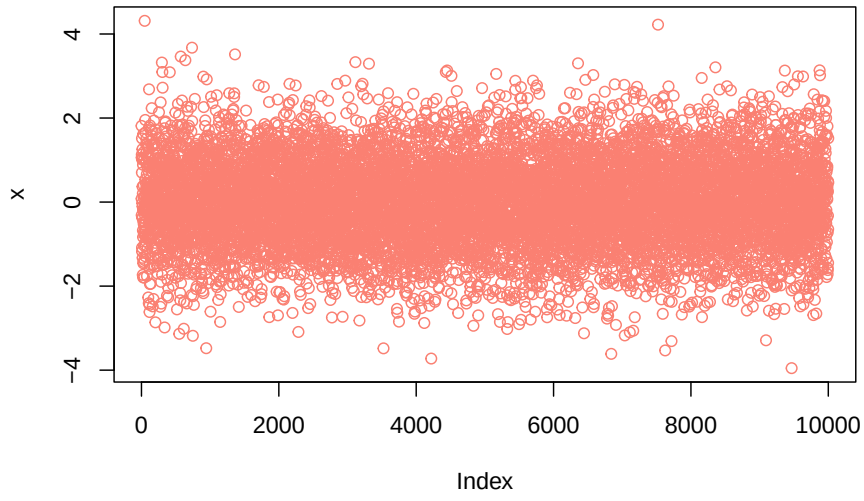
3.11 Gráficos con vectores

Podemos graficar los vectores de manera inmediata en `R`

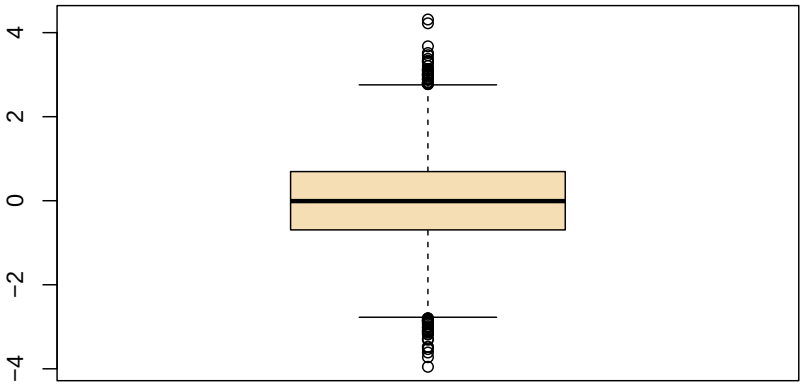
```r
x<- rnorm(10000)

hist(x,col="tomato2") #Histograma: grafica la distribución de las frecuencias de los datos
```




```r
plot(x,col="salmon") #Gráfica los datos en el orden de aparición en el vector
```




```r
boxplot(x,col="wheat") #Boxplot: muestra la mediana y quantiles
```



3.12 Vectores con nombre

Una de las cracterísticas de R es que se puede asignar nombres a los vectores, para ello usamos la función `names()`

Ejemplo:

```
edades <- c(35,35,70,17,14) #Definimos un vector llamado "edades"
nombres <- c("Jerry","Beth","Rick", "Summer","Morty") #Definimos un vector llamado "edades", del

names(edades) <- nombres #Se nombran los elementos del vector "edades"
edades
```

```
##  Jerry   Beth   Rick Summer  Morty
##     35     35     70     17     14
```

También se selecciona de la manera usual, por ejemplo, si quiero ver cuál es la edad de Rick, debo seleccionar el elemento 3:

```
edades[3]
```

```
## Rick
##   70
```

Esto es muy poco eficiente y propenso al error, sobre todo con vectores muy grandes. Por ello podemos usar los nombres de los vectores:

```
edades["Rick"]
```

```
## Rick
##   70
```

Recuerda que los nombres S-I-E-M-P-R-E van entre comillas

```
edades[c("Rick","Morty")]
```

```
##  Rick Morty
##    70    14
```

Ejercicios:

1. ¿Cuál es el promedio de las edades, sin contar el de Beth?
2. Quiten a Morty del vector, ordénenlo y guárdenlo como un nuevo objeto.
3. ¿Hay alguna edad que sea mayor de 75? ¿Menor de 12? ¿Entre 12 y 20?

3.12.1 Tamaños de genomas

Ahora veamos un ejemplo más "biológico"

```
genomeSizeM_BP<-c(3234.83,2716.97,143.73,0.014281,12.1)
```

NOTA Si se desea ver el tamaño en bp, simplemente multiplicamos por el valor del prefijo (Mega = 1 millón)

```
genomeSizeM_BP*1e6
```

```
## [1] 3234830000 2716970000  143730000      14281   12100000
```

```
organismo<-c("Human","Mouse","Fruit Fly","Roundworm","Yeast")
```

```
names(genomeSizeM_BP)<- organismo
```

```
genomeSizeM_BP
```

```
##       Human       Mouse   Fruit Fly   Roundworm       Yeast
## 3234.830000 2716.970000  143.730000    0.014281   12.100000
```

¿Qué hay de diferente entre el primer vector al que le se asiganaron los tamaños de genomas & esta última versión?

RECUERDA Se pueden seleccionar elementos de un vector utilizando corchetes:

```
genomeSizeM_BP[1]
```

```
##   Human
## 3234.83
```

Para obtener elementos consecutivos:

```
genomeSizeM_BP[1:4]
```

```
##       Human       Mouse   Fruit Fly   Roundworm
## 3234.830000 2716.970000  143.730000    0.014281
```

Para obtener elementos NO consecutivos:

```
genomeSizeM_BP[c(1,2,5)]
```

```
##   Human   Mouse   Yeast
## 3234.83 2716.97   12.10
```

Para descartar (no eliminar, ni quitar) ciertos elementos:

```
genomeSizeM_BP[-c(1,3,5)]
```

```
##       Mouse   Roundworm
## 2716.970000    0.014281
```

Para referirnos a los elementos por el nombre asignado:

```
genomeSizeM_BP[c("Yeast","Human")]
```

```
##   Yeast   Human
##   12.10 3234.83
```

Además de algunas operaciones aritméticas se pueden calcular con la media, máximo, mediana, mínimo, suma y longitud de los vectores

Ejercicio

1. Generar un vector de las edades de 10 de tus compañeros
2. Asignales nombre.
3. Encuentra el mínimo,máximo, media, mediana, la desviación estándar, la longitud del vector y selecciona sólo los elementos impares.
4. Elimina el máximo y el mínimo y con el vector resultante realiza un histograma.
5. Crea un vector de caracteres con diez nombres de especies y asocialo con su número de acceso de NCBI para su genoma en nucleótidos.

3.13 ¿Cómo lidiar con las NAs ?

Es (muy) freceunte que en bases de datos se tengan valores `NA`, es decir medidas que no pudieron realizarse, medidas perdidas, etc. Para ello se utiliza `NA`. `R` trata de manera especial a las NAs

```
x <- c(88,NA,12,168,13)
```

Existe una fución para determinar si un elemento es o no una `NA`. La función es `is.na()`

```
x <- c(88,NA,12,168,13)
```

```
is.na(x)
```

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

Si queremos calcular ciertas funciones numéricas `R` no sabrá qué hacer

```
x <- c(88,NA,12,168,13)
mean(x)
```

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

Sin embargo, podemos decirle a `R` que las omita, indicando como argumento de la función `mean()` `na.rm=TRUE` que significa *na remove*

```
x <- c(88,NA,12,168,13)
mean(x,na.rm=TRUE)
```

```
## [1] 70.25
```

¿Qué otras funciones tienen esta opción `na.rm=TRUE` ?

3.14 Filtrado de elementos de un vector

Podemos generar vectores de que sean subconjuntos de vectores más grandes que cumplan cierta(s) condición(es)

```
un_vector <- c(1,2,3,5,7,11,13,17,19)
otro_vector <- un_vector[un_vector*un_vector > 10]   #Se lee el vector desde dentro ha
otro_vector
```

```
## [1]  5  7 11 13 17 19
```

Veamos paso a paso qué es lo que hace este proceso

```
un_vector
```

```
## [1]  1  2  3  5  7 11 13 17 19
```

```
un_vector*un_vector > 10    # Mira, de adentro hacía afuera
```

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

```
indices<-un_vector*un_vector > 10
un_vector[indices]
```

```
## [1]  5  7 11 13 17 19
```

```
un_vector[c(FALSE,FALSE,FALSE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE)]
```

```
## [1]  5  7 11 13 17 19
```

La representación interna de los valores booleanos `FALSE` y `TRUE`son 0 y 1, respectivamente

```
un_vector[c(rep(0,3),rep(1,1))]
```

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

3.14.1 Filtrado con `subset()`

Podemos usar la función `subset()`para hacer lo mismo que en el caso anterior **excepto que omite los NA**

```
un_vector<-c(1,2,3,5,7,11,13,17,19)
otro_vector <- subset(un_vector,un_vector*un_vector > 10)
otro_vector
```

```
## [1]  5  7 11 13 17 19
```

Qué pasa si tenemos `NA`s. Si usamos el método anterior obtendríamos

```
un_vector<-c(1,2,3,5,7,11,NA,13,17,NA,19)
otro_vector <- un_vector[un_vector*un_vector > 10]   # Leeme de adentro hacia afuera
otro_vector  # Aquí salen las NAs
```

```
## [1]  5  7 11 NA 13 17 NA 19
```

En cambio con `subset()`

```
un_vector<-c(1,2,3,5,7,11,NA, 13,17,NA, 19)
otro_vector <- subset(un_vector,un_vector*un_vector > 10)
otro_vector  # Aquí ya no aparecen las NAs
```

```
## [1]  5  7 11 13 17 19
```

3.14.2 La función de selección `which()`

La función `which()` nos regresa los índices, es decir, dice **quiénes** cumplen cierta condición

```
z <- c(5,2,-3,8)
which(z*z > 8)
```

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

Acá nos dicen quiénes

```
z[which(z*z > 8)]
```

```
## [1]  5 -3  8
```

3.15 ¿Cómo podemos ver si dos vectores son iguales?

Dos vectores son iguales si elemento a elemento son idénticos. Por lo tanto deben de ser del mismo tamaño. **RECUERDA** Para probar si dos elementos son iguales se utiliza el operador de comparación `==` son dos signos iguales juntos, sin espacio. No confundir con el operador `=` que se puede usar como operador de asignación (aunque no es recomendable su uso. De hecho está prohibido en este curso >:(

```
x <- c(1,4,9,16,25)
y <- 1:5
y <- y*y
```

```
x==y
```

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

¿Qué pasaría si me confundo y escribo el operador de igualdad en lugar del de comparación?

```
y <- 5:9
y
```

```
## [1] 5 6 7 8 9
```

```
x = y
```

```
x
```

```
## [1] 5 6 7 8 9
```

```
y
```

```
## [1] 5 6 7 8 9
```

Para vectores grandes puedo usar la función `all()` que ya vimos arriba

```
x <- seq(1,10000,1)
y <- seq(1,10000,1)
all(x==y)
```

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

¿Cómo podríamos corrobar que son iguales usando `any`?

También podríamos utilizar que `TRUE` es `1` y que `FALSE` es `0`

¿Por qué este código nos dice que sí son iguales?

```
sum(x==y)
```

```
## [1] 10000
```

3.15.1 Factor

Los factores son un tipo de vector que puede tomar un número "limitado" de valores, que normalmente se utilizan como variables categóricas. Por ejemplo: macho/hembra. Es útil tener este tipo de objeto porque varios modelos estadísticos que se pueden correr en `R` los utilizan. A los valores que pueden tomar los elementos del factor se les conoce como *levels*.

```
x <- c(1,2,2,3,1,2,3,3,1,2,3,3,1)
x
```

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

```
as.factor(x)
```

```
##  [1] 1 2 2 3 1 2 3 3 1 2 3 3 1
## Levels: 1 2 3
```

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

```
##  [1] 1 2 2 3 1 2 3 3 1 2 3 3 1
## Levels: 1 2 3
```

Los factores son una manera computacionalmente eficiente de almacenar caracteres, pues cada valor único (*level*) se guarda solo una vez y a los datos se les asigna un valor entero.

```
meses = c("March","April","January","November","January",
       "September","October","September","November","August",
        "January","November","November","February","May","August",
        "July","December","August","August","September","November",
         "February","April")
meses
```

```
##  [1] "March"     "April"     "January"   "November"  "January"
##  [6] "September" "October"   "September" "November"  "August"
## [11] "January"   "November"  "November"  "February"  "May"
## [16] "August"    "July"      "December"  "August"    "August"
## [21] "September" "November"  "February"  "April"
```

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

```
##  [1] March     April     January   November  January   September
##  [7] October   September November  August    January   November
## [13] November  February  May       August    July      December
## [19] August    August    September November  February  April
## 11 Levels: April August December February January July March ... September
```

El que existan los *levels* permite realizar ciertas operaciones y manipular el contenido del factor.

```
table(meses)
```

```
## meses
##     April    August  December  February   January      July
##         2         4         1         2         3         1
##     March       May  November   October September
##         1         1         5         1         3
```

```
levels(meses)
```

```
##  [1] "April"     "August"    "December"  "February"  "January"
##  [6] "July"      "March"     "May"       "November"  "October"
## [11] "September"
```

```
levels(meses)[1]
```

```
## [1] "April"
```

```
levels(meses)[1]<-"Abril"
levels(meses)
```

```
##  [1] "Abril"     "August"    "December"  "February"  "January"
```

```
##  [6] "July"      "March"     "May"       "November"  "October"
## [11] "September"
```

```
meses
```

```
##  [1] March     Abril     January   November  January   September
##  [7] October   September November  August    January   November
## [13] November  February  May       August    July      December
## [19] August    August    September November  February  Abril
## 11 Levels: Abril August December February January July March ... September
```

3.15.2 Ejercicio: Temperaturas de Incubación

Supongamos que estamos realizando un experimento de cultivo bacteriano y registramos las temperaturas de incubación para diferentes muestras. Queremos calcular la temperatura media y la desviación estándar.

```
# Temperaturas de incubación (en grados Celsius)
temperaturas <- c(37, 37, 25, 30, 30, 37, 25, 25)

# Cálculo de la temperatura media
temp_media <- mean(temperaturas)
cat("Temperatura media:", temp_media, "°C\n")
```

```
## Temperatura media: 30.75 °C
```

```
# Cálculo de la desviación estándar
temp_desviacion <- sd(temperaturas)
cat("Desviación estándar:", temp_desviacion, "°C\n")
```

```
## Desviación estándar: 5.574175 °C
```

Supongamos que tenemos un conjunto de temperaturas de incubación de diferentes muestras bacterianas, y queremos identificar las muestras que están dentro de un rango de temperatura óptimo para el crecimiento bacteriano (entre 25°C y 37°C).

```
# Vectores con nombre: Muestras y Temperaturas
muestras <- c("Muestra1", "Muestra2", "Muestra3", "Muestra4")
temperaturas <- c(Muestra1 = 37, Muestra2 = 25, Muestra3 = 30, Muestra4 = 40)

# Filtrado de temperaturas dentro del rango óptimo
temperaturas_optimas <- temperaturas[temperaturas >= 25 & temperaturas <= 37]

# Muestras dentro del rango óptimo
muestras_optimas <- names(temperaturas_optimas)

cat("Muestras con temperatura óptima:", muestras_optimas, "\n")
```

```
## Muestras con temperatura óptima: Muestra1 Muestra2 Muestra3
```

Ejercicio 1. Lee la ayuda de `as.factor` para determinar cómo crear un factor "ordenado" 2. Crea un vector con los meses en los que todas las alumnas del grupo cumplen años. 3. Aprovecha los levels para generar un objeto que guarde el número de estudiantes que cumplen años cada mes.

