INTRODUCCIÓN A LA CIENCIA DE DATOS (O BIOESTADÍSTICA) USANDO R/RSTUDIO

Dra. Stephanie Hereira-Pacheco

Dr. Mauricio Hernández

Dr. Daniel Rivera

MsC. Braulio Pérez Alva

MsC. Yefrin Valladares

# Breve semblanza de los autores

La Dra. Stephanie Hereira-Pacheco (PhD) es Ingeniera Agroindustrial de profesión, posee un máster en y un doctorado en ciencias en la especialidad en Biotecnología. Es investigadora del Centro en Ciencias Biológicas de la Universidad Autónoma de Tlaxcala (México). Dentro de sus líneas de investigación destacan los temas relacionados a ecología microbiana, metagenómica, bioinformática y metabolitos secundarios. Ha publicado una gran cantidad de artículos científicos en diferentes revistas de alto impacto entre ellas PeerJ, Scientific Reports, Molecular Ecology. Fungal Ecoloy, Microbology Spectrum, etc.

El Dr. Mauricio Hernández (PhD) es biólogo, posee un máster en geología y un doctorado en Ciencias Biológicas. El Dr. Hernández es profesor e investigador en el Departamento de Biología Celular y Genética, Escuela de Biología, Facultad de Ciencias (Universidad Nacional Autónoma de Honduras). Sus líneas de investigación cubren áreas relacionadas a la ecología molecular, microbiología, ambientes contaminados por compuestos orgánicos persistentes, entre otros. Sus trabajos de investigación han sido publicados en revistas internaciones como ser Molecular Ecology, Animal Microbiome, Integrative Zoology, Trends in Ecology and Evolution, Environmental Pollution, Journal of Hazardous Materials, etc.

## Protección de propiedad intelectual

El contenido de este manual se encuentra bajo la protección de las normativas vigentes en materia de Propiedad Intelectual en los marcos legales de México y Honduras. Por lo tanto, su reproducción o distribución no autorizada con fines comerciales o lucrativos está prohibida.

## Índice

Prefacio

Introducción a R y Rstudio

* Requisitos para la instalación de R
* Instalación de R y Rstudio
* Tipos de datos
* Dataframes
* Directorio de trabajo
* Funciones básicas en R

Manejo de dataframes y bases de datos

Construcción de gráficos

Valor de significancia p en estadística (p-value)

* ¿Qué es el valor p?
* Interpretación general del valor p
* Marco de prueba de hipótesis: hipótesis nula y alternativa
* Regla de decisión: umbrales comunes para el valor p
* Errores en la prueba de hipótesis

Introducción a la estadística

* Diferencia entre estadística y bioestadística
* Estadística descriptiva: medidas de tendencia central y medidas de dispersión
* Estadística inferencial: univariada y multivariada
* Estadística inferencial paramétrica: supuestos principales de las pruebas paramétricas
* Estadística inferencial no paramétrica: supuestos principales de las pruebas no paramétricas
* Test estadísticos para determinar distribución y homogeneidad de varianzas en un conjunto de datos: prueba de Shapiro-Wilk, pruebas de ajuste o distribución (Kolmogorov-Smirnov, Anderson-Darling) y prueba de Levins (varianzas).

Tipos de pruebas paramétricas

* Prueba t (t de Student) para un conjunto de datos
* Prueba t (t de Student) para dos conjuntos de datos
* Prueba t (t de Student) para datos pareados
* Prueba t de Welch
* Prueba F de Fisher
* Análisis de varianza “ANOVA”: una vía (one-way) y dos vías (two-way)
* Análisis de covarianza (ANCOVA)
* Correlación de Pearson
* Regresión lineal simple y múltiple

Tipos de pruebas no paramétricas

* Prueba de Mann-Whitney U (o Wilcoxon rank-sum test) – muestras independientes
* Prueba de Wilcoxon para muestras relacionadas
* Prueba de Kruskal-Wallis – muestras independientes
* Prueba de Friedman – muestras relacionadas
* Prueba de Chi-Cuadrado – datos categóricos (frecuencias)
* Prueba de McNemar – datos categóricos (frecuencias)
* Correlación de rango de Spearman
* Correlación de Kendall

Análisis de riqueza, diversidad alfa y beta

* Números de Hill (diversidad al órden q0, q1 y q2)

Análisis multivariados

* Análisis de componentes principales (PCA)
* Análisis de coordenadas principales (PCoA)
* Escalamiento multidimensional no métrico (NMDS)
* Análisis de varianza multivariado permutacional (PERMANOVA)
* PERMDIS
* ANOSIM
* RDA
* CCA

Modelización estadística

* Modelo lineal
* Modelo lineal general (GLM)
* GLM con distribución Poisson
* GLM con distribución quasi-Poisson
* GLM con distribución de Bernoulli (binomail)
* GLMM: modelos lineales generales mixtos

## Prefacio

Actualmente, las ciencias biológicas son consideradas una de las áreas más activas en la investigación científica. Dentro de este campo, los investigadores recurren cada vez más a las herramientas computacionales para analizar e interpretar datos biológicos complejos. Entre las diversas herramientas de programación disponibles, R ha sido considerado como uno de los lenguajes más robustos, versátiles y ampliamente utilizados para los análisis de datos. Sin embargo, consideramos que existe una brecha importante entre la recolección, análisis e interpretación de datos, particularmente para los estudiantes universitarios. Aunado a esta falta de conocimiento, reconocemos que los estudiantes egresados deben poseer las habilidades básicas para realizar análisis estadísticos y brindar explicaciones sencillas a fenómenos naturales biológicos. Este manual tiene como objetivo proporcionar una introducción al uso del software R, orientado principalmente a estudiantes de pregrado y posgrado, así como investigadores de las ciencias biológicas. Debido a la gran cantidad de paquetes desarrollados por la comunidad científica, R ofrece una amplia gama de herramientas relacionadas con la manipulación de datos, modelado estadístico, visualización de gráficos, entre otros.

A lo largo de texto, trataremos de cubrir aspectos básicos de R, como la importación y exportación de datos, pruebas para evaluar distribución y varianza de los datos, aplicación de pruebas paramétricas y no paramétricas, gráficos de alta calidad, entre otros. El contenido ha sido cuidadosamente estructurado para brindar una guía detallada con ejemplos prácticos en el contexto de las ciencias biológicas. Por lo tanto, los estudiantes serán capaces de manejar conjuntos de datos, visualizar distribuciones y tendencias, realizar pruebas de hipótesis, aplicar modelos estadísticos y otros aspectos fundamentales para sus proyectos de investigación.

# 1 Introducción a R y Rstudio

**Instalación de Rsudio y R**

* R versión de 4.0 en adelante.
* Rstudio versiones 2024 en adelante.

En caso de no tener instalado R y Rstudio que son los prerrequisitos para realizar cada uno de los ejercicios, entonces debe visitar los siguientes links para descargar e instalarlos. Es recomendable instalar la última versión para que disponga de los paquetes y funciones más actualizados y, le permitan a su vez, un mejor uso del software.

* **R**
  + Para [MacOS](https://cran.r-project.org/bin/macosx/)
  + Para [Ubuntu](https://cran.r-project.org/bin/linux/ubuntu/)
  + Para [Windows](https://cran.r-project.org/bin/windows/base/)
* **Rstudio**
  + En [MacOS](https://download1.rstudio.org/electron/macos/RStudio-2024.12.1-563.dmg)
  + En [Ubuntu](https://download1.rstudio.org/electron/jammy/amd64/rstudio-2024.12.1-563-amd64.deb)
  + En [Windows](https://download1.rstudio.org/electron/windows/RStudio-2024.12.1-563.exe)

## 1.1 Comenzando con R y RStudio

### 1.1.1 Breve descripción de R

R es un lenguaje de programación como C, Java o Python, pero enfocado principalmente a la estadística. Cuando instalamos R en nuestra computadora, en realidad estamos instalando un entorno de programación, pero es necesario escribir correctamente las instrucciones para que puedan ser interpretarlas y ejecutarlas adecuadamente. Como en otros lenguajes de programación, en R podemos guardar nuestros proyectos de trabajo mediante un **script**, que puede ser utilizado eventualmente y compartido con otros usuarios de R.

### 1.1.2 Un poco de historia…[[1]](#footnote-31)

R tiene su origen en el lenguaje de programación S, desarrollado en los Laboratorios Bell de Estados Unidos. Posteriormente, en 1993, Ross Ihaka y Robert Gentleman, de la Universidad de Auckland en Nueva Zelanda, crearon una versión de S de código abierto y gratuita, lo que permitió su acceso y uso generalizado. No obstante, fue en el año 2000 cuando se lanzó una versión estable de R. Actualmente, el desarrollo y mantenimiento del software están a cargo del R Development Core Team, un grupo internacional de expertos en ciencias computacionales y estadísticas, provenientes de diversas instituciones académicas.

### 1.1.3 ¿Por qué usar R?[[2]](#footnote-34)

1. R es gratuito y de código abierto3.
2. Se ejecuta en todas las plataformas principales: Windows, Mac Os, UNIX/Linux.
3. Los *scripts* y los objetos de datos se pueden compartir entre plataformas y usuarios.
4. Existe una comunidad creciente y activa de usuarios en R y, como resultado, hay numerosos recursos para aprender4 5.
5. Es fácil para otras personas contribuir con complementos (*add-ons* en inglés) o paquetes que les permiten a los desarrolladores compartir implementaciones de software relacionados a nuevas metodologías de ciencia de datos. Esto les aporta a los usuarios de R acceso directo a los métodos y herramientas más recientes que se desarrollan para una amplia variedad de disciplinas, incluyendo la ecología, la biología molecular, las ciencias sociales y la geografía, entre otros campos.
6. Es fácil para otras personas contribuir con complementos (*add-ons* en inglés) o paquetes que les permiten a los desarrolladores compartir implementaciones de software relacionados a nuevas metodologías. Esto aporta acceso directo a los métodos y herramientas más recientes para una amplia variedad de disciplinas, incluyendo la ecología, la biología molecular, las ciencias sociales, la geografía, entre otros campos.

### 1.1.4 RStudio

RStudio será nuestra plataforma de uso directo, ya que nos provee un editor visual e interactivo para crear y editar *scripts*. Además, de otras herramientas útiles que iremos evaluando a lo largo de los diferentes temas.

### 1.1.5 Interfaz de Rstudio

Rstudio posee 4 paneles principales (Figura 1.1):

1. En el panel superior izquierdo aparece el editor de códigos, que comprende la sección donde se crean los *scripts* que deben ser ejecutados.
2. En el panel inferior izquierdo se encuentra la consola. Aquí es donde R evalúa y corre los códigos. Al comienzo de la consola encontrará el carácter >. Este es un mensaje indicando que R está listo para recibir un código nuevo.
3. A la derecha, el panel superior incluye pestañas como *Environment* e *History*. La pestaña *Environment* (ambiente) muestra el nombre de todos los objetos (como vectores, matrices, bases de datos, etc.) creados en la consola. Por otra parte, la pestaña *History* le muestra un historial de todo el código que ha desarrollado previamente en la consola.
4. En el panel inferior derecho se muestran cinco pestañas: *File, Plots, Packages, Help* y *viewer*. A grandes rasgos, *File* le brinda acceso al directorio de archivos en su computadora y en *Plots* se visualizan todos los gráficos creados. Además, *Packages* muestra una lista de todos los paquetes instaladas en R, mientras que *Help* es es el menú de ayuda para las diferentes funciones en R.

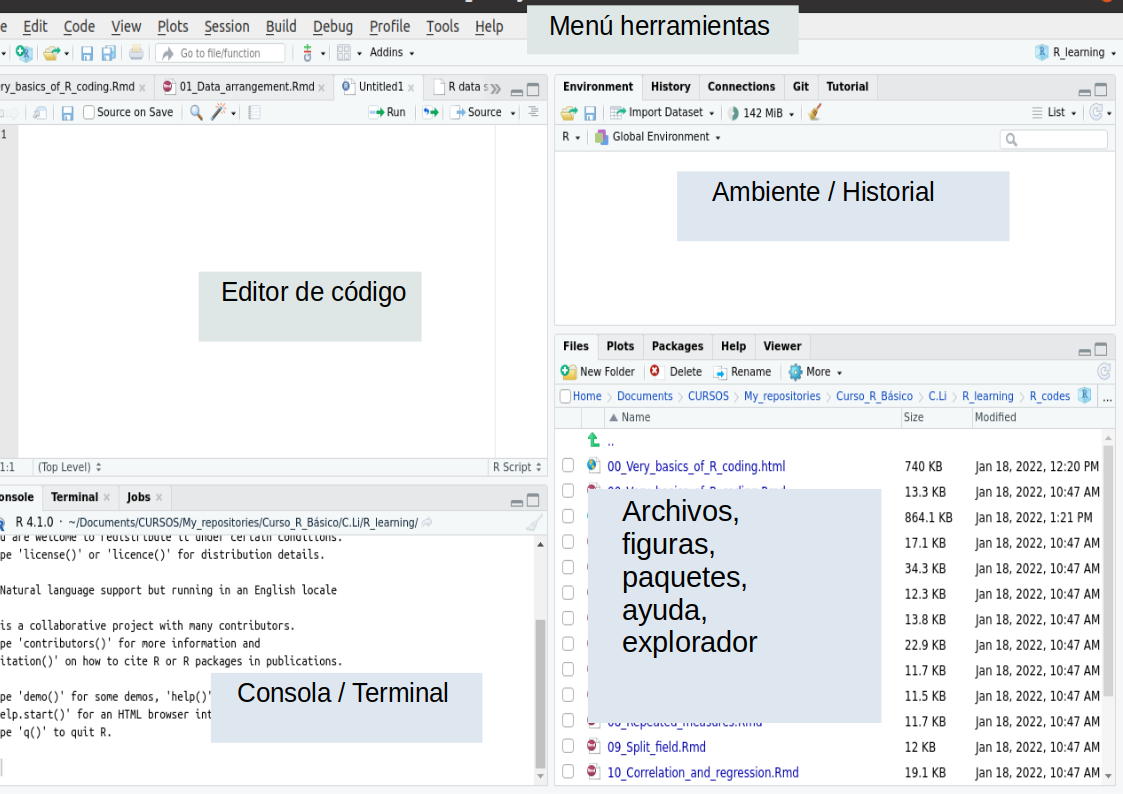
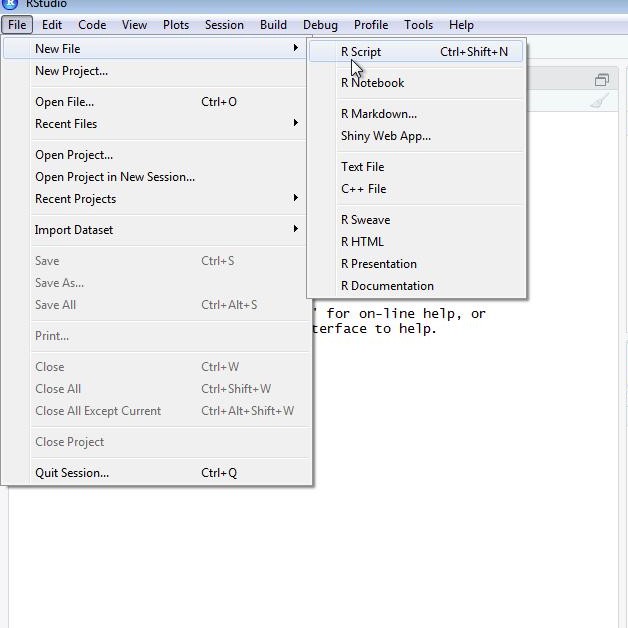


Figure 1.1: Se representan los cuatro paneles de Rstudio

### 1.1.6 *Scripts*

Una de las principales ventajas de R es la posibilidad de guardar los códigos y comandos mediante *scripts*. Un *script* es un archivo de texto que contiene una serie de instrucciones y comentarios. Para crear un nuevo *script*, solo tiene que hacer clic en *“File”*, luego seleccionar *“New File”* y elegir *“R Script”*. Esto abrirá un nuevo panel en la pantalla, donde podrá comenzar a escribir sus códigos y comentarios (Figura 1.2).



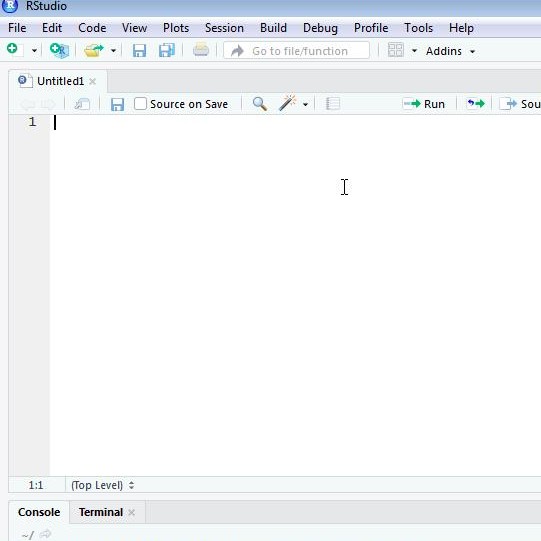


Figure 1.2: Representación gráfica de como abrir un nuevo script (File > New file > R script).

Podemos abrir y ejecutar scripts en R usando la función *source()*, indicando la ruta del archivo ***.R*** en nuestra computadora, o dar click en File > Open File y luego buscar la carpeta donde se encuentre el *script*. Por ejemplo.

source("C:/mi\_script.R")

#### 1.1.6.1 Cómo ejecutar comandos mientras se editan *scripts*

Para comenzar, abrimos un nuevo *script* y le asignamos un nombre. Esto se puede hacer guardando el script sin nombre a través del editor (Figura 1.3). Al guardar el *script* por primera vez, es importante usar un nombre descriptivo, en minúsculas, sin espacios y preferiblemente con guiones bajos (“\_“) para separar las palabras. Se deben evitar los guiones (”-“). Un ejemplo de nombre adecuado para el script podría ser: *“mi\_script.R”*.

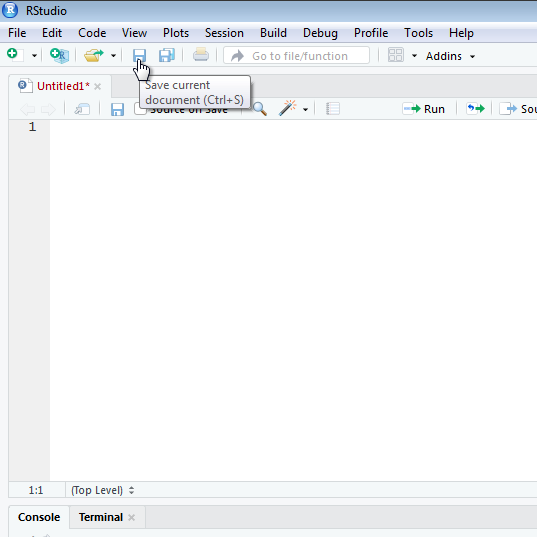


Figure 1.3: Con el editor de scripts como ventana activa, guarde y asigne un nombre a su archivo, junto con la extensión de archivo “. R”.

Ahora podemos editar nuestro primer script. Las primeras líneas de código se utilizan para agregar títulos o comentarios, los cuales se indican con el símbolo “#” al inicio de cada línea. A continuación, cargamos los paquetes y los datos que vamos a utilizar en el análisis. En una sección posterior, profundizaremos más sobre cómo cargar datos y paquetes de manera adecuada. Como ejemplo, utilizaremos la base de datos “Iris”. Primero, cargamos la librería “datasets” y posteriormente los datos. Luego, obtendremos un resumen de los datos y, finalmente, creamos un gráfico boxplot. Para ejecutar cada línea de código, debemos hacer clic en el botón “Run” en la parte superior derecha del panel de edición. También se puede ejecutar una línea de código usando las combinaciones de teclas: Control+Enter en Windows y Linux, o Command+Return en Mac.

Ejecute las siguientes líneas de código y obtenga los resultados provistos en la Figura 1.4:

library(datasets)  
data(iris)  
summary(iris)  
boxplot(iris)

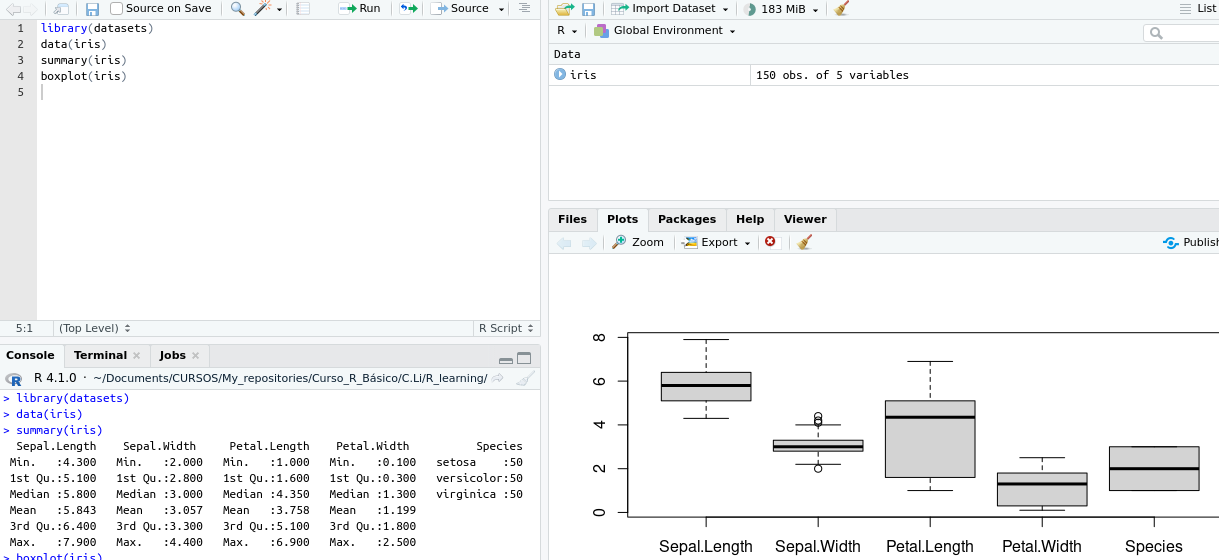


Figure 1.4: Resumen (panel inferior izquierdo) y gráfico boxplot (panel inferior derecho) de la base de datos “Iris”.

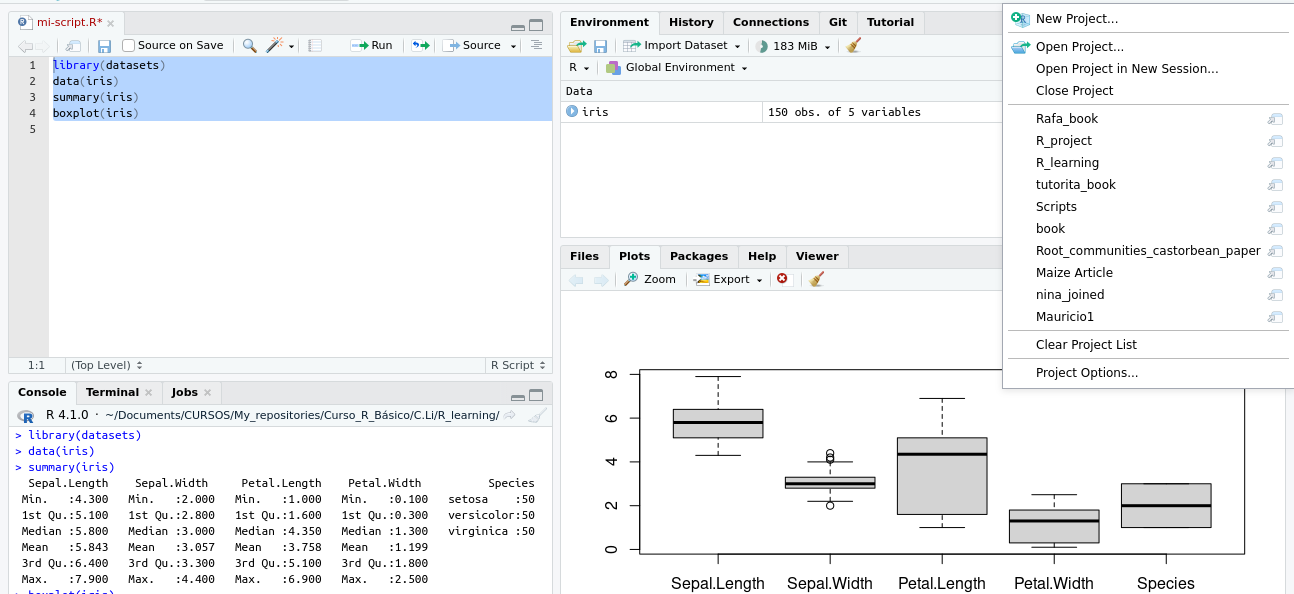
### 1.1.7 Directorio de trabajo

El directorio de trabajo es la carpeta donde se encuentran los archivos (base de datos) que posteriormente son importados a R. Para saber cuál es la ubicación del directorio de trabajo use la siguiente función *“getwd”*. Para cambiar el directorio de trabajo, debe usar la función *“setwd”* y establecer la respectiva ruta (e.g. “/home/Desktop/”). Además, se puede conocer el contenido del directorio, utilizando las siguientes funciones *“list.files”* y *“list.dirs”*.

getwd()  
setwd("/home/steph/Desktop/")  
list.files()  
list.dirs()

### 1.1.8 Sesión

En R, los objetos y funciones se almacenan en la memoria durante una sesión y no se comparten entre sesiones. Cada vez que inicia R, se crea una nueva sesión. Debe guardar los objetos generados durante la sesión, ya que los scripts se guardan automáticamente, pero no los objetos. Estos se guardan en un archivo **.RData** en su directorio de trabajo. Para ver todos los objetos guardados, utilice la función *“ls”*. En la Figura 1.5 se muestra cómo crear un nuevo proyecto en R.



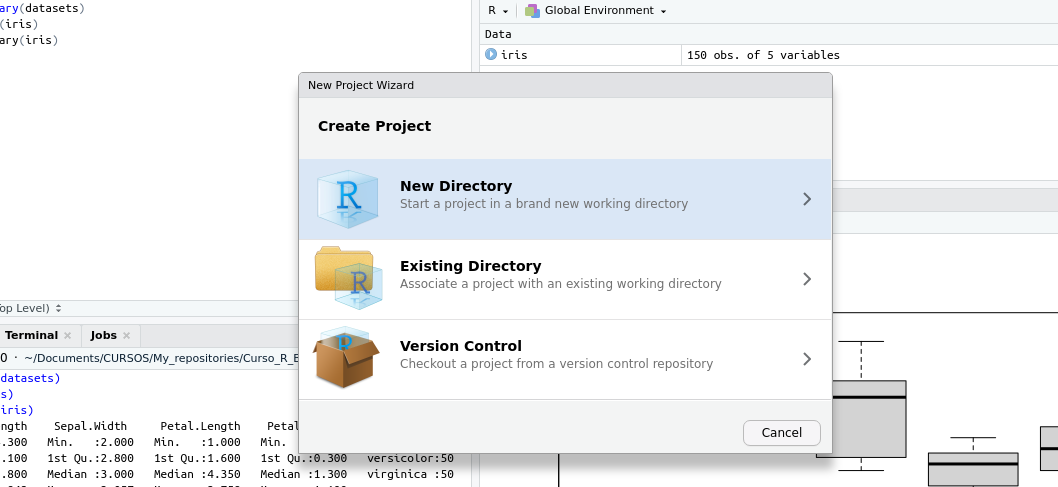


Figure 1.5: Creación de un nuevo proyecto en R.

