

MANUAL DE R Y RSTUDIO

Stephanie Hereira-Pacheco Mauricio Hernández

Contents

Prefacio	5
0.1 Objetivo del libro	5
0.2 Requisitos	5
0.3 Instalación de Rstudio y R	5
0.4 Aviso Legal: Protección de Propiedad Intelectual.	6
1 : Introducción a R	7
1.1 Comenzando con R y RStudio	7
1.2 Tipos de datos	20
1.3 <i>Data frames</i>	25
1.4 Creando subconjuntos o Indexación.	28
1.5 El directorio de trabajo y rutas	31
1.6 Funciones de importación	32
1.7 Algunas funciones Básicas de R	35
2 Manejo de data frames y bases de datos	43
2.1 Manipulación de datos con Rbase	43
2.2 Manejo de datos con tidyverse	55
3 Gráficos en R	79
3.1 Gráficas en R base	79
3.2 Gráficas con <i>ggplot2()</i>	89
3.3 Figuras Multi-panel / Facets	105
3.4 Extras	107
4 Valor de significancia p en estadística (p-value)	121
4.1 ¿Qué es el valor p?	121
4.2 Interpretación general del valor p	121
4.3 Marco de prueba de hipótesis: relación con la hipótesis nula y alternativa	121
4.4 Regla de decisión: umbrales comunes para el valor p	121
4.5 Errores en la prueba de hipótesis: error tipo I (falso positivo) y error tipo II (falso	121

5 Introducción a la estadística	123
5.1 Diferencia entre estadística y bioestadística	124
5.2 Estadística descriptiva: medidas de tendencia central y medidas de dispersión	124
5.3 Estadística inferencial: Univariada y multivariada	124
5.4 Estadística inferencial paramétrica: supuestos principales de las pruebas paramétricas	124
5.5 Estadística inferencial no paramétrica: supuestos principales de las pruebas no paramétricas	124
5.6 ¿Cómo saber si mis datos presentan una distribución normal y homogeneidad de las variancias?	124
5.7 Estadísticos descriptivos	125
5.8 Gráficos descriptivos	128
5.9 Explorando normalidad en los datos	129
5.10 Correlación y Regresión	132
5.11 ANOVA (1 vía)	135
5.12 Prueba de Tukey	136
5.13 ANOVA (2 vías)	138
6 Tipos de pruebas paramétricas	143
6.1 Prueba t de una muestra	143
6.2 Prueba t de dos muestras	143
6.3 Prueba t de Welch	143
6.4 Prueba F de Fisher	143
6.5 Prueba t pareada	143
6.6 ANOVA (one-way-anova, two-way-anova)	143
6.7 ANCOVA	143
6.8 Correlación de Pearson	143
6.9 Regresión lineal simple y múltiple	143
6.10 Etc.	143
7 Tipos de pruebas no paramétricas	145
7.1 Prueba de Mann-Whitney U (Wilcoxon rank-sum test)	145
7.2 Prueba de Wilcoxon para muestras relacionadas	145
7.3 Prueba de Kruskal-Wallis	145
7.4 Prueba de Friedman	145
7.5 Prueba de Chi-cuadrado	145
7.6 Prueba de McNemar	145
7.7 Correlación de rango de Spearman	145
7.8 Correlación de Kendall	145
8 Análisis de riqueza y diversidad alfa y beta	147
9 Análisis multivariados	149
9.1 PCA	149
9.2 PCoA	149

CONTENTS	5
----------	---

9.3 NMDS	149
9.4 Permanova	149
9.5 permdis	149
9.6 Anosim	149
9.7 RDA	149
9.8 CCA	149
10 Modelización estadística:	151
10.1 Modelo lineal general	151
10.2 GLM con distribución Poisson	151
10.3 GLM con distribución quasi-Poisson	151
10.4 GLM con distribución de Bernoulli (Binomial)	151
10.5 GLMM: modelos lineales generales mixtos	151

Prefacio

Este libro contiene la recopilación de temas impartidos y tratados a lo largo de diferentes cursos y lecciones sobre R y sus aplicaciones en Ciencias Biológicas.