Ejercicios

1. Genera un vector con el nombre de 10 virus
2. Asocia esos nombres con su número de acceso en NCBI
3. Genera otro vector que contega los tamaños en pb y los nombres
4. Determina cuáles son mayores de 300 bp
5. Asocia un subconjunto de vectores que sean mayores (menores a 300 bp) y mayores (mayores a 300 bp)
6. Haz un histograma con los tamaños de todos
7. Dibuja un boxplot con los tamaños de todos.Pon en el eje los nombres de todos.

Ejercicios adicionales

1. Crea un vector llamado "v1" con los números 2, 4, 6, 8 y 10.
2. Crea un vector llamado "v2" con los números 1, 3, 5, 7 y 9.
3. Suma los vectores "v1" y "v2" elemento por elemento.
4. Multiplica el vector "v1" por el escalar 3.
5. Calcula la media del vector "v2".
6. Encuentra el valor mínimo del vector "v1".
7. Crea un vector llamado "v3" con los números 2, 4, 6, 8 y 10.
8. Compara los vectores "v1" y "v3" y determina si son iguales.
9. Crea un vector "v4" con los primeros 10 números impares.
10. Encuentra los elementos comunes entre los vectores "v2" y "v4"
11. Crea un vector llamado "v1" con números aleatorios enteros entre 1 y 50.
12. Ordena el vector "v1" de forma descendente.
13. Encuentra la mediana del vector "v1".
14. Crea un vector llamado "v2" con números aleatorios enteros entre 10 y 20, de longitud 5. 15 .Calcula el producto punto entre "v1" y "v2". 16 .Crea un vector llamado "v3" con números aleatorios entre 0 y 1, de longitud 10.
15. Normaliza el vector "v3".

Capítulo 4

Matrices

En R, una matriz es un tipo de dato bidimensional que se utiliza para almacenar elementos de datos del mismo tiposí **solo del mismo tipo** organizados en filas y columnas. Las matrices son una extensión de los vectores y pueden ser útiles para realizar operaciones matemáticas y estadísticas.

4.1 Creación de matrices

Para crear una matriz podemos usar la función `matrix()`. Dicha función requiere de, al menos, un vector e indicar al menos una dimensión.

Ejemplo:

```
y <- matrix(c(1,5,8,-4), nrow=2, ncol=2) #nrow: indica el número de renglones & ncol: indica el n
y
```

```
##      [,1] [,2]
## [1,]    1    8
## [2,]    5   -4
```

Por default, la matriz va agregando los datos por columnas.

```
z <- matrix(c(TRUE, FALSE,rep(c(TRUE, FALSE),3)),nrow=4)
z
```

```
##       [,1]  [,2]
## [1,]  TRUE  TRUE
## [2,] FALSE FALSE
## [3,]  TRUE  TRUE
## [4,] FALSE FALSE
```

¿Por qué sólo es necesario indicar una dimensión (renglones)?

También se puede indicar que se cambien el orden de llenado de la matriz, es decir, en lugar de que lo haga por columnas, lo haga por renglones.

```
m <- matrix(c(1,2,3,4,5,6),nrow=2,byrow=TRUE)
m
```

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

4.2 Dimensiones de un matriz

La dimensión de una matriz es el número de renglones y de columnas respectivamente. Se puede obtener usando la función `dim()` de *dimensión*. **NOTA** `dim()` no se puede emplear en elementos unidemsionales (Ej: vectores)

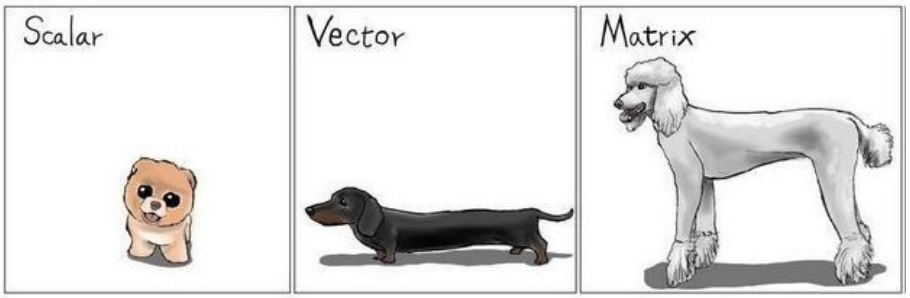
```
dim(y)
```

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

```
dim(z)
```

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

Así una matriz se distingue de un vector ya que tiene, además de renglones, columnas.



Elementos de una matriz

Para acceder a elementos de un objeto con *indices* (componentes que tienen una posición asignada), debemos usar *corchetes* `[]`. En el caso de la matriz se debe indicar la posición de ambas dimensiones `[renglón,columna]`.

Ejemplo: En este caso se desea seleccionar el elemento del primer renglón, segunda columna.

```
y[1,2]
```

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

Ejemplo: En este caso se quiere seleccionar todos los elementos del primer renglón.

```
y[1,]
```

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

Ejemplo: En este caso se quiere seleccionar todos los elementos de la segunda columna.

```
y[,2]
```

```
## [1]  8 -4
```

4.3 Creación de matriz "vacia"

Una forma **mucho menos eficiente** de definir una matriz es declarando una matriz sin elementos (matriz vacía) y después llenándolos de forma explícita asignando un valor distinto a cada posición.

```
 y <- matrix(nrow=2,ncol=2)
y[1,1] <- "Esta"
y[2,1] <- "es"
y[1,2] <- "una"
y[2,2] <- "matriz"
y
```

```
##      [,1]   [,2]
## [1,] "Esta" "una"
## [2,] "es"   "matriz"
```

4.4 Operaciones rbind y cbind en R para Matrices

En R, las funciones `rbind()` y `cbind()` se utilizan para unir matrices por renglones y columnas, respectivamente. Además, la función `t()` se utiliza para transponer una matriz.

4.5 Uso de `rbind()` para unir matrices por renglones

La función `rbind()` se utiliza para unir matrices por renglones. Por ejemplo, considera las siguientes dos matrices:

```
# Matrices de ejemplo
matriz1 <- matrix(1:6, nrow = 2)
```

```
matriz2 <- matrix(7:12, nrow = 2)

# Unir matrices por renglones
matriz_unida <- rbind(matriz1, matriz2)
matriz_unida
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
## [3,]    7    9   11
## [4,]    8   10   12
```

4.6 Uso de cbind() para unir matrices por columnas

La función cbind() se utiliza para unir matrices por columnas. A continuación, se muestra un ejemplo de cómo unir dos matrices por columnas:

```
# Unir matrices por columnas
matriz_unida_columnas <- cbind(matriz1, matriz2)
matriz_unida_columnas
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    3    5    7    9   11
## [2,]    2    4    6    8   10   12
```

Ejercicio

¿Cómo se llenaría una matriz vacia a partir de vectores? ¿El vector tendría que tener la misma longitud que la columna o el renglón de la matriz? ¿Qué pasaría si la longitud del vector es diferente a la columna o renglón de la matriz? ¿Cómo podrías emplear `cbind()` & `rbind()`?‘

4.7 Operaciones con matrices

4.7.1 Multiplicación de un escalar con una matriz

```
3*m
```

```
##      [,1] [,2] [,3]
## [1,]    3    6    9
## [2,]   12   15   18
```

4.7.2 Suma de dos matrices

```
m + m
```

```
##      [,1] [,2] [,3]
## [1,]    2    4    6
## [2,]    8   10   12
```

```
n<-matrix(c(2,3,4,5,6,7),ncol=3)
m+n
```

```
##      [,1] [,2] [,3]
## [1,]    3    6    9
## [2,]    7   10   13
```

Para sumar matrices deben tener las mismas dimensiones

```
dim(n)
```

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

```
dim(m)
```

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

```
(dim(n)-dim(m))==0
```

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

4.7.3 Multiplicación de matrices

Se utiliza el operador `%*%`. Sí. Son tres caracteres. E incluyen dos `%`. No hay espacios y es un sólo operador .

```
n<-matrix(c(2,3,4,5,6,7),ncol=2)
n
```

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

```
m %*% n
```

```
##      [,1] [,2]
## [1,]   20   38
## [2,]   47   92
```

¿Recuerdas cuál es el criterio para calcular el producto de matrices? ¿Recuerdas cómo se multiplican dos matrices?

4.8 Uso de la función `t()` para transponer una matriz

La función `t()` se utiliza para transponer una matriz, es decir, intercambiar renglones por columnas y viceversa. Veamos un ejemplo:

```
# Transponer una matriz
matriz_transpuesta <- t(matriz_unida_columnas)
matriz_transpuesta
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
## [3,]    5    6
## [4,]    7    8
## [5,]    9   10
## [6,]   11   12
```

4.9 Seleccionar elementos de matrices

Para seleccionar elementos de matrices se hace de forma análoga a vectores, es decir, se utiliza el operador `[]`. Sólo que esta vez hay que indicar tanto los renglones como la columna en ese orden

```
m[2,3]  # Este es el segundo renglón tercera columna de m
```

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

```
n[3,2]  # Este es el elemento que está en el renglón 3 y columna 2 de la matriz n
```

```
## [1] 7
```

4.9.1 Seleccionar todo(a) un(a) renglón(columna)

Para seleccionar todos los elementos de un renglón dado se utiliza la siguiente sintáxis

```
m[2,]  # Todos los elementos que están en el segundo renglón
```

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

Para una columna

```
m[,3] # Toda la tercera columna
```

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

4.9.2 Selecccionar elementos de una matriz

¿Qué hace lo siguiente?

```
m[1:2,1]
```

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

```
m[1:2,2:3]
```

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

```
m[-1,]
```

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

```
m[-1,-c(1,3)]
```

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

4.10 Nombres a renglones y columnas

Al igual que con vectores le podemos poner nombres tanto a renglones como a columnas para ello utilizamos `rownames()` y `colnames()`

```
m   # No tengo nombres :(
```

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

```
colnames(m)<-LETTERS[1:3]
rownames(m)<-letters[5:6]
```

```
m   # Ahora sí. Feos, pero nombres :) :)
```

```
##   A B C
## e 1 2 3
## f 4 5 6
```

```
m["e","C"]
```

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

```
m["e","C"]==m[1,3]
```

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

4.10.1 Ejercicios

1. Genera dos matrices aleatorias de tamaño 3×3 y luego suma ambas matrices.
2. Crea dos matrices aleatorias, una de tamaño 2×3 y otra de tamaño 3×4. Luego, calcula su producto matricial.
3. Crea una matriz aleatoria de tamaño 4×3 y encuentra su matriz transpuesta.
4. Genera una matriz cuadrada aleatoria de tamaño 4×4 y calcula su determinante.
5. Crea una matriz cuadrada aleatoria de tamaño 3×3 y encuentra su inversa.
6. Genera una matriz aleatoria de tamaño 5×5 y extrae el tercer renglón y la segunda columna.
7. Crea una matriz diagonal aleatoria de tamaño 4×4 y encuentra sus elementos diagonales.
8. Define una matriz de coeficientes $\mathcal{A}$ y un vector de términos constantes b. Luego, resuelve el sistema de ecuaciones lineales $\mathcal{A}x = b$.
9. Genera una matriz aleatoria de tamaño 3×3 y multiplica cada uno de sus elementos por un escalar, por ejemplo, 2.
10. Crea una matriz simétrica aleatoria de tamaño 4x4 y verifica si es simétrica.
11. **Comparación de Expresión Génica entre Condiciones**

Descripción: Supongamos que tienes una matriz de expresión génica con 6 genes y 4 condiciones experimentales.

- Crea una matriz llamada `expresion_genica` con 6 genes y 4 condiciones (rellena con datos aleatorios).
- Asigna nombres de genes a las filas y nombres de condiciones a las columnas.
- Calcula el promedio de expresión génica para cada gen.

12. **Red de interacciones proteína-proteína**

Descripción: Modela una matriz de interacciones entre proteínas donde 1 indica interacción y 0 indica no interacción.

- Crea una matriz interacciones de 5x5 con valores binarios.
- Asigna nombres de proteínas a las filas y columnas.
- Encuentra cuántas interacciones tiene cada proteína (suma de cada fila).

13. **Variación de concentraciones de metabolitos**

Descripción: Supongamos que tienes datos de concentraciones de 4 metabolitos en 3 tipos de tejidos diferentes.

- Crea una matriz concentraciones de 4×3 con datos aleatorios.
- Asigna nombres de metabolitos a las filas y tipos de tejidos a las columnas.
- Normaliza las concentraciones para cada metabolito (dividiendo por el máximo valor de cada fila).

14. **Análisis de tasa de crecimiento bacteriano**

Descripción: Modela una matriz con las tasas de crecimiento de 5 cepas bacterianas en 4 medios de cultivo diferentes.

- Crea una matriz crecimiento de 5x4 con datos aleatorios.
- Asigna nombres de cepas bacterianas a las filas y nombres de medios de cultivo a las columnas.
- Calcula la media y la desviación estándar de la tasa de crecimiento para cada medio de cultivo.

15. **Matriz de distancias genéticas**

Descripción: Supongamos que tienes una matriz de distancias genéticas entre 6 especies.

- Crea una matriz distancias de 6x6 con valores aleatorios.
- Asigna nombres de especies a las filas y columnas.
- Encuentra la especie más cercana y más lejana para cada especie (índices de los mínimos y máximos valores en cada fila, excluyendo la diagonal). ### Ejercicios avanzados

1. **Ejercicio 1: Transformaciones de Matrices**

- Crea una matriz de 5x5 con números aleatorios entre 1 y 100.
- Encuentra la transpuesta de la matriz.
- Calcula la inversa de la matriz original (asegúrate de que la matriz sea invertible).
- Multiplica la matriz original por su inversa y verifica si el resultado es la matriz identidad.