### 1.1.9 Proyecto

Un proyecto de R (con extensión .Rproj) es una carpeta que contiene todos los archivos y recursos relacionados. Dentro del proyecto, los scripts, archivos de datos, resultados e historial se organizan en subcarpetas, lo que facilita la gestión del trabajo. Al crear un proyecto, todos los archivos quedan vinculados, lo que mejora la organización y el flujo de trabajo. Para crear un proyecto debe realizar los siguientes pasos (File>New File> New directory>New Project> Colocar el nombre del directorio y dónde se guardará>Create Project; ver Figura 1.6)

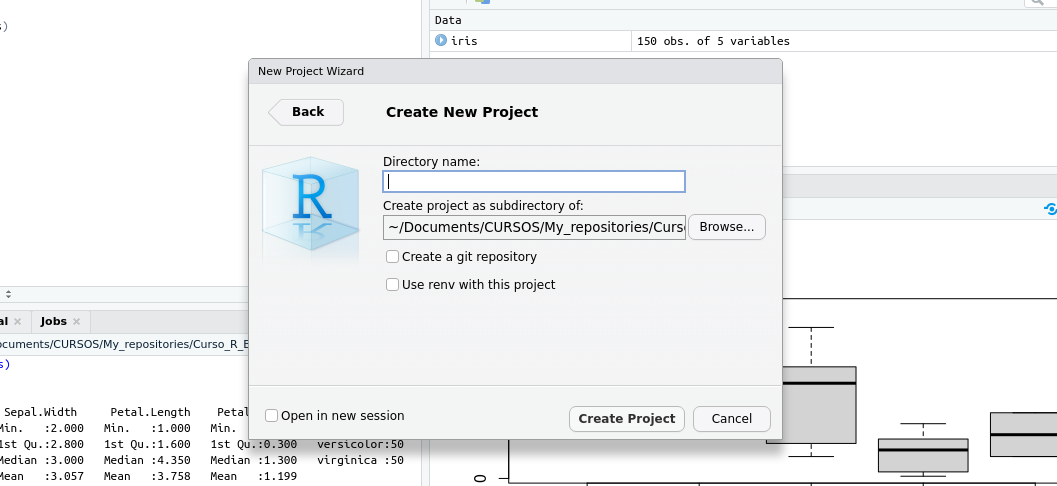


Figure 1.6: Nombrando el nuevo proyecto y escogiendo dónde se guardará.

### 1.1.10 Instalación de paquetes de R

Una de las principales ventajas de R es que numerosos desarrolladores y programadores crean paquetes continuamente, los cuales están disponibles de forma gratuita y ofrecen diversas funcionalidades. Actualmente, existen muchos paquetes accesibles a través de CRAN (Comprehensive R Archive Network), una red global de servidores que proporcionan versiones actualizadas. Además, se pueden encontrar numerosos paquetes en plataformas como GitHub y Bioconductor. Un ejemplo de ello es el paquete *tidyverse* que exploraremos en sesiones posteriores. Para instalarlo, utilice el siguiente código:

install.packages("tidyverse")

En RStudio, se puede ir a la pestaña “Packages” y hacer clic en “Install”. Luego, debe escribir el nombre del paquete que desea instalar, siempre y cuando esté disponible en CRAN. Posteriormente, cargue la librería utilizando la función **library()**:

library(tidyverse)

Una vez instalados los paquetes, no deben instalarse nuevamente. Sin embargo, cada vez que cerramos o reiniciamos sesión es necesario volver a cargarlos. Además, debemos tener en cuenta que la instalación de tidyverse incluye varios paquetes. Esto ocurre cuando un paquete tiene dependencias. Algunos paquetes no están en CRAN y se requiere de otros para su instalación. Por ejemplo, para instalar la versión en desarrollo de rmarkdown, se debe utilizar el paquete devtools. Los dos puntos “::” permiten llamar una función específica de un paquete sin cargarlo completamente.

devtools::install\_github('rstudio/rmarkdown')

### 1.1.11 Tipos de objetos en R

La información que manipulamos en R se estructura en forma de objetos y los podemos ver almacenados en el panel del ambiente de trabajo o *Enviroment*. Una ventaja de los lenguajes de programación es la posibilidad de definir variables y escribir expresiones matemáticas para almacenar valores que se usarán eventualmente. En R, usamos <- para asignar valores a las variables. Aunque también se puede usar =, se recomienda evitarlo ya que el signo = se utiliza para igualdad lógica, lo que podría generar confusión.

Los objetos pueden ser de diferentes tipos:

* Números escalares o letras
* Vectores y matrices
* Dataframes, tablas y listas

En las siguientes líneas de comando se puede observar diferentes tipos de objetos:

a <- 1 #escalar  
letra <- "a" #caracter o letra  
b <- c(1,2,3) #vector  
c<- matrix(1:10) #matriz  
d<- data.frame(Especie=c("A", "B"), Longitud=c(c(1,2))) #dataframe o tabla  
e<- list(c(1:20), c(1:10)) #lista

Para visualizar la información almacenada en un objeto, simplemente le pedimos a R que evalúe esa información mediante la función print o escribiendo directamente el objeto creado. Por Ejemplo:

print(a)

## [1] 1

Otra forma de examinar los objetos es buscarlos directamente en el ambiente de trabajo. Si intentamos visualizar el valor de un objeto que no está definido en el ambiente de trabajo se recibirá un mensaje de error. Por ejemplo, si escriben la letra f, encontrarán el siguiente error: ***object “f” not found***.

#### 1.1.11.1 Algunos tips para asignar variables u objetos en R…

* Los nombres de variables tienen que comenzar con una letra, no pueden contener espacios y no deben ser variables predefinidas en R (como funciones o argumentos de funciones). Por ejemplo, nombrar un objeto de la siguiente manera “install.packages <- 2” provocará confusiones para R, ya que lo reconocerá como una función.
* Los nombres deben ser descriptivos para que reflejen el contenido almacenado. Es recomendable usar minúsculas y separar las palabras con guiones bajos (\_) en lugar de espacios. Además, se deben evitar caracteres especiales como guiones (-), puntos (.), punto y coma (;), arroba (@) o signos de interrogación (?).

#### 1.1.11.2 Guardar los espacios de trabajo y exportar objetos de R

Los objetos evaluados permanecen en el espacio de trabajo hasta finalizar sesión. Sin embargo, puede guardar el espacio de trabajo para usarlo posteriormente. Al finalizar sesión, el programa muestra una ventana emergente sugiriendo si desea guardar el *script*. Además, al reiniciar R, se restaurará con los objetos previamente guardados. Por otra parte, es recomendable crear un proyecto de trabajo (como vimos previamente) para guardar todos los archivos y scripts que estén relacionados con el mismo.

Usando como ejemplo el “data.frame” creado en el ejemplo anterior “d”. R posee funciones para la exportación de datos en formato de textos editables, como son:

library(readr)  
write\_tsv(d, "database.tsv")  
write.table(d, "datase.txt", sep = "\t")  
write\_csv(da, "database.csv")

También existe la forma de guardar objetos sin declararlos texto, sino como la identidad de R. Por ejemplo:

saveRDS(d, "database.RDS")

Y para abrir o cargar este tipo de objetos con extensión .RDS usamos la siguiente función:

readRDS("database.RDS")

### 1.1.12 Funciones en R[[3]](#footnote-78)

Una vez definidos los objetos o variables, se usan funciones específicas para continuar con el análisis de datos. R incluye funciones preestablecidas y otras que se encuentran en determinados paquetes. En apartados previos utilizamos funciones como *write\_csv()* para exportar datos o *library()* para cargar paquetes. También existen funciones como *log()* o *sqrt()* para calcular logaritmos o raíces cuadradas. ***(Nótese que la mayoría de las funciones están definidas en inglés)***.

La sintaxis de R indica que se deben usar paréntesis para evaluar funciones. Por ejemplo, con la función *log()*, debemos asignar un valor a su argumento para obtener un resultado (e.g. “*log(a)*”. Sin embargo, algunas funciones como *ls()* no requieren argumentos y devuelven información sin necesidad de incluir información dentro de los paréntesis.

log(a)

## [1] 0

En R, podemos explorar funciones desde la pestaña “Paquetes”. Al hacer clic en un paquete, se despliegan las funciones que contiene junto con su documentación. Para conocer los argumentos de una función, podemos usar el comando *help* o preceder la función con un signo de interrogación (*?*). También es posible consultar manuales y obtener ayuda detallada sobre las funciones.:

help("log")  
?log

La página de ayuda de cada función muestra qué argumentos requiere. Por ejemplo, la función log() necesita los argumentos x y base. Algunos argumentos son obligatorios y otros opcionales, los cuales se indican en la documentación con valores predeterminados asignados con el signo =. El valor predeterminado de base en la función log() es exp(1), lo que hace que el logaritmo calculado sea el logaritmo natural:

args(log)

## function (x, base = exp(1))   
## NULL

log(x = 8, base = 2)

## [1] 3

Podemos escribir de manera más breve si omitimos el nombre de los argumentos.:

log(8,2)

## [1] 3

Existen algunas funciones que no necesitan paréntesis para ser evaluadas. Entre estas destacan los operadores aritméticos y relacionales:

a + a

## [1] 2

a - 2

## [1] -1

1 \* pi

## [1] 3.141593

2 / 3

## [1] 0.6666667

4 ^ a

## [1] 4

También podemos crear funciones personalizadas en R. Por ejemplo, si queremos definir nuestra propia versión de la función mean(), que calcula el promedio, podríamos hacerlo de la siguiente manera:

average<- function(x){sum(x)/length(x)}  
x<- 1:100  
average(x)

## [1] 50.5

mean(x)

## [1] 50.5

Note que ambas devuelven el mismo resultado. Lo importante es que hemos creado una nueva función, definiendo primero las variables (o variables) y luego las operaciones asignadas a un objeto R.

nombre <- function(argumentos) {  
 operaciones}

## 1.2 Tipos de datos

Como se ejemplificó previamente, los objetos en R pueden ser de diversos tipos, como números o caracteres. Para identificarlos, podemos usar la función *class()*, que nos permite determinar su clase o tipo de datos:

a <- 5  
class(a)

## [1] "numeric"

| Tipo | Nombre en inglés/en R | Ejemplo |
| --- | --- | --- |
| Numérico | numeric | 5.1 |
| Entero | integer | 4 |
| Real | double/float | 3.4 |
| Cadena de texto, letra | character | “a” |
| Factor | factor | Bajo |
| Lógico | logic | TRUE, FALSE |
| Perdido/Omitido | NA | NA |
| Vacío | null | NULL |

#### 1.2.0.1 Para tener en cuenta

* Un valor de tipo “character”es una cadena de caracteres, que puede incluir letras, números y símbolos. Se pueden convertir los objetos en valores de caracteres mediante la función as.character().
* Un factor es un tipo de dato exclusivo de R, que representa valores categóricos usando etiquetas en lugar de números. Cada etiqueta o valor que puede tomar un factor se llama “nivel”. Los niveles no siguen necesariamente el orden en que aparecen en los datos, y por defecto, R los ordena alfabéticamente.
* En los conjuntos de datos estadísticos, a menudo encontramos datos faltantes, que en R son representados como valores NA. En cambio, NULL indica que el valor en cuestión simplemente no existe.

#### 1.2.0.2 Coerción de datos

En R, los datos pueden ser coercionados, es decir, forzados, para transformarlos de un tipo a otro.

| Función de coerción | Tipo |
| --- | --- |
| as.integer() | Entero |
| as.numeric() | Numérico |
| as.character() | Caracter |
| as.factor() | Factor |
| as.logical() | Lógico |

Veamos algunos ejemplos:

a<- 5  
as.character(a)

## [1] "5"

as.factor("medio")

## [1] medio  
## Levels: medio

### 1.2.1 Tipos de estructura de los datos

Los datos se estructuran de diferentes formas dependiendo de su propósito, en todo caso, la función *class()* también nos puede dar información sobre los tipos de estructuras de datos.

#### 1.2.1.1 Vectores

Los vectores son colecciones de uno o más datos del mismo tipo. Por ejemplo, si tenemos un vector con datos numéricos tenemos un vector de tipo numérico. No es posible mezclar datos de tipos diferentes dentro de ellos. Por ejemplo, un vector de colores puede ser:

colores<- c("red", "black", "blue")  
is.vector(colores)

## [1] TRUE

class(colores)

## [1] "character"

Al usar las funcion is.vector() corroboramos que efectivamente es un vector al darnos TRUE pero al pedirle que nos indique el tipo con la función class nos dice que es un “character” es decir que es un vector de una cadena de texto.

Existen algunas operaciones al aplicarlas a un vector, se aplican a cada uno de sus elementos. A este proceso le llamamos **vectorización**. Las operaciones aritméticas y relacionales pueden vectorizarse. Si las aplicamos a un vector, la operación se realizará para cada uno de los elementos que contiene.

un\_vector <- c(1:10)  
un\_vector\*10

## [1] 10 20 30 40 50 60 70 80 90 100

un\_vector+1

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

un\_vector + un\_vector

## [1] 2 4 6 8 10 12 14 16 18 20

#### 1.2.1.2 Matrices y arreglos

Las matrices y arreglos no son más que vectores multidimensionales, es decir un conjunto de vectores. Al igual que un vector deben contener un sólo tipo de datos. En sentido estricto, una arreglo es una matrix pero con n dimensiones, mientras que las matrices tienen solo dos dimensiones. Las matrices y los arreglos suelen ser usados de manera regular en matemáticas y estadística, por ser sencillas y contener solo un tipo de datos (usualmente de tipo numérico). En general, es preferible usar listas en lugar de arrays, una estructura de datos que además tienen ciertas ventajas que se revisará más adelante. En R, podemos usar el símbolo : para indicar una secuencia de números que tiene un principio y fin, por ejemplo:

vect<- 1:20

Este es un vector con números que va desde el 1 al 20. Pero para hacerlo matriz hacemos:

matr<- matrix(1:20)

O para dividirlo en varias renglones y columnas:

matri<-matrix(1:20, nrow = 5, ncol = 4)  
dim(matri)

## [1] 5 4

matri

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

Con la función *dim()* podemos saber cuales son las dimensiones (es decir, largo y ancho) de nuestra matriz. Las operaciones aritméticas también son vectorizadas al aplicarlas a una matriz. La operación es aplicada a cada uno de los elementos de la matriz al igual que los vectores.

matri\*2

## [,1] [,2] [,3] [,4]  
## [1,] 2 12 22 32  
## [2,] 4 14 24 34  
## [3,] 6 16 26 36  
## [4,] 8 18 28 38  
## [5,] 10 20 30 40

Algo más que podemos hacer es tranponer una matriz para rotarla 90°.

t(matri)

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

#### 1.2.1.3 Listas

Las listas, al igual que los vectores, son estructuras de datos unidimensionales, sólo tienen largo, pero a diferencia de los vectores cada uno de sus elementos puede ser de diferente tipo o incluso de diferente clase, por lo que son estructuras heterogéneas. Podemos tener listas que contengan escalares, vectores, matrices, data frames u otras listas. Para crear una lista usamos la función list(), que nos pedirá los elementos que deseamos incluir en nuestra lista. Para esta estructura, no importan las dimensiones o largo de los elementos que queramos incluir en ella. Al igual que con un data frame, tenemos la opción de poner nombre a cada elemento de una lista.

un\_vector <- 1:20  
una\_matriz <- matrix(1:4, nrow = 5)

## Warning in matrix(1:4, nrow = 5): la longitud de los datos [4] no es un submúltiplo o múltiplo del  
## número de filas [5] en la matriz

una\_df <- data.frame("numeros" = 1:3, "letras" = c("a", "b", "c"))  
  
una\_lista <- list("vector" = un\_vector, "matriz" = una\_matriz, "df" = una\_df)  
  
una\_lista

## $vector  
## [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20  
##   
## $matriz  
## [,1]  
## [1,] 1  
## [2,] 2  
## [3,] 3  
## [4,] 4  
## [5,] 1  
##   
## $df  
## numeros letras  
## 1 1 a  
## 2 2 b  
## 3 3 c

No es posible vectorizar operaciones aritméticas usando listas. Al intentarlo nos es devuelto un error.

Finalmente, en caso de que sea necesario utilizar funciones que requieran a fuerza una matrix o un *data frame* también aplica la coerción entre estas dos estructuras de datos:

coerción\_df<- as.data.frame(una\_matriz)  
coerción\_mat<- as.matrix(una\_df)  
  
class(coerción\_df); class(coerción\_mat)

## [1] "data.frame"

## [1] "matrix" "array"

Esta coerción es muy útil por ejemplo al utilizar la función t() que transponer una *data frame*:

df\_transpuesta<- t(una\_df)  
class(df\_transpuesta)

## [1] "matrix" "array"

Como vemos la función t() cambia la estructura de los datos, para evitar esto, coercionamos esta salida.

df\_transpuesta<- as.data.frame(t(una\_df))  
class(df\_transpuesta)

## [1] "data.frame"

## 1.3 *Data frames*

La forma más común de almacenar un set de datos en R es usando un *data frame*. Los data frames son estructuras de datos de dos dimensiones (rectangulares) que pueden contener datos de diferentes tipos, por lo tanto, son heterogéneas. Podemos entender a los data frames como una versión más flexible de una matriz. Mientras que en una matriz todas las celdas deben contener datos del mismo tipo, los renglones de un data frame admiten datos de distintos tipos, pero sus columnas conservan la restricción de contener datos de un sólo tipo.

Vamos a trabajar con una data de ejemplo y exploraremos esta *data frame*, es una data en la que evaluan el efecto de la dosis de vitamina C sobre el crecimiento de los dientes de unos tipos de cerdos. La función str es útil para obtener más información sobre la estructura de un objeto:

data(ToothGrowth)  
str(ToothGrowth)

## 'data.frame': 60 obs. of 3 variables:  
## $ len : num 4.2 11.5 7.3 5.8 6.4 10 11.2 11.2 5.2 7 ...  
## $ supp: Factor w/ 2 levels "OJ","VC": 2 2 2 2 2 2 2 2 2 2 ...  
## $ dose: num 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 ...

Esto nos dice mucho más sobre el objeto. Vemos que la tabla tiene 60 filas y 3 variables. Podemos mostrar las primeras seis líneas usando la función head:

head(ToothGrowth)

## len supp dose  
## 1 4.2 VC 0.5  
## 2 11.5 VC 0.5  
## 3 7.3 VC 0.5  
## 4 5.8 VC 0.5  
## 5 6.4 VC 0.5  
## 6 10.0 VC 0.5

Para crear un data frame usamos la función *data.frame()*. Esta función nos pedirá un número de vectores igual al número de columnas que deseemos. Todos los vectores que proporcionemos deben tener el mismo largo. Es decir, un data frame está compuesto por vectores. Veamos un ejemplo:

df <- data.frame(  
 "entero" = 1:3,   
 "factor" = c("alto", "medio", "bajo"),   
 "letras" = as.character(c("a", "b", "c"))  
)  
  
df

## entero factor letras  
## 1 1 alto a  
## 2 2 medio b  
## 3 3 bajo c

dim(df)

## [1] 3 3

La función dim nos permite conocer también las dimensiones de nuestra data frame. En este caso tenemos 3 filas y 3 columnas.Las funciones names y colnames nos permiten conocer los nombres de los headers o de las columnas.

names(df)

## [1] "entero" "factor" "letras"

colnames(df)

## [1] "entero" "factor" "letras"

### 1.3.1 El operador $ y otras formas de acceso

Para tener acceso a las diversas variables o columnas de un *data.frame* utilizamos el operador de acceso $, por ejemplo, si quisieramos tener acceso a la variable ‘factor’ de la *data.frame* **df** de la siguiente manera:

df$factor

## [1] "alto" "medio" "bajo"

class(df$factor)

## [1] "character"

is.vector(df$factor)

## [1] TRUE

Cuando usamos el operador $ el tipo de objeto que obtenemos es un vector, en el ejemplo como la columna ‘factor’ es una cadena de caracteres entonces al usar las funciones *class()* y *is.vector()* nos confirma lo antes mencionado.

**Tip**: R viene con una muy buena funcionalidad de autocompletar que nos ahorra la molestia de escribir todos los nombres. Escriban df$f y luego presionen la tecla *tab* en su teclado. Esta funcionalidad y muchas otras características útiles de autocompletar están disponibles en RStudio, esto aplica también para las funciones.

En el caso de las listas también podemos acceder con el operador $, aunque también podemos usar corchetes dobles ([[) así. Por ejemplo declaramos una lista:

notas\_estudiantes <- list(nombres = c("Ana", "Clara", "Sofy"),  
 id\_estudiante = c("i1", "i2", "i3"),  
 notas = c(10, 9,7))

Y queremos extraer los nombres de los estudiantes, entonces hacemos\_

notas\_estudiantes$nombres

## [1] "Ana" "Clara" "Sofy"

notas\_estudiantes[["nombres"]]

## [1] "Ana" "Clara" "Sofy"

Y obtenemos el mismo resultado.

Para el caso de las matrices se puede acceder usando corchetes ([). Si desean la primera fila y la pimera columna, entonces:

mat<- matrix(1:10, ncol = 2, nrow = 5)  
mat[1,1]

## [1] 1

mat

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

Para acceder solo a la primera fila y solo a la primera columa usamos las comas, así: s

mat[1, ] #acceder primera fila

## [1] 1 6

mat[, 1] #acceder a la primera columna

## [1] 1 2 3 4 5

is.vector(mat[, 1])

## [1] TRUE

Notese que esto devuelve un vector, no una matriz.

Del mismo modo, si desean la segunda columna completa, dejen el lugar de la fila vacío:

mat[, 2]

## [1] 6 7 8 9 10

Esto también es un vector, no una matriz. Lo corroboramos con la función *is.vector()*

Se pueden crear subconjuntos basados tanto en las filas como en las columnas:

mat[2:4 , 1:2] #en orden de posición es filas primero y luego columnas

## [,1] [,2]  
## [1,] 2 7  
## [2,] 3 8  
## [3,] 4 9

Podemos convertir las matrices en *data frames* usando la función as.data.frame:

as.data.frame(mat)

## V1 V2  
## 1 1 6  
## 2 2 7  
## 3 3 8  
## 4 4 9  
## 5 5 10

## 1.4 Creando subconjuntos o Indexación.

En R, podemos obtener subconjuntos de nuestras estructuras de datos. Es decir, podemos extraer partes de una estructura de datos (nuestro conjunto).

También podemos usar corchetes individuales ([) para acceder a las filas y las columnas de un *data frame* y es exactamente igual que lo que se aplicó con las matrices. A esto es lo que llamamos **Subconjuntos** de los *data.frame*. Como las listas de datos que usamos para *notas\_estudiantes* tienen las mismas dimensiones entonces podemos coercionarlo a ser una *data.frame*:

evaluaciones<- as.data.frame(notas\_estudiantes)

Y para obtener más de una entrada se puede utilizar un vector de entradas múltiples como índice:

evaluaciones[c(1,2)]

## nombres id\_estudiante  
## 1 Ana i1  
## 2 Clara i2  
## 3 Sofy i3

Obtenemos las dos primeras columnas. Las secuencias definidas anteriormente son particularmente útiles si necesitamos acceso, digamos, a los dos primeros elementos:

evaluaciones[1:2]

## nombres id\_estudiante  
## 1 Ana i1  
## 2 Clara i2  
## 3 Sofy i3

Ahora bien, si queremos **NO** elegir por ejemplo la primera columna o dejarla por fuera, entonces usamos el signo **‘-’**:

evaluaciones[,-1]

## id\_estudiante notas  
## 1 i1 10  
## 2 i2 9  
## 3 i3 7

Si los elementos tienen nombres de columna o *headers* también podemos acceder a las entradas utilizando estos nombres:

evaluaciones[c("nombres","notas")]

## nombres notas  
## 1 Ana 10  
## 2 Clara 9  
## 3 Sofy 7

Ahora bien, podemos seleccionar datos que tengan características específicas, por ejemplo, todos los valores mayores a cierto número o aquellos que coinciden exactamente con un valor de nuestro interés. Para realizar esta operación haremos uso de índices y operadores lógicos.

| Operador | Comparación |
| --- | --- |
| < | Menor que |
| <= | Menor o igual que |
| > | Mayor que |
| >= | Mayor o igual que |
| == | Exactamente igual que |
| ! = | No es igual que |
| ! | No es |
| = | Igual que |
| &, | | y, ó |

Por ejemplo en el caso de la tabla de **evaluaciones**, si queremos escoger los valores que sean mayor de 8 en las notas obtenidas:

evaluaciones$notas > 8

## [1] TRUE TRUE FALSE

Observamos cuales cumplen con la condición si muestran TRUE. Ahora para usar este filtro y hacer un subconjunto con el *data.frame*, hacemos:

mas\_de\_8<-evaluaciones[evaluaciones$notas > 8,]  
mas\_de\_8

## nombres id\_estudiante notas  
## 1 Ana i1 10  
## 2 Clara i2 9

Si queremos usar más de una condición pero indicando negación:

evaluaciones[!(evaluaciones$notas > 8 & evaluaciones$nombres == "Clara"), ]

## nombres id\_estudiante notas  
## 1 Ana i1 10  
## 3 Sofy i3 7

Para escoger un valor que sea exactamente igual a una condición usamos ‘==’:

evaluaciones[evaluaciones$nombres == "Sofy",]

## nombres id\_estudiante notas  
## 3 Sofy i3 7

### 1.4.1 Importando datos

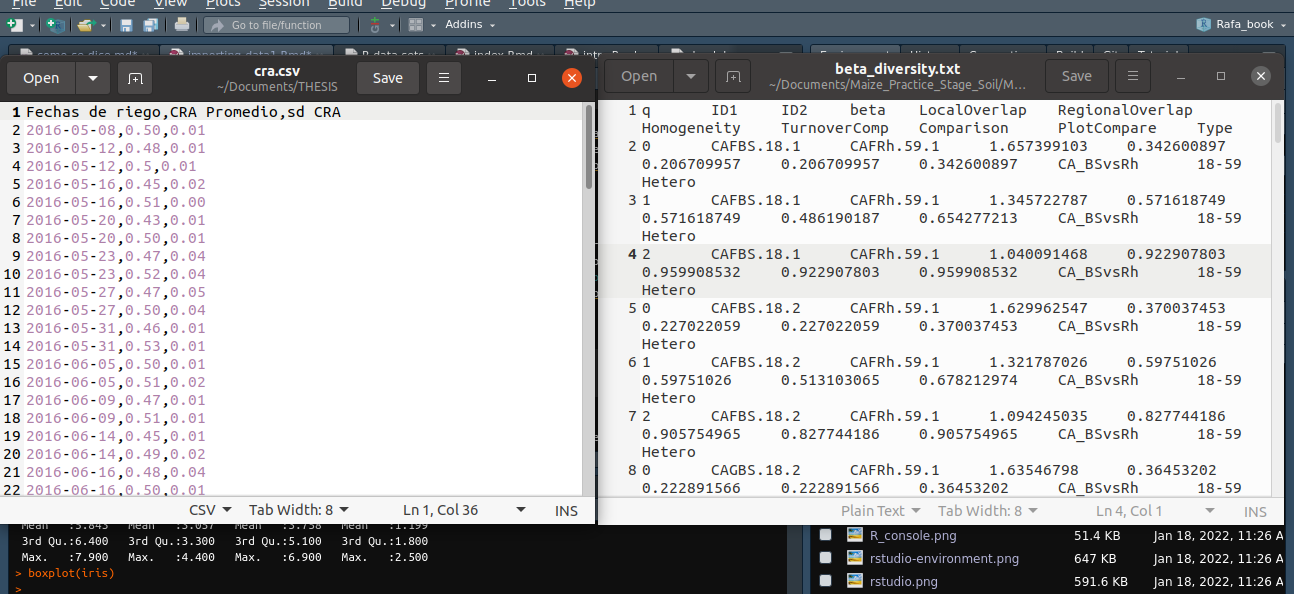
Para el ejemplo que vimos en el capítulo pasado usamos un dataset que está en el ambiente de R por default, si queremo saber cuales son los datasets que tenemos en nuestro ambiente, podemos usar el comando *data()* y nos desplegará la lista:

data()

Si queremos utilizar los datos de nuestro trabajo o usar datos de una base de datos o que de una ‘dataset’ que se encuentre en internet, debemos *Importar* estos datos a nuestra sesión de R. Usualmente tenemos nuestros datos guardados en hojas de cálculo en diferentes formatos con diferentes extensiones, estos son los más populares:

* separados con *coma* o *punto y coma* (,,;): csv,
* separados con tabulaciones o espacios (*tab,* \t) : .txt o .tsv,
* Hojas de cálculo de excel: .xls, son las más usadas.