0.1 Objetivo del libro

Que los estudiantes de nivel pregrado y posgrado puedan aprender temas básicos y aplicados del lenguaje de R y su aplicación en sus estudios y trabajos enfocados en ciencias biológicas.

0.2 Requisitos

- Rstudio versiones 2024 en adelante.
- R versiones de 4.0 en adelante.

0.3 Instalación de Rsudio y R

En caso de no tener instalado Rstudio y R que son los prerequisitos para llevar a cabo los ejercicios y temas de este libro, entonces visita estas ligas para descargarlos e instalarlos.

0.3.1 Rstudio

- En MacOS
- En Ubuntu
- En Windows

0.3.2 R

- Para MacOS
- Para Ubuntu
- Para Windows

0.4 Aviso Legal: Protección de Propiedad Intelectual.

El material contenido en este manual está protegido por las leyes mexicanas de Propiedad Intelectual. Por lo que su reproducción o distribución no autorizada con fines de lucro está prohibida.

Chapter 1

: Introducción a R

1.1 Comenzando con R y RStudio

1.1.1 Breve descripción de R

R es un lenguaje de programación como C o Java o Python pero enfocado principalmente a la estadística. No fue creado por ingenieros de software para el desarrollo de software, sino por estadísticos como un ambiente interactivo para el análisis de datos. Otros programas como los que mencioné anteriormente sí están más enfocados en el desarrollo de programas aunque también pueden usarse para hacer cálculos estadísticos. Cuando instalamos R en nuestra computadora en realidad lo que estamos instalando es el entorno computacional y para que podamos hacer uso de ese entorno necesitamos conocer la manera de escribir que el software pueda interpretar y ejecutar las instrucciones que le damos. Eso es lo que aprenderemos a hacer en este curso.

Como en otros lenguajes de programación, en R pueden guardar su trabajo como una secuencia de comandos o instrucciones, conocida como un *script*, que se pueden ejecutar fácilmente en cualquier momento, los podemos guardar y nos servirán siempre.

1.1.2 Un poco de historia...¹

R proviene del lenguaje S, creado en los Laboratorios Bell (Estados Unidos). Los mismos que inventaron el transistor, el láser, el sistema operativo Unix y algunas otras cosas más.

Ross Ihaka y Robert Gentleman, de la Universidad de Auckland de Nueva Zelanda, decidieron crear una implementación abierta y gratuita de S. Este tra-

¹<https://bookdown.org/ndphillips/YaRrr/arranging-plots-with-parmfrow-and-layout.html>

bajo, que culminaría en la creación de R inició en 1992 y no fue hasta el 2000 que se obtuvo una versión final estable.

Hoy día, el mantenimiento y desarrollo de R es realizado por el R Development Core Team, un equipo de especialistas en ciencias computacionales y estadística provenientes de diferentes instituciones y lugares alrededor del mundo.

R posee una Licencia Pública General de GNU, esto, para que pueda ser distribuido de manera gratuita, por lo que es software libre y de código abierto. Esta licencia también te permite usar R para los fines que deseas, sin limitaciones, no importando si son personales, académicos o comerciales.

1.1.3 ¿Por qué usar R?²

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 sin problemas entre plataformas.
4. Existe una comunidad grande, creciente y activa de usuarios de R y, como resultado, hay numerosos recursos para aprender y hacer preguntas^{4 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 de nuevas metodologías de ciencia de datos. Esto les da a los usuarios de R acceso temprano 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.

1.1.4 RStudio

RStudio será nuestra plataforma para los proyectos usados con el lenguaje R. Nos provee un editor visual e interactivo para crear y editar nuestros *scripts*, además de otras herramientas útiles que iremos viendo con el pasar de los temas.