2. **Ejercicio 4: eigenvalores y eigenvectores**

- Crea una matriz simétrica de 5x5 con números aleatorios entre 1 y 10.
- Calcula los eigenvalores y eigenvectores de la matriz. Verifica la propiedad de los autovalores y autovectores: $\mathcal{A} \cdot v = \lambda \cdot v$

Capítulo 5

Data Frames

Un Dataframe en R es una estructura de datos rectangular que se compone de renglones y columnas, donde cada columna puede tener un tipo de datos diferente. Los Dataframes son una de las estructuras de datos más utilizadas en R, ya que son la forma estándar de almacenar datos tabulares.

5.1 Crear un Dataframe en R

Podemos crear un DataFrame en R utilizando la función `data.frame()`. Aquí hay un ejemplo de cómo crear un DataFrame simple con datos de estudiantes:

```
# Crear un DataFrame de estudiantes
estudiantes <- data.frame(
  nombre = c("Juan", "María", "Pedro", "Laura"),
  edad = c(20, 22, 21, 23),
  puntaje = c(85, 90, 88, 92)
)

# Ver el DataFrame
print(estudiantes)
```

```
##   nombre edad puntaje
## 1   Juan   20      85
## 2  María   22      90
## 3  Pedro   21      88
## 4  Laura   23      92
```

Para crear un Dataframe en R, puedes utilizar la función `data.frame()`. Por ejemplo, para crear un Dataframe con información del genoma de algunos microorganismos, podrías escribir lo siguiente:

```
dna_data <- data.frame(
  secuencia = c("ATCGATCG", "GCTAGCTA", "TTAAGGCT"),
  tamaño = c(8, 8, 8),
  contenido_GC = c(0.5, 0.4, 0.3)
)

print(dna_data)
```

```
##   secuencia tamaño contenido_GC
## 1  ATCGATCG      8          0.5
## 2  GCTAGCTA      8          0.4
## 3  TTAAGGCT      8          0.3
```

NOTA También puedes emplear `View()` para visualizar el DataFrame.

```
View(dna_data)
```

En este ejemplo, el dataframe tiene tres columnas: "secuencia", "tamaño" y "contenido_GC". La columna "secuencia" contiene cadenas de caracteres (strings) que representan las bases del ADN, mientras que las otras dos columnas contienen valores numéricos.

5.2 Acceder a los datos de un dataframe

Para seleccionar elementos o acceder a algún dato de un dataframe, se hace de forma análoga a las matrices, es decir, se utiliza el operador de subíndice `[]`.

Por ejemplo, para acceder al segundo renglón de la columna "secuencia" en el Dataframe "dna_data", se puede escribir lo siguiente:

```
dna_data[2, "secuencia"]
```

```
## [1] "GCTAGCTA"
```

Esto devolvería la cadena de caracteres "GCTAGCTA".

También puedes acceder a varios renglones o columnas a la vez. Por ejemplo, para acceder a los primeros dos renglones de las columnas "tamaño" y "contenido_GC" en el Dataframe "dna_data", podrías escribir lo siguiente:

```
dna_data[1:2, c("tamaño", "contenido_GC")]
```

```
##   tamaño contenido_GC
## 1      8          0.5
## 2      8          0.4
```

Esto devolvería un dataframe con dos renglones y dos columnas.

5.3 Agregar y eliminar renglones y columnas en un Dataframe en R

Para agregar una nueva columna a un Dataframe en R, puedes utilizar el operador de asignación `<-`. Por ejemplo, para agregar una columna que represente la Temperatura de Melting (o Fusión) en el Dataframe "dna_data", podrías escribir lo siguiente:

```
dna_data$temperatura_melting <- c(24, 24, 20)
```

Esto crearía una nueva columna llamada "temperatura_melting" en el Dataframe "dna_data" y la inicializaría con los valores proporcionados.

Para eliminar una columna de un Dataframe en R, puedes utilizar el operador de subíndice `[]` con un valor nulo para la columna que deseas eliminar. Por ejemplo, para eliminar la columna "temperatura_melting" del Dataframe "dna_data", podrías escribir lo siguiente:

```
dna_data$temperatura_melting <- NULL
```

Para agregar un nuevo renglón a un Dataframe en R, puedes utilizar la función `rbind()`. ¿También se puede aplicar `cbind()` en Dataframe?

Ejercicio

Empleando la siguiente fórmula para calcular la Temperatura de Fusión: $T_m = 4(G + C) + 2(A + T)$, diseña una secuencia que cumpla con una T_m de 55°. Agrega sus características (secuencia, tamaño, cantidad GC & T_m) en un nuevo renglón. También será necesario agregar la columna Tm para indicar la Temperatura de Fusión de cada secuencia. Usa el Dataframe `dna_data` como base.

5.4 Importar archivos externos

Normalmente queremos trabajar con datos generados en excel o en google sheets que vienen de una encuesta, de un experimento etc. R tiene la capacidad de hacer eso con varias funciones y normalmente el resultado es un data.frame

5.4.1 Importar un archivo csv en R

Un archivo csv (Comma Separated Values) es un archivo de texto que contiene datos en formato tabular, donde cada renglón representa un registro y cada columna representa una variable. Para importar un archivo CSV en R, puedes utilizar la función `read.csv()`. Por ejemplo, si tienes un archivo llamado "ventas.csv" en tu directorio de trabajo actual, puedes importarlo de la siguiente manera:

```
#ventas <- read.csv("ventas.csv")
```

Esto creará un dataframe llamado "ventas" en R, que contendrá los datos del archivo CSV.

Si el archivo CSV utiliza un separador de campos diferente a la coma, puedes utilizar la función read.csv2() en su lugar. Por ejemplo, si el archivo CSV utiliza un punto y coma como separador de campos, puedes importarlo de la siguiente manera:

```
#ventas <- read.csv2("ventas.csv")
```

5.4.2 Importar un archivo TSV en R

Un archivo TSV (Tab Separated Values) es similar a un archivo CSV, pero utiliza **tabulaciones** en lugar de comas para separar los campos. Para importar un archivo TSV en R, puedes utilizar la función read.delim(). Por ejemplo, si tienes un archivo llamado "ventas.tsv" en tu directorio de trabajo actual, puedes importarlo de la siguiente manera:

```
#ventas <- read.delim("ventas.tsv")
```

Esto creará un Dataframe llamado "ventas" en R, que contendrá los datos del archivo TSV.

Si el archivo TSV utiliza un separador de campos diferente a la tabulación, puedes utilizar la función `read.delim2()` en su lugar.

5.4.3 Importar un archivo Excel en R

Un archivo de Excel es un formato de archivo popular para almacenar datos en formato tabular. Para importar un archivo de Excel en R, puedes utilizar la función readxl::read_excel() del paquete "readxl". Por ejemplo, si tienes un archivo llamado "ventas.xlsx" en tu directorio de trabajo actual, puedes importarlo de la siguiente manera:

```
library(readxl)
#ventas <- read_excel("ventas.xlsx")
```

5.4.4 Ejercicios

1. **Datos de crecimiento de cultivos bacterianos**

Descripción: Supón que tienes un data.frame con datos de crecimiento de cultivos bacterianos en diferentes condiciones.

- Crea un data.frame crecimiento con las siguientes columnas: Cepa, Medio, TasaCrecimiento, Temperatura.

- Llena el data.frame con datos aleatorios para 10 cepas en 3 medios diferentes y 4 temperaturas distintas.
- Encuentra la media y la desviación estándar de la tasa de crecimiento por cada medio.

2. **Perfil de resistencia antibiótica**

Descripción: Supón que tienes un data.frame con datos de resistencia antibiótica de diferentes cepas bacterianas.

- Crea un data.frame resistencia con las columnas: Cepa, Antibiótico, Resistencia (0 para sensible, 1 para resistente).
- Llena el data.frame con datos aleatorios para 5 cepas y 5 antibióticos diferentes.
- Calcula el porcentaje de resistencia para cada antibiótico.

3. **Datos de abundancia de secuencias**

Descripción: Supón que tienes un data.frame con datos de abundancia de secuencias de diferentes microorganismos en distintas muestras.

- Crea un data.frame secuencias con las columnas: Muestra, Microorganismo, Abundancia.
- Llena el data.frame con datos aleatorios para 8 microorganismos en 5 muestras diferentes.
- Encuentra la abundancia total y promedio por muestra.

4. ** Datos de Producción de Ácido Láctico**

Descripción: Supón que tienes un data.frame con datos de producción de ácido láctico por diferentes cepas en distintas condiciones.

- Crea un data.frame fermentacion con las columnas: Cepa, Condición, Producción.
- Llena el data.frame con datos aleatorios para 7 cepas y 4 condiciones diferentes.
- Encuentra la cepa que produce la mayor cantidad de ácido láctico en cada condición.

Capítulo 6

Listas

Las listas son una estructura de datos muy versátil en R, que permiten almacenar una colección de elementos. A diferencia de los vectores, una lista puede contener elementos de diferentes tipos, como números, cadenas y hasta otras listas.

En R, las listas pueden contener una amplia variedad de tipos de datos, incluyendo números, vectores, matrices, y data frames. Esto las hace extremadamente versátiles para la gestión de datos complejos.

Ejemplos

- Creación de una Lista

```
# Creando una lista con diferentes tipos de datos
compleja_lista <- list(
  numero = 42,
  vector = c(1, 2, 3),
  matriz = matrix(c(1, 2, 3, 4), nrow = 2),
  data_frame = data.frame(nombre = c("E. coli", "S. aureus"), gram = c("+", "-"))
)

print(compleja_lista)
```

```
## $numero
## [1] 42
##
## $vector
## [1] 1 2 3
##
## $matriz
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

```
##
## $data_frame
##      nombre gram
## 1   E. coli    +
## 2 S. aureus    -
```

Para acceder a los elementos de una lista, puedes usar el doble corchete [[ ]] o el operador de dólar $. El doble corchete es útil para acceder a los elementos por su índice, mientras que el operador de dólar se usa con nombres.

Accceso por índice

```
# Accediendo al vector dentro de la lista
vector_en_lista <- compleja_lista[[2]]
print(vector_en_lista)
```

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

6.0.1 Acceso por Nombre

```
# Accediendo al data frame por nombre
data_frame_en_lista <- compleja_lista$data_frame
print(data_frame_en_lista)
```

```
##      nombre gram
## 1   E. coli    +
## 2 S. aureus    -
```

6.1 Ejercicios

6.1.1 Ejercicios Propuestos con listas

1. Crea una lista que contenga al menos cuatro tipos diferentes de datos (incluyendo al menos un vector, una matriz, y un data frame). Luego, escribe código para acceder a cada uno de estos elementos por su índice.
2. Añade un nuevo elemento a la compleja_lista que sea otra lista conteniendo información relevante a un experimento microbiológico (p.ej., fechas, resultados de crecimiento, tipo de medio de cultivo). Accede a un elemento específico dentro de esta lista anidada.

Recuerda, el uso efectivo de listas en R puede ayudarte a gestionar y manipular una amplia gama de conjuntos de datos complejos, especialmente útil en campos como la Microbiología.

Capítulo 7

Estructuras de selección

1. `if`
2. `if` … `else`
3. `ifelse`
4. `if ... else if ...else if ...else`

7.1 If (si condicional)

La instrucción `if` nos permite probar una condición y esa condición debe arrojar un valor booleano, es decir, un valor de verdad (`TRUE` o `FALSE`) Si la condición es verdadera se ejecuta lo que está dentro de los corchetes, de lo contrario, ejecuta lo que sigue después del corchete de cierre.

Definición: Lo que se encuentra dentra del corchete se llama cuerpo (*body*) de la declaración `if`

La sintaxis de una condición `if` consiste en lo siguiente:

```
if (condicion){

si la condicion es verdadera

Ejecuta TODO lo que está en los corchetes

}
```

7.1.1 Sintaxis

```
if(<condition>) {
        ## Hace algo
}
## Continúa con el resto del código
```

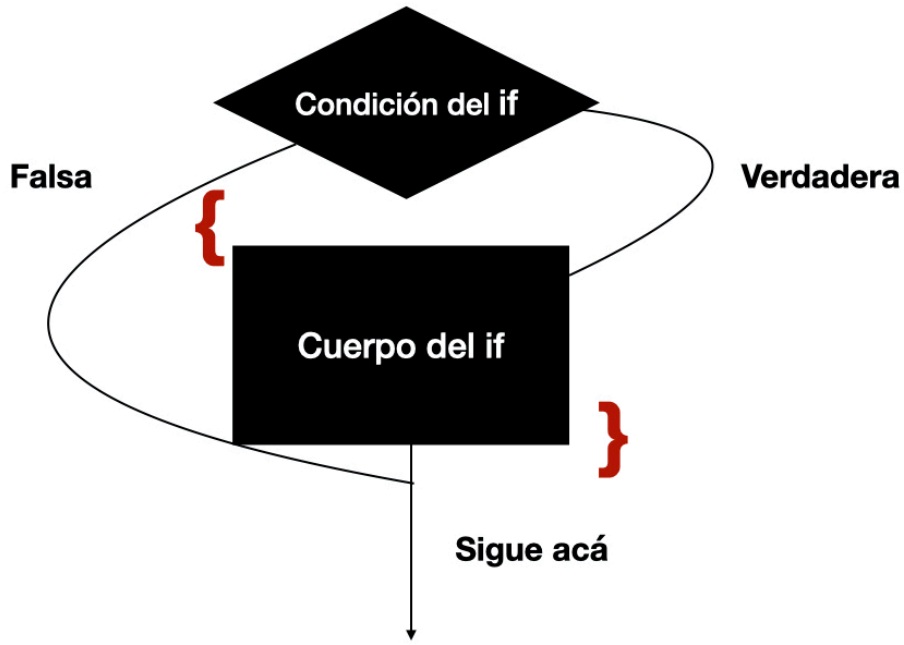


Figure 7.1: Diagrama de flujo del If

7.1.2 Errores comunes en el `if`

1. No incializar la variable de la condición.
2. La condición no arroja un valor de verdad.
3. No poner todo lo que quieres que haga **dentro** de los corchetes.
4. Este no es un error, es más bien una advertencia, si la condición arroja un sólo valor de verdad sólo toma en cuenta el primero de ellos.

###Ejemplos de uso del `if`