A continuación muestro una imagen de como se ven un .csv y .txt:



## 1.5 El directorio de trabajo y rutas

Antes de importar nuestros propios archivos, tablas o datos debemos estar seguros en qué directorio nos encontramos, para estar seguros que vamos a importar el archivo deseado a R.

Existen tres opciones para esto:

1. Utilizar **getwd()** y **setwd()**, como lo vimos anteriormente, para establecer y saber en qué directorio nos encontramos y si es el caso, cambiarlo.
2. Poner la ruta completa de nuestro archivo, sin importar donde esté.
3. Utilizar **“Import Dataset”** de nuestro panel de ambiente y ubicar manualmente la ubicación del archivo.

El reto de la primera opción es permitir que las funciones de importación de R sepan dónde buscar el archivo que contiene los datos. La forma más sencilla de hacer esto es tener una copia del archivo en la carpeta donde las funciones de importación buscan por defecto, es decir guardar este archivo en nuestro directorio de trabajo.

### 1.5.1 Descargando un archivo de la web

Para descargar algún archivo en la web a utilizar, podemos correr el siguiente código:

download.file(  
 url = "https://raw.githubusercontent.com/sagar3122/Machine-Learning/refs/heads/master/iris.data.txt",   
 destfile = "iris.data")

Si observamos nuestros archivos en nuestro directorio de trabajo con el código *list.files()* veremos que se encuentra esta data que hemos descargado.

El código anterior no lee los datos sólo, en este caso, descarga la data. Otra forma de descargarlo y a la vez abrirlo es de la siguiente manera, con el paquete *readr*:

iris\_dat<-readr::read\_csv("https://raw.githubusercontent.com/sagar3122/Machine-Learning/refs/heads/master/iris.data.txt")

## 1.6 Funciones de importación

Una vez descargado o que se encuentre en nuestro directorio de trabajo, podemos importar los datos con solo una línea de código. Aquí usamos la función read.csv o read.delim de R base (que viene default cuando descargamos R).

iris\_data<- read.csv("iris.data", header = F)  
iris\_data<- read.delim("iris.data",header = F, sep = ",")

Los datos se importan y almacenan en el obejto iris\_dat. Los argumentos *header = F* y *sep=“,”* son parámetros extras que podemos agregar a la función para indicarle algunas cosas. Por ejemplo *header=F*, le estamos diciendo que la prima fila no contiene los títulos o *headers* de la tabla, en caso de que si fuera así, le daríamos *TRUE*. Podemos usar la tecla *‘tab’* para explorar las demás opciones que podemos utilizar en estas funciones.

También el paquete *readr* tiene otras funciones de importación muy parecidas:

library(readr)  
iris\_data<-read\_csv("iris.data", col\_names = c("Longitud.sepalo", "Ancho.Sepalo" ,  
 "Longitud.Petalo" ,"Ancho.Petalo" , "Especies" ))

En esta función usamos el argumento *col\_names* para establecer los *Headers* o nombre de las columnas de esta tabla.

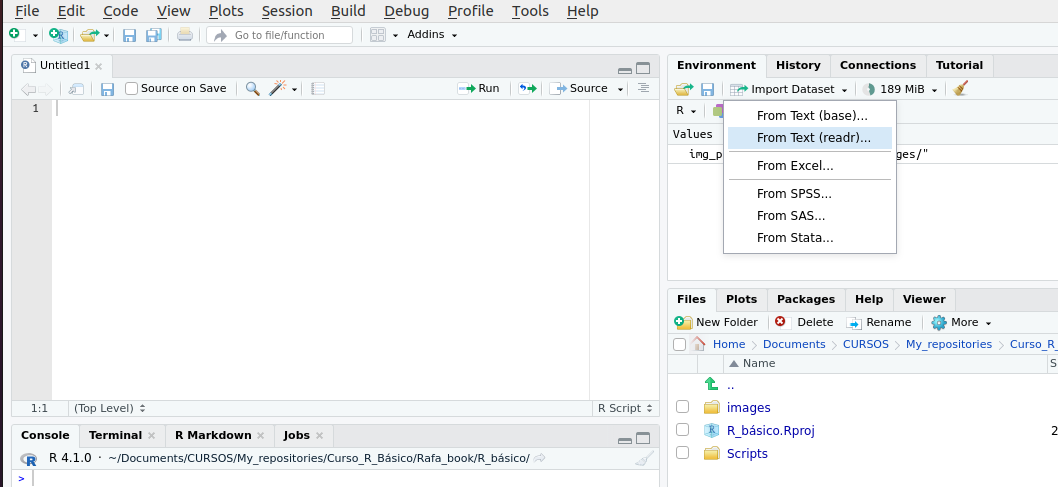
La segunda opción que vimos es utilizar la ruta completa del archivo, por ejemplo:

data<- read\_csv("Data/penguins\_size.csv")

## Rows: 344 Columns: 7  
## ── Column specification ────────────────────────────────────────────────────────────────────────────  
## Delimiter: ","  
## chr (3): species, island, sex  
## dbl (4): culmen\_length\_mm, culmen\_depth\_mm, flipper\_length\_mm, body\_mass\_g  
##   
## ℹ Use `spec()` to retrieve the full column specification for this data.  
## ℹ Specify the column types or set `show\_col\_types = FALSE` to quiet this message.

En este ejemplo, la data que importamos está ubicada en la carpeta de ‘Data’ de este proyecto. Y vemos que esta función de *readr* también nos da información del archivo como el tipo de variable y el nombre de las mismas.

La última opción, un poco más fácil para algunos y más interactiva es usar *Import Dataset* del planel del ambiente.



En esta opción podemos importar tablas con cualquiera de las funciones que despliega dependiendo del tipo de archivo, yo recomiendo si es .csv o .txt usar *readr* que es el mismo que usa tidyverse, como lo vimos anteriormente. Paquete que veremos más detalladamente junto con *readxl* en la siguiente sección.

### 1.6.1 Los paquetes readr y readxl[[4]](#footnote-101)

#### 1.6.1.1 readr

El paquete **readr** un paquete de tiyverse, tiene las siguientes funciones para importar arvhivos con diferentes extensiones:

| Función | Tipo de archivo | Extensión |
| --- | --- | --- |
| read\_table | valores separados por espacios en blanco | txt |
| read\_csv | valores separados por comas | csv |
| read\_csv2 | valores separados por punto y coma | csv |
| read\_tsv | valores separados delimitados por tab | tsv o txt |
| read\_delim | formato de archivo de texto general, debe definir delimitador | txt, csv o tsv |

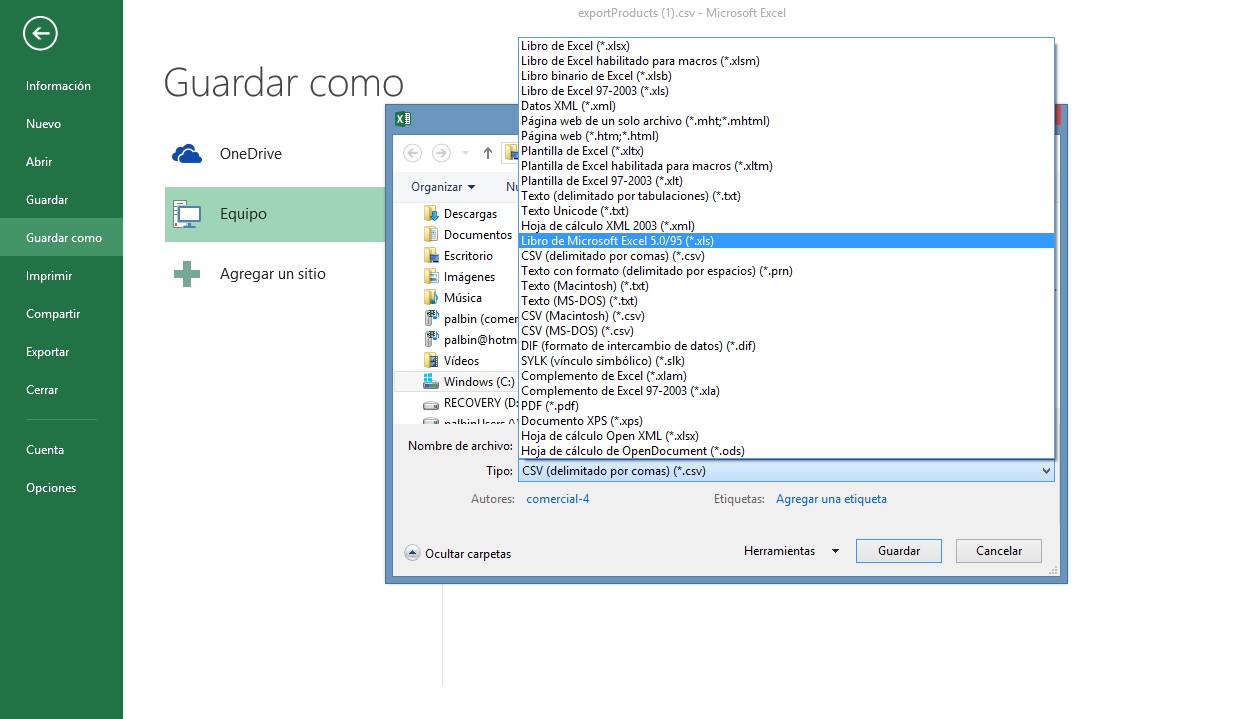
#### 1.6.1.2 readxl

Este paquete ofrece funciones para leer archivos provienetes de Microsoft Excel:

| Función | Formato | Sufijo típico |
| --- | --- | --- |
| read\_excel | detectar automáticamente el formato | xls, xlsx |
| read\_xls | formato original | xls |
| read\_xlsx | nuevo formato | xlsx |

### 1.6.2 Algunos tips para hojas de cálculo de excel…

* Evitar cosas como tener muchos diferentes formatos (colores, subrayados, etc).
* Evitar en lo posible celdas vacías y poner un dato en cada celda
* Evitar celdas con cálculo o aplicación de fórmulas.
* Tenga en cuenta que nuestros datos que tenemos en hojas de cálculo en excel también podemos guardarlos en otros formatos un poco más fáciles para ser importados como los que ya vimos (.csv y .txt). Esto utilizando la opción *Guardar como* y escogiendo el tipo de formato deseado. En la imagen podemos ver un ejemplo de esto:



## 1.7 Algunas funciones Básicas de R

### 1.7.1 sort y order

Digamos que queremos clasificar las notas de la mayor a la menor, podemos usar alguna de estas dos funciones:

sort(evaluaciones$notas)

## [1] 7 9 10

order(evaluaciones$notas)

## [1] 3 2 1

### 1.7.2 max y which.max

Si solo estamos interesados en la entrada con el mayor valor, podemos usar max:

max(evaluaciones$notas)

## [1] 10

y which.max nos dice que valor es el mayor, posicionalmente:

which.max(evaluaciones$notas)

## [1] 1

Para el mínimo, podemos usar min y which.min del mismo modo.

### 1.7.3 which

La función which nos dice qué entradas de un vector lógico son TRUE. Entonces podemos escribir:

ind <- which(evaluaciones$nombres == "Ana")  
ind

## [1] 1

evaluaciones[ind,]

## nombres id\_estudiante notas  
## 1 Ana i1 10

De esta forma también podemos usarlo para filtrar y hacer subconjuntos.

### 1.7.4 match

La función match nos dice qué índices de un segundo vector coinciden con cada una de las entradas de un primer vector:

v1<- c("Uvas", "Peras", "Mandarinas", "Plátanos", "Manzanas")  
v2<- c("Uvas","Cerezas", "Mandarinas", "Naranjas", "Manzanas")   
match(v1, v2)

## [1] 1 NA 3 NA 5

match(c("Peras", "Plátanos"), v1)

## [1] 2 4

ind<-match(c("Peras", "Plátanos"), v1)  
v1[ind]

## [1] "Peras" "Plátanos"

Este filtro puede aplicarse de igual manera a un *data.frame*:

ind2<- match(v1, v2)  
frutas<- data.frame(persona1=v1,persona2=v2)  
frutas[ind,]

## persona1 persona2  
## 2 Peras Cerezas  
## 4 Plátanos Naranjas

na.omit(frutas[ind2,]) #na.omit() nos permite quitar las celdas que contienen NA's

## persona1 persona2  
## 1 Uvas Uvas  
## 3 Mandarinas Mandarinas  
## 5 Manzanas Manzanas

### 1.7.5 %in%

Si en lugar de un índice queremos un lógico que nos diga si cada elemento de un primer vector está en un segundo vector, podemos usar la función %in%. Siguiendo el ejemplo pasado:

c("Peras", "Plátanos") %in% frutas$persona1

## [1] TRUE TRUE

Nos dice que los dos elementos qu buscamos están presente en el *data.frame()*

**Avanzado**: match y %in% pueden dar el mismo output usando which:

match(c("Peras", "Plátanos"), frutas$persona1)

## [1] 2 4

which(frutas$persona1 %in% c("Peras", "Plátanos"))

## [1] 2 4

### 1.7.6 La familia de funciones apply

Esta familia de funciones es usada para aplicar una función a cada elemento de una estructura de datos. En particular, es usada para aplicar funciones en matrices, data frames, arreglos y listas. Para entender más fácilmente el uso de la familia apply, recordemos la vectorización de operaciones. Hay operaciones que, si las aplicamos a un vector, son aplicadas a todos sus elementos. La familia apply esta formada por las siguientes funciones:

* **apply()**
* **lapply()**
* **mapply()**
* **sapply()**
* eapply()
* rapply()
* tapply()
* vapply()

Es una familia numerosa y esta variedad de funciones se debe a que varias de ellas tienen aplicaciones sumamente específicas. Las más usadas son las que están en negrita, repasaremos la función apply pero no nos detendremos mucho porque muchas de estas no están al alcance del presente curso.

### 1.7.7 apply

apply aplica una función a todos los elementos de una **matriz**.

La estructura de esta función es la siguiente.

apply(X, MARGIN, FUN)

apply tiene tres argumentos:

* X: Una matriz o un objeto que pueda coercionarse a una matriz, generalmente, un data frame.
* MARGIN: La dimensión (margen) que agrupará los elementos de la matriz X, para aplicarles una función. Son identificadas con números, **1** son renglones y **2** son colummnas.
* FUN: La función que aplicaremos a la matriz X en su dimención MARGIN.

Si queremos sumar todas las columnas de una matriz, podemos aplicar esta función, para comparar usaremos también la función ColSums() que realiza esta misma operación:

matriz<- matrix(1:20, ncol = 5, nrow = 4)  
matriz

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

apply(X = matriz, MARGIN = 2, FUN = sum)

## [1] 10 26 42 58 74

colSums(matriz)

## [1] 10 26 42 58 74

También podemos aplicar múltiples funciones a una matriz:

multiples.func <- function(x) {  
 c(sum = sum(x), prom = mean(x), max = max(x))}  
apply(X = matriz, MARGIN = 2, FUN = multiples.func)

## [,1] [,2] [,3] [,4] [,5]  
## sum 10.0 26.0 42.0 58.0 74.0  
## prom 2.5 6.5 10.5 14.5 18.5  
## max 4.0 8.0 12.0 16.0 20.0

### 1.7.8 Estructuras de control

Estas estructuras nos permiten controlar la manera en que se ejecuta nuestro código.Se establecen como condicionales en nuestros código. Por ejemplo, qué condiciones deben cumplirse para realizar una operación o qué debe ocurrir para ejecutar una función.

Las estructuras de control más usadas son:

| Estructura de control | Descripción |
| --- | --- |
| if, else | Si, de otro modo |
| while | mientras |
| for | Para |
| break | interrumpe |
| next | siguiente |

También las tocaremos pero no profundizaremos mucho en ellas, pero conoceremos como se utilizan.

#### 1.7.8.1 If, else

If y else se utilizan para crear condiciones, por ejemplo, si cumple esta condición entonces haz esto, de otra manera, haz esto.

Ejemplo:

if(10>2) {"Verdadero"  
} else {  
 "Falso"  
}

## [1] "Verdadero"

if(10<2) {"Verdadero"  
} else {  
 "Falso"  
}

## [1] "Falso"

También hay una función que reune estas dos condiciones, esifelse() y se usa de igual manera:

ifelse((10>2), "Verdadero", "Falso")

## [1] "Verdadero"

Podemos aplicarlo en los *data.frames* usando como ejemplo el dataset anterior:

ifelse(evaluaciones$notas>7, "Aprobado", "Reprobado")

## [1] "Aprobado" "Aprobado" "Reprobado"

#### 1.7.8.2 for

La estructura for nos permite ejecutar un bucle (*loop*), realizando una operación para cada elemento de un conjunto de datos.

Ejemplo:

un\_vector<- 1:10  
for(i in un\_vector) {  
 print(i\*2)  
}

## [1] 2  
## [1] 4  
## [1] 6  
## [1] 8  
## [1] 10  
## [1] 12  
## [1] 14  
## [1] 16  
## [1] 18  
## [1] 20

### 1.7.9 while

Este es un tipo de bucle que ocurre **mientras** una condición es verdadera (TRUE). La operación se realiza hasta que se se llega a cumplir un criterio previamente establecido. Ejemplo:

umbral <- 3  
valor <- 0  
  
while(valor < umbral) {  
 print("Aún no llegas al umbral")  
 valor <- valor + 1  
}

## [1] "Aún no llegas al umbral"  
## [1] "Aún no llegas al umbral"  
## [1] "Aún no llegas al umbral"

Para revisar las demás estructuras, podemos revisar la referencia citada[[5]](#footnote-120)

### 1.7.10 Tratando con datos NA

#### 1.7.10.1 1. Probando qué tenemos NA en nuestros datos:

data <- data.frame(x1 = c(NA, 5, 6, 8, 9),   
 x2 = c(2, 4, NA, NA, 1),   
 x3 = c(3,6,7,0,3),   
 x4 = c("Hola", "algo",  
 NA, "Chao", NA))   
is.na(data)

## x1 x2 x3 x4  
## [1,] TRUE FALSE FALSE FALSE  
## [2,] FALSE FALSE FALSE FALSE  
## [3,] FALSE TRUE FALSE TRUE  
## [4,] FALSE TRUE FALSE FALSE  
## [5,] FALSE FALSE FALSE TRUE

is.na(data$x2)

## [1] FALSE FALSE TRUE TRUE FALSE

which(is.na(data))

## [1] 1 8 9 18 20

sum(is.na(data))

## [1] 5

#### 1.7.10.2 2. Omitir NAs

mean(data$x1, na.rm=TRUE)

## [1] 7

data[complete.cases(data),]

## x1 x2 x3 x4  
## 2 5 4 6 algo

na.omit(data)

## x1 x2 x3 x4  
## 2 5 4 6 algo

data[!is.na(data$x2),]

## x1 x2 x3 x4  
## 1 NA 2 3 Hola  
## 2 5 4 6 algo  
## 5 9 1 3 <NA>

#### 1.7.10.3 3. Reemplazar NA’s

#reemplazar con 0  
data[is.na(data)] <- 0  
#reemplazar con el promedio o mediana  
data$x1[is.na(data$x1)] <- mean(data$x1, na.rm = TRUE)  
data$x2[is.na(data$x2)] <- median(data$x2, na.rm = TRUE)

Para ver soluciones más complejas podemos buscar los paquetes *Hmisc* (impute), *mice* (mice) y *rpart* (repart).

# 2 Manejo de data frames y bases de datos

## 2.1 Manipulación de datos con Rbase

### 2.1.1 Filtrado y obtención de subconjuntos

En los capítulos anteriores ya vimos algunos ejemplos de subconjuntos y filtrados, retomaremos algunos de estos y también veremos unos nuevos. Para ejemplificar mejor esta parte, de nuevo trabajaremos con el dataset de **ToothGrowth** y utilizaré la función **head()** para ver solo las 6 primeras filas:

data("ToothGrowth")

#### 2.1.1.1 Seleccionando columnas

head(ToothGrowth[1:2])

## len supp  
## 1 4.2 VC  
## 2 11.5 VC  
## 3 7.3 VC  
## 4 5.8 VC  
## 5 6.4 VC  
## 6 10.0 VC

head(ToothGrowth[-3])

## len supp  
## 1 4.2 VC  
## 2 11.5 VC  
## 3 7.3 VC  
## 4 5.8 VC  
## 5 6.4 VC  
## 6 10.0 VC

head(ToothGrowth[c("dose", "len")])

## dose len  
## 1 0.5 4.2  
## 2 0.5 11.5  
## 3 0.5 7.3  
## 4 0.5 5.8  
## 5 0.5 6.4  
## 6 0.5 10.0

head(ToothGrowth[-1:-2,1:2])

## len supp  
## 3 7.3 VC  
## 4 5.8 VC  
## 5 6.4 VC  
## 6 10.0 VC  
## 7 11.2 VC  
## 8 11.2 VC

#### 2.1.1.2 Filtrando filas

Tradicionalmente indexando podemos filtrar nuestra tabla usando:

head(ToothGrowth[-1:-2,])

## len supp dose  
## 3 7.3 VC 0.5  
## 4 5.8 VC 0.5  
## 5 6.4 VC 0.5  
## 6 10.0 VC 0.5  
## 7 11.2 VC 0.5  
## 8 11.2 VC 0.5

head(ToothGrowth[which(ToothGrowth$supp == "VC"),])

## len supp dose  
## 1 4.2 VC 0.5  
## 2 11.5 VC 0.5  
## 3 7.3 VC 0.5  
## 4 5.8 VC 0.5  
## 5 6.4 VC 0.5  
## 6 10.0 VC 0.5

ind<-ToothGrowth$len>27  
ind2<- ToothGrowth$sup=="OJ"  
head(ToothGrowth[ind,])

## len supp dose  
## 23 33.9 VC 2  
## 26 32.5 VC 2  
## 30 29.5 VC 2  
## 50 27.3 OJ 1  
## 56 30.9 OJ 2  
## 58 27.3 OJ 2

head(ToothGrowth[ind2,])

## len supp dose  
## 31 15.2 OJ 0.5  
## 32 21.5 OJ 0.5  
## 33 17.6 OJ 0.5  
## 34 9.7 OJ 0.5  
## 35 14.5 OJ 0.5  
## 36 10.0 OJ 0.5

Un detalle importante en los **dataframes** son la inserción de los llamados *“NA”* que son datos que no han sido introducidos por error o porque no se tienen los datos. En algunos análisis estos tipos de datos no son deseados porque pueden generar ruido por lo que se sugiere identificarlos, omitirlos y/o eliminarlos. Veamos un ejemplo:

promedio\_clases<- data.frame(clase=c("M", "M", "M", "B", "B", "B"),  
 notas=c(7,8,9, 10, 9, NA))  
mean(promedio\_clases$notas)

## [1] NA

Una opción que tenemos es colocar el argumento **na.rm=TRUE** para que nos ignore los NA’s la función:

mean(promedio\_clases$notas, na.rm=TRUE)

## [1] 8.6

Pero si queremos identificar cuales son los datos que nos dan NA y filtrarlos usamos la función *is.na()*:

is.na(promedio\_clases$notas)

## [1] FALSE FALSE FALSE FALSE FALSE TRUE

Indexando podemos remover estos **NA**:

promedio\_clases[!is.na(promedio\_clases$notas),]

## clase notas  
## 1 M 7  
## 2 M 8  
## 3 M 9  
## 4 B 10  
## 5 B 9

También R tiene una función que nos hace más fácil esto:

na.omit(promedio\_clases)

## clase notas  
## 1 M 7  
## 2 M 8  
## 3 M 9  
## 4 B 10  
## 5 B 9

#### 2.1.1.3 Filtrando con *subset()*

Hay una función en R básico que nos permite obtener subconjuntos o filtrar las filas de nuestras tablas de manera más intuitiva. Usaremos de nuevo la data de *ToothGrowth*:

head(subset(ToothGrowth, dose=="0.5"))

## len supp dose  
## 1 4.2 VC 0.5  
## 2 11.5 VC 0.5  
## 3 7.3 VC 0.5  
## 4 5.8 VC 0.5  
## 5 6.4 VC 0.5  
## 6 10.0 VC 0.5

subset(ToothGrowth, dose=="0.5" & supp =="OJ" & len >10)

## len supp dose  
## 31 15.2 OJ 0.5  
## 32 21.5 OJ 0.5  
## 33 17.6 OJ 0.5  
## 35 14.5 OJ 0.5  
## 39 16.5 OJ 0.5

### 2.1.2 Creando una nueva columna

Para crear una nueva columna de una *dataframe* podemos utilizar varios métodos:  
- El primero sería declarar una variable nueva de la dataframe y de ahí indicarle que se desea como nueva columna en el dataframe utilizando el *“$”*. Creemos una data como la anterior de las notas:

promedio\_notas<- data.frame(estudiante = c("E1", "E2", "E3","E1", "E2", "E3"),  
 clase=c("M", "M", "M", "B", "B", "B"),  
 notas=c(7,8,9, 10, 9, 8))  
  
promedio\_notas$ponderacion <- c(0.7,0.7,0.7, 0.3,0.3,0.3)  
  
promedio\_notas

## estudiante clase notas ponderacion  
## 1 E1 M 7 0.7  
## 2 E2 M 8 0.7  
## 3 E3 M 9 0.7  
## 4 E1 B 10 0.3  
## 5 E2 B 9 0.3  
## 6 E3 B 8 0.3

- También podemos usar las funciones *within()* y *transform():*

promedio\_notas<-within(promedio\_notas, nota\_ponderada <- notas\*ponderacion)   
  
promedio\_notas<-transform(promedio\_notas, nota\_ponderada = notas\*ponderacion)   
  
promedio\_notas

## estudiante clase notas ponderacion nota\_ponderada  
## 1 E1 M 7 0.7 4.9  
## 2 E2 M 8 0.7 5.6  
## 3 E3 M 9 0.7 6.3  
## 4 E1 B 10 0.3 3.0  
## 5 E2 B 9 0.3 2.7  
## 6 E3 B 8 0.3 2.4

### 2.1.3 *aggregate()* : Resumiendo los datos

Con la función *aggregate()* podemos resumir nuestros datos, por ejemplo:

aggregate(notas ~ clase, data = promedio\_notas, mean)

## clase notas  
## 1 B 9  
## 2 M 8

aggregate(notas ~ estudiante, data = promedio\_notas, median)

## estudiante notas  
## 1 E1 8.5  
## 2 E2 8.5  
## 3 E3 8.5

aggregate(notas ~ clase+estudiante, data = promedio\_notas, sum)

## clase estudiante notas  
## 1 B E1 10  
## 2 M E1 7  
## 3 B E2 9  
## 4 M E2 8  
## 5 B E3 8  
## 6 M E3 9

### 2.1.4 Renombrando columnas y datos

Para renombrar columnas podemo sólo reescribir el nuevo nombre por el viejo, por ejemplo:

colnames(promedio\_notas)

## [1] "estudiante" "clase" "notas" "ponderacion" "nota\_ponderada"

colnames(promedio\_notas)[2] <- "curso"  
colnames(promedio\_notas)

## [1] "estudiante" "curso" "notas" "ponderacion" "nota\_ponderada"

También si queremos cambiar todos los nombres de las columnas (no lo correré pero dejaré el ejemplo):

names(promedio\_notas) <- c("a", "b", "c", "d")

#### 2.1.4.1 Renombrando valores en una columna:

promedio\_notas$curso <- ifelse(promedio\_notas$curso == "M", "Matemáticas", "Biología")   
promedio\_notas$curso

## [1] "Matemáticas" "Matemáticas" "Matemáticas" "Biología" "Biología" "Biología"

### 2.1.5 *cbind() y rbind()*

*cbind()* y *rbind()* son funciones que nos permiten combinar y juntar vectores, matrices y tablas.

“c” es para juntar por columnas (horizontalmente, una al lado de otra) y “r” para combinar combinar por filas (verticalmente, una abajo de otra).

Ejemplos:

cbind:

correcion\_nota<- c(10,9,8,8,9,10)  
cbind(promedio\_notas, correcion\_nota)

## estudiante curso notas ponderacion nota\_ponderada correcion\_nota  
## 1 E1 Matemáticas 7 0.7 4.9 10  
## 2 E2 Matemáticas 8 0.7 5.6 9  
## 3 E3 Matemáticas 9 0.7 6.3 8  
## 4 E1 Biología 10 0.3 3.0 8  
## 5 E2 Biología 9 0.3 2.7 9  
## 6 E3 Biología 8 0.3 2.4 10

cbind(promedio\_notas, promedio\_notas)

## estudiante curso notas ponderacion nota\_ponderada estudiante curso notas ponderacion  
## 1 E1 Matemáticas 7 0.7 4.9 E1 Matemáticas 7 0.7  
## 2 E2 Matemáticas 8 0.7 5.6 E2 Matemáticas 8 0.7  
## 3 E3 Matemáticas 9 0.7 6.3 E3 Matemáticas 9 0.7  
## 4 E1 Biología 10 0.3 3.0 E1 Biología 10 0.3  
## 5 E2 Biología 9 0.3 2.7 E2 Biología 9 0.3  
## 6 E3 Biología 8 0.3 2.4 E3 Biología 8 0.3  
## nota\_ponderada  
## 1 4.9  
## 2 5.6  
## 3 6.3  
## 4 3.0  
## 5 2.7  
## 6 2.4

rbind:

notas\_fisica<- data.frame(estudiante = c("E1", "E2", "E3"),  
 curso=c("F", "F", "F", "F", "F", "F"),  
 notas=c(7, 9, 8))  
rbind(promedio\_notas, notas\_fisica)

Error in rbind(deparse.level, …) : numbers of columns of arguments do not match  
Vemos este error es debido a que tanto para hacer el cbind o el rbind se necesitan tener las mismas dimensiones entre objetos (tablas, vectores y matrices). Y en el caso del rbind, deben tener el mismo nombre de columnas (colnames). En este caso quería agregar unas filas abajo en esta tabla pero nos faltó la columna de ponderación, probemos de nuevo:

notas\_fisica<- data.frame(estudiante = c("E1", "E2", "E3"),  
 curso=c("Fisica", "Fisica", "Fisica"),  
 notas=c(7, 9, 8),  
 ponderacion = c(0.1, 0.1, 0.1),   
 nota\_ponderada= c(10, 9, 8))  
rbind(promedio\_notas, notas\_fisica)

## estudiante curso notas ponderacion nota\_ponderada  
## 1 E1 Matemáticas 7 0.7 4.9  
## 2 E2 Matemáticas 8 0.7 5.6  
## 3 E3 Matemáticas 9 0.7 6.3  
## 4 E1 Biología 10 0.3 3.0  
## 5 E2 Biología 9 0.3 2.7  
## 6 E3 Biología 8 0.3 2.4  
## 7 E1 Fisica 7 0.1 10.0  
## 8 E2 Fisica 9 0.1 9.0  
## 9 E3 Fisica 8 0.1 8.0

### 2.1.6 Uniendo tablas con *merge()*

Con la funcion *merge()* podemos unir dos *dataframes* con nombres de columnas y filas comunes:

x <- data.frame(k1 = c(1,3,3,4,5), k2 = c("a1","a2","a3","a4","a5"), data = 1:5)  
y <- data.frame(k3 = c(2,2,6,4,5), k2 = c("a1","a2","a3","a4","a5"), data = 1:5)  
  
merge(x, y, by = "k2")

## k2 k1 data.x k3 data.y  
## 1 a1 1 1 2 1  
## 2 a2 3 2 2 2  
## 3 a3 3 3 6 3  
## 4 a4 4 4 4 4  
## 5 a5 5 5 5 5

merge(x, y, by = c("k2", "data"), all = TRUE)

## k2 data k1 k3  
## 1 a1 1 1 2  
## 2 a2 2 3 2  
## 3 a3 3 3 6  
## 4 a4 4 4 4  
## 5 a5 5 5 5

Vimos como se pueden unir tablas con columnas en común estas pueden ser una o más y preferiblemente que tengan nombres diferentes las columnas que no se van a unir.

### 2.1.7 Ordenando tablas por un criterio o columna

Podemos ordenar nuestra tabla con uno o más criterios:

promedio\_notas <- promedio\_notas[order(promedio\_notas$notas, decreasing = TRUE),]  
promedio\_notas

## estudiante curso notas ponderacion nota\_ponderada  
## 4 E1 Biología 10 0.3 3.0  
## 3 E3 Matemáticas 9 0.7 6.3  
## 5 E2 Biología 9 0.3 2.7  
## 2 E2 Matemáticas 8 0.7 5.6  
## 6 E3 Biología 8 0.3 2.4  
## 1 E1 Matemáticas 7 0.7 4.9

promedio\_notas <- promedio\_notas[order(promedio\_notas$notas, promedio\_notas$ponderacion),]  
promedio\_notas

## estudiante curso notas ponderacion nota\_ponderada  
## 1 E1 Matemáticas 7 0.7 4.9  
## 6 E3 Biología 8 0.3 2.4  
## 2 E2 Matemáticas 8 0.7 5.6  
## 5 E2 Biología 9 0.3 2.7  
## 3 E3 Matemáticas 9 0.7 6.3  
## 4 E1 Biología 10 0.3 3.0

### 2.1.8 Funciones adicionales de agregación

para calcular fácilmente los promedos o sumas de todas las columnas y las filas usamos rowMeans(), colMeans(), rowSums() y colSums().

examen <- data.frame("q1" = c(1, 0, 0, 0, 0),  
 "q2" = c(1, 0, 1, 1, 0),  
 "q3" = c(1, 0, 1, 0, 0),  
 "q4" = c(1, 1, 1, 1, 1),  
 "q5" = c(1, 0, 0, 1, 1))

rowMeans(examen)

## [1] 1.0 0.2 0.6 0.6 0.4

colMeans(examen)

## q1 q2 q3 q4 q5   
## 0.2 0.6 0.4 1.0 0.6

rowSums(examen)

## [1] 5 1 3 3 2

colSums(examen)

## q1 q2 q3 q4 q5   
## 1 3 2 5 3

También si queremos saber cuántas columnas y filas tienen nuestros datos además de *dim()* y *str()* podemos usar:

nrow(examen)

## [1] 5

ncol(examen)

## [1] 5

### 2.1.9 Ejemplo aplicado[[6]](#footnote-140)

Las siguientes dos tablas muestran los resultados de dos encuestas hechas a 10 personas. En la primera encuesta preguntaron su género y su edad. Y en la segunda preguntaron su superhéroe favorito y cantidad de tatuajes que tenía.

primera<- data.frame(Nombre= c("Astrid", "Lea", "Sarina", "Remon", "Letizia",   
 "Babice", "Jonas", "Wendy", "Nivedithia", "Gioia"),  
 Sexo= c("F", "F", "F", "M", "F", "F", "M", "F", "F", "F"),  
 Edad= c(30,25,25,29,22,22,35,19,32,21))  
segunda<- data.frame(Nombre= c("Astrid", "Lea", "Sarina", "Remon", "Letizia",   
 "Babice", "Jonas", "Wendy", "Nivedithia", "Gioia"),  
 Superhéroe= c("Batman", "Superman", "Batman", "Spiderman", "Batman",   
 "Antman", "Batman", "Superman", "Maggot", "Superman"),  
 Tatuajes= c(11,15,12,5,65,3,9,13,900,0))  
knitr::kable(primera); knitr::kable(segunda)

| Nombre | Sexo | Edad |
| --- | --- | --- |
| Astrid | F | 30 |
| Lea | F | 25 |
| Sarina | F | 25 |
| Remon | M | 29 |
| Letizia | F | 22 |
| Babice | F | 22 |
| Jonas | M | 35 |
| Wendy | F | 19 |
| Nivedithia | F | 32 |
| Gioia | F | 21 |

| Nombre | Superhéroe | Tatuajes |
| --- | --- | --- |
| Astrid | Batman | 11 |
| Lea | Superman | 15 |
| Sarina | Batman | 12 |
| Remon | Spiderman | 5 |
| Letizia | Batman | 65 |
| Babice | Antman | 3 |
| Jonas | Batman | 9 |
| Wendy | Superman | 13 |
| Nivedithia | Maggot | 900 |
| Gioia | Superman | 0 |

Para hacer:

1. Combina las dos tablas en una sola y completa las siguientes asignaciones.
2. ¿Cuál es la edad media de las mujeres y hombres por separado?
3. ¿Cuál fue el número más alto de tatuajes en un hombre?
4. ¿Cuál es el porcentaje de personas debajo de 32 años que son mujeres?
5. Agrega una nueva columna a a la data llamada tatuajes.por.año que muestre cuántos tatuajes por año se ha hecho cada persona por cada año en su vida.
6. ¿Cuál persona tiene el mayor número de tatuajes por año?
7. ¿Cuáles son los nombres de las mujeres a las que su superheroe favorito es superman?
8. ¿Cuál es la mediana del número de tatuajes de cad apersona que está por encima de los 20 años y que su personaje favorito es Batman?

#### 2.1.9.1 Resolviendo

1. Combina las dos tablas en una sola y completa las siguientes asignaciones.

encuestas<- merge(primera, segunda, by = "Nombre")

2. ¿Cuál es la edad media de las mujeres y hombres por separado? cómo podemos hacer esto?

aggregate(Edad ~ Sexo, data = encuestas, mean)

## Sexo Edad  
## 1 F 24.5  
## 2 M 32.0

3. ¿Cuál fue el número más alto de tatuajes en un hombre?

males<- subset(encuestas, Sexo=="M")  
max(males$Tatuajes)

## [1] 9

4. ¿Cuál es el porcentaje de mujeres debajo de 32 años?

fem<- subset(encuestas, Sexo=="F")  
fem\_32<- fem[fem$Edad<32,]  
  
(nrow(fem\_32)/nrow(fem))\*100

## [1] 87.5

5. Agrega una nueva columna a a la data llamada tatuajes.por.año que muestre cuántos tatuajes por año se ha hecho cada persona por cada año en su vida.

encuestas$tatuajes.por.año<- encuestas$Tatuajes/encuestas$Edad  
encuestas$tatuajes.por.año

## [1] 0.3666667 0.1363636 0.0000000 0.2571429 0.6000000 2.9545455 28.1250000 0.1724138  
## [9] 0.4800000 0.6842105

6. ¿Cuál persona tiene el mayor número de tatuajes por año?

mayor\_tatuaje<-which.max(encuestas$tatuajes.por.año)  
encuestas[mayor\_tatuaje,]

## Nombre Sexo Edad Superhéroe Tatuajes tatuajes.por.año  
## 7 Nivedithia F 32 Maggot 900 28.125

7. ¿Cuáles son los nombres de las mujeres a las que su superheroe favorito es superman?

sup<-fem[fem$Superhéroe=="Superman",]  
sup$Nombre

## [1] "Gioia" "Lea" "Wendy"

8. ¿Cuál es la mediana del número de tatuajes de cada persona que está por encima de los 20 años y que su personaje favorito es superman?

ocho<- subset(encuestas, Edad>20 & Superhéroe =="Batman")  
median(ocho$Tatuajes)

## [1] 11.5

## 2.2 Manejo de datos con tidyverse

### 2.2.1 *tidyverse*

Hasta ahora hemos estado manipulando las tablas o *dataframes* creando subconjuntos mediante la indexación y utlizando otras funciones del Rbase. Sin embargo, existe todo un universo llamado *tidyverse* que nos perimte hacer todo esto que vimos y más de manera más intuitiva.

Podemos cargar todos los paquetes del *tidyverse* a la vez al instalar y cargar el paquete **tidyverse**:

library(tidyverse)

Los paquetes que se activan son dplyr, purr, tidyr, stringr, tibble para el manejo de tablas; readr para importar y exportar datos y forcats para manejo de factores. Además de ggplot2 que es el paquete para graficar.

### 2.2.2 Datos *tidy*

Hemos estado trabajando con tablas o *dataframes*, sin embargo, el *tidyverse* presenta un nuevo tipo de forma de almacenamiento de datos. Decimos que una tabla de datos está en formato *tidy* si cada fila representa una observación y las columnas representan las diferentes variables disponibles para cada una de estas observaciones. El set de datos us\_rent\_income o como lo he denominado *rentas\_us* es un ejemplo de un *data frame tidy*.

| estado | variable | estimado |
| --- | --- | --- |
| Alabama | ingreso | 24476 |
| Alabama | renta | 747 |
| Alaska | ingreso | 32940 |
| Alaska | renta | 1200 |
| Arizona | ingreso | 27517 |
| Arizona | renta | 972 |
| Arkansas | ingreso | 23789 |
| Arkansas | renta | 709 |
| California | ingreso | 29454 |
| California | renta | 1358 |

Cada fila representa una medida con cada una de las otras dos columnas proveyendo una variable diferente relacionada con las otras dos: variable (si es el ingreso o la renta) y el estado.

Ahora bien, también podemos ver la misma información pero organizada de otra forma:

| estado | ingreso | renta |
| --- | --- | --- |
| Alabama | 24476 | 747 |
| Alaska | 32940 | 1200 |
| Arizona | 27517 | 972 |
| Arkansas | 23789 | 709 |
| California | 29454 | 1358 |

Se provee la misma información, pero hay dos diferencias importantes en el formato: 1) cada fila incluye varias observaciones y 2) una de las variables, “variable”, se almacena en el encabezado.  
Para que los paquetes del *tidyverse* se utilicen de manera óptima, le tenemos que cambiar la forma a los datos para que estén en formato *tidy*, como las primera tabla. Esto podemos hacerlo fuera de R o también R tiene funciones para hacerlo, que veremos más adelante.

#### 2.2.2.1 Tibbles

Los datos o tablas resultantes luego de aplicar funciones del *tidyverse* se conocen como **tibbles** y son prácticamente igual que los *dataframes* pero con unas ligeras diferencias. Veamos:

df<- data.frame(letras=c("a", "b", "c"),  
 números=1:3)  
tib<- as\_tibble(df)

class(df)

## [1] "data.frame"

class(tib)

## [1] "tbl\_df" "tbl" "data.frame"

print(df)

## letras números  
## 1 a 1  
## 2 b 2  
## 3 c 3

print(tib)

## # A tibble: 3 × 2  
## letras números  
## <chr> <int>  
## 1 a 1  
## 2 b 2  
## 3 c 3

Se ven ligeramente diferentes, como que la tibble te muestra información como tipo de dato de columna y las dimensiones, además de omitir los rownames por defecto. Algunas funciones pueden dar error si no es de un tipo u otro, por ejemplo si en vez de ser un *dataframe* es un *tibble* pero en escencia son lo mismo y se manejan igual. De aquí en adelante nos dirijiremos indistintamente sobre las dos, aunque ya sabemos la diferencia entre una y otra.

### 2.2.3 Manipulación de *data frames*

El paquete **dplyr** del *tidyverse* ofrece funciones que realizan algunas de las operaciones más comunes y que ya vimos el capítulo anterior con R base. Las funciones principales de *dplyr* son: *select*, *mutate*, *filter* y *summarise*. Pero antes, revisemos lo que es el *pipe*.

### 2.2.4 El *pipe*: %>%

El *pipe* es la herramienta que nos permite darle *dplyr* las órdenes, comandos o funciones a realizar. Lo podemos poner con el atajo del teclado **“Ctrl + Shift + M (Windows/linux)”** y **“Cmd + Shift + M (Mac)”**.  
Con **dplyr**, podemos realizar una serie de operaciones, por escoger una columna, crear una nueva, filtrar nuestras filas y demás . En Rbase tendríamos que hacer paso por paso, por ejemplo :

#asignando la data a la variable "mi\_data"  
mi\_data<-ToothGrowth  
head(mi\_data)

## len supp dose  
## 1 4.2 VC 0.5  
## 2 11.5 VC 0.5  
## 3 7.3 VC 0.5  
## 4 5.8 VC 0.5  
## 5 6.4 VC 0.5  
## 6 10.0 VC 0.5

#escoger sólo las columnas len y dose  
mi\_data<- mi\_data[c("len", "dose")]  
head(mi\_data)

## len dose  
## 1 4.2 0.5  
## 2 11.5 0.5  
## 3 7.3 0.5  
## 4 5.8 0.5  
## 5 6.4 0.5  
## 6 10.0 0.5

#hacer una nueva columna declarando la variable "dose" como un factor   
mi\_data$dose<- factor(mi\_data$dose, levels = c(0.5,1.0, 2.0), labels =c("D0.5", "D1", "2"))  
head(mi\_data)

## len dose  
## 1 4.2 D0.5  
## 2 11.5 D0.5  
## 3 7.3 D0.5  
## 4 5.8 D0.5  
## 5 6.4 D0.5  
## 6 10.0 D0.5

#filtrar solo los que sean de dosis = 1  
mi\_data<- mi\_data[mi\_data$dose=="D1",]  
head(mi\_data)

## len dose  
## 11 16.5 D1  
## 12 16.5 D1  
## 13 15.2 D1  
## 14 17.3 D1  
## 15 22.5 D1  
## 16 17.3 D1

En cambio en dplyr y con funciones que veremos a continuación, todos estos pasos podemos resumirlos a la siguiente línea de código:

mi\_data<- ToothGrowth %>% select(len, dose) %>% mutate(  
 dose=case\_when(dose==0.5~"D0.5",dose==1~"D1", dose==2~"D2")) %>% filter(dose=="D1")  
head(mi\_data)

## len dose  
## 1 16.5 D1  
## 2 16.5 D1  
## 3 15.2 D1  
## 4 17.3 D1  
## 5 22.5 D1  
## 6 17.3 D1

Como vimos para realizar la secuencia de estos pasos y unir estas funciones en una sólo línea de código hicimos uso del *pipe* %>%. En general, el *pipe* envía el resultado que se encuentra en el lado izquierdo del *pipe* para ser el primer argumento de la función en el lado derecho del *pipe*. Aquí vemos un ejemplo sencillo:

16 %>% sqrt()

## [1] 4

Podemos continuar canalizando (*piping* en inglés) valores a lo largo de:

16 %>% sqrt() %>% log2()

## [1] 2

La declaración anterior es equivalente a:

log2(sqrt(16))

## [1] 2

Ahora veremos todas las funciones que nos permitirán manipular nuestras tablas en el mismo orden que las vimos en Rbase (el capítulo pasado).

### 2.2.5 Seleccionando columnas

*dplyr* tiena una función muy intuitiva para seleccionar las columnas que queremos en un *dataframe* y es **select()**

head(select(.data = ToothGrowth, len, dose))

## len dose  
## 1 4.2 0.5  
## 2 11.5 0.5  
## 3 7.3 0.5  
## 4 5.8 0.5  
## 5 6.4 0.5  
## 6 10.0 0.5

Usandola con el *pipe* sería algo así:

#seleccionando columnas que queremos  
ToothGrowth %>% select(len, dose) %>% head()

## len dose  
## 1 4.2 0.5  
## 2 11.5 0.5  
## 3 7.3 0.5  
## 4 5.8 0.5  
## 5 6.4 0.5  
## 6 10.0 0.5

ToothGrowth %>% select(len:dose) %>% glimpse()

## Rows: 60  
## Columns: 3  
## $ len <dbl> 4.2, 11.5, 7.3, 5.8, 6.4, 10.0, 11.2, 11.2, 5.2, 7.0, 16.5, 16.5, 15.2, 17.3, 22.5, 1…  
## $ supp <fct> VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, V…  
## $ dose <dbl> 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, …

#Seleccionando columnas que no queremos  
ToothGrowth %>% select(-len) %>% glimpse()

## Rows: 60  
## Columns: 2  
## $ supp <fct> VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, V…  
## $ dose <dbl> 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, …

La función **glimpse** nos permite ver los datos como la función *str* pero uniendolo al pipe.

También podemos seleccionar columnas con criterios, por ejemplo:

ToothGrowth %>% select(starts\_with("d")) %>% glimpse()

## Rows: 60  
## Columns: 1  
## $ dose <dbl> 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, …

ToothGrowth %>% select(contains("ose")) %>% glimpse()

## Rows: 60  
## Columns: 1  
## $ dose <dbl> 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, …

ToothGrowth %>% select(ends\_with("ose")) %>% glimpse()

## Rows: 60  
## Columns: 1  
## $ dose <dbl> 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, …

ToothGrowth %>% select(matches("o.+e")) %>% glimpse()

## Rows: 60  
## Columns: 1  
## $ dose <dbl> 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, …

Seleccionando usando un vector, notemos también que en el orden que ponemos el vector así va a apareciendo reordenando las columnas en nuestra tabla:

colum <- c("supp", "len")  
ToothGrowth %>% select(!!colum) %>% glimpse()

## Rows: 60  
## Columns: 2  
## $ supp <fct> VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, V…  
## $ len <dbl> 4.2, 11.5, 7.3, 5.8, 6.4, 10.0, 11.2, 11.2, 5.2, 7.0, 16.5, 16.5, 15.2, 17.3, 22.5, 1…

Cada función del *tidyverse* tiene tres variantes que son *at*, *if* y *all* que al combinarlos con nuestras funciones principales como *select* nos permiten hacer muchas cosas más.  
Por ejemplo, si usamos *if* sería bajo un criterio como un tipo de dato :

#selección positiva  
ToothGrowth %>% select\_if(is.numeric) %>% glimpse()

## Rows: 60  
## Columns: 2  
## $ len <dbl> 4.2, 11.5, 7.3, 5.8, 6.4, 10.0, 11.2, 11.2, 5.2, 7.0, 16.5, 16.5, 15.2, 17.3, 22.5, 1…  
## $ dose <dbl> 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, …

#selección negativa  
ToothGrowth %>% select\_if(~!is.numeric(.)) %>% glimpse()

## Rows: 60  
## Columns: 1  
## $ supp <fct> VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, V…

Con *all* podemos reformatear los nombres de nuestras columnas:

ToothGrowth %>% select\_all(toupper) %>% glimpse()

## Rows: 60  
## Columns: 3  
## $ LEN <dbl> 4.2, 11.5, 7.3, 5.8, 6.4, 10.0, 11.2, 11.2, 5.2, 7.0, 16.5, 16.5, 15.2, 17.3, 22.5, 1…  
## $ SUPP <fct> VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, V…  
## $ DOSE <dbl> 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, …

Y con *at* también podemos escoger columnas basadas en criterios, por ejemplo:

#npositiva  
ToothGrowth%>% select\_at(vars(contains("ose"))) %>% glimpse()

## Rows: 60  
## Columns: 1  
## $ dose <dbl> 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, …

#negativa  
ToothGrowth%>% select\_at(vars(-contains("ose"))) %>% glimpse()

## Rows: 60  
## Columns: 2  
## $ len <dbl> 4.2, 11.5, 7.3, 5.8, 6.4, 10.0, 11.2, 11.2, 5.2, 7.0, 16.5, 16.5, 15.2, 17.3, 22.5, 1…  
## $ supp <fct> VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, V…

ToothGrowth%>% select\_at(vars(!contains("ose"))) %>% glimpse()

## Rows: 60  
## Columns: 2  
## $ len <dbl> 4.2, 11.5, 7.3, 5.8, 6.4, 10.0, 11.2, 11.2, 5.2, 7.0, 16.5, 16.5, 15.2, 17.3, 22.5, 1…  
## $ supp <fct> VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, V…

### 2.2.6 Filtrando filas

Para filtrar nuestra data a nivel de filas usamos la función *filter*:

head(filter(ToothGrowth, supp=="OJ") )

## len supp dose  
## 1 15.2 OJ 0.5  
## 2 21.5 OJ 0.5  
## 3 17.6 OJ 0.5  
## 4 9.7 OJ 0.5  
## 5 14.5 OJ 0.5  
## 6 10.0 OJ 0.5

o con el pipe:

ToothGrowth %>% filter(supp=="OJ") %>% glimpse()

## Rows: 30  
## Columns: 3  
## $ len <dbl> 15.2, 21.5, 17.6, 9.7, 14.5, 10.0, 8.2, 9.4, 16.5, 9.7, 19.7, 23.3, 23.6, 26.4, 20.0,…  
## $ supp <fct> OJ, OJ, OJ, OJ, OJ, OJ, OJ, OJ, OJ, OJ, OJ, OJ, OJ, OJ, OJ, OJ, OJ, OJ, OJ, OJ, OJ, O…  
## $ dose <dbl> 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, …

ToothGrowth %>% filter(len>30) %>% glimpse()

## Rows: 3  
## Columns: 3  
## $ len <dbl> 33.9, 32.5, 30.9  
## $ supp <fct> VC, VC, OJ  
## $ dose <dbl> 2, 2, 2

ToothGrowth %>% filter(supp=="OJ", len>30) %>% glimpse()

## Rows: 1  
## Columns: 3  
## $ len <dbl> 30.9  
## $ supp <fct> OJ  
## $ dose <dbl> 2

Filtrando basado en un vector:

len\_quiero<-c(26.4, 27.3 ,29.4, 23.0)  
ToothGrowth %>% filter(len %in% len\_quiero) %>% glimpse()

## Rows: 8  
## Columns: 3  
## $ len <dbl> 26.4, 26.4, 27.3, 26.4, 26.4, 27.3, 29.4, 23.0  
## $ supp <fct> VC, OJ, OJ, OJ, OJ, OJ, OJ, OJ  
## $ dose <dbl> 2, 1, 1, 2, 2, 2, 2, 2

ToothGrowth %>% filter(!len %in% len\_quiero) %>% glimpse()