```
mayor_de_edad<-18

edad<-20

if(edad >=mayor_de_edad){
  print("Eres mayor de edad")
}
```

```
## [1] "Eres mayor de edad"
```

```
x<-5+4
print(x)
```

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

```
minimo<-20000
dinero<-15000

if(dinero>=minimo){
  print("¿Cómo está Cancún?")
  print("La vida es buena")
  sobrante<-dinero-minimo
  print(paste("Me queda $", sobrante))
}
print("Acá sigue")
```

```
## [1] "Acá sigue"
```

7.1.2.1 Ejercicios

1. Elabora un programa que compare tu estatura con tu ídolx y determine si eres más altx.
2. Toma dos archivos fasta de virus distintos. Leelos con Biostrings y compara sus tamaños (en bp) y determina si el primero es más grande que el segundo.
3. A partir del archivo de anotación del genoma de un organismo determina toma dos proteínas al azar y compara sus tamaños. Toma todos los genes de la cadena positiva y todos los de la negativa compara sus tamaños promedio y determina cuál de estos es más grande.

7.2 Combinación de operadores booleanos

Los operadores lógicos o booleanos se pueden combinar para formar enunciados complejos por ejemplo:

1. Tengo vacaciones (del trabajo y/o la escuela)
2. Tengo dinero

Si las dos condiciones son ciertas entonces puedo hacer algo

También podría ser que basta con que una de ellas sea cierta para que haga algo.

7.2.1 And (&)

El operador booleano `&` representa el "Y" lógico. Estos operadores binarios nos sirven para unir al menos dos enunciados que tienen valor de verdadero o falso (Tengo dinero (V/F), Tengo vacaciones (V/F))

Con estas dos operaciones puedo unirlas utilizando el operador "Y" lógico (AND (`&`)) representado en R con el símbolo del *ampersand* (`&`)

Tengo dinero AND Tengo vaciones

Para saber el valor booleano (V/F) del enunciado anterior debemos conocer los valores de verdad de los enunciados por separado

Por ejemplo, podemos representar al primer enunciado por `p` y al segundo enunciado por `q`

```
p: Tengo dinero
```

```
q: Tengo vacaciones
```

Para saber cuál es el valor de verdad del enunciado compuesto debemos ver cuáles son todas las combinaciones de valores de verdad de los enunciados que la componen: p verdadero y q verdadero, p verdadero y q falso, p falso y q verdadero, p falso y q falso. Eso se resumen en las tablas de verdad de los operadores

Table 7.1: Tabla de verdad del AND

p	q	p & q
V	V	V
V	F	F
F	V	F
F	F	F

Es decir, el `&` solo es **verdadero** cuando ambas condiciones son **verdaderas**.

Esto representa lo que se observa en la realidad: es decir, solo hago algo si tengo y tengo vacaciones. Si una de ellas no se cumple (es decir, es falsa) entonces no se lleva a cabo la acción.

7.2.2 OR (|)

El operador booleano `|` representa el "O" lógico. Estos operadores binarios nos sirven para unir al menos dos enunciados que tienen valor de verdadero o falso (Tengo dinero (V/F), Tengo vacaciones (V/F))

Con estas dos operaciones puedo unirlas utilizando el operador `O` lógico (OR (`|`)) representado en R con el símbolo de *la barrita* (`|`)

Tengo dinero OR Tengo vaciones

Para saber el valor booleano (V/F) del enunciado anterior debemos conocer los valores de verdad de los enunciados por separado

Por ejemplo podemos representar al primer enunciado por `p` y al segundo enunciado por `q`

```
p: Tengo dinero
```

`q: Tengo vacaciones`

Para saber cuál es el valor de verdad del enunciado compuesto debemos ver cuáles son todas las combinaciones de valores de verdad de los enunciados que la componen: p verdadero y q verdadero, p verdadero y q falso, p falso y q verdadero, p falso y q falso. Eso se resumen en las tablas de verdad de los operadores

Table 7.2: Tabla de verdad del operador OR

p	q	p | q
V	V	V
V	F	V
F	V	V
F	F	F

Es decir haría algo, por ejemplo, irme a la playa cuando **al menos** una condición se cumpla. Por ejemplo que tenga dinero aunque no tenga vacaciones, que tenga vacaciones aunque no tenga dinero y, obviamente, también cuando las dos se cumplen.

Es decir, el `|` solo es **falso** cuando ambas condiciones son **falsas**.

7.2.3 Ejemplos de combinaciones

Por ejemplo es útil para intervalos

$$18 \leq edad \leq 29$$

Esta condición la podemos expresar mediante la combinación de dos: la edad debe ser mayor igual a 18 y (**AND**, `&`) la edad debe ser menor o igual que 29

```
if (edad >= 18 & edad <=29){
  print("Te toca vacunarte")
}
```

```
## [1] "Te toca vacunarte"
```

Pregunta: ¿qué pasaría si se pone un **OR** como unión entre las dos condiciones

```
if (edad >= 18 | edad <=29){
  print("Te toca vacunarte")
}
```

```
## [1] "Te toca vacunarte"
```

o así (¿es lo mismo?)

```
if (edad <= 29 | edad >= 18){
  print("Te toca vacunarte")
}
```

```
## [1] "Te toca vacunarte"
```

7.3 Ejercicio

1. ¿Cómo harías una condición que considere que te gusta el mole y el pozole?
2. ¿Cómo harías una condición que considere que te gusta el mole o el pozole?

7.4 If ... else (si ... de otro modo)

Si además quieres que se ejecute algo cuando la condición es **falsa** entonces debes usar la declaración `if … else`

```
if (condición) { # Si la condición es cierta
hace esto
y esto
y esto
} else { # De otro modo, es decir si es falsa hace lo que #está en el corchete
entonces hace esto otro
y esto otro
y esto
}
```

```
minimo<-20000
vacaciones<-"SI"
dinero<-21000
if(dinero>=minimo & vacaciones=="SI"){
  print("Me voy a la playa, looser")
}else{
  print("Me quedo en mi casa")
}
```

```
## [1] "Me voy a la playa, looser"
```

7.5 ifelse

Si la condición es muy simple ,tanto para cuando es verdadero como cuando es falso se puede implementar la función `ifelse` en una línea. Es equivalente a la condición compuesta pero ahorramos código.

```
edad<-21
ifelse(edad>=18, "Ya eres grande","Todavia no puedes beber (legalmente)")
```

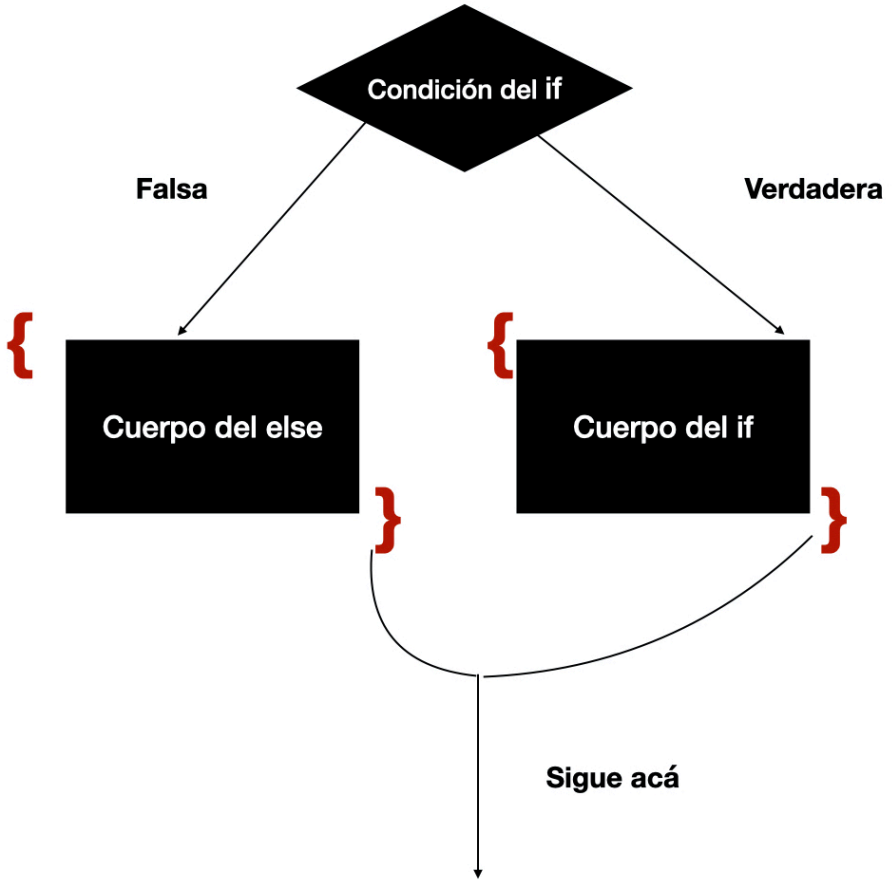


Figure 7.2: Diagrama_if_else

```
## [1] "Ya eres grande"
```

```
edad<-12
ifelse(edad>=18, "Ya eres grande","Todavia no puedes beber (legalmente)")
```

```
## [1] "Todavia no puedes beber (legalmente)"
```

7.6 If ... else if ... else (si, si no si , si no si, si no)

Si tienes más opciones, es decir no alternativas, puedes usar la sentencia `if … else if …else if …else`

Importante Esta estructura se ejecuta solo en la primera que sea verdadera o si no hay una verdadera ejecuta lo que esta en el `else`

```
if ( condicion 1) {
Hace cosas
} else if ( condcion 2) {
Hace otras cosas
} else if ( condicion 3) {
Hace estas otras cosas
} else {
No le queda de otra y hace esto
}
```

```
numero<-3
if(numero > 0){
  print("Tu número es positivo")
}else if (numero <0){
  print("Tu número es negativo")
}else{
  print("Tu número es cero")
}
```

```
## [1] "Tu número es positivo"
```

```
numero<- -27
if(numero > 0){
  print("Tu número es positivo")
}else if (numero <0){
  print("Tu número es negativo")
}else{
  print("Tu número es cero")
}
```

```
## [1] "Tu número es negativo"
```

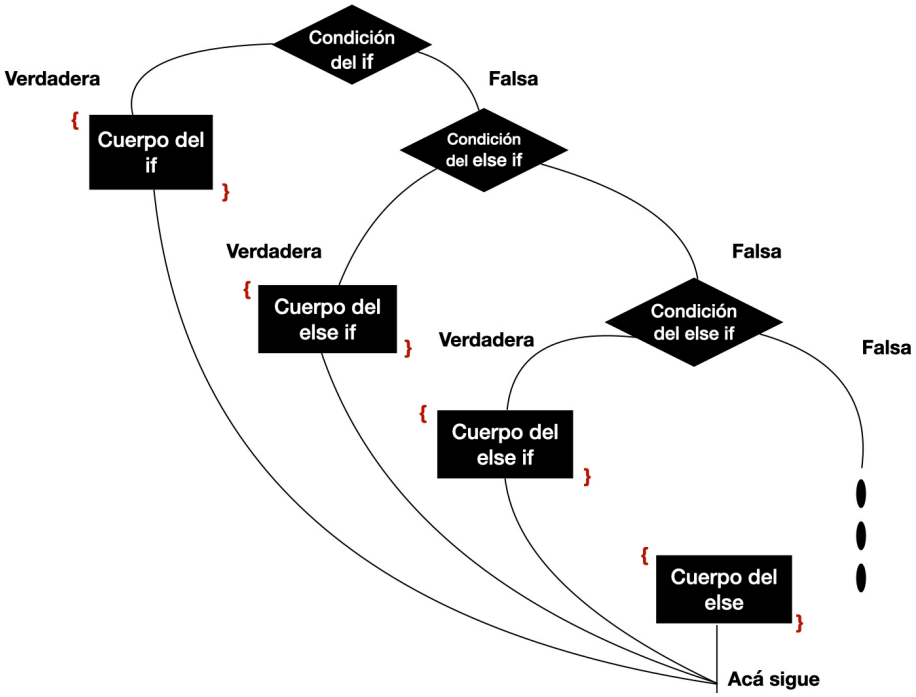


Figure 7.3: Diagrama de flujo del if… else if… else if…else

```
numero<- 0
if(numero > 0){
  print("Tu número es positivo")
}else if (numero <0){
  print("Tu número es negativo")
}else{
  print("Tu número es cero")
}
```

```
## [1] "Tu número es cero"
```

Pregunta: ¿por qué no es necesario poner un if en el último else?

7.7 Ejercicios

1. Elabora un programa que con tu fecha de cumpleaños te diga en qué estación del año naciste.
2. Elabora un programa que a partir de las calificaciones de tus examenes parciales y 8 quincenales arroje si exentarás o no este curso usando los criterios definidos en el programa del curso. Asume que en las tareas y demás actividades tienes 10.

Capítulo 8

Funciones en R

Las funciones definidas por el usuario en R son bloques de código que realizan una tarea específica y se pueden llamar desde cualquier lugar del programa. Las funciones en R se definen usando la palabra clave function seguida del nombre de la función, paréntesis y llaves.

Las funciones toman argumentos de entrada y producen un resultado como salida. Las funciones definidas por el usuario son una herramienta poderosa en R que permiten reutilizar código y automatizar tareas. Una vez que domines la sintaxis básica de la función en R, puedes comenzar a crear funciones más avanzadas y complejas para adaptarse a tus necesidades.

8.1 Sintaxis básica de una función en R

La sintaxis básica para definir una función en R es la siguiente:

```
nombre_de_la_funcion <- function(arg1, arg2, ...) {
  # Cuerpo de la función
  resultado <- ...
  return(resultado)
}
```

Donde:

- `nombre_de_la_funcion`: el nombre que le das a tu función. -`function`: la palabra *function*
- `arg1`, `arg2`, ...: los argumentos de entrada que toma la función (opcional).
- `resultado`: el resultado que devuelve la función (opcional).

El cuerpo de la función es donde se escribe el código que realiza la tarea específica. El resultado de la función se devuelve con la función `return()`.

8.2 Ejemplo de función definida por el usuario en R

Aquí hay un ejemplo de una función definida por el usuario que toma dos argumentos x e y y devuelve la suma de ambos:

```
mi_suma <- function(x, y) {
  resultado <- x + y
  return(resultado)
}
```

Para usar esta función, simplemente llámala con los argumentos que deseas pasar:

```
mi_suma(3, 5)
```

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

8.3 Ejemplo de función con argumentos por defecto en R

Las funciones en R también pueden tener argumentos por defecto que se utilizan si no se proporciona ningún valor para ellos. Aquí hay un ejemplo de una función que tiene dos argumentos, `x` e `y`, y `y` tiene un valor por defecto de 2:

```
mi_funcion <- function(x, y = 2) {
  resultado <- x * y
  return(resultado)
}
```

En este caso, si no se proporciona un valor para `y`, se utilizará el valor por defecto de 2:

```
mi_funcion(3)
```

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

También puede proporcionar un valor diferente para y, si es necesario:

```
mi_funcion(3, 5)
```

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

8.4 Definir una función con un parámetro opcional

```
saludar <- function(nombre, saludo = "Hola") {
  mensaje <- paste(saludo, nombre)
  return(mensaje)
}
```

8.4.1 Llamar a la función saludar sin proporcionar el parámetro opcional

```
mensaje1 <- saludar("Juan")
print(mensaje1)
```

```
## [1] "Hola Juan"
```

8.4.2 Llamar a la función saludar proporcionando el parámetro opcional

```
mensaje2 <- saludar("María", "Buenos días")
print(mensaje2)
```

```
## [1] "Buenos días María"
```

8.5 Definir una función para calcular el área de un círculo

```
area_circulo <- function(radio) {
  area <- pi * radio^2
  return(area)
}
# Ejemplo de uso

radio <- 3
area <- area_circulo(radio)
print(paste("El área del círculo con radio", radio, "es:", area))
```

```
## [1] "El área del círculo con radio 3 es: 28.2743338823081"
```

8.6 Definir una función para calcular el factorial de un número

```
factorial <- function(n) {
  if (n == 0) {
```

```
    return(1)
  } else {
    return(n * factorial(n - 1))
  }
}

# Ejemplo de uso
numero <- 5
resultado <- factorial(numero)
print(paste("El factorial de", numero, "es:", resultado))
```

```
## [1] "El factorial de 5 es: 120"
```

8.7 Definir una función para verificar si un número es primo

```
es_primo <- function(n) {
  if (n <= 1) {
    return(FALSE)
  } else if (n <= 3) {
    return(TRUE)
  } else if (n %% 2 == 0 | n %% 3 == 0) {
    return(FALSE)
  }
  i <- 5
  while (i * i <= n) {
    if (n %% i == 0 | n %% (i + 2) == 0) {
      return(FALSE)
    }
    i <- i + 6
  }
  return(TRUE)
}

# Ejemplo de uso
numero <- 11
if (es_primo(numero)) {
  print(paste(numero, "es un número primo."))
} else {
  print(paste(numero, "no es un número primo."))
}
```

```
## [1] "11 es un número primo."
```

Ejercicios

1. Escribe una función que tome un vector numérico como argumento de entrada y devuelva la media aritmética de los valores.
2. Escribe una función que tome dos vectores numéricos como argumentos de entrada y devuelva su producto punto (también conocido como producto escalar).
3. Escribe una función que tome un vector numérico como argumento de entrada y devuelva el valor mínimo y máximo en una lista.
4. Escribe una función que tome una matriz cuadrada como argumento de entrada y devuelva su determinante.
5. Escribe una función que tome una matriz de 2×2 como argumento de entrada y devuelva su transpuesta.
6. Escribe una función que tome una lista como argumento de entrada y devuelva la longitud de cada elemento en una lista.
7. Escribe una función que tome una cadena de texto como argumento de entrada y devuelva una lista con todas las palabras en la cadena.
8. Escribe una función que tome un vector numérico como argumento de entrada y devuelva un vector con los valores ordenados de menor a mayor.
9. Escribe una función que tome un vector de caracteres como argumento de entrada y devuelva un vector con los mismos elementos en orden inverso.
10. Escribe una función que tome una matriz como argumento de entrada y devuelva la diagonal principal en un vector.

Capítulo 9

Gráficos

Agrupar los datos, ya sea numéricamente o gráficamente, es un paso muy importante en el análisis de datos. Una de las ventajas de R es su gran capacidad para elaborar gráficos, sus aplicaciones van desde una primera consulta exploratoria de datos hasta la generación de imágenes complejas y de alta cálidad.

Existen tres principales sistemas para generar gráficos en R:

- **base R graphics**: Es el sistema de graficación base que está implementado en R desde sus inicios. Es bastante útil para realizar gráficas simples con el objetivo de un primer análisis exploratorio a los datos. Su principal función es `plot`, este comando permite crear un gráfico sencillo, adicionalmente se deben elegir y emplear funciones menores, como `lines` y `text`, para agregar información o mejorar el diseño del gráfico.

- **lattice**: Es un paquete de R. Una de sus ventajas es que la mayoría de sus gráficos son generados empleando una sola función, por lo tanto no hay necesidad de implementar funciones menores para modificar la apariencia del gráfico, ya que muchas de las específicaciones vienen por *default* en los comandos. Principalmente `lattice` está impelmentado para crear gráficos de alto nivel, como gráficas de líneas, gráficas de barras apiladas, gráficas de contorno y dividirlos por variables de agrupación.

- **ggplot2**: Es la librería de graficación que se empleará principalmente en este curso. En el siguiente apartado se presentan de manera más amplia sus aplicaciones. El paquete `ggplot2` es bastante popular por su capacidad para realizar gráficos de gran calidad de una manera relativamente sencilla basado en *La Grámatica del Gráfico*.

9.1 Visualización de datos con ggplot2

9.1.1 Introducción

La visualización de datos es una herramienta poderosa en el análisis de datos, permitiendo comunicar información compleja de manera intuitiva y efectiva. `ggplot2` es un paquete en R que facilita la creación de gráficos de calidad de publicación de manera sencilla. Este documento proporciona una introducción a `ggplot2` y ejercicios para practicar.

9.1.1.1 La Grámatica de los Gráficos

El paquete `ggplot2` está insipirado en el libro *Grammar of Graphics* por Leland Wilkinson, de ahí parte de su nombre "gg". Su principal enfoque es la creación de gráficos considerando sus distintos componentes y que el usuario sea capaz de manipular cada una de esas partes con base en necesidades específicas que permitan una óptima representación e interpretación de los datos.

Los componentes que considera `ggplot2` para la construcción de un gráfico son los siguientes:

- **Data**: Datos que se emplean para la creación del gráfico.
- **Mapping**: La manera de mapear las variables de los datos a propiedades visuales del gráfico como el color, el tamaño, la forma o la posición.
- **Geoms**: El tipo de gráfico empleado: puntos, barras, líneas…
- **Facets**: Permite dividir el gráfico en múltiples paneles según los niveles de una o más variables categóricas.
- **Statistics**: Son las técnicas estadísticas que son aplicadas a los datos previo a su visualización.
- **Coords**: El tipo de coordenadas empleadas para mapear los datos. Por ejemplo: cartesianas, polares, logarítmicas…
- **Scales**: Asigna los valores de los datos a los valores visuales. Por ejemplo: Que a valores más pequeños les corresponda un tono más claro de color.
- **Themes**: Controlan los aspectos visuales del gráfico, como los colores, las fuentes, los tamaños de los ejes, los márgenes…

NOTA Para conocer a profundidad e identificar de manera específica cada uno de estos componentes se puede consultar la siguiente Guía para la contrucción de un gráfico con ggplot2

9.1.2 Configuración inicial

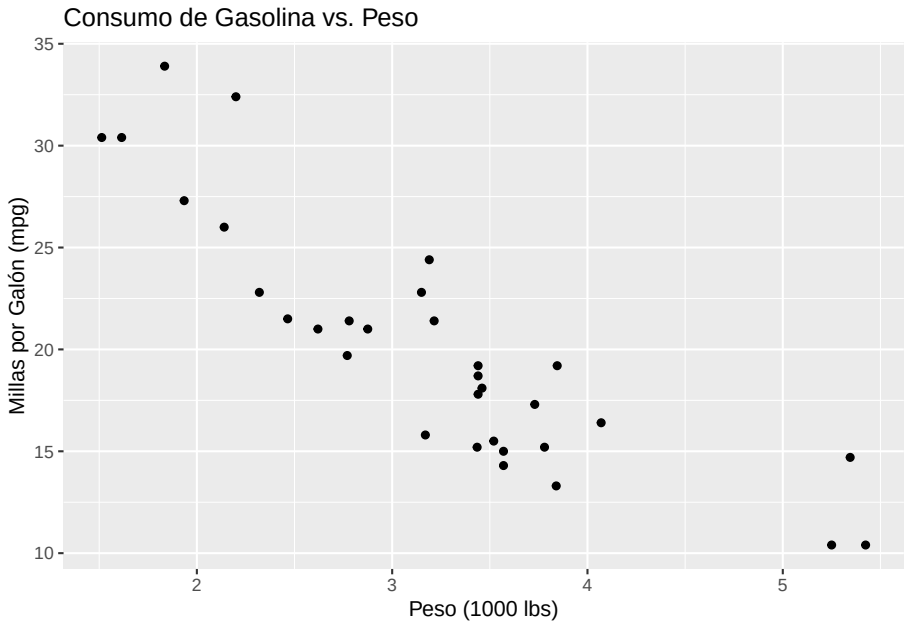
Primero, asegúrate de tener instalado `ggplot2`.

```
library(ggplot2)
```

9.1.3 Creación de un gráfico básico

Vamos a comenzar con un gráfico de dispersión simple utilizando el conjunto de datos mtcars.

```
ggplot(data = mtcars, aes(x = wt, y = mpg)) +
  geom_point() +
  labs(title = "Consumo de Gasolina vs. Peso",
       x = "Peso (1000 lbs)",
       y = "Millas por Galón (mpg)")
```



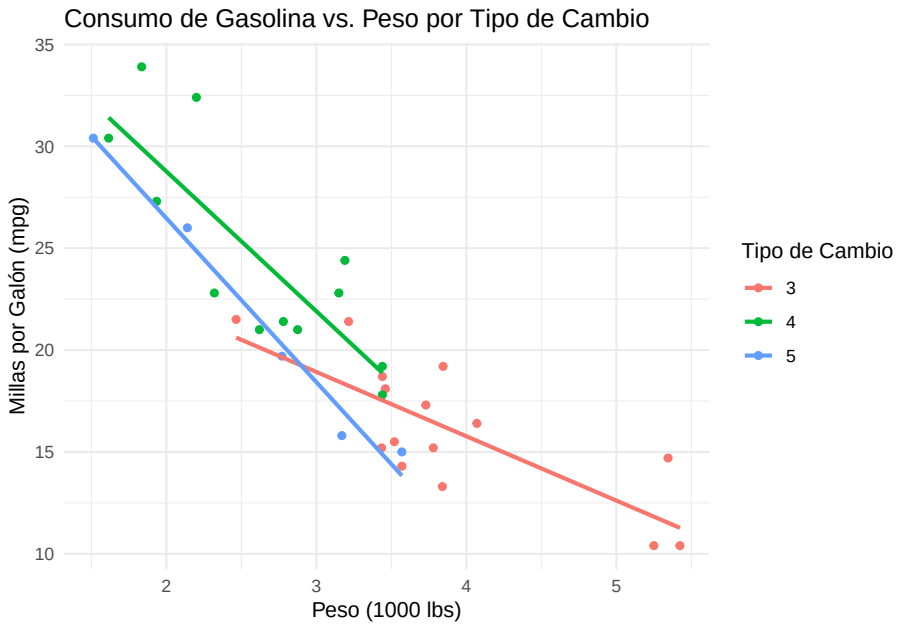
9.1.4 Personalización de gráficos

Ahora, personalizaremos el gráfico cambiando colores y añadiendo una línea de tendencia.

```
ggplot(data = mtcars, aes(x = wt, y = mpg, color = factor(gear))) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE) +
  labs(title = "Consumo de Gasolina vs. Peso por Tipo de Cambio",
       x = "Peso (1000 lbs)",
       y = "Millas por Galón (mpg)",
       color = "Tipo de Cambio") +
```