## Rows: 52  
## Columns: 3  
## $ len <dbl> 4.2, 11.5, 7.3, 5.8, 6.4, 10.0, 11.2, 11.2, 5.2, 7.0, 16.5, 16.5, 15.2, 17.3, 22.5, 1…  
## $ supp <fct> VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, V…  
## $ dose <dbl> 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, …

Filtrando NA’s (en caso de que este dataset tuviera NA’s):

ToothGrowth %>% filter(!is.na(len))

En el caso de *if* , *all* y *at* , nos permiten filtrar con condiciones y a través de varias columnas:

ToothGrowth %>% filter\_if(is.numeric, all\_vars(between(.,2,20)))

## len supp dose  
## 1 18.5 VC 2

ToothGrowth %>% filter\_all(any\_vars(. > 30))

## Warning: There was 1 warning in `filter()`.  
## ℹ In argument: `len > 30 | supp > 30 | dose > 30`.  
## Caused by warning in `Ops.factor()`:  
## ! '>' not meaningful for factors

## len supp dose  
## 1 33.9 VC 2  
## 2 32.5 VC 2  
## 3 30.9 OJ 2

ToothGrowth %>% filter\_at(vars(len, dose), all\_vars(.>1)) %>% head()

## len supp dose  
## 1 23.6 VC 2  
## 2 18.5 VC 2  
## 3 33.9 VC 2  
## 4 25.5 VC 2  
## 5 26.4 VC 2  
## 6 32.5 VC 2

### 2.2.7 Creando una nueva columna

Para crear una nueva columna en *dplyr* usamos la función *mutate()*:

ToothGrowth %>% mutate(lenlog = log(len)) %>% head()

## len supp dose lenlog  
## 1 4.2 VC 0.5 1.435085  
## 2 11.5 VC 0.5 2.442347  
## 3 7.3 VC 0.5 1.987874  
## 4 5.8 VC 0.5 1.757858  
## 5 6.4 VC 0.5 1.856298  
## 6 10.0 VC 0.5 2.302585

Al igual que las funciones pasadas también podemos utilizar *if*, *all* y *at*:

ToothGrowth %>% mutate\_if(is.numeric, round) %>% head()

## len supp dose  
## 1 4 VC 0  
## 2 12 VC 0  
## 3 7 VC 0  
## 4 6 VC 0  
## 5 6 VC 0  
## 6 10 VC 0

ToothGrowth %>% mutate\_all(tolower) %>% head()

## len supp dose  
## 1 4.2 vc 0.5  
## 2 11.5 vc 0.5  
## 3 7.3 vc 0.5  
## 4 5.8 vc 0.5  
## 5 6.4 vc 0.5  
## 6 10 vc 0.5

La acción de mutar o la función que se pone después del argumento (como round y tolower), muchas veces se pone sin paréntesis pero otras las requiere. Vemos también que usando el *mutate\_all* cambia todas las columnas (lo que quiere decir que las numéricas las convierte en character). Ahora si quisieramos usarla al revés:

ToothGrowth %>% mutate\_all(round) %>% head()

Error: Problem with mutate() column supp

En estos casos es mejor usar if, como vimos en el ejemplo anterior, *all* aplica mejor si tenemos una data con el mismo tipo de datos (números, caracterres, factores). Ahora bien, *at* nos permite hacer cambios a columnas específicas:

ToothGrowth %>% mutate\_at(vars(contains("ose")), ~(.\*100)) %>% head()

## len supp dose  
## 1 4.2 VC 50  
## 2 11.5 VC 50  
## 3 7.3 VC 50  
## 4 5.8 VC 50  
## 5 6.4 VC 50  
## 6 10.0 VC 50

ToothGrowth %>% mutate\_at("dose", ~(.\*100)) %>% head()

## len supp dose  
## 1 4.2 VC 50  
## 2 11.5 VC 50  
## 3 7.3 VC 50  
## 4 5.8 VC 50  
## 5 6.4 VC 50  
## 6 10.0 VC 50

### 2.2.8 Datos discretos

Existen varias herramientas que nos sirven para trabajar con datos discretos, por ejemplo si queremos cambiar los datos de una columna y modificarlos:

ToothGrowth %>% mutate(supp2 = recode\_factor(supp,  
 "OJ" = "Jugo",  
 "VC" = "Ascórbico",  
 .default = "other",  
 .ordered = TRUE)) %>% tail()

## len supp dose supp2  
## 55 24.8 OJ 2 Jugo  
## 56 30.9 OJ 2 Jugo  
## 57 26.4 OJ 2 Jugo  
## 58 27.3 OJ 2 Jugo  
## 59 29.4 OJ 2 Jugo  
## 60 23.0 OJ 2 Jugo

Otra cosa que podemos hacer es crear una nueva columna con valores discretos usando valores numéricos, por ejemplo:

ToothGrowth%>% mutate(dose2 = ifelse(dose > 1, "alto", "bajo")) %>% head()

## len supp dose dose2  
## 1 4.2 VC 0.5 bajo  
## 2 11.5 VC 0.5 bajo  
## 3 7.3 VC 0.5 bajo  
## 4 5.8 VC 0.5 bajo  
## 5 6.4 VC 0.5 bajo  
## 6 10.0 VC 0.5 bajo

Y si queremos renombrar los datos en una columna, entonces:

ToothGrowth%>%mutate(dose = case\_when(  
 dose == 0.5 ~ "D\_0.5",  
 dose == 1 ~ "D\_1",  
 dose == 2 ~ "D\_2")) %>% mutate(  
 dose = factor(dose, levels = c("D\_2", "D\_1", "D\_0.5"))) %>% head()

## len supp dose  
## 1 4.2 VC D\_0.5  
## 2 11.5 VC D\_0.5  
## 3 7.3 VC D\_0.5  
## 4 5.8 VC D\_0.5  
## 5 6.4 VC D\_0.5  
## 6 10.0 VC D\_0.5

Para separar o unir datos de una columna con data discreta (caracter), podemos usar las funciones *unite()* y *separate()*, por ejemplo:

ToothGrowth %>% unite("interaccion", supp:dose, sep = "\_") %>% head()

## len interaccion  
## 1 4.2 VC\_0.5  
## 2 11.5 VC\_0.5  
## 3 7.3 VC\_0.5  
## 4 5.8 VC\_0.5  
## 5 6.4 VC\_0.5  
## 6 10.0 VC\_0.5

Para ejemplifcar *separate* usaremos el ejemplo anterior:

ToothGrowth%>%mutate(dose = case\_when(  
 dose == 0.5 ~ "D\_0.5",  
 dose == 1 ~ "D\_1",  
 dose == 2 ~ "D\_2")) %>% separate(dose, c("D", "dose"),  
 sep = "\_") %>% head()

## len supp D dose  
## 1 4.2 VC D 0.5  
## 2 11.5 VC D 0.5  
## 3 7.3 VC D 0.5  
## 4 5.8 VC D 0.5  
## 5 6.4 VC D 0.5  
## 6 10.0 VC D 0.5

### 2.2.9 Resumiendo los datos

Hay varias funciones en *tidyverse* que nos permiten hacer un resumen de nuestros datos, como por ejemplo la función *count()*:

ToothGrowth %>% count(supp, sort=TRUE)

## supp n  
## 1 OJ 30  
## 2 VC 30

ToothGrowth %>% count(dose, supp, sort=TRUE)

## dose supp n  
## 1 0.5 OJ 10  
## 2 0.5 VC 10  
## 3 1.0 OJ 10  
## 4 1.0 VC 10  
## 5 2.0 OJ 10  
## 6 2.0 VC 10

Otra forma es usar *group\_by()* que nos permite agrupar nuestros datos bajo alguna condición y luego aplicar una función a estos:

ToothGrowth %>% group\_by(supp) %>%count()

## # A tibble: 2 × 2  
## # Groups: supp [2]  
## supp n  
## <fct> <int>  
## 1 OJ 30  
## 2 VC 30

Con este *group\_by()* podemos aplicar cualquier cantidad de funciones, por ejemplo para en vez que me de el conteo me de el promedio, mediana, cuenta, suma, etc; para esto, debemos ocupar una nueva función muy útil llamada *summarise()*:

ToothGrowth %>% group\_by(supp) %>% summarise(promedio = mean(len), suma = sum(len),  
 n = n(), mediana =median(len))

## # A tibble: 2 × 5  
## supp promedio suma n mediana  
## <fct> <dbl> <dbl> <int> <dbl>  
## 1 OJ 20.7 620. 30 22.7  
## 2 VC 17.0 509. 30 16.5

Esta función también viene en todas las presentaciones, es decir, *at*, *if* y *all*, Ejemplos:

ToothGrowth %>% group\_by(supp) %>% summarise\_if(is.numeric, mean)

## # A tibble: 2 × 3  
## supp len dose  
## <fct> <dbl> <dbl>  
## 1 OJ 20.7 1.17  
## 2 VC 17.0 1.17

ToothGrowth%>% group\_by(supp) %>% summarise\_at(vars(contains("ose")), mean)

## # A tibble: 2 × 2  
## supp dose  
## <fct> <dbl>  
## 1 OJ 1.17  
## 2 VC 1.17

ToothGrowth %>% group\_by(supp) %>% summarise\_all(mean)

## # A tibble: 2 × 3  
## supp len dose  
## <fct> <dbl> <dbl>  
## 1 OJ 20.7 1.17  
## 2 VC 17.0 1.17

### 2.2.10 Renombrando columnas Con *tidyverse*

Existen diferentes formas de renombrar las columnas una es con la función *select()*:

ToothGrowth %>% select(dosis=dose, longitud=len, suplemento=supp) %>% glimpse()

## Rows: 60  
## Columns: 3  
## $ dosis <dbl> 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,…  
## $ longitud <dbl> 4.2, 11.5, 7.3, 5.8, 6.4, 10.0, 11.2, 11.2, 5.2, 7.0, 16.5, 16.5, 15.2, 17.3, 2…  
## $ suplemento <fct> VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC,…

También existe la función rename en todas sus versiones, ejemplos:

ToothGrowth %>% rename(dosis=dose) %>% glimpse()

## Rows: 60  
## Columns: 3  
## $ len <dbl> 4.2, 11.5, 7.3, 5.8, 6.4, 10.0, 11.2, 11.2, 5.2, 7.0, 16.5, 16.5, 15.2, 17.3, 22.5, …  
## $ supp <fct> VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, …  
## $ dosis <dbl> 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,…

ToothGrowth %>% rename\_if(is.numeric, ~paste0("Num\_", .)) %>% glimpse()

## Rows: 60  
## Columns: 3  
## $ Num\_len <dbl> 4.2, 11.5, 7.3, 5.8, 6.4, 10.0, 11.2, 11.2, 5.2, 7.0, 16.5, 16.5, 15.2, 17.3, 22.…  
## $ supp <fct> VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, V…  
## $ Num\_dose <dbl> 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1…

ToothGrowth%>% rename\_at(vars(contains("ose")), ~paste0("Num\_", .)) %>% glimpse()

## Rows: 60  
## Columns: 3  
## $ len <dbl> 4.2, 11.5, 7.3, 5.8, 6.4, 10.0, 11.2, 11.2, 5.2, 7.0, 16.5, 16.5, 15.2, 17.3, 22.…  
## $ supp <fct> VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, V…  
## $ Num\_dose <dbl> 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1…

ToothGrowth%>% rename\_all(toupper) %>% glimpse()

## Rows: 60  
## Columns: 3  
## $ LEN <dbl> 4.2, 11.5, 7.3, 5.8, 6.4, 10.0, 11.2, 11.2, 5.2, 7.0, 16.5, 16.5, 15.2, 17.3, 22.5, 1…  
## $ SUPP <fct> VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, VC, V…  
## $ DOSE <dbl> 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, …

### 2.2.11 Ordenando tablas por un criterio o columna

Conocemos las funciones order y sort, pero para ordenar tablas enteras, la función arrange de **dplyr** es útil. Por ejemplo:

ToothGrowth %>% arrange(len) %>% head()

## len supp dose  
## 1 4.2 VC 0.5  
## 2 5.2 VC 0.5  
## 3 5.8 VC 0.5  
## 4 6.4 VC 0.5  
## 5 7.0 VC 0.5  
## 6 7.3 VC 0.5

ToothGrowth %>% arrange(-len) %>% head()

## len supp dose  
## 1 33.9 VC 2  
## 2 32.5 VC 2  
## 3 30.9 OJ 2  
## 4 29.5 VC 2  
## 5 29.4 OJ 2  
## 6 27.3 OJ 1

También podemos ordenar por varios criterios:

ToothGrowth%>% arrange(dose, supp) %>% head()

## len supp dose  
## 1 15.2 OJ 0.5  
## 2 21.5 OJ 0.5  
## 3 17.6 OJ 0.5  
## 4 9.7 OJ 0.5  
## 5 14.5 OJ 0.5  
## 6 10.0 OJ 0.5

### 2.2.12 Uniendo tablas

Para la unión de tablas usaremos una familia de funciones denominadas **joins**.

Las diferentes presentaciones de esta función nos permite juntar tablas:  
- inner\_join() : incluye todas las filas en x y y (es decir la inteserccion o las que comparten).

* left\_join(): incluye todas las filas en x.
* right\_join(): incuye todas las filas en y.
* full\_join(): incluye todas las filas en x o y (este incluye todos, incluso las que no comparten)

band\_members; band\_instruments

## # A tibble: 3 × 2  
## name band   
## <chr> <chr>   
## 1 Mick Stones   
## 2 John Beatles  
## 3 Paul Beatles

## # A tibble: 3 × 2  
## name plays   
## <chr> <chr>   
## 1 John guitar  
## 2 Paul bass   
## 3 Keith guitar

band\_members %>% full\_join(band\_instruments)

## Joining with `by = join\_by(name)`

## # A tibble: 4 × 3  
## name band plays   
## <chr> <chr> <chr>   
## 1 Mick Stones <NA>   
## 2 John Beatles guitar  
## 3 Paul Beatles bass   
## 4 Keith <NA> guitar

band\_members %>% inner\_join(band\_instruments, by = "name")

## # A tibble: 2 × 3  
## name band plays   
## <chr> <chr> <chr>   
## 1 John Beatles guitar  
## 2 Paul Beatles bass

band\_members %>% left\_join(band\_instruments)

## Joining with `by = join\_by(name)`

## # A tibble: 3 × 3  
## name band plays   
## <chr> <chr> <chr>   
## 1 Mick Stones <NA>   
## 2 John Beatles guitar  
## 3 Paul Beatles bass

band\_members %>% right\_join(band\_instruments)

## Joining with `by = join\_by(name)`

## # A tibble: 3 × 3  
## name band plays   
## <chr> <chr> <chr>   
## 1 John Beatles guitar  
## 2 Paul Beatles bass   
## 3 Keith <NA> guitar

### 2.2.13 Funciones adicionales del *tydiverse*

Anteriormente, usamos la función head para evitar que la página se llene con todo el set de datos. Si queremos ver una mayor proporción, podemos usar la función top\_n. Esta función toma un *data frame* como primer argumento, el número de filas para mostrar en el segundo y la variable para filtrar en el tercero. Aquí hay un ejemplo de cómo ver las 5 filas superiores:

ToothGrowth %>% top\_n(5, len)

## len supp dose  
## 1 33.9 VC 2  
## 2 32.5 VC 2  
## 3 29.5 VC 2  
## 4 30.9 OJ 2  
## 5 29.4 OJ 2

También hay otras funciones que son útiles en los diferentes análisis donde a veces ocupamos los rownames o a veces no, estas funciones nos permiten hacer una columna que sea rownames y viceversa, por ejemplo:

ToothGrowth %>% rownames\_to\_column(var = "row") %>% head()

## row len supp dose  
## 1 1 4.2 VC 0.5  
## 2 2 11.5 VC 0.5  
## 3 3 7.3 VC 0.5  
## 4 4 5.8 VC 0.5  
## 5 5 6.4 VC 0.5  
## 6 6 10.0 VC 0.5

ToothGrowth %>% rownames\_to\_column(  
 var = "row") %>% column\_to\_rownames(var = "row") %>% head()

## len supp dose  
## 1 4.2 VC 0.5  
## 2 11.5 VC 0.5  
## 3 7.3 VC 0.5  
## 4 5.8 VC 0.5  
## 5 6.4 VC 0.5  
## 6 10.0 VC 0.5

Si queremos convertir las tablas como al principio del capítulo, es decir, convertir una tabla que no está en formato *tidy* a una que sí esté y viceversa podemos usar las funciones **pivot\_longer()** y **pivot\_wider()** (antes denominadas **gather()** y **spread()**, pero ya están reemplazadas aunque aún no eliminadas).

data(iris)  
data\_iris<- iris %>% select(Sepal.Length, Sepal.Width) %>% rownames\_to\_column(var = "ids")  
head(data\_iris)

## ids Sepal.Length Sepal.Width  
## 1 1 5.1 3.5  
## 2 2 4.9 3.0  
## 3 3 4.7 3.2  
## 4 4 4.6 3.1  
## 5 5 5.0 3.6  
## 6 6 5.4 3.9

Vamos a cambiarla a formato tidy:

tidy\_iris<-pivot\_longer(names\_to = "variable",   
 values\_to = "longitud",   
 data = data\_iris,   
 cols = Sepal.Length:Sepal.Width)   
head(tidy\_iris)

## # A tibble: 6 × 3  
## ids variable longitud  
## <chr> <chr> <dbl>  
## 1 1 Sepal.Length 5.1  
## 2 1 Sepal.Width 3.5  
## 3 2 Sepal.Length 4.9  
## 4 2 Sepal.Width 3   
## 5 3 Sepal.Length 4.7  
## 6 3 Sepal.Width 3.2

Y si queremos regresar a como lo teníamos:

notidy\_iris<- tidy\_iris %>% pivot\_wider(names\_from = variable,  
 values\_from = longitud)  
head(notidy\_iris)

## # A tibble: 6 × 3  
## ids Sepal.Length Sepal.Width  
## <chr> <dbl> <dbl>  
## 1 1 5.1 3.5  
## 2 2 4.9 3   
## 3 3 4.7 3.2  
## 4 4 4.6 3.1  
## 5 5 5 3.6  
## 6 6 5.4 3.9

### 2.2.14 Ejemplo Aplicado

primera<- data.frame(Nombre= c("Astrid", "Lea", "Sarina", "Remon", "Letizia",   
 "Babice", "Jonas", "Wendy", "Nivedithia", "Gioia"),  
 Sexo= c("F", "F", "F", "M", "F", "F", "M", "F", "F", "F"),  
 Edad= c(30,25,25,29,22,22,35,19,32,21))  
segunda<- data.frame(Nombre= c("Astrid", "Lea", "Sarina", "Remon", "Letizia",   
 "Babice", "Jonas", "Wendy", "Nivedithia", "Gioia"),  
 Superhéroe= c("Batman", "Superman", "Batman", "Spiderman", "Batman",   
 "Antman", "Batman", "Superman", "Maggot", "Superman"),  
 Tatuajes= c(11,15,12,5,65,3,9,13,900,0))  
knitr::kable(primera); knitr::kable(segunda)

| Nombre | Sexo | Edad |
| --- | --- | --- |
| Astrid | F | 30 |
| Lea | F | 25 |
| Sarina | F | 25 |
| Remon | M | 29 |
| Letizia | F | 22 |
| Babice | F | 22 |
| Jonas | M | 35 |
| Wendy | F | 19 |
| Nivedithia | F | 32 |
| Gioia | F | 21 |

| Nombre | Superhéroe | Tatuajes |
| --- | --- | --- |
| Astrid | Batman | 11 |
| Lea | Superman | 15 |
| Sarina | Batman | 12 |
| Remon | Spiderman | 5 |
| Letizia | Batman | 65 |
| Babice | Antman | 3 |
| Jonas | Batman | 9 |
| Wendy | Superman | 13 |
| Nivedithia | Maggot | 900 |
| Gioia | Superman | 0 |

Para hacer:

1. Combina las dos tablas en una sola y completa las siguientes asignaciones.
2. ¿Cuál es la edad media de las mujeres y hombres por separado?
3. ¿Cuál fue el número más alto de tatuajes en un hombre?
4. ¿Cuál es el porcentaje de personas debajo de 32 años que son mujeres?
5. Agrega una nueva columna a a la data llamada tatuajes.por.año que muestre cuántos tatuajes por año se ha hecho cada persona por cada año en su vida.
6. ¿Cuál persona tiene el mayor número de tatuajes por año?
7. ¿Cuáles son los nombres de las mujeres a las que su superheroe favorito es superman?
8. ¿Cuál es la mediana del número de tatuajes de cad apersona que está por encima de los 20 años y que su personaje favorito es Batman?

#### 2.2.14.1 Resolviendo

1. Combina las dos tablas en una sola y completa las siguientes asignaciones.

encuestas<-primera %>% full\_join(segunda)

## Joining with `by = join\_by(Nombre)`

2. ¿Cuál es la edad media de las mujeres y hombres por separado?

encuestas %>% group\_by(Sexo) %>% summarise\_at(c("Edad"), mean)

## # A tibble: 2 × 2  
## Sexo Edad  
## <chr> <dbl>  
## 1 F 24.5  
## 2 M 32

3. ¿Cuál fue el número más alto de tatuajes en un hombre?

encuestas %>% filter(Sexo=="M") %>% filter(Tatuajes == max(Tatuajes))

## Nombre Sexo Edad Superhéroe Tatuajes  
## 1 Jonas M 35 Batman 9

4. ¿Cuál es el porcentaje de mujeres debajo de 32 años?

fem<- encuestas %>% filter(Sexo=="F")  
fem\_32<- encuestas %>% filter( Sexo=="F", Edad<32)  
(nrow(fem\_32)/nrow(fem))\*100

## [1] 87.5

5. Agrega una nueva columna a a la data llamada tatuajes.por.año que muestre cuántos tatuajes por año se ha hecho cada persona por cada año en su vida.

encuestas<- encuestas %>% mutate("tatuajesporaño"=Tatuajes/Edad)  
encuestas$tatuajesporaño

## [1] 0.3666667 0.6000000 0.4800000 0.1724138 2.9545455 0.1363636 0.2571429 0.6842105  
## [9] 28.1250000 0.0000000

6. ¿Cuál persona tiene el mayor número de tatuajes por año?

encuestas %>% filter(tatuajesporaño == max(tatuajesporaño))

## Nombre Sexo Edad Superhéroe Tatuajes tatuajesporaño  
## 1 Nivedithia F 32 Maggot 900 28.125

7. ¿Cuáles son los nombres de las mujeres a las que su superheroe favorito es superman?

encuestas %>% filter(Sexo=="F", Superhéroe=="Superman") %>% select(Nombre)

## Nombre  
## 1 Lea  
## 2 Wendy  
## 3 Gioia

8. ¿Cuál es la mediana del número de tatuajes de cada persona que está por encima de los 20 años y que su personaje favorito es batman?

encuestas %>% filter(Edad>20, Superhéroe =="Batman") %>% summarise(mediana=median(Tatuajes))

## mediana  
## 1 11.5

# 3 Gráficos en R

## 3.1 Gráficas en R base

R cuenta con un sistema de generación de gráficas poderoso y flexible. En este capítulo revisaremos como crear las gráficas más comunes con R base y luego un poco más complejas usando *ggplot2* del *tidyverse*.

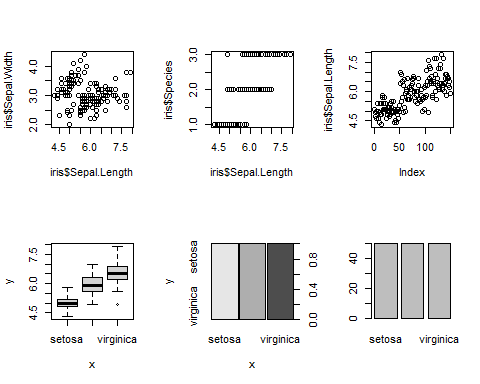
### 3.1.1 La función *plot()*

La función *plot()* es usada de manera general para generar gráficos. Esta función es muy especial porque depende del tipo de datos que le demos generará diferentes tipos de gráficos. El argumento principal que pide esta función es *“x”* también podemos poner *“y”*. Y depende de estos el tipo de gráfica que se generará. Diremos:

* **Continuo**: Cuando nos referimos a un vector numérico, entero, lógico o complejo.
* **Discreto**: Cuando nos referimos a un vector de factores o cadenas de texto.

| x | y | Tipo Gráfico |
| --- | --- | --- |
| Continuo | Continuo | Dispersión /Scatter |
| Continuo | Discreto | Dispersión y coercionada a numérica |
| Continuo | Ninguno | Dispersión por número de renglón |
| Discreto | Continuo | Boxplot/Cajas |
| Discreto | Discreto | Mosaico |
| Discreto | Ninguno | Barras |

#Ejemplos  
par(mfrow= c(2,3) )  
plot(x=iris$Sepal.Length, y=iris$Sepal.Width)  
plot(x=iris$Sepal.Length, y = iris$Species)  
plot(x=iris$Sepal.Length)  
plot(x = iris$Species, y = iris$Sepal.Length)  
plot(x=iris$Species, y=iris$Species)  
plot(x=iris$Species)

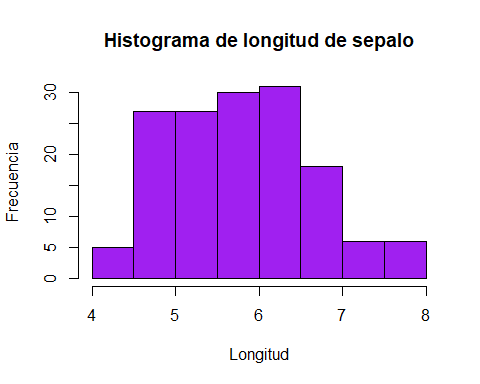


Además de estas, también hay otras funciones de Rbase que nos permiten graficar tipos específicos como son *barplot()*, *boxplot()* o *hist()*.

### 3.1.2 Histrogramas

Los histogramas ya los vimos en el capítulo de estadísticos. Sabemos que nos permiten ver la distribución de nuestros datos (si son normales o no). Sin embargo, podemos usar más argmentos para darle formato, veamoslo con un ejemplo:

hist(x = iris$Sepal.Length, main = "Histograma de longitud de sepalo",   
 xlab = "Longitud", ylab = "Frecuencia",  
 col = "purple")



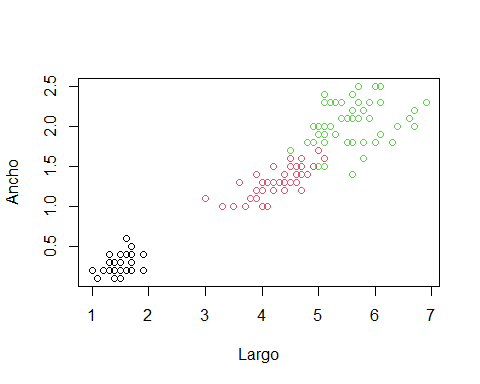
Como vemos en este ejemplo hay argumentos que aplicamos que puede ser aplicados para todos los gráficos de Rbase, tales como:

* *main* : Titulo de la gráfica
* *xlab* y *ylab*: Títulos de los axis x y y
* *col* : color de las barras o gráfica.

### 3.1.3 Diagramas de dispersión

Estos diagramas nos son útiles para ver las relaciones que hay entre dos variables continuas. En el siguiente ejemplo veremos la relación entre las variables longitud y ancho del pétalo, pero en este caso los colorearemos por especies:

plot(x = iris$Petal.Length, y = iris$Petal.Width, col = iris$Species, xlab = "Largo", ylab = "Ancho")



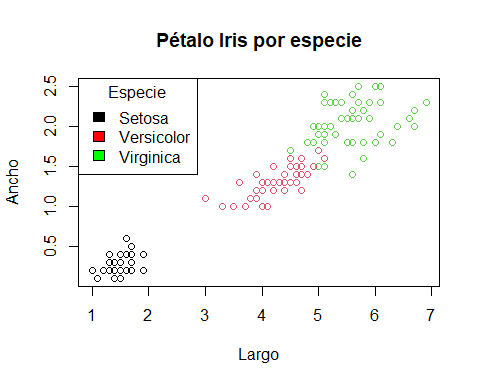
Si queremos agregar una leyenda a alguna de nuestras figuras de Rbase, usamos la función *legend()*:

legend() siempre nos pide siempre los siguientes argumentos.

* legend: Las etiquetas de los datos que queremos describir con la leyenda. Por ejemplo, si tenemos cuatro categorías a describir, proporcionamos un vector de cuatro cadenas de texto.
* fill: Los colores que acompañan a las etiquetas definidas con legend. Estos colores tienen que coincidir con los que hemos usado en el gráfico.
* x y y: Las coordenadas en pixeles, en las que estará ubicada la leyenda. Podemos dar como argumento a x alguno de los siguientes, para ubicar automáticamente la leyenda: “bottomright”, “bottom”, “bottomleft”, “left”, “topleft”, “top”, “topright”, “right”, “center”.
* title: Para poner título a la leyenda.

Veámoslo con el mismo ejemplo, primero ponemos *plot()* con la gráfica que queremos y debajo *legend()* para ponerlo encima de la figura que ya habíamos hecho:

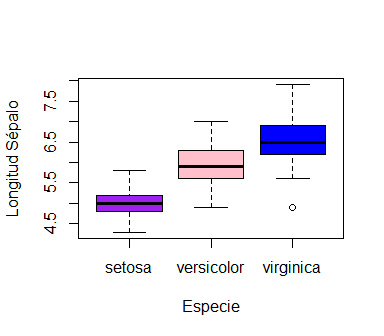
plot(x = iris$Petal.Length, y = iris$Petal.Width, col = iris$Species,   
 main = "Pétalo Iris por especie", xlab = "Largo", ylab = "Ancho")  
legend(x = "topleft", legend = c("Setosa", "Versicolor", "Virginica"),   
 fill = c("black", "red", "green"), title = "Especie")



### 3.1.4 Boxplots o diagramas de cajas

Los diagramas de cajas o boxplots son gráficos que nos muestran la distribución de una variable mostrando sus cuartiles (parte baja primer cuartil y parte alta tercer cuartil), de manera que podamos ver su distribución y simetría. La línea horizontal media representa la “mediana” y las dos líneas verticales que muestran el valor máximo y mínimo. Veamos un ejemplo:

plot(x=iris$Species, y = iris$Sepal.Length, xlab = "Especie", ylab = "Longitud Sépalo",   
 col = c("purple", "pink", "blue"))

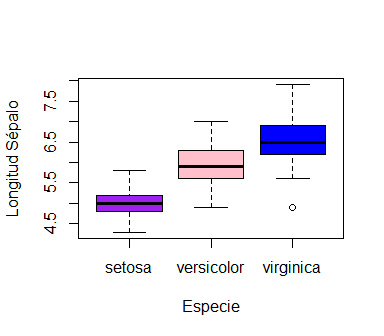


También podemos usar la función *boxplot()*. En esta segunda forma de hacer diagramas de cajas necesitamos declarar dos argumentos principales:

* formula: Para esta función las fórmulas tienen el formato y ~ x, donde x es el nombre de la variable continua a graficar, y la x es la variable que usaremos como agrupación.
* data: Es el data frame del que serántomadas las variables.

Además declarar los demás argumentos extras como colores, títulos y demás.

boxplot(formula = Sepal.Length ~ Species, data = iris, xlab = "Especie",   
 ylab = "Longitud Sépalo", col = c("purple", "pink", "blue"))



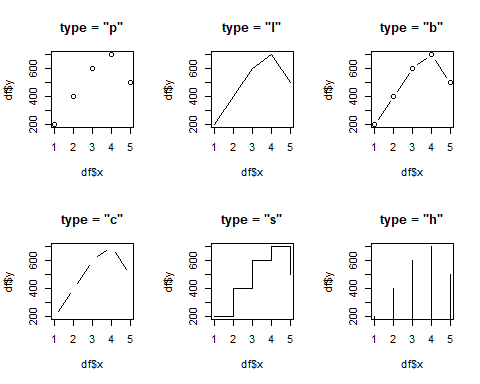
### 3.1.5 Otros gráficos

Usando la función *plot()* podemos dibujar más gráficos de los vistos que son los estándares, como fue el caso cuando graficas modelos lineales. También existe otro argumento de esta función que se denomina *type* y nos permite escoger otros tipos de gráficos. Esto lo podemos ver usando el ‘?’ para ver la ayuda y los argumentos.

?plot

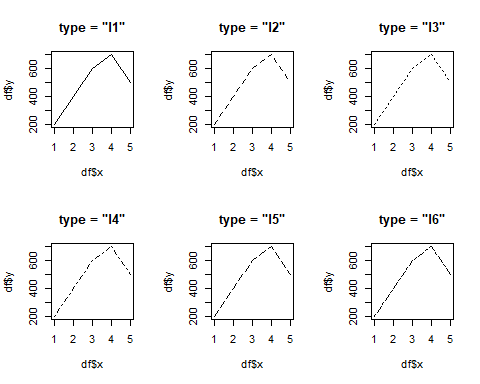
Son posibles los siguientes valores: “p” para puntos, “l” para líneas, “b” para puntos y líneas, “c” para puntos vacíos unidos por líneas, “o” para puntos sobretrazados y líneas, “s” y “S” para escalones y “h” para líneas verticales similares a histogramas. Finalmente, “n” no produce ningún punto o línea. Veamos algunos ejemplos:

#dataset de ensayo  
df<- data.frame(x= c(1:5),  
 y= c(200, 400, 600, 700, 500))  
par(mfrow = c(2, 3))  
plot(df$x, df$y, type = "p", main = 'type = "p"')  
plot(df$x, df$y, type = "l", main = 'type = "l"')  
plot(df$x, df$y, type = "b", main = 'type = "b"')  
plot(df$x, df$y, type = "c", main = 'type = "c"')  
plot(df$x, df$y, type = "s", main = 'type = "s"')  
plot(df$x, df$y, type = "h", main = 'type = "h"')



Ahora bien, en cuanto a líneas podemos escoger el tipo de línea con el argumento *lty*:

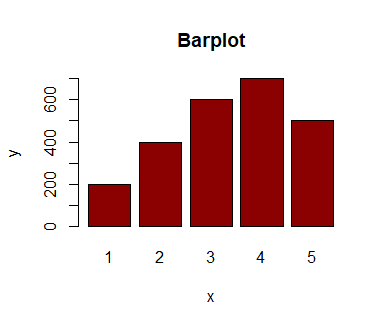
par(mfrow = c(2, 3))  
plot(df$x, df$y, type = "l", lty=1, main = 'type = "l1"')  
plot(df$x, df$y, type = "l", lty=2, main = 'type = "l2"')  
plot(df$x, df$y, type = "l", lty=3, main = 'type = "l3"')  
plot(df$x, df$y, type = "l", lty=4, main = 'type = "l4"')  
plot(df$x, df$y, type = "l", lty=5, main = 'type = "l5"')  
plot(df$x, df$y, type = "l", lty=6, main = 'type = "l6"')



### 3.1.6 Barplots o gráficas de barras

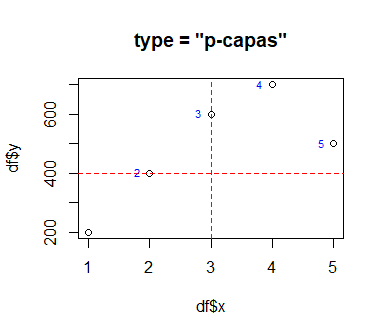
Las gráficas de barras nos permiten ver los valores de una manera diferente a las líneas:

barplot(y ~ x , data = df, main = "Barplot", col = "darkred")



También hay capas que podemos agregarle a las gráficas, como la capa *text* o la capa *abline* para agregar texto y líneas sobre la gráfica, por ejemplo:

plot(df$x, df$y, type = "p", main = 'type = "p-capas"')  
abline(h=400, v=3, col="red", lty=2)  
text(df, labels=rownames(df), cex=0.7, pos=2, col="blue")



### 3.1.7 Guardando una gráfica

Para guardar o exportar una gráfica debemos:

1. Indicar las instrucciones de cómo exportaremos la imagen
2. Usar *plot()* y graficarla y luego,
3. *dev.off*() para quitarla del panel y exportarla

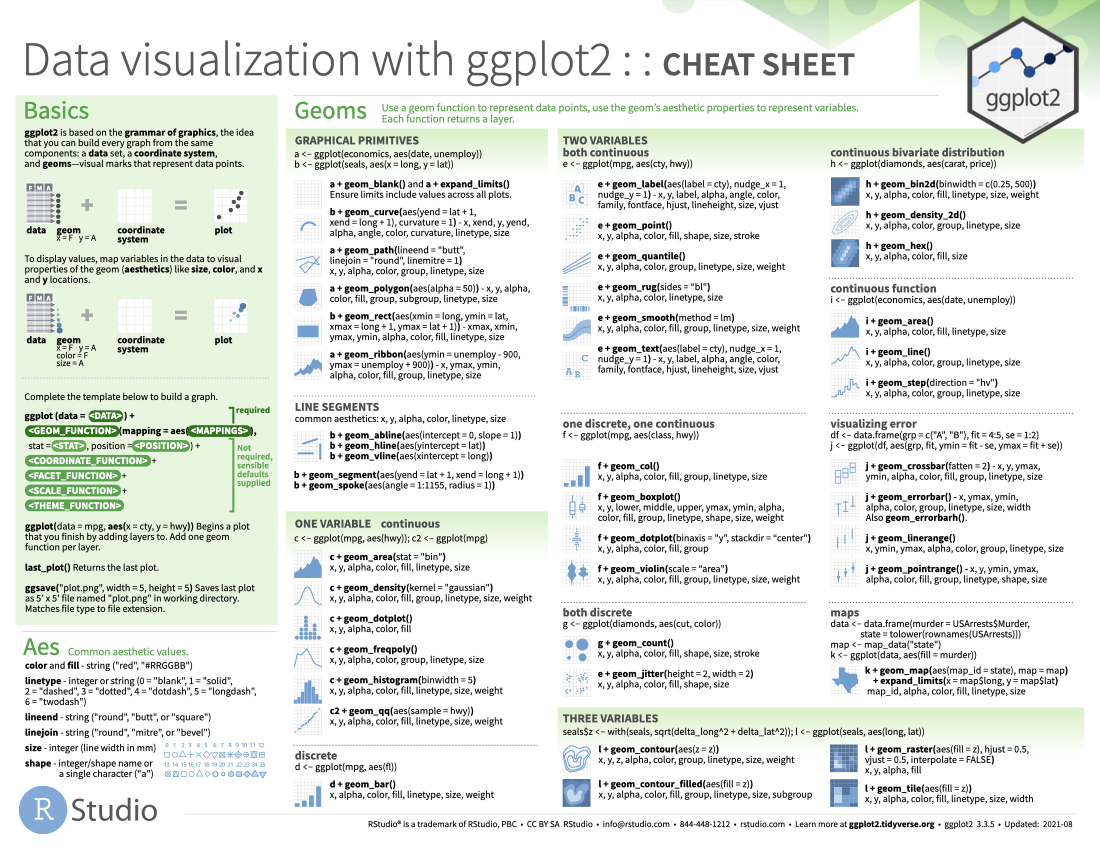
png(filename="gráfica1.png", width=648, height=432)  
plot(df$x, df$y, type = "p", main = 'type = "p"')  
dev.off()

## 3.2 Gráficas con *ggplot2()*

Las gráficas con *ggplot2()* es quizás de las cosas más poderosas y atractivas de R si lo comparamos con otros lenguajes y/o programas. *ggplot2* es generalmente más intuitiva porque usa una gramática de gráficos además de ser visualmente agradable.

La limitación de ggplot2 es que está diseñado para trabajar exclusivamente con tablas de datos en formato tidy (donde las filas son observaciones y las columnas son variables). Sin embargo,ya vimos en anteriores temas cómo podemos convertir nuestras tablas para que tengan este formato.

Para usar ggplot2, tendrán que aprender varias funciones y argumentos. Estos son difíciles de memorizar, por lo que les recomendamos que tengan a mano la hoja de referencia de ggplot2. Pueden obtener una copia en línea o simplemente realizar una búsqueda en internet de “ggplot2 cheat sheet.” y econtraran algo como esto:



O también en este link: [***cheaseet***](https://www.rstudio.com/resources/cheatsheets/), donde podrán ver además estas hojas con indicaciones de otros paquetes de R que nos facilitarán un poco más las cosas.

Es fácil de usar, pero puedes crear figuras complejas con una sintaxis bastante simple.

“gg” significa gramática gráfica - Significa que se superponen diferentes capas de objetos y elementos sobre los anteriores para generar la figura.

En esta parte se repasará las diferentes características de ggplot2:

1. geomas o “geoms”
2. escalas o “scales”
3. guías o “guides”
4. temas o “theme”
5. facetas o “facets”

El primer paso para aprender **ggplot2** es poder separar un gráfico en componentes. Las figuras aquí, se construyen por capas. Los tres componentes principales para considerar son:

* **Data**: nuestro set de datos o *dataframe*
* **Mapeo estético o ‘aes’**: El gráfico usa varias señales visuales para representar la información proveída por el set de datos. Las dos señales más importantes en este gráfico son las posiciones de los puntos en el eje-x y el eje-y.
* **Geometría**: Nos indica el tipo de gráfica a realizar. Unas posibles geometrías son diagrama de barras, dispersión, histograma, densidades suaves (*smooth densities* en inglés), gráfico Q-Q y diagrama de cajas.
* **Elementos adicionales**: temas, guías, colores, etc.

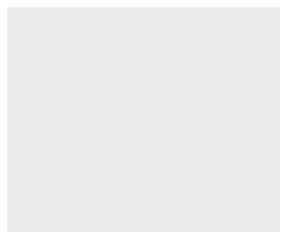
Para esta parte usaremos como ejemplo el set de Datos de “ToothGrotwth” de nuevo y contruiremos por partes la gráfica.

### 3.2.1 ggplot

El primer paso para crear un gráfico **ggplot2** es definir un objeto ggplot. Hacemos esto con la función ggplot, que inicializa el gráfico.

Esta parte podemos hacerla evaluando dentro de la función o usando el pipe, así:

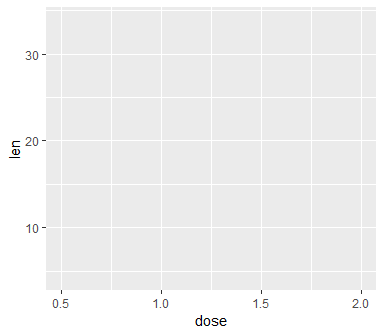
data("ToothGrowth")  
library(dplyr)  
library(ggplot2)  
ggplot(data = ToothGrowth)  
ToothGrowth %>% ggplot()



El código crea un gráfico, en este caso una pizarra en blanco ya que no se ha definido la geometría. La única opción de estilo que vemos es un fondo gris.

También dentro de esta capa de *ggplot()* se declara algo que se conoce como ***mapeo estético*** que describe cómo las propiedades de los datos se conectan con las características del gráfico. En otras palabras, más coloquiales, qué con qué graficamos, también podemos colocar aquí si queremos colorear o llenar un factor. Por ejemplo, con la data de ToothGrowth, queremos graficar el tamaño o longitud de los dientes versus la dosis aplicada y destacando o coloreando el método de aplicación de la vitamina C.

ToothGrowth %>% ggplot(aes(x = dose, y = len, fill=supp))

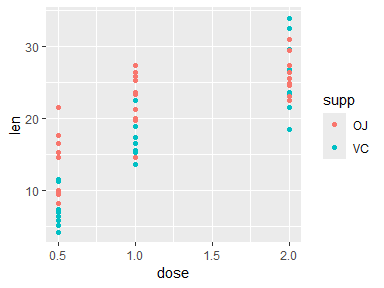


Aquí ya vemos que el eje x y el eje y ya aparecen con nombres de las variables, pero aún no nos muestra ningún gráfico, esto es porque no le indicamos aún que geometría o qué tipo de gráfico queremos.

### 3.2.2 Geometrías

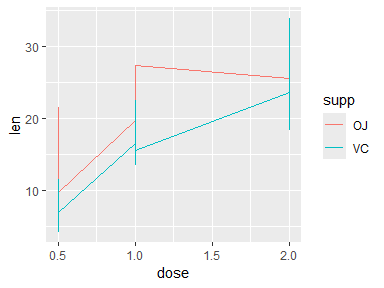
Esta es la siguiente capa en *ggplot2*. Los nombres de las funciones de geometría siguen el patrón: geom\_$ donde $ es el nombre de la geometría. Algunos ejemplos incluyen geom\_bar, geom\_point y geom\_histogram.  
Digamos que queremos ahora sí dibujar la gráfica del paso anterior pero usando puntos, para esto usamos la geometría geom\_point . Luego de cada capa de *ggplot2* en vez de usar pipe usamos el signo **+** y así vamos agregando cada capa.

ToothGrowth %>% ggplot(aes(x = dose, y = len, color=supp)) +  
 geom\_point()



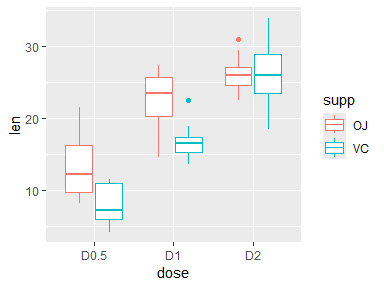
Ahora sí podemos ver lo que queríamos, cómo lucen los datos de longitud en cada dosis aplicada destacando o coloreando el modo de administración. Si en vez de puntos quisieramos, líneas, pues sólo cambiamos la capa de geometría:

ToothGrowth %>% ggplot(aes(x = dose, y = len, color=supp)) +  
 geom\_line()



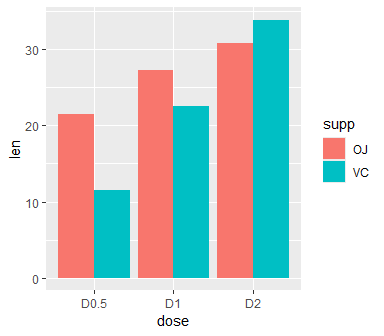
También podemos combinar lo visto en *tidyverse* con *ggplot2* por ejemplo si queremos hacer un boxplot o barplot debemos cambiar la variable de dosis para hacerla discreta en vez de numérica o continua.

ToothGrowth %>% mutate(dose=case\_when(  
 dose==0.5~"D0.5",dose==1~"D1", dose==2~"D2")) %>% ggplot(  
 aes(x = dose, y = len, color=supp)) + geom\_boxplot()



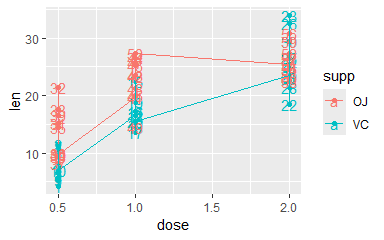
En el caso de *geom\_bar* debemos cambiar y poner más argumentos, por ejemplo, como las barras son huecas en vez de usar ‘color’ usamos ‘fill’ y dentro de la geomtría escogemos una stat y una position , esto es para indicar cómo estarán las barras ubicadas y representadas. Los argumentos más usados para barras tradicionales los presento a continuación:

ToothGrowth %>% mutate(dose=case\_when(  
 dose==0.5~"D0.5",dose==1~"D1", dose==2~"D2")) %>% ggplot(  
 aes( x = dose, y = len, fill=supp)) + geom\_bar(stat = "identity", position = "dodge")



Podemos también agregar más de una capa de geometría y así hacer gráficas combinadas, por ejemplo:

ToothGrowth %>% ggplot(aes(x = dose, y = len, color=supp)) +  
 geom\_point()+  
 geom\_text(aes(label=rownames(ToothGrowth)))+  
 geom\_line()



En la gráfica pasada combinamos tres geometrías, puntos, texto y línea. De esta forma podemos hacer gráficas más elaboradas y mejor representadas.

### 3.2.3 Colores, títulos, escalas y otros ajustes.

Hasta ahora hemos construido gráficas con *ggplot2()* declarando el mapa estético y las diferentes geometrías. Sin embargo podemos notar de todas nuestras figuras anteriores que lucen muy parecidas en formato, mismos colores, mismo fondo y demás.

Todos estos parámetros podemos modificarlos para personalizar nuestras figuras.

#### 3.2.3.1 Colores

*ggplot2()* tiene su escala personalizadas de colores, como lo vimos anteriormente. Sin embargo esto podemos cambiarlo usando una capa denominada scale\_color\_$, tomando $ varias formas, como scale\_color\_manual(), scale\_color\_continuous(),scale\_color\_discrete(), scale\_color\_brewer() , entre otros.

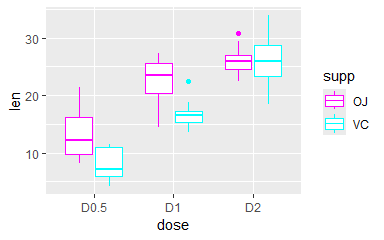
Todas estas opciones son depende de nuestro tipo de datos o del tipo de escala que queremos utilizar. Para declarar los colores en R podemos utilizar los nombres de los colores inglés como lo hicimos en la parte de R base o usando el formato de colores de html. También hay muchas paletas de varios paquetes ya definidas como la de RColorBrewer y viridis.

Para buscar los colores en formato html pomos usar el siguiente link: [***html-colors***](https://htmlcolorcodes.com/es/)***.*** El código html se ve de esta manera:



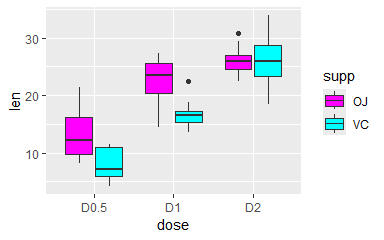
Y de esta manera los declaramos en nuestra gráfica usando scale\_color\_manual():

ToothGrowth %>% mutate(dose=case\_when(  
 dose==0.5~"D0.5",dose==1~"D1", dose==2~"D2")) %>% ggplot(  
 aes(x = dose, y = len, col=supp)) + geom\_boxplot()+  
 scale\_color\_manual(values = c("#FF00FF","#00FFFF"))

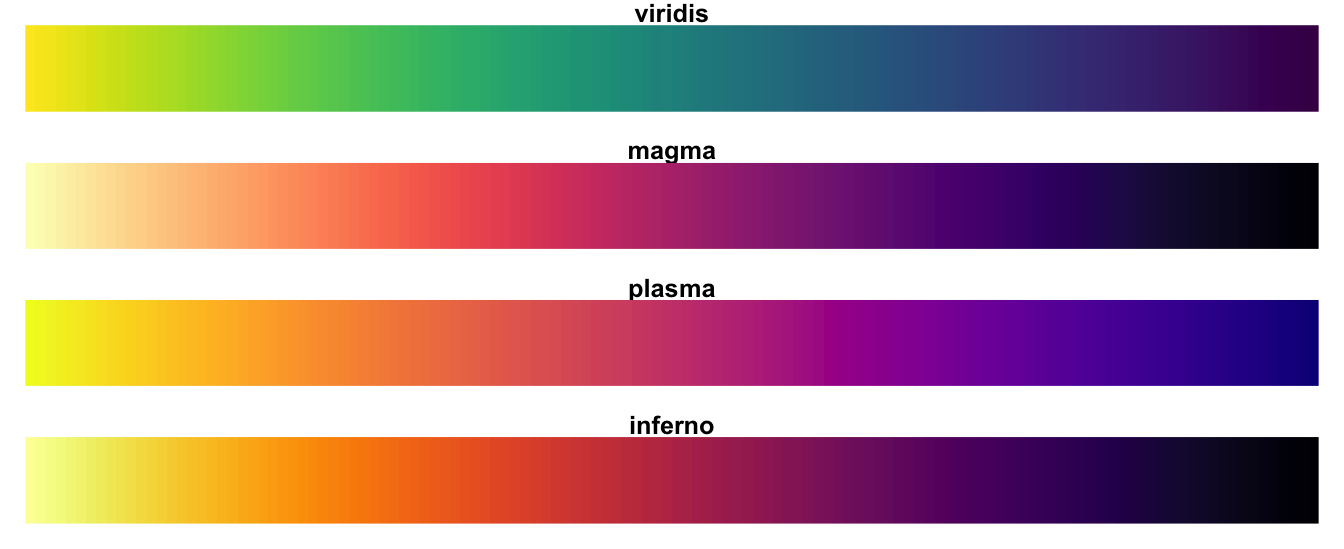
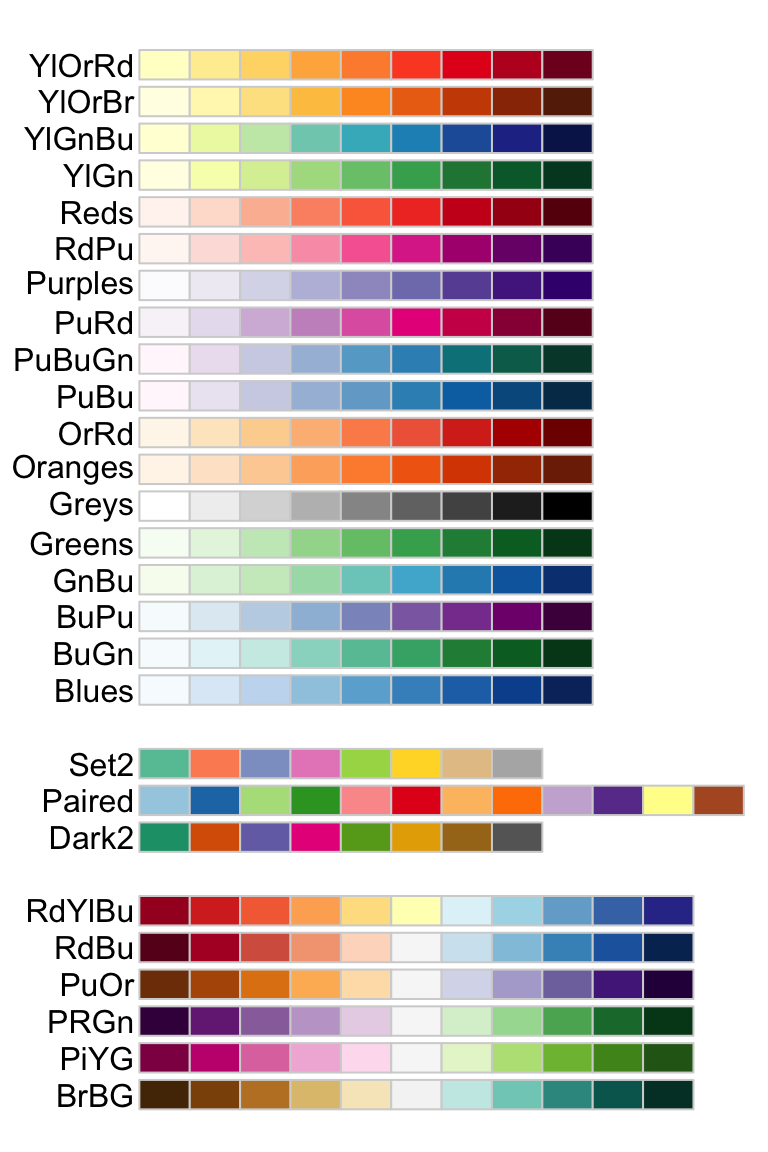


Ahora bien, como les mencioné anteriormente los diagramas de cajas y barras son elementos huecos entonces cuando indicados *col* lo que realmente coloreamos son los bordes, pero si queremos colorear las cajas adentro debemos usar *fill* como usamos en el *geom\_bar()*, pero, en este caso en vez de usar scale\_color\_$, usamos scale\_fill\_$ y de esta manera podemos reemplazarlo por todas las opciones anteriormente mencionadas:

ToothGrowth %>% mutate(dose=case\_when(  
 dose==0.5~"D0.5",dose==1~"D1", dose==2~"D2")) %>% ggplot(  
 aes(x = dose, y = len, fill=supp)) + geom\_boxplot()+  
 scale\_fill\_manual(values = c("#FF00FF","#00FFFF"))

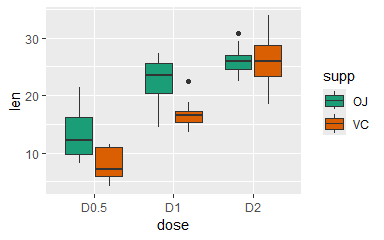


Ahora, si dudamos mucho o no estamos seguros de cómo escoger los colores, podemos utilizar paletas de colores ya determinadas como las que mencioné RColorBrewer y viridis. Estas son las paletas de RColorBrewer y viridis:

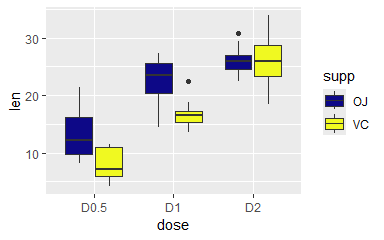


Y así las podemos usar:

ToothGrowth %>% mutate(dose=case\_when(  
 dose==0.5~"D0.5",dose==1~"D1", dose==2~"D2")) %>% ggplot(  
 aes(x = dose, y = len, fill=supp)) + geom\_boxplot()+  
 scale\_fill\_brewer(palette = "Dark2")



ToothGrowth %>% mutate(dose=case\_when(  
 dose==0.5~"D0.5",dose==1~"D1", dose==2~"D2")) %>% ggplot(  
 aes(x = dose, y = len, fill=supp)) + geom\_boxplot()+  
 scale\_fill\_viridis\_d(option = "C")

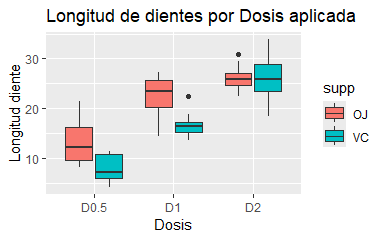


Entonces para *brewer* ponemos el nombre de la paleta que queremos uscar y el *viridis* ponemos la opción de paleta que queremos: “magma” (o “A”), “inferno” (“B”), “plasma” (“C”), “viridis” (“D”, default) and “cividis” (o “E”).