```
  theme_minimal()
```

```
## `geom_smooth()` using formula = 'y ~ x'
```

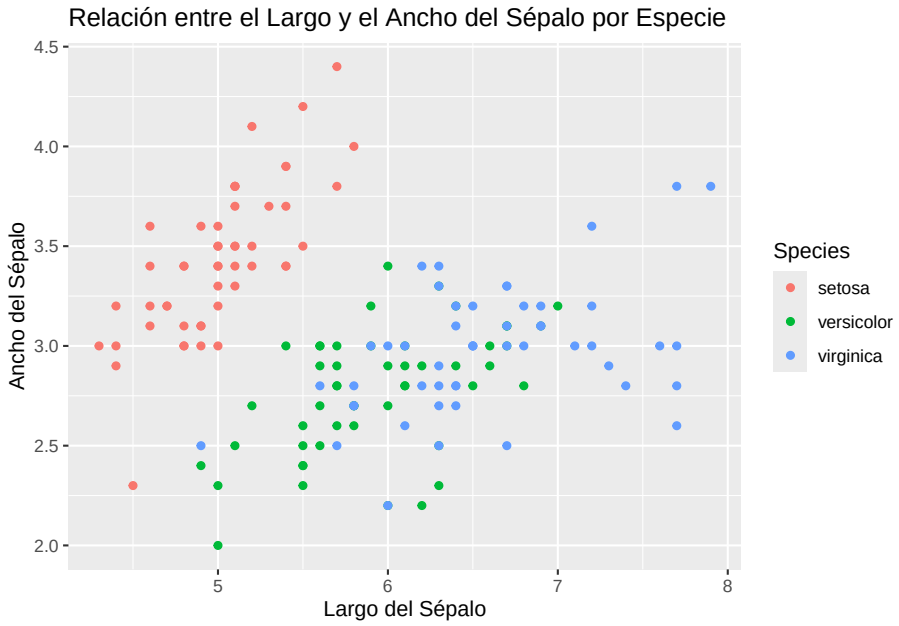


Ejercicios sugeridos

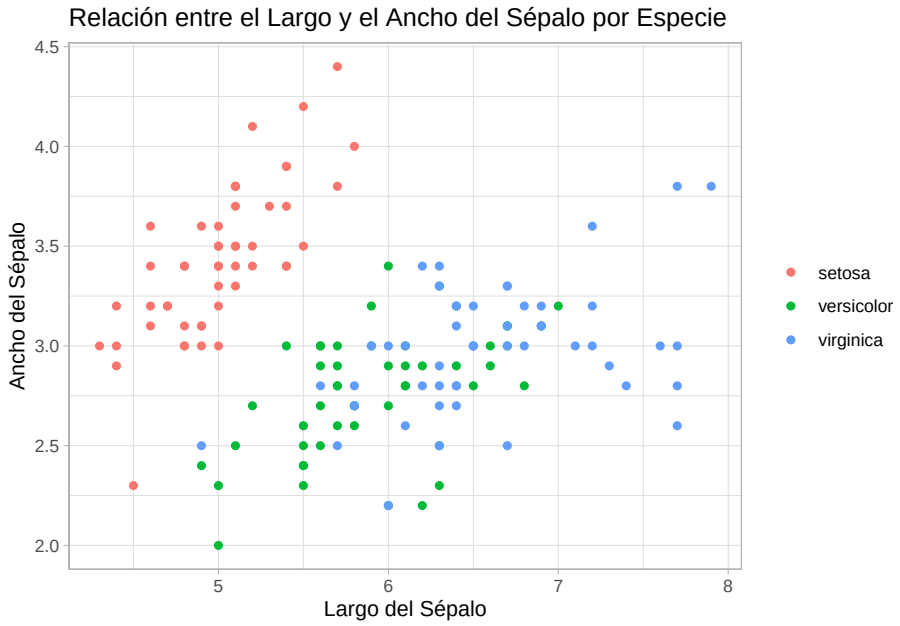
1. Explora otro conjunto de datos: Utiliza el conjunto de datos iris para crear un gráfico de dispersión que muestre la relación entre Sepal.Length y Sepal.Width. Colorea los puntos según la especie.
2. Personaliza tu gráfico: Añade títulos personalizados a los ejes y al gráfico. Experimenta con diferentes temas, como theme_bw() o theme_light().
3. Exploración de geométricas: Utiliza geom_histogram() para crear un histograma del Sepal.Length en el conjunto de datos iris. Ajusta los parámetros binwidth y fill.
4. Facetas para múltiples gráficos: Utiliza facet_wrap(~ species) para crear gráficos separados para cada especie en el conjunto de datos iris, mostrando la relación entre Sepal.Length y Sepal.Width.

9.1.5 Soluciones a los ejercicios sugeridos

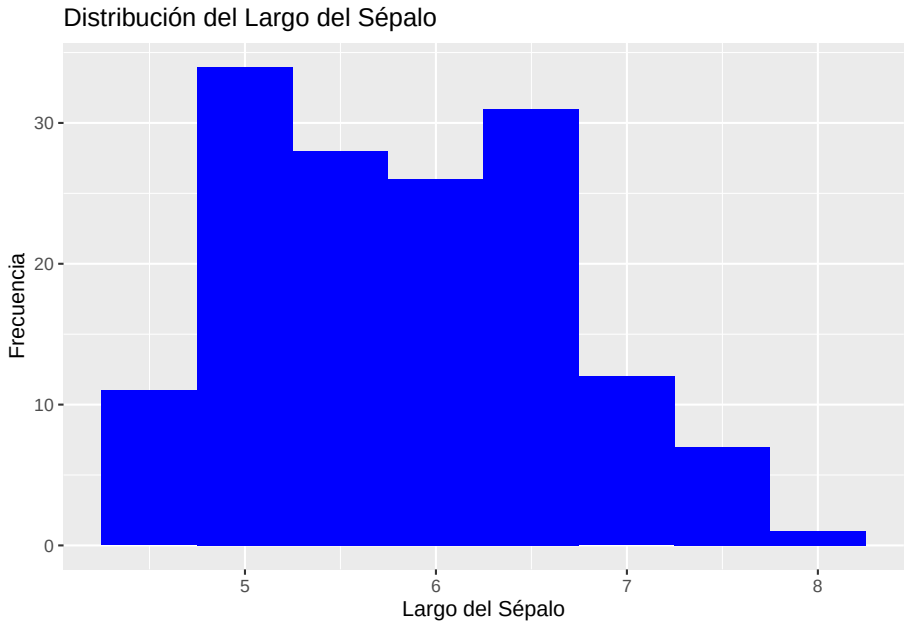
```
ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width, color = Species)) +
  geom_point() +
  labs(title = "Relación entre el Largo y el Ancho del Sépalo por Especie",
       x = "Largo del Sépalo",
       y = "Ancho del Sépalo")
```



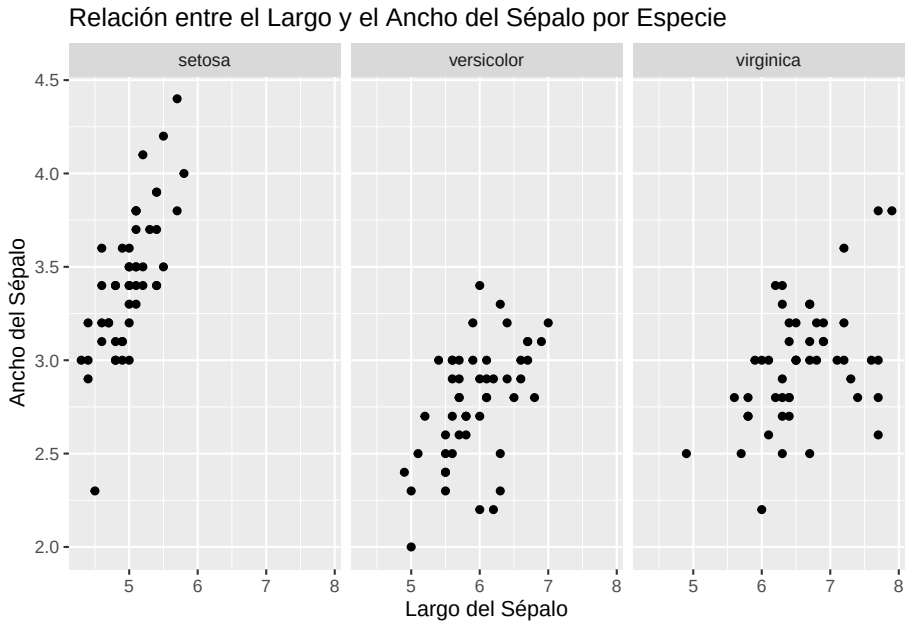

```
ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width, color = Species)) +
  geom_point() +
  labs(title = "Relación entre el Largo y el Ancho del Sépalo por Especie",
       x = "Largo del Sépalo",
       y = "Ancho del Sépalo") +
  theme_light() +
  theme(legend.title = element_blank())
```




```
ggplot(data = iris, aes(x = Sepal.Length)) +
  geom_histogram(binwidth = 0.5, fill = "blue") +
  labs(title = "Distribución del Largo del Sépalo",
       x = "Largo del Sépalo",
       y = "Frecuencia")
```




```
ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width)) +
  geom_point() +
  facet_wrap(~ Species) +
  labs(title = "Relación entre el Largo y el Ancho del Sépalo por Especie",
       x = "Largo del Sépalo",
       y = "Ancho del Sépalo")
```



9.1.6 Ejercicios resueltos

En esta sección usaremos la librería `palmerpenguins`

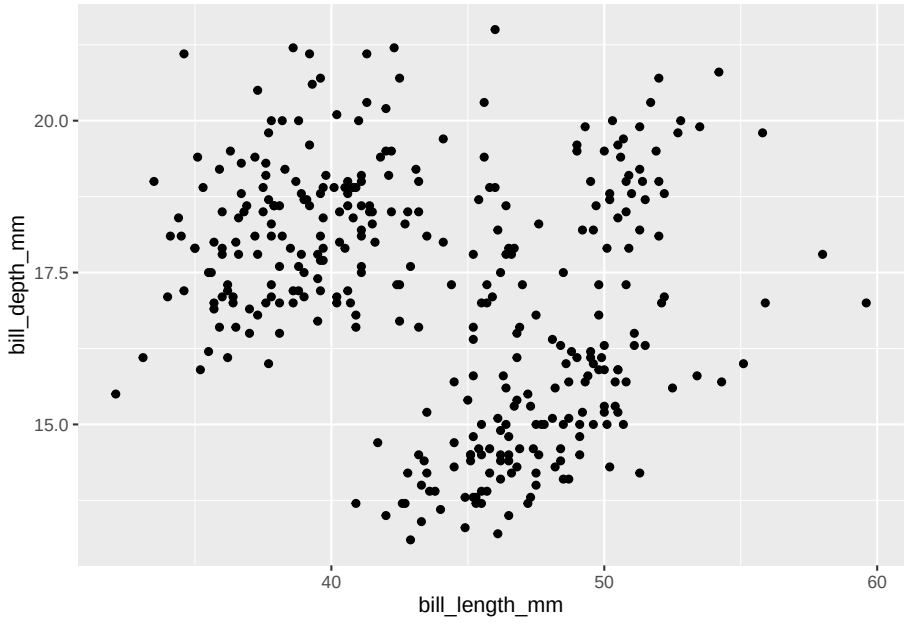
```
library(palmerpenguins)
library(ggplot2)
```

9.1.6.1 Ejercicio 1: Gráfico de dispersión básico

Crea un gráfico de dispersión para visualizar la relación entre la longitud del culmen y la profundidad del culmen de los pingüinos.

```
ggplot(penguins, aes(x = bill_length_mm, y = bill_depth_mm)) +
  geom_point()
```

```
## Warning: Removed 2 rows containing missing values or values outside the scale
## range (`geom_point()`).
```

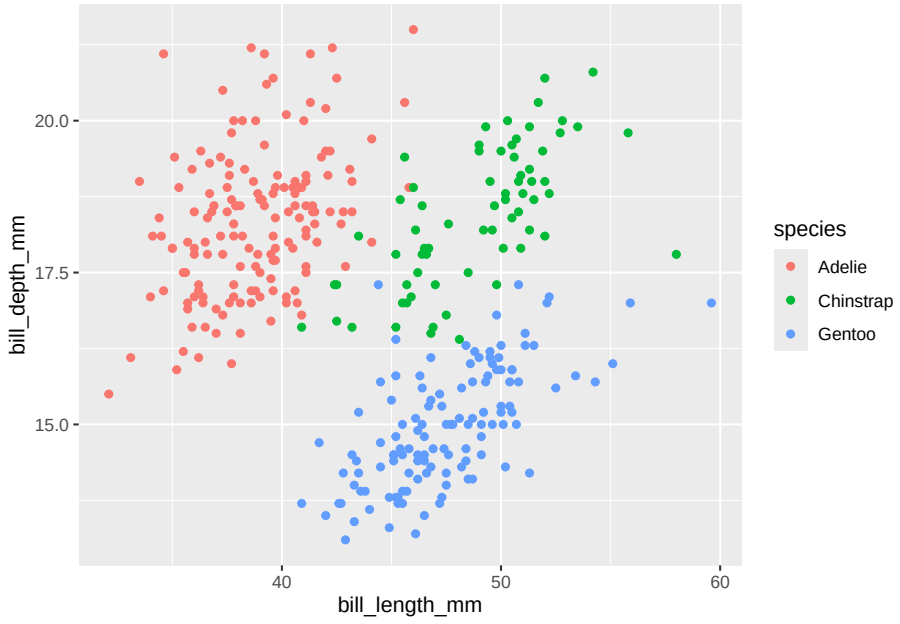


9.1.6.2 Ejercicio 2: Diferenciación por especie

Modifica el gráfico de dispersión anterior para diferenciar los puntos por especie de pingüino.

```
ggplot(penguins, aes(x = bill_length_mm, y = bill_depth_mm, color = species)) +
  geom_point()
```

```
## Warning: Removed 2 rows containing missing values or values outside the scale
## range (`geom_point()`).
```

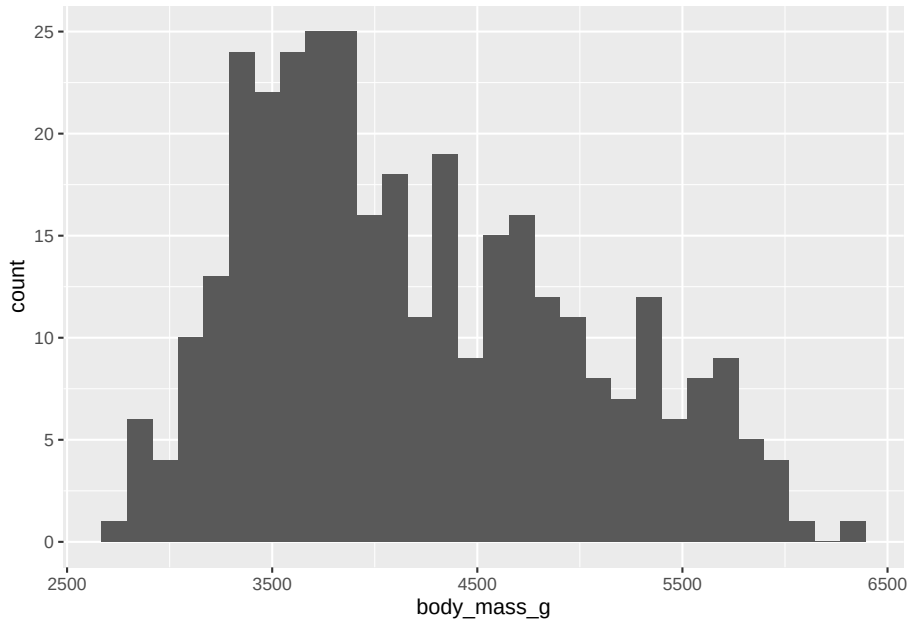


9.1.6.3 Ejercicio 3: Histograma de masa corporal

Crea un histograma para explorar la distribución de la masa corporal de los pingüinos.

```
ggplot(penguins, aes(x = body_mass_g)) +
  geom_histogram(bins = 30)
```

```
## Warning: Removed 2 rows containing non-finite outside the scale range
## (`stat_bin()`).
```

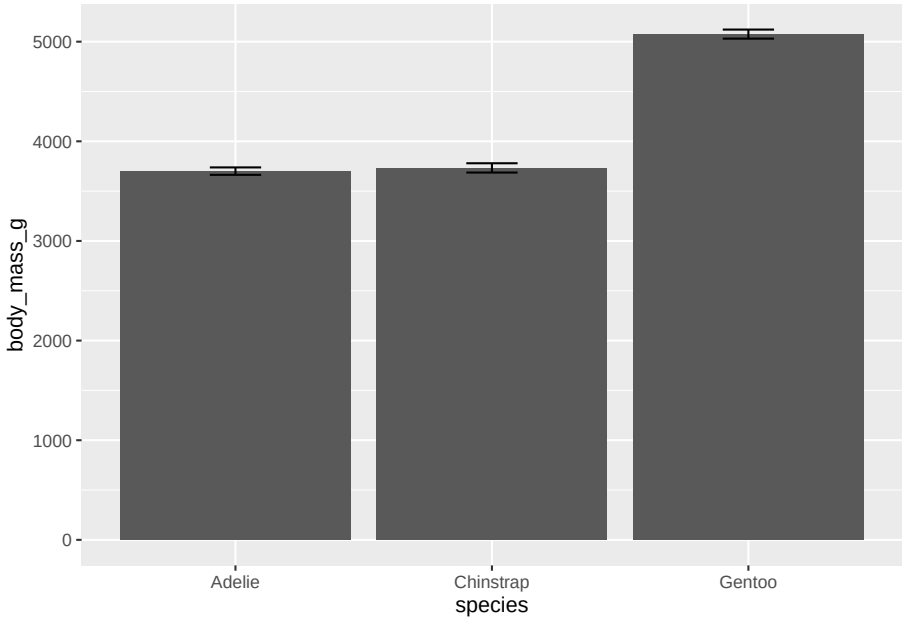


9.1.6.4 Ejercicio 4: Barras de error por especie

Genera un gráfico de barras que muestre la masa corporal media de los pingüinos por especie, incluyendo barras de error.