#### 3.2.3.2 Títulos

Para asignar los títulos al eje x y al eje y podemos usar las capas xlab() y ylab(). Y para el título principal usamos ggtitle() . Así:

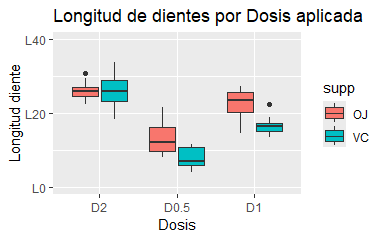
ToothGrowth %>% mutate(dose=case\_when(  
 dose==0.5~"D0.5",dose==1~"D1", dose==2~"D2")) %>% ggplot(  
 aes(x = dose, y = len, fill=supp)) + geom\_boxplot()+  
 ylab("Longitud diente")+  
 xlab("Dosis")+  
 ggtitle("Longitud de dientes por Dosis aplicada")



### 3.2.4 Escalas

Para establecer o cambiar las escalas de nuestra figura usamos las funciones scale\_x\_discrete() o scale\_y\_continuos()dependiendo del tipo de datos que tengamos (cotinuos o numéricos y discretos o categóricos) en cada axis (x o y) y.

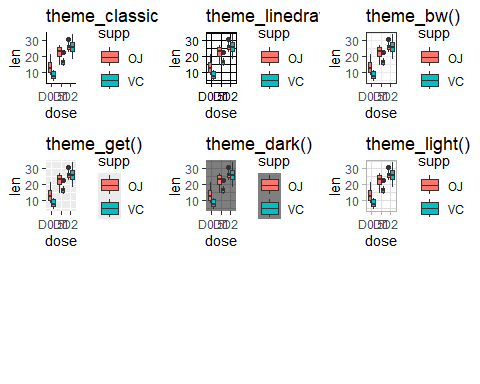
ToothGrowth %>% mutate(dose=case\_when(  
 dose==0.5~"D0.5",dose==1~"D1", dose==2~"D2")) %>% ggplot(  
 aes(x = dose, y = len, fill=supp)) + geom\_boxplot()+  
 ylab("Longitud diente")+  
 xlab("Dosis")+  
 ggtitle("Longitud de dientes por Dosis aplicada") +  
 scale\_x\_discrete(limits = c("D2", "D0.5", "D1"), position ="bottom" )+  
 scale\_y\_continuous(breaks = c(0, 20,40), limits = c(0,40), labels = c("L0", "L20", "L40"))



Como vemos, podemos explorar y probar diferentes parámetros para cambiar en el eje x y en el eje y sin tener que modificar nuestros datos sino solamente para ser graficado.

### 3.2.5 Otros ajustes

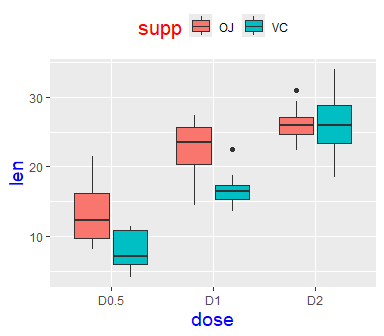
Hay otros ajustes más geneales o específicos que podemos realizar a nuestros gráficos. Por ejemplo, los temas. *ggplot2()* usa como *default* el tema theme\_grey() pero hay otros que podemos usar para cambiar el formarto de nuestras figuras. Veamos algunos ejemplos:



Y así muchos otros más, los podemos explorar dando theme\_ y usando la tecla TAB para ver las demás opciones que nos aparecen.

Otros ajustes más específicos como tamaño y color de letras, posición de la leyenda, entre otros, los hacemos usando la capa denominada theme(). Vamos a dar un ejemplo de cómo hacer esto, pero para usar **thene()** es todo un capítulo muy extenso para entrar en detalles, si queremos conocer muchos de estos detalles hacemos **?theme()** y nos depliega la página de ayuda y veremos los argumentos posibles.

ToothGrowth %>% mutate(dose=case\_when(  
 dose==0.5~"D0.5",dose==1~"D1", dose==2~"D2")) %>% ggplot(  
 aes(x = dose, y = len, fill=supp)) + geom\_boxplot()+   
 theme(axis.title = element\_text(size = "14", colour = "blue"),  
 title = element\_text(size = 16, colour = "red"),  
 legend.position = "top")



## 3.3 Figuras Multi-panel / Facets

Para dibujar varias figuras en una misma página o output hay varias estrategias. Varias de ellas están fuera del alcance de este curso, pero para mencionarlas:

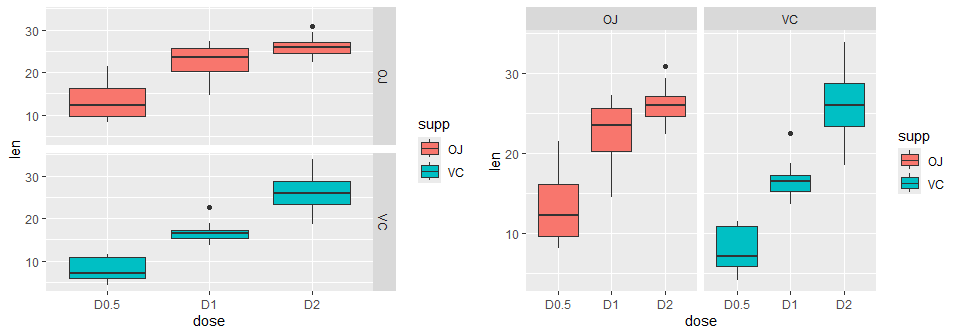
* par (multi-panel con R base)[[7]](#footnote-267)
* cowplot (ggplot2)[[8]](#footnote-268)
* grid.arrange(ggplot2)[[9]](#footnote-269)
* Facets

### 3.3.1 Facets

Esta forma de graficar en multi-panel es quizás la más sencilla por ser una capa de *ggplot2()*. Para esta hay dos funciones facet\_grid()y facet\_wrap(). Los trataremos como ejemplos porque en detalle no son del alcance del presente curso.

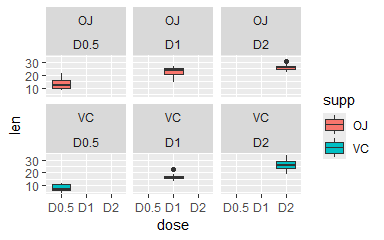
Para facet\_grid() hay dos opciones, presentarla por columnas o por filas:

ToothGrowth %>% mutate(dose=case\_when(  
 dose==0.5~"D0.5",dose==1~"D1", dose==2~"D2")) %>% ggplot(  
 aes(x = dose, y = len, fill=supp)) + geom\_boxplot()+   
 facet\_grid(supp~.)  
  
  
ToothGrowth %>% mutate(dose=case\_when(  
 dose==0.5~"D0.5",dose==1~"D1", dose==2~"D2")) %>% ggplot(  
 aes(x = dose, y = len, fill=supp)) + geom\_boxplot()+   
 facet\_grid(.~supp)



Y con facet\_wrap() podemos combinar columnas y filas, por ejemplo:

ToothGrowth %>% mutate(dose=case\_when(  
 dose==0.5~"D0.5",dose==1~"D1", dose==2~"D2")) %>% ggplot(  
 aes(x = dose, y = len, fill=supp)) + geom\_boxplot()+   
 facet\_wrap(~supp+dose, ncol = 3, nrow = 2)



#### 3.3.1.1 Guardando una gráfica en ggplot2

Para guardar una gráfica de tipo *ggplot2()* usamos la función *ggsave()*

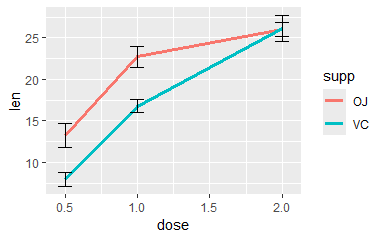
ggsave(filename = "plot.png", plot = a1, dpi = 300,width = 4, height = 3.5)

## 3.4 Extras

### 3.4.1 stat\_summary() y stat\_smoot()

Esta es una capa que nos permite poner desviaciones estándar o líneas que representan el promedio de los datos, por ejemplo:

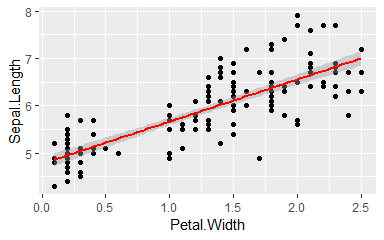
ToothGrowth %>%   
 ggplot(aes(x = dose, y = len)) +  
 stat\_summary(geom = "line", fun = mean, aes(group = supp, color = supp), size = 1.2) +   
 stat\_summary(geom = "errorbar", fun.data = mean\_se, aes(group = supp), width = 0.1)



stat\_smooth() nos permite colorear o resaltar la línea que representa nuestro modelo lineal:

ggplot(iris, aes(x = Petal.Width, y = Sepal.Length)) +   
 geom\_point() +  
 stat\_smooth(method = "lm", col = "red")

## `geom\_smooth()` using formula = 'y ~ x'

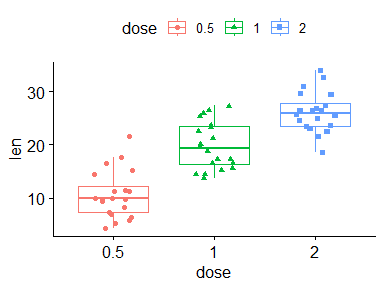


### 3.4.2 *ggpubr()*

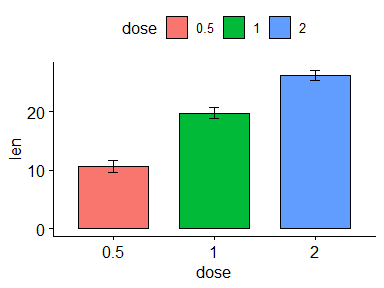
*ggpubr()* es una paqutería tipo compatible con *ggplot2()* pero un poco más fácil de declarar y más intuitivo.

Veamos algunos pocos ejemplos de sus utilidades:

library(ggpubr)  
ggboxplot(ToothGrowth, x = "dose", y = "len", color = "dose",add = "jitter", shape = "dose")



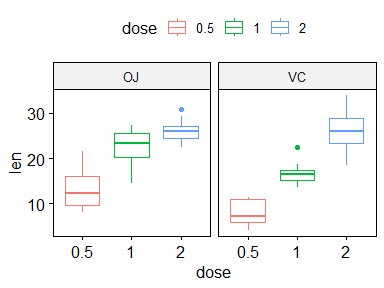
ggbarplot(ToothGrowth, x = "dose", y = "len", fill = "dose", add = "mean\_se")



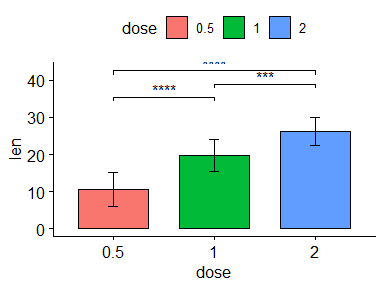
Vemos que los parámetros se definen similarmente pero con algunas diferencias, además que nos permite agregarles más elementos sin agregar más capas. Y el ‘output’ o tipo de gráfica es igual del tipo **ggplot2()**.

Otras funciones interesantes de este paquete son **facet.by** que es un argumento que puede ser declarado dentro de la función principal y nos permite hacer las gráficas multi-panel y otro que es muy útil es el de **stat\_compare\_means()** que nos permite agregar una capa que hace análisis estátidísticos como t.test, wilcoxon, kruskal.walis, anova, entre otros y que nos evita hacer análisis fuera y que podamos agragarlas a la gráfica en un sólo paso.

ggboxplot(ToothGrowth, x = "dose", y = "len", color = "dose",facet.by = "supp")



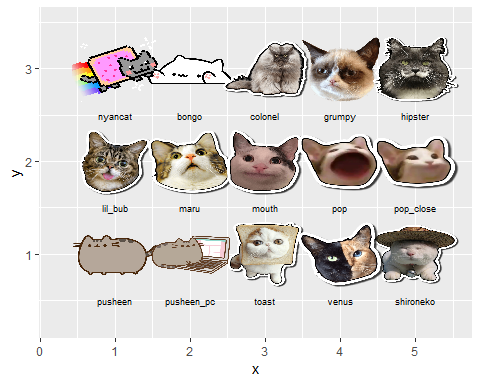
comparaciones <- list( c("0.5", "1"), c("1", "2"), c("0.5", "2") )  
  
ggbarplot(ToothGrowth, x = "dose", y = "len", fill = "dose", add = "mean\_sd")+  
 stat\_compare\_means(comparisons = comparaciones, label = "p.signif")



### 3.4.3 Misceláneos: *ggcats()*[[10]](#footnote-300)

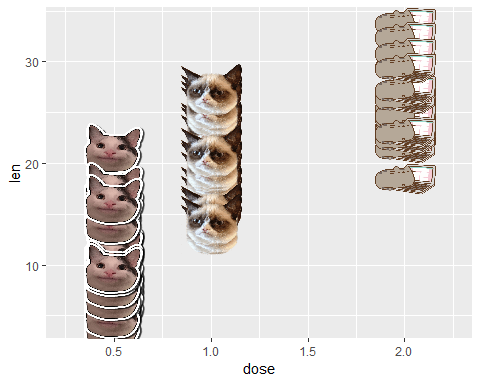
*ggplot2()* nos permite usar diversos paquetes desarrollados por muchos usarios con fines divertidos o misceláneos, podemos verlo en el pie de página el link para explorar todos los que hay. Veremos *ggcats()*. Estas son las opciones que podemos usar:

# install.packages("magick")  
# remotes::install\_github("R-CoderDotCom/ggcats@main")  
library(ggcats)  
grid <- expand.grid(1:5, 3:1)  
  
df <- data.frame(x = grid[, 1],  
 y = grid[, 2],  
 image = c("nyancat", "bongo",  
 "colonel", "grumpy",  
 "hipster", "lil\_bub",  
 "maru", "mouth",  
 "pop", "pop\_close",   
 "pusheen", "pusheen\_pc",  
 "toast", "venus",  
 "shironeko"))  
ggplot(df) +  
 geom\_cat(aes(x, y, cat = image), size = 5) +  
 geom\_text(aes(x, y - 0.5, label = image), size = 2.5) +  
 xlim(c(0.25, 5.5)) +   
 ylim(c(0.25, 3.5))



Entonces, por ejemplo si queremos usarlos como puntos de nuestra gráfica, sería algo así:

#hacemos la variable cats para escoger los que queremos  
ToothGrowth$cats <- factor(ToothGrowth$dose,  
 levels = c(0.5,1,2),  
 labels = c("mouth", "grumpy", "pusheen\_pc"))  
  
ToothGrowth %>% ggplot(aes(y = len, x = dose)) +   
 geom\_cat(aes(cat = cats), size = 4) + xlim(c(0.25, 2.25))

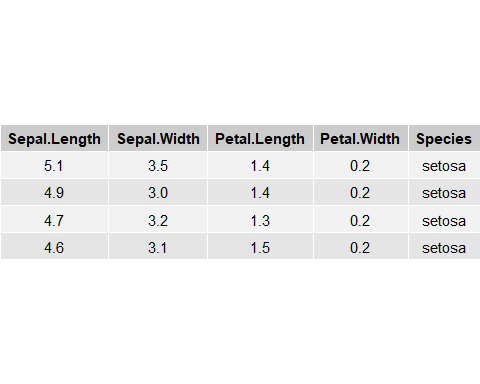


### 3.4.4 Misceláneos: *ggtexttable()*

Esta función me permite hacer mis tablas como figuras que puedo exportar como imágenes y darle formato. Hace parte del paquete **ggpubr()** que vimos anteriormente.

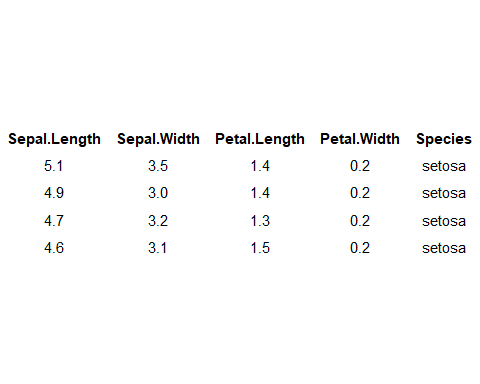
Haremos un ejemplo con los primeros cuatro datos de **iris**:

library(dplyr)  
df<- iris %>% slice(c(1:4))  
ggtexttable(df, rows = NULL)

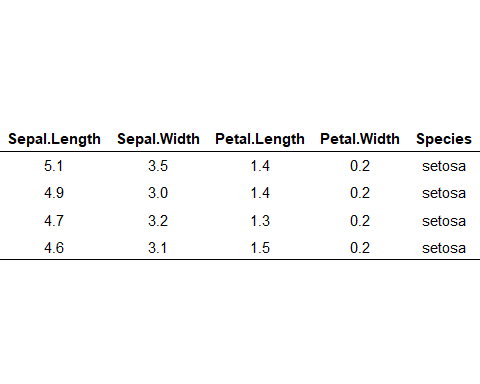


También podemos cambiar el tema con el que se formatea la tabla, por ejemplo:

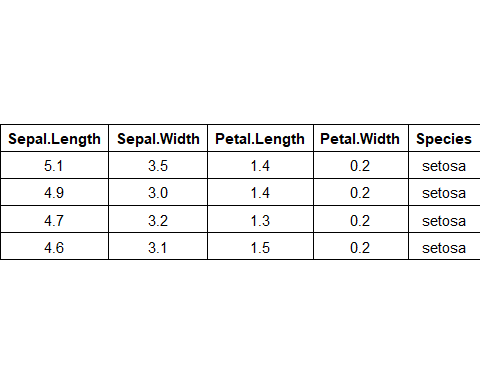
ggtexttable(df, rows = NULL, theme = ttheme("blank"))



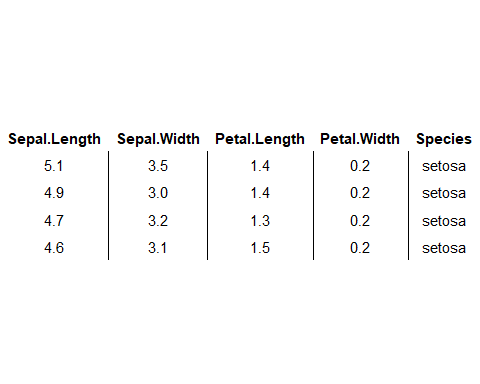
ggtexttable(df, rows = NULL, theme = ttheme("light"))



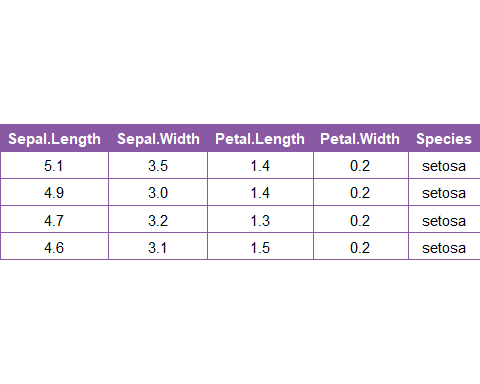
ggtexttable(df, rows = NULL, theme = ttheme("classic"))



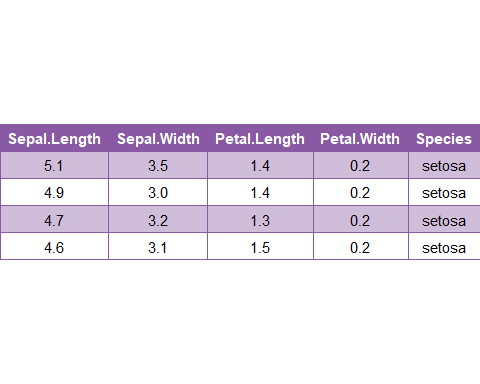
ggtexttable(df, rows = NULL, theme = ttheme("minimal"))



ggtexttable(df, rows = NULL, theme = ttheme("lVioletWhite"))

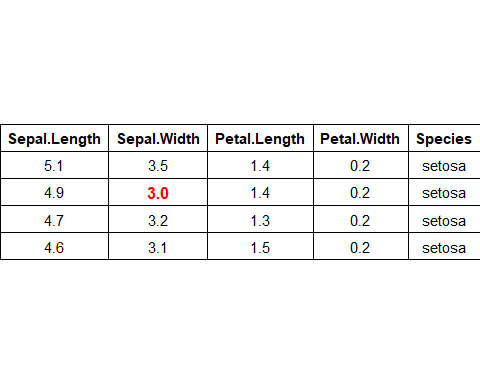
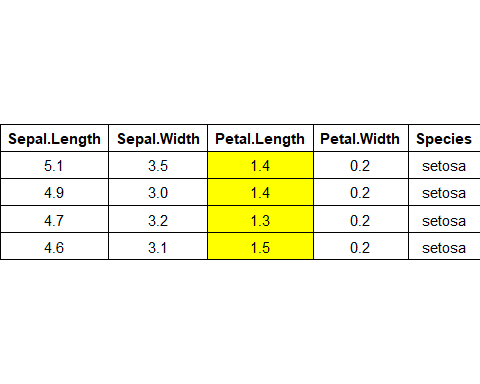
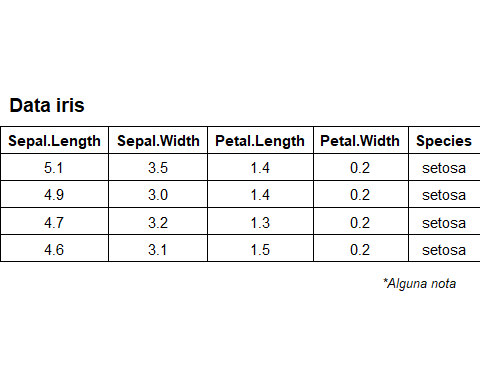


ggtexttable(df, rows = NULL, theme = ttheme("mVioletWhite"))



Para ver otros temas: <https://rpkgs.datanovia.com/ggpubr/files/ggtexttable-theme.pdf>

Otros formateos:

* Poner en negrita una celda:
* ggtexttable(df, rows = NULL, theme = ttheme("classic")) %>%   
   table\_cell\_font(row = 3, column = 2, face = "bold", color = "red")
* 
* Resaltar una columna:
* ggtexttable(df, rows = NULL, theme = ttheme("classic")) %>%   
   table\_cell\_bg(row = 2:5, column = 3, fill="yellow")
* 
* Poner títulos y pies de notas:
* ggtexttable(df, rows = NULL, theme = ttheme("classic")) %>%   
   tab\_add\_title(text = "Data iris", size = 14, face="bold") %>%  
   tab\_add\_footnote(text = "\*Alguna nota", size = 10, face = "italic")
* 

# 4 Valor de significancia p en estadística (p-value)

## 4.1 ¿Qué es el valor p?

## 4.2 Interpretación general del valor p

## 4.3 Marco de prueba de hipótesis: relación con la hipótesis nula y alternativa

## 4.4 Regla de decisión: umbrales comunes para el valor p

## 4.5 Errores en la prueba de hipótesis: error tipo I (falso positivo) y error tipo II (falso

# 5 Introducción a la estadística

## 5.1 Diferencia entre estadística y bioestadística

## 5.2 Estadística descriptiva: medidas de tendencia central y medidas de dispersión

## 5.3 Estadística inferencial: Univariada y multivariada

## 5.4 Estadística inferencial paramétrica: supuestos principales de las pruebas paramétricas

Se conoce como estadística paramétrica a aquella que se basa en el muestreo de una población con una distribución conocida y con parámetros fijos.