```
ggplot(penguins, aes(x = species, y = body_mass_g)) +
  geom_bar(stat = "summary", fun = "mean") +
  geom_errorbar(stat = "summary", fun.data = mean_se, width = 0.2)
```

```
## Warning: Removed 2 rows containing non-finite outside the scale range
## (`stat_summary()`).
## Removed 2 rows containing non-finite outside the scale range
## (`stat_summary()`).
```

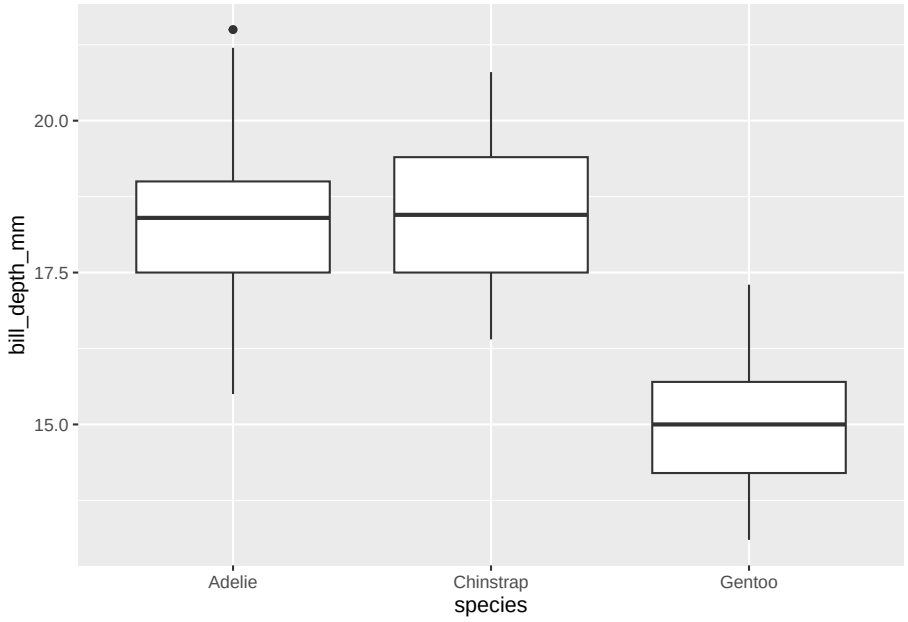


9.1.6.5 Ejercicio 5: Boxplot de profundidad del culmen

Crea un boxplot para comparar la profundidad del culmen entre las diferentes especies de pingüinos.

```
ggplot(penguins, aes(x = species, y = bill_depth_mm)) +
  geom_boxplot()
```

```
## Warning: Removed 2 rows containing non-finite outside the scale range
## (`stat_boxplot()`).
```

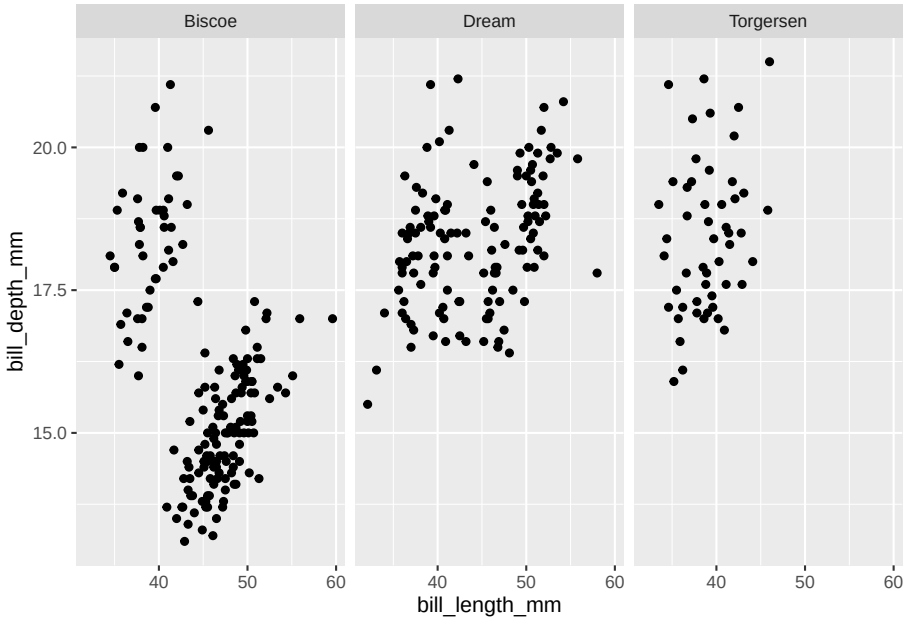


9.1.6.6 Ejercicio 6: Facetado por islas

Repite el gráfico de dispersión de longitud vs. profundidad del culmen, pero esta vez facetado por la isla de origen.

```
ggplot(penguins, aes(x = bill_length_mm, y = bill_depth_mm)) +
  geom_point() +
  facet_wrap(~island)
```

```
## Warning: Removed 2 rows containing missing values or values outside the scale
## range (`geom_point()`).
```

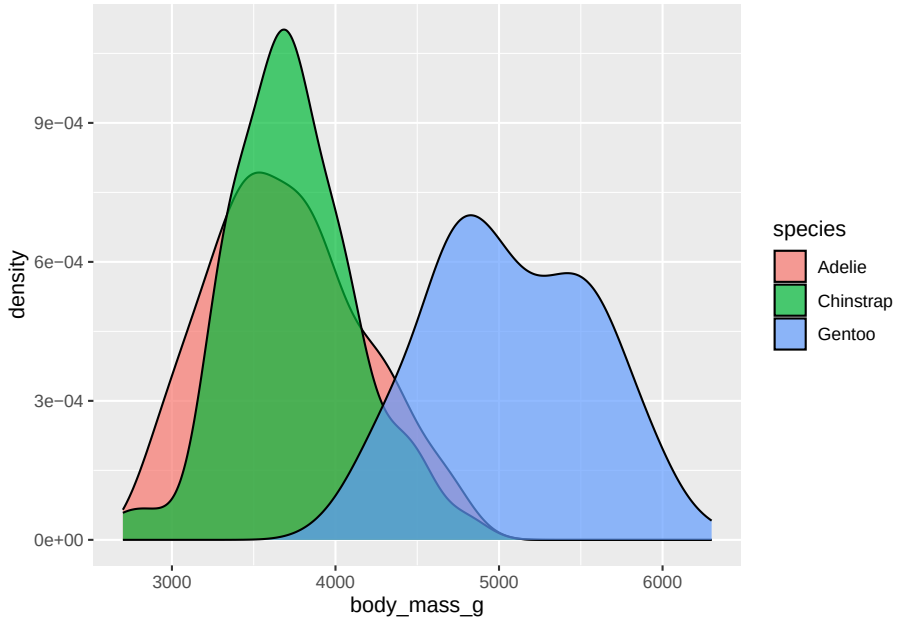


9.1.6.7 Ejercicio 7: Densidad de masa corporal

Muestra la densidad de la distribución de la masa corporal de los pingüinos utilizando un gráfico de densidad.

```
ggplot(penguins, aes(x = body_mass_g, fill = species)) +
  geom_density(alpha = 0.7)
```

```
## Warning: Removed 2 rows containing non-finite outside the scale range
## (`stat_density()`).
```

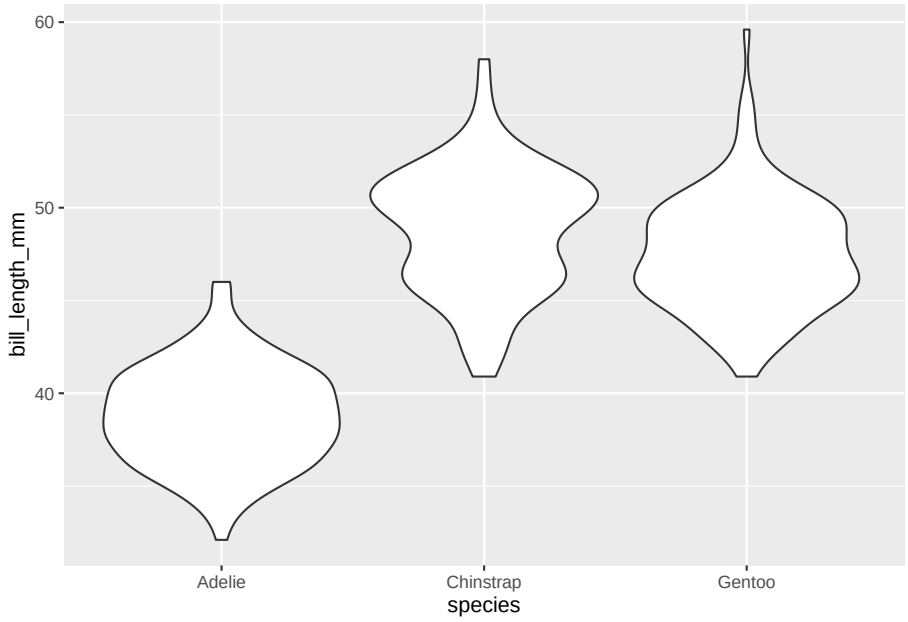


9.1.6.8 Ejercicio 8: Gráfico de violín de la longitud del culmen

Genera un gráfico de violín para visualizar la distribución de la longitud del culmen por especie.

```
ggplot(penguins, aes(x = species, y = bill_length_mm)) +
  geom_violin()
```

```
## Warning: Removed 2 rows containing non-finite outside the scale range
## (`stat_ydensity()`).
```



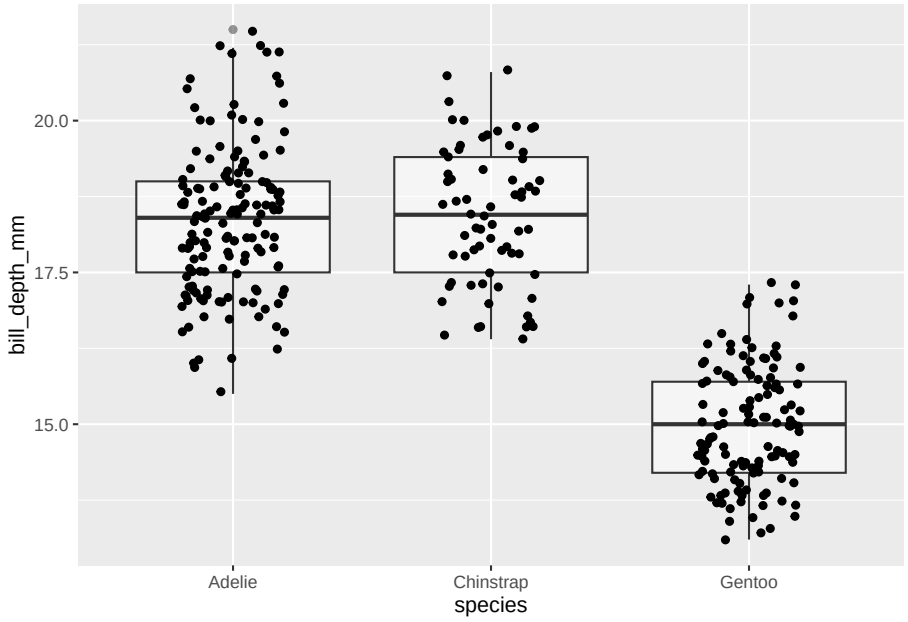
9.1.6.9 Ejercicio 9: Puntos superpuestos en boxplot

Crea un boxplot de la profundidad del culmen por especie y superpón los puntos de datos individuales.

```
ggplot(penguins, aes(x = species, y = bill_depth_mm)) +
  geom_boxplot(alpha = 0.5) +
  geom_jitter(width = 0.2)
```

```
## Warning: Removed 2 rows containing non-finite outside the scale range
## (`stat_boxplot()`).
```

```
## Warning: Removed 2 rows containing missing values or values outside the scale
## range (`geom_point()`).
```

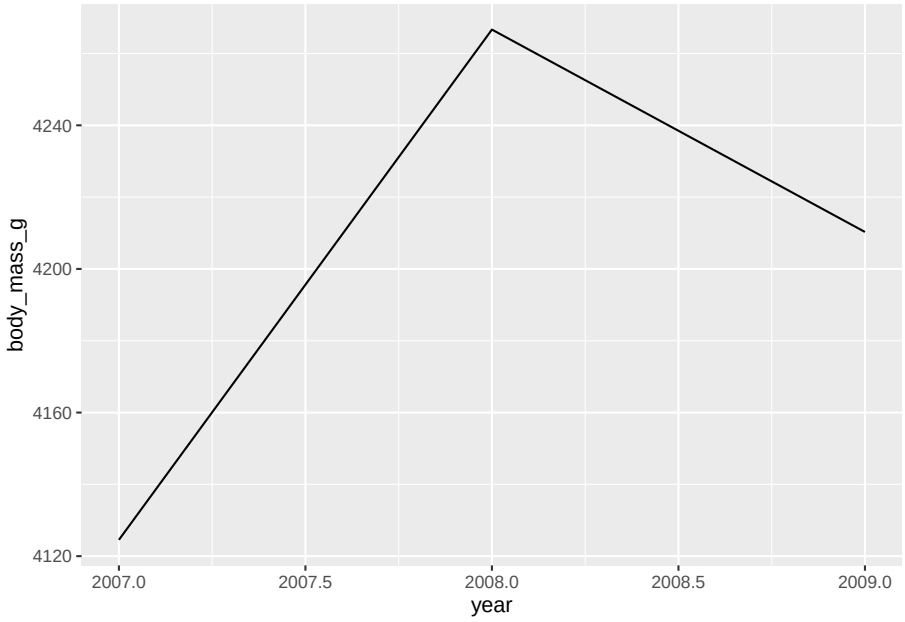


9.1.6.10 Ejercicio 10: Gráfico de líneas de masa corporal promedio a lo largo del tiempo

Asumiendo que los datos estén ordenados temporalmente, muestra cómo cambia la masa corporal promedio a lo largo del tiempo.