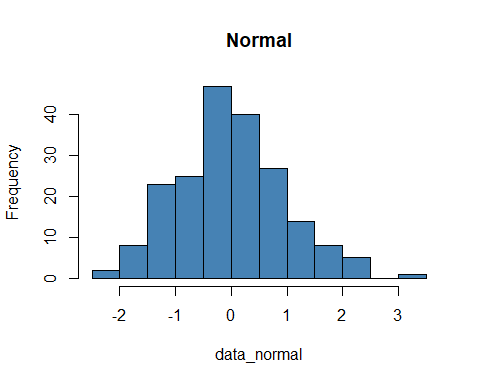
Los supuestos de las pruebas paramétricas en general son:

* Distribución conocida (**normal**): visual y pruebas numéricas.
* Homocedasticidad: visual y pruebas numéricas.
* Otros: tamaño de la muestra, variables cuantitativas o continuas, outliers, aleatoriedad, independencia de las observaciones, linealidad.

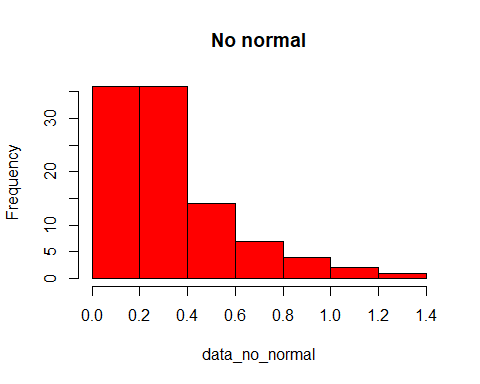
## 5.5 ¿Cómo saber si mis datos presentan una distribución normal y homogeneidad de las variancias?:

### 5.5.1 Métodos visuales

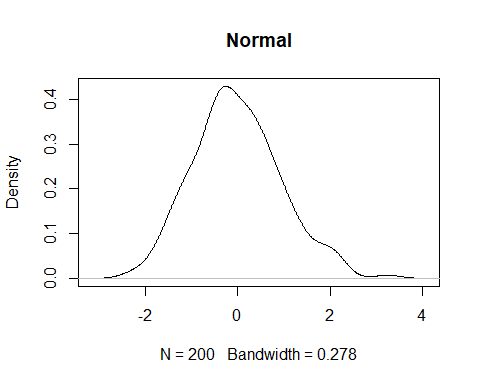
set.seed(123)  
data\_normal<- rnorm(200)  
hist(data\_normal, col='steelblue', main='Normal')



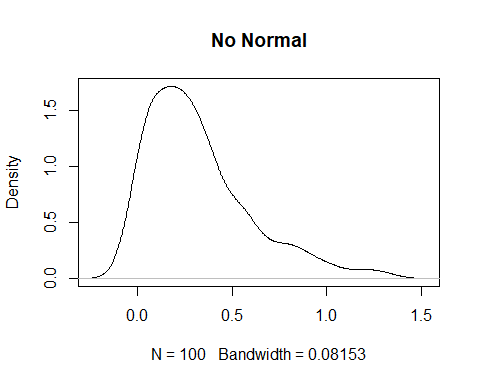
data\_no\_normal<- rexp(100, rate=3)  
hist(data\_no\_normal, col='red', main='No normal')



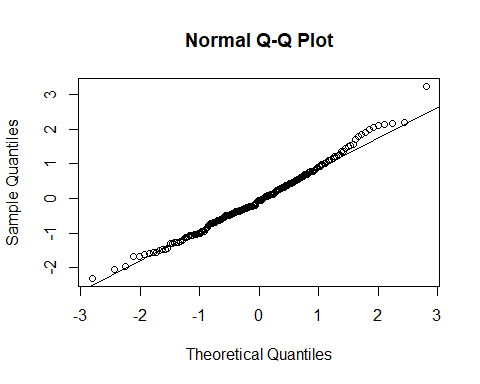
plot(density(data\_normal), main="Normal")



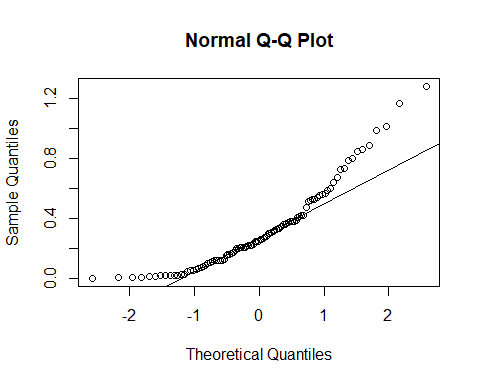
plot(density(data\_no\_normal), main="No Normal")



qqnorm(data\_normal)  
qqline(data\_normal)



qqnorm(data\_no\_normal)  
qqline(data\_no\_normal)



### 5.5.2 Prueba de Shapiro-Wilk

shapiro.test(data\_normal)

##   
## Shapiro-Wilk normality test  
##   
## data: data\_normal  
## W = 0.99076, p-value = 0.2298

shapiro.test(data\_no\_normal)

##   
## Shapiro-Wilk normality test  
##   
## data: data\_no\_normal  
## W = 0.89172, p-value = 5.884e-07

### 5.5.3 Pruebas de ajuste o distribución: Prueba de Kolmogorov-Smirnov, Prueba de Anderson-Darling

ks.test(data\_normal, "pnorm")

##   
## Asymptotic one-sample Kolmogorov-Smirnov test  
##   
## data: data\_normal  
## D = 0.053155, p-value = 0.6243  
## alternative hypothesis: two-sided

ks.test(data\_no\_normal, "pnorm")

##   
## Asymptotic one-sample Kolmogorov-Smirnov test  
##   
## data: data\_no\_normal  
## D = 0.50105, p-value < 2.2e-16  
## alternative hypothesis: two-sided

### 5.5.4 Probando Homocedasticidad o heterocedasticidad

aggregate(len ~ supp, data = ToothGrowth, var)

## supp len  
## 1 OJ 43.63344  
## 2 VC 68.32723

Ratio

68.32 / 43.63

## [1] 1.565895

aggregate(Petal.Width ~ Species, data = iris, var)

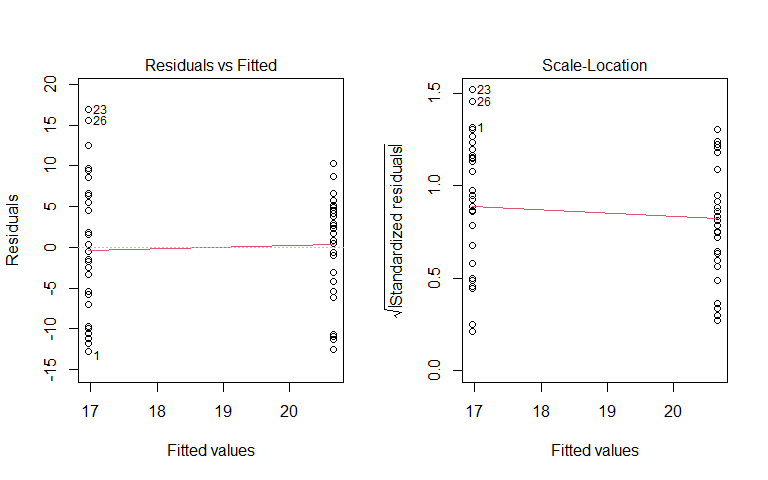
## Species Petal.Width  
## 1 setosa 0.01110612  
## 2 versicolor 0.03910612  
## 3 virginica 0.07543265

Ratio

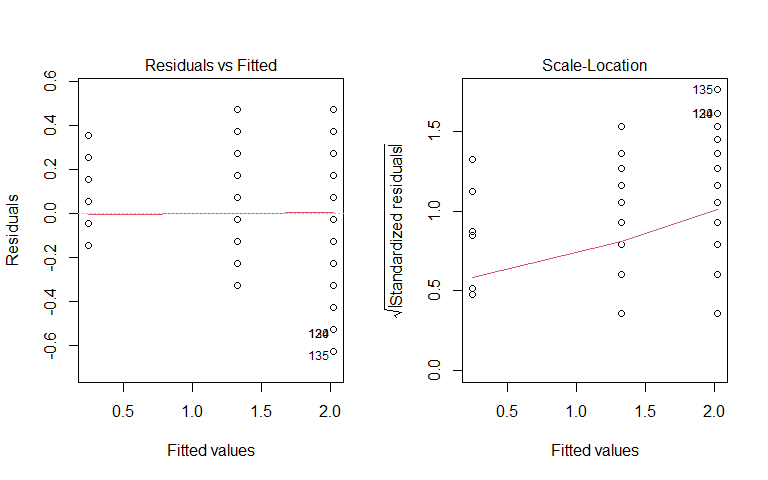
r1<-0.03910612 / 0.01110612 #versicolor vs setosa  
r2<-0.07543265 / 0.01110612 #virginca vs setosa  
r3<-0.07543264 / 0.03910612 #virginica vs versicolor  
cbind(r1,r2,r3)

## r1 r2 r3  
## [1,] 3.521132 6.791989 1.928922

m1<-lm(len ~ supp, data=ToothGrowth)  
par(mfrow = c(1, 2))  
plot(m1, which=c(1,3))



m2<-lm(Petal.Width ~ Species, data=iris)  
par(mfrow = c(1, 2))  
plot(m2, which=c(1,3))



Prueba para dos niveles = F

var.test(len ~ supp, data = ToothGrowth)

##   
## F test to compare two variances  
##   
## data: len by supp  
## F = 0.6386, num df = 29, denom df = 29, p-value = 0.2331  
## alternative hypothesis: true ratio of variances is not equal to 1  
## 95 percent confidence interval:  
## 0.3039488 1.3416857  
## sample estimates:  
## ratio of variances   
## 0.6385951

lmtest::bptest(m2) #sobre un modelo

##   
## studentized Breusch-Pagan test  
##   
## data: m2  
## BP = 25.099, df = 2, p-value = 3.546e-06

#### 5.5.4.1 Otras pruebas

GAD::C.test(m2)

##   
## Cochran test of homogeneity of variances  
##   
## data: m2  
## C = 0.60036, n = 50, k = 3, p-value = 5.531e-06  
## alternative hypothesis: Group virginica has outlying variance  
## sample estimates:  
## setosa versicolor virginica   
## 0.0111 0.0391 0.0754

car::leveneTest(m2)

## Levene's Test for Homogeneity of Variance (center = median)  
## Df F value Pr(>F)   
## group 2 19.892 2.261e-08 \*\*\*  
## 147   
## ---  
## Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

**Ejemplo:**

data(iris)  
cols<- c("Largo\_Sepalo", "Ancho\_Sepalo", "Largo\_Petalo", "Ancho\_Petalo", "Especies")  
colnames(iris)<- cols

Este conjunto de datos describe tres especies de las flores iris y como cambia el ancho y largo de su pétalo y sépalo. Veamos la estructura de los datos:

str(iris)

## 'data.frame': 150 obs. of 5 variables:  
## $ Largo\_Sepalo: num 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...  
## $ Ancho\_Sepalo: num 3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...  
## $ Largo\_Petalo: num 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...  
## $ Ancho\_Petalo: num 0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...  
## $ Especies : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...

dim(iris)

## [1] 150 5

nrow(iris)

## [1] 150

ncol(iris)

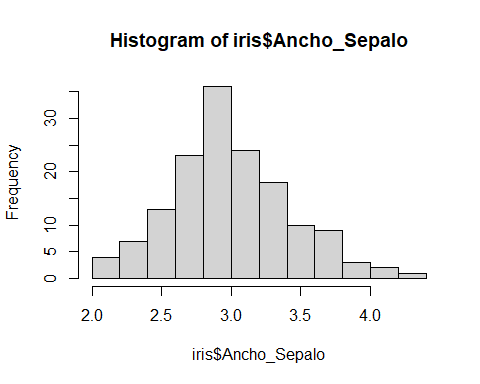
## [1] 5

Como vemos, posee 4 variables de respuesta y un factor que sería la especie de flor.

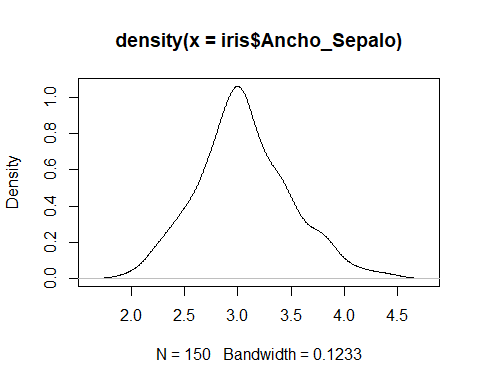
* Explorando normalidad en los datos

Existen diversas gráficas que podemos realizar para probar o explorar si nuestros datos siguen una distribución normal (también llamada distribución gaussiana) y su gráfica debe tener una forma acampanada y simétrica. La aplicación de muchas pruebas y estadísticos depende de si los datos siguen esta distribución o no. Por esto es importante antes de aplicar cualquier prueba estadística, explorar la distribución de nuestros datos y sí la prueba o estadístico que aplicamos asume que nuestros datos sean normales o no. Para este ejemplo, usaremos el ancho del sepalo en vez del largo del sepalo. ¨

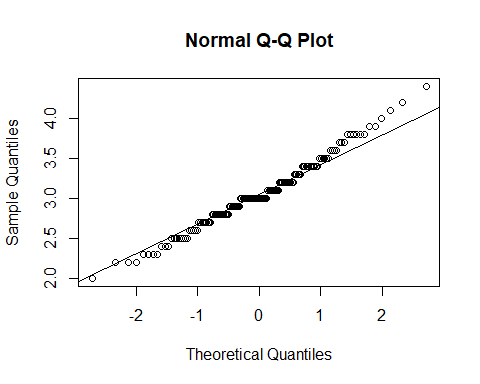
hist(iris$Ancho\_Sepalo)



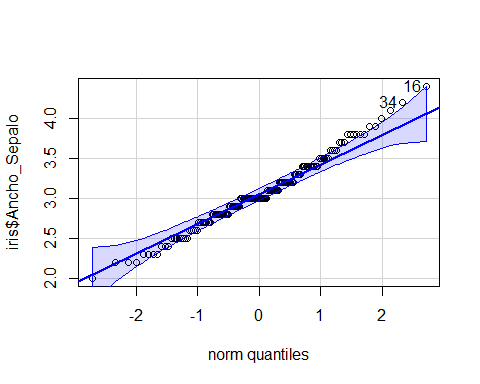
plot(density(iris$Ancho\_Sepalo))



qqnorm(iris$Ancho\_Sepalo)  
qqline(iris$Ancho\_Sepalo)



library(car) # cargamos el paquete car  
qqPlot(iris$Ancho\_Sepalo )



## [1] 16 34

Al parecer nuestros datos tienen una distribución normal, según los gráficos, sin embargo, para estar seguros de esto, haremos una prueba llamada test de shapiro que nos permitirá confirmar esto:

shapiro.test(iris$Ancho\_Sepalo)

##   
## Shapiro-Wilk normality test  
##   
## data: iris$Ancho\_Sepalo  
## W = 0.98492, p-value = 0.1012

La hipótesis nula que estamos aceptando o rechazando con esta prueba es que la distribución es normal y escogiendo un valor de probabilidad de 0.05 y dado que 0.1012 > 0.05 no podemos rechazar la hipótesis nula. En caso que este valor de p-value < 0.05 entonces los datos no serían normales.

## 5.6 Estadística inferencial no paramétrica: supuestos principales de las pruebas no paramétricas

# 6 Tipos de pruebas paramétricas

## 6.1 Prueba t de una muestra

## 6.2 Prueba t de dos muestras

## 6.3 Prueba t de Welch

## 6.4 Prueba F de Fisher

## 6.5 Prueba t pareada

## 6.6 ANOVA (one-way-anova, two-way-anova)

## 6.7 ANCOVA

## 6.8 Correlación de Pearson

## 6.9 Regresión lineal simple y múltiple

## 6.10 Etc.

| TIPO | PRUEBA |
| --- | --- |
| Comparación de 2 grupos | t de Student/Welch |
| Comparación de >2 grupos | Anova |
| Correlación de dos variables | Coeficiente de Pearson |
| Variables cualitativas | Prueba de Z |

# 7 t de Student

* Desarrollada por William Sealy Gosset. La prueba t permite comparar las medias de una muestra con la media teórica de una población, o comparar dos poblaciones.
* Una de las características de la prueba de student, es que permite la hipótesis alternativa de ver si dos medias son diferentes o si, una media es mayor, o menor que otra.
* Esta prueba tiene variantes dependiendo de nuestros datos.
* Para ser utilizada es necesario que se cumplan los siguientes supuestos: >- Muestreo aleatorio >- Los datos y errores deben tener una distribución normal >- Cuando se comparan dos distribuciones las varianzas deben ser iguales (σ1=σ2) y/o homogéneas.

# 8 t de Student una muestra

En la prueba t de Student para una muestra, se usa la siguiente fórmula:

Dónde es el estadístico, es la media muestral, es la desviación estándar muestral, es el tamaño de la muestra y es la media verdadera de la población.

El estadístico posee un valor de p asociado dependiendo de los grados de libertad de la prueba.

Mide qué tan inusual es la media de la muestra observada de la media de la muestra hipotética. Lo hace midiendo qué tan lejos está la media de la muestra observada de la media de la población hipotética en términos de errores estándar.

### 8.0.1 Ejemplo de una muestra

data<- data.frame(Congelador=1:10,  
 Temperatura=c(-2.14,-0.8,  
 -2.75,-2.58,   
 -2.26,-2.46,  
 -1.33,-2.85,  
 -0.93,-2.01))  
shapiro.test(data$Temperatura)$p.value

## [1] 0.1823153

Manualmente

prom = mean(data$Temperatura); desves = sd(data$Temperatura); sn = sqrt(nrow(data));mu = 0  
t=(prom-mu)/(desves/sn)  
t;pt(t, df=9)

## [1] -8.572431

## [1] 6.345623e-06

.pull-rigth[ Con R

t.test(data$Temperatura, mu = 0, alternative = "less")

##   
## One Sample t-test  
##   
## data: data$Temperatura  
## t = -8.5724, df = 9, p-value = 6.346e-06  
## alternative hypothesis: true mean is less than 0  
## 95 percent confidence interval:  
## -Inf -1.580971  
## sample estimates:  
## mean of x   
## -2.011

## 8.1 t de Student para medidas pareadas

En la prueba t de Student para medidas pareadas se usa la siguiente fórmula:

Dónde es el estadístico, es la media muestral de la diferencia de las medidas, es la desviación estándar muestral, es el tamaño de la muestra y es la diferencia de medias de mis grupos.

* Data ejemplo:

data\_pareada<- data.frame(Pan=1:10,  
 Temp\_pre=c(20.83,19.72,19.64,20.09,22.25,  
 20.83,21.31,22.50,21.17,19.57),  
 Temp\_post=c(100.87,98.58,109.09,121.83,122.78,  
 111.41,103.96,121.81,127.85,115.17))  
  
data\_pareada$Diferencia<- data\_pareada$Temp\_post - data\_pareada$Temp\_pre

## 8.2 Ejemplo muestras pareadas

| Pan | Temp\_pre | Temp\_post | Diferencia |
| --- | --- | --- | --- |
| 1 | 20.83 | 100.87 | 80.04 |
| 2 | 19.72 | 98.58 | 78.86 |
| 3 | 19.64 | 109.09 | 89.45 |
| 4 | 20.09 | 121.83 | 101.74 |
| 5 | 22.25 | 122.78 | 100.53 |
| 6 | 20.83 | 111.41 | 90.58 |
| 7 | 21.31 | 103.96 | 82.65 |
| 8 | 22.50 | 121.81 | 99.31 |
| 9 | 21.17 | 127.85 | 106.68 |
| 10 | 19.57 | 115.17 | 95.60 |

# 9 Ejemplo muestras pareadas

shapiro.test(data\_pareada$Diferencia)

##   
## Shapiro-Wilk normality test  
##   
## data: data\_pareada$Diferencia  
## W = 0.93611, p-value = 0.5106

Manualmente

prom = mean(data\_pareada$Diferenci); desves = sd(data\_pareada$Diferenci); sn = sqrt(nrow(data\_pareada));mu = 0  
t=(prom-mu)/(desves/sn)  
t;pt(t, df=9, lower.tail = FALSE)

## [1] 29.96903

## [1] 1.253345e-10

.pull-rigth[ Con R

t.test(data\_pareada$Temp\_post, data\_pareada$Temp\_pre, mu = 0, alternative = "greater", paired = TRUE)

##   
## Paired t-test  
##   
## data: data\_pareada$Temp\_post and data\_pareada$Temp\_pre  
## t = 29.969, df = 9, p-value = 1.253e-10  
## alternative hypothesis: true mean difference is greater than 0  
## 95 percent confidence interval:  
## 86.88337 Inf  
## sample estimates:  
## mean difference   
## 92.544

# 10 t de Student para dos grupos

En la prueba t de Student para dos grupós se usa la siguiente fórmula:

Dónde es el estadístico, son los promedios de los grupos y es el promedio teórico de la distribución nula de la diferencia de promedios (usualmente se usa 0), el viene dado por:

En este caso los grados de libertad se calculan :

# 11 t de Student para dos grupos

* Debemos tener en cuenta si hay dependencia o no de los grupos y si la varianza es igual y/o homogénea.
* Cuando los grupos son independientes, se supone que proceden de diferentes poblaciones que tienen la misma varianza.
* La t de muestras independientes es una medida de la diferencia de las medias de los dos grupos.
* La distribución de la que se muestrea la diferencia de medias no es de una sola población sino de la combinación de las dos poblaciones.

-Por ejemplo, tengo una población con promedio de 10 y otra de 7, varianzas de 4 y 4, entonces mi combinación tendrá un promedio de 3 (diferencia) y varianza de 8.

## 11.1 Ejemplo para dos grupos

freidora<- data.frame(  
 Oster1=c(4.72,7.40,3.50,3.85,4.47,  
 4.09,6.34,3.30,7.13,4.99),  
 Oster2=c(9.19,11.44,9.64,12.09,10.80,  
 12.71,10.04,9.06,6.31,9.44))  
ro1<- freidora$Oster1-mean(freidora$Oster1)  
ro2<- freidora$Oster2-mean(freidora$Oster2)  
  
shapiro.test(c(ro1,ro2))

##   
## Shapiro-Wilk normality test  
##   
## data: c(ro1, ro2)  
## W = 0.9483, p-value = 0.342

.

| Oster1 | Oster2 |
| --- | --- |
| 4.72 | 9.19 |
| 7.40 | 11.44 |
| 3.50 | 9.64 |
| 3.85 | 12.09 |
| 4.47 | 10.80 |
| 4.09 | 12.71 |
| 6.34 | 10.04 |
| 3.30 | 9.06 |
| 7.13 | 6.31 |
| 4.99 | 9.44 |

# 12 Ejemplo para dos grupos

Manualmente

prom1 = mean(freidora$Oster1)  
prom2=mean(freidora$Oster2)  
  
s=((var(freidora$Oster1)\*9)+(var(freidora$Oster2)\*9))/(10+10-2)  
t=(prom1-prom2-0)/(sqrt(s/10+s/10))  
t; pt(-6.8527, df=18, lower.tail=TRUE)+pt(6.8527, df=18, lower.tail=FALSE)

## [1] -6.852696

## [1] 2.062577e-06

.pull-rigth[ Con R

t.test(freidora$Oster1, freidora$Oster2, mu=0, paired=FALSE, alternative="two.sided", var.equal=TRUE)

##   
## Two Sample t-test  
##   
## data: freidora$Oster1 and freidora$Oster2  
## t = -6.8527, df = 18, p-value = 2.063e-06  
## alternative hypothesis: true difference in means is not equal to 0  
## 95 percent confidence interval:  
## -6.654429 -3.531571  
## sample estimates:  
## mean of x mean of y   
## 4.979 10.072

# 13 Prueba de Welch

Para los casos donde no se cumplen que las varianzas sean iguales u homogéneas.

Los grados de libertad se calculan:

.

t.test(freidora$Oster1, freidora$Oster2, mu=0, paired=FALSE, alternative="two.sided", var.equal=FALSE)

##   
## Welch Two Sample t-test  
##   
## data: freidora$Oster1 and freidora$Oster2  
## t = -6.8527, df = 17.268, p-value = 2.58e-06  
## alternative hypothesis: true difference in means is not equal to 0  
## 95 percent confidence interval:  
## -6.659188 -3.526812  
## sample estimates:  
## mean of x mean of y   
## 4.979 10.072

## 13.1 Pruebas no paramétricas y supuestos estadísticos

* Las pruebas no paramétricas no requieren que conozcamos la distribución de nuestros datos ni parámetros de la población de estudio como su media o varianza poblacional.
* Las pruebas no paramétricas nos permiten analizar datos en escala nominal u ordinal y en su mayoría los resultados se derivan a partir de procedimientos de ordenación y recuento, por lo que su base lógica es de fácil comprensión.
* Cuando trabajamos con muestras pequeñas (n < 10) en las que se desconoce si es válido suponer la normalidad de los datos, conviene utilizar pruebas no paramétricas
* Pocos supuestos: aleatoriedad e independencia.

# 14 PRUEBAS NO PARAMÉTRICAS

| VENTAJAS | DESVENTAJAS |
| --- | --- |
| - Mayor aplicabilidad | - Eficiencia estadística menor |
| - Se pueden usar variables ordinales | - Poder estadísitico menor |
| - Son más fáciles de calcular | - No se pueden evaluar interacciones |

# 15 Tipos de pruebas no paramétricas

## 15.1 Prueba de Mann-Whitney U (Wilcoxon rank-sum test)

## 15.2 Prueba de Wilcoxon para muestras relacionadas

## 15.3 Prueba de Kruskal-Wallis

## 15.4 Prueba de Friedman

## 15.5 Prueba de Chi-cuadrado

## 15.6 Prueba de McNemar

## 15.7 Correlación de rango de Spearman

## 15.8 Correlación de Kendall

# 16 Análisis de riqueza y diversidad alfa y beta

# 17 Análisis de riqueza y diversidad alfa y beta

# 18 Análisis multivariados

## 18.1 PCA

## 18.2 PCoA

## 18.3 NMDS

## 18.4 Permanova

## 18.5 permdis

## 18.6 Anosim

## 18.7 RDA

## 18.8 CCA

# 19 Modelización estadística:

## 19.1 Modelo lineal general

## 19.2 GLM con distribución Poisson

## 19.3 GLM con distribución quasi-Poisson

## 19.4 GLM con distribución de Bernoulli (Binomial)

## 19.5 GLMM: modelos lineales generales mixtos

1. <https://bookdown.org/ndphillips/YaRrr/arranging-plots-with-parmfrow-and-layout.html> [↑](#footnote-ref-31)
2. <https://cran.r-project.org/web/packages/cowplot/vignettes/introduction.html> [↑](#footnote-ref-34)
3. <https://rafalab.github.io/dslibro/r-basics.html#> [↑](#footnote-ref-78)
4. <https://bookdown.org/ndphillips/YaRrr/arranging-plots-with-parmfrow-and-layout.html> [↑](#footnote-ref-101)
5. <https://cran.r-project.org/web/packages/cowplot/vignettes/introduction.html> [↑](#footnote-ref-120)
6. <https://bookdown.org/ndphillips/YaRrr/arranging-plots-with-parmfrow-and-layout.html> [↑](#footnote-ref-140)
7. <https://bookdown.org/ndphillips/YaRrr/arranging-plots-with-parmfrow-and-layout.html> [↑](#footnote-ref-267)
8. <https://cran.r-project.org/web/packages/cowplot/vignettes/introduction.html> [↑](#footnote-ref-268)
9. <https://cran.r-project.org/web/packages/egg/vignettes/Ecosystem.html> [↑](#footnote-ref-269)
10. <https://r-charts.com/es/miscelanea/> [↑](#footnote-ref-300)