```
# Asumiendo que 'year' representa el tiempo en tus datos
ggplot(penguins, aes(x = year, y = body_mass_g, group = 1)) +
  geom_line(stat = "summary", fun = "mean")
```

```
## Warning: Removed 2 rows containing non-finite outside the scale range
## (`stat_summary()`).
```

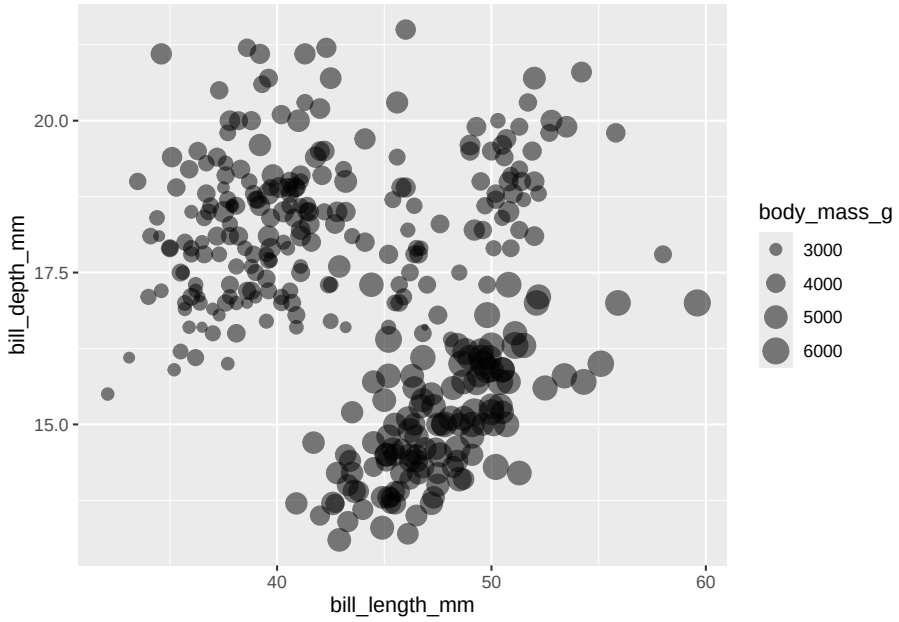


9.1.6.11 Ejercicio 11: Gráfico de dispersión con tamaño de punto

Modifica el gráfico de dispersión de longitud vs. profundidad del culmen para que el tamaño de los puntos refleje la masa corporal de los pingüinos.

```
ggplot(penguins, aes(x = bill_length_mm, y = bill_depth_mm, size = body_mass_g)) +
  geom_point(alpha = 0.5)
```

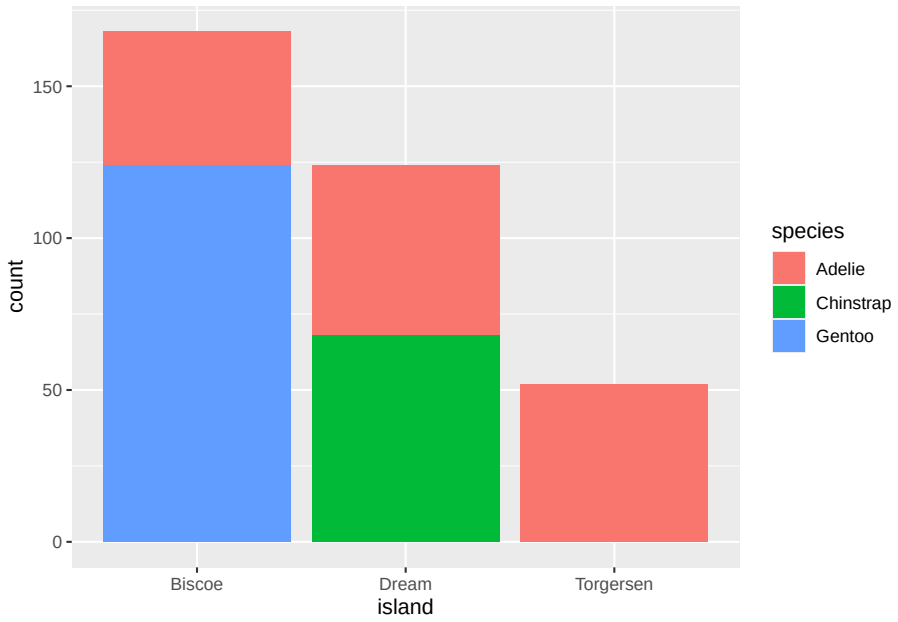
```
## Warning: Removed 2 rows containing missing values or values outside the scale
## range (`geom_point()`).
```



9.1.6.12 Ejercicio 12: Barras apiladas de especies por isla

Crea un gráfico de barras apiladas que muestre la cantidad de pingüinos de cada especie presentes en cada isla.

```
ggplot(penguins, aes(x = island, fill = species)) +
  geom_bar(position = "stack")
```



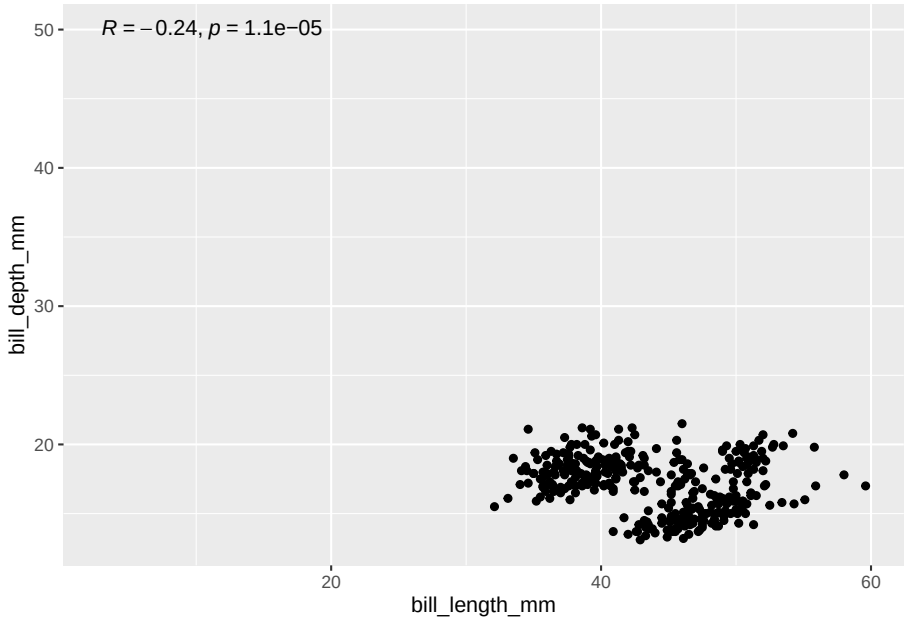
9.1.6.13 Ejercicio 13: Gráfico de correlación con texto

Genera un gráfico de dispersión entre la longitud y la profundidad del culmen e incluye un texto que muestre el coeficiente de correlación en el gráfico.

```
library(ggpubr)
ggplot(penguins, aes(x = bill_length_mm, y = bill_depth_mm)) +
  geom_point() +
  stat_cor(method = "pearson", label.x = 3, label.y = 50)
```

```
## Warning: Removed 2 rows containing non-finite outside the scale range
## (`stat_cor()`).
```

```
## Warning: Removed 2 rows containing missing values or values outside the scale
## range (`geom_point()`).
```



9.1.6.14 Ejercicio 14: Mapa de calor de la longitud y profundidad del culmen

Crea un mapa de calor que muestre la distribución conjunta de la longitud y la profundidad del culmen de los pingüinos.

```
ggplot(penguins, aes(x = bill_length_mm, y = bill_depth_mm)) +
  geom_bin2d()
```

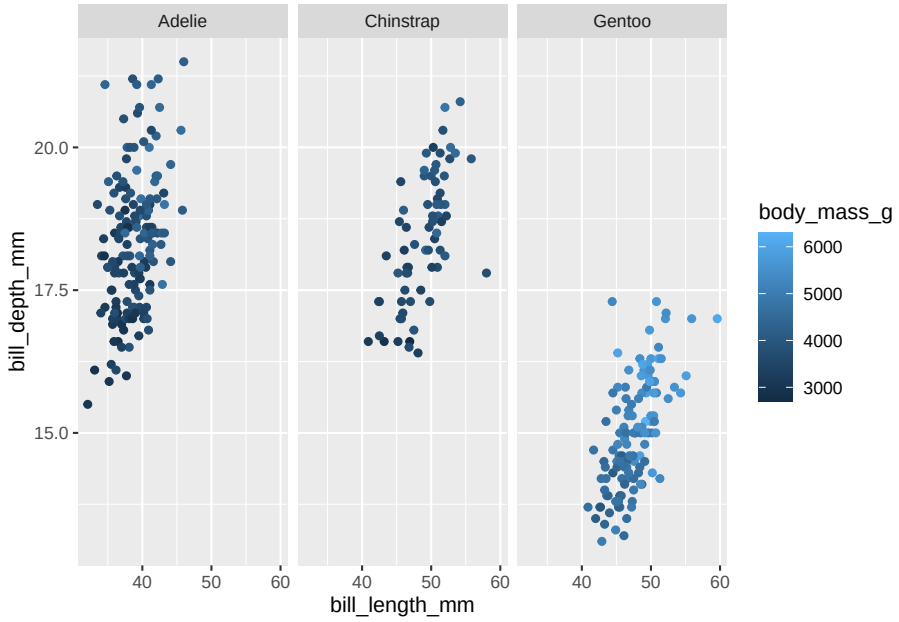
```
## Warning: Removed 2 rows containing non-finite outside the scale range
## (`stat_bin2d()`).
```

9.1.6.15 Ejercicio 15: Gráfico de interacción entre tres variables

Explora la relación entre la longitud del culmen, la profundidad del culmen y la masa corporal, diferenciando por especie.

```
ggplot(penguins, aes(x = bill_length_mm, y = bill_depth_mm, color = body_mass_g)) +
  geom_point() +
  facet_wrap(~species)
```

```
## Warning: Removed 2 rows containing missing values or values outside the scale
## range (`geom_point()`).
```



9.1.6.16 Ejercicios propuestos

1. Gráfico de dispersión con facetas por sexo Crea un gráfico de dispersión para examinar la relación entre la longitud del culmen y la profundidad del culmen, separando los datos por sexo de los pingüinos utilizando facetas.

2. Comparación de masa corporal entre islas Utiliza un gráfico de barras para comparar la masa corporal media de los pingüinos en las diferentes islas.

3. Gráfico de densidad por sexo Genera gráficos de densidad para la longitud del culmen, diferenciados por sexo de los pingüinos.

4. Gráfico de líneas de tendencia para la profundidad del culmen Crea un gráfico que muestre la tendencia de la profundidad del culmen a lo largo del tiempo para cada especie de pingüino.

5. Mapa de calor de la correlación entre variables numéricas Utiliza funciones de `ggplot2` para crear un mapa de calor que muestre la correlación entre las variables numéricas de los datos de pingüinos.

6. Gráfico de barras de conteo por especie Crea un gráfico de barras que muestre el número de observaciones (conteo) para cada especie de pingüino.

7. Análisis de outliers en la masa corporal Utiliza un boxplot para identificar outliers en la masa corporal de los pingüinos y diferencia por especie.

8. Gráfico de dispersión con modelado lineal Crea un gráfico de dispersión

de la longitud vs. profundidad del culmen e incluye una línea de tendencia lineal.

9. Barras apiladas de conteo por isla y especie Genera un gráfico de barras apiladas que muestre el número de pingüinos de cada especie en cada isla.

10. Gráfico de violín con puntos individuales Crea gráficos de violín para la masa corporal de los pingüinos por especie e incluye los puntos individuales de los datos.

Capítulo 10

RProjects

Los proyectos en R (R Projects) son una manera eficiente de organizar y gestionar todo el trabajo relacionado con tus análisis en R. Facilitan la colaboración, el control de versiones y la reproducibilidad de tu investigación o análisis de datos.

10.1 Introducción a R Projects en RStudio

RStudio permite crear proyectos, que son básicamente carpetas que contienen todos los archivos relacionados con un análisis específico. Al trabajar dentro de un proyecto, RStudio automáticamente establece el directorio del proyecto como el directorio de trabajo, simplificando la gestión de archivos y el flujo de trabajo.

Crear un nuevo proyecto:

1. En RStudio, ve a `File > New Project`.
2. Elige crear un proyecto en un nuevo directorio o en un directorio existente.
3. Asigna un nombre al proyecto y selecciona la ubicación para el directorio del proyecto.
4. Haz clic en `Create Project`.

10.2 Gestión de proyectos: organización y buenas prácticas

Una buena gestión de proyectos en R implica una estructura de carpetas organizada, nomenclatura consistente de archivos y un entendimiento claro del flujo de trabajo del proyecto.

Estructura de carpetas recomendada:

- `/01_RawData`: Guarda tus bases de datos originales (preferentemente en formato de solo lectura como .csv o .xlsx)..
- `/02_Scripts`: Guardar tus scripts con extensiones .R o .Rmd
- `/03_ProceesedData`: Almacena archivos de salida procesados, como datos limpios o tablas resumen
- `/04_output`: .Guarda las figuras generadas por tus scripts.

Buenas prácticas:

- Mantén un script de R (por ejemplo, `main.R`) que sirva como punto de entrada para entender y ejecutar el proyecto.
- Documenta tus scripts detalladamente para que otros (o tú en el futuro) puedan entender el propósito y funcionamiento de tu código.
- Utiliza nombres de archivos y variables claros y descriptivos. Por ejemplo: "SeqCluster: Clustering de Secuencias Biológicas" o "GeneAnalyzer: Análisis de Datos Genéticos".

10.3 Colaboración y control de versiones con Git y GitHub

El control de versiones es crucial para la colaboración en proyectos de programación. Git, integrado con GitHub, permite a múltiples personas trabajar en el mismo proyecto sin conflictos.

Configurar Git en RStudio:

1. Instala Git en tu computadora y configúralo con tus credenciales de GitHub.
2. En RStudio, ve a `Tools > Global Options > Git/SVN` para configurar tu cuenta de Git.
3. Crea un nuevo repositorio en GitHub y clónalo en tu computadora a través de la URL del repositorio.

Uso básico de Git con RStudio:

- **Commit**: Guarda los cambios realizados en tus archivos al repositorio local. Describe brevemente los cambios realizados.
- **Push**: Envía tus commits locales a GitHub para actualizar el repositorio remoto.
- **Pull**: Actualiza tu repositorio local con los cambios realizados por otros colaboradores en el repositorio remoto.

Colaborar en proyectos:

- Utiliza *branches* para trabajar en nuevas características o experimentos sin afectar la rama principal (*main* o *master*).
- Abre *pull requests* para discutir cambios antes de integrarlos a la rama principal.

- Realiza revisiones de código y utiliza *issues* para rastrear tareas y discutir problemas.

Capítulo 11

Referencias