1.1.5 Interfaz de Rstudio

Rstudio posee 4 paneles principales:

1. El panel izquierdo superior aparece nuestro editor de códigos. Donde ponemos los scripts que vamos a correr.
2. A la derecha, el panel superior incluye pestañas como *Environment* y *History*, que son el ambiente y el historial, aquí podremos observar los

²<https://cran.r-project.org/web/packages/cowplot/vignettes/introduction.html>

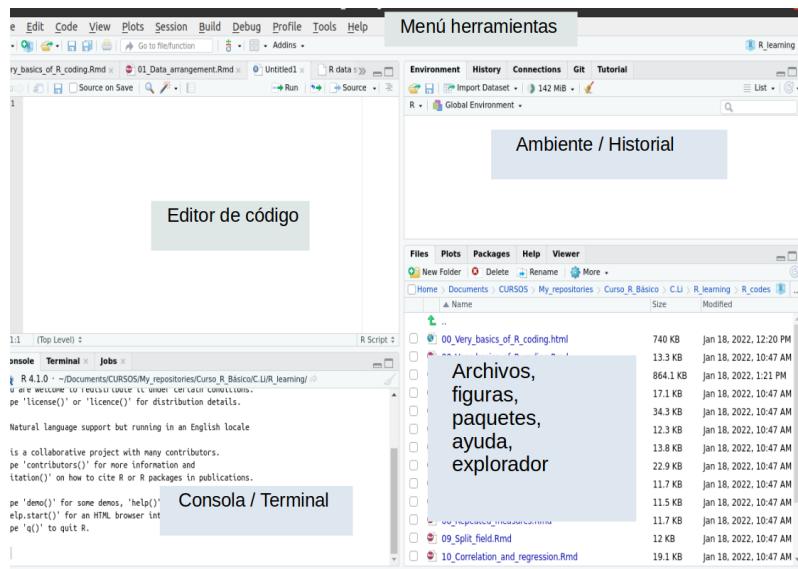
³<https://cran.r-project.org/web/packages/egg/vignettes/Ecosystem.html>

⁴<https://r-charts.com/es/misclanea/>

⁵<https://www.r-project.org/help.html>

objetos que vayamos declarando y data que subamos, además del historial de scripts.

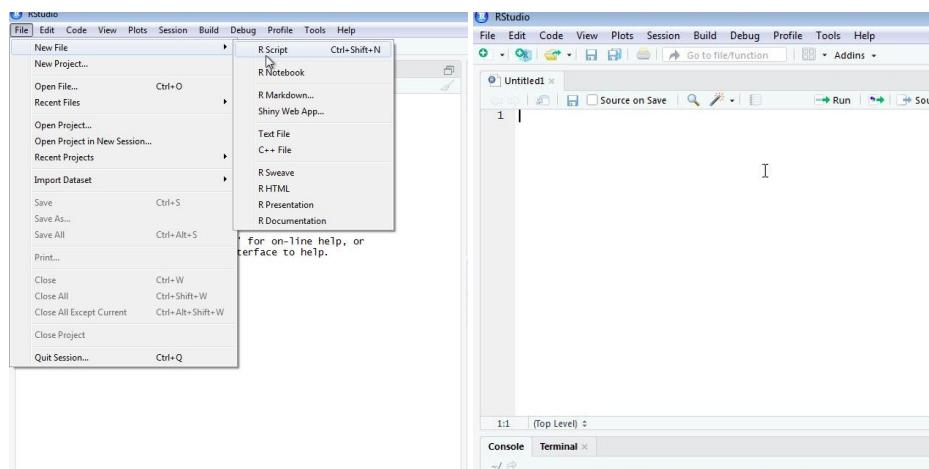
3. En el panel inferior izquierdo nos aparece nuestra consola de R que es donde se corren los códigos.
4. En el panel inferior derecho se muestran cinco pestañas: *File*, *Plots*, *Packages*, *Help* y *Viewer*. Pueden hacer clic en cada pestaña para moverse por las diferentes opciones. Pero a grandes rasgos, en file vemos donde nos encontramos situados y los archivos que hay en nuestro directorio. En plots se visualizan las imágenes que generamos, en packages los paquetes que poseemos instalados y cargados, en Help cuando necesitamos información extra de nuestros paquetes y en Viewer exploramos scripts de rmarkdown.



1.1.6 Scripts

Una de las grandes ventajas de R y Rstudio es que se pueden guardar los diversos códigos e instrucciones, los famosos conocidos como *scripts*, que entonces se pueden editar y guardar con un editor de texto.

Para iniciar un nuevo *script*, hagan clic en *Archivo*, entonces *Nuevo Archivo* y luego *R Script*. Esto inicia un nuevo panel a la izquierda y es aquí donde pueden comenzar a escribir su *script*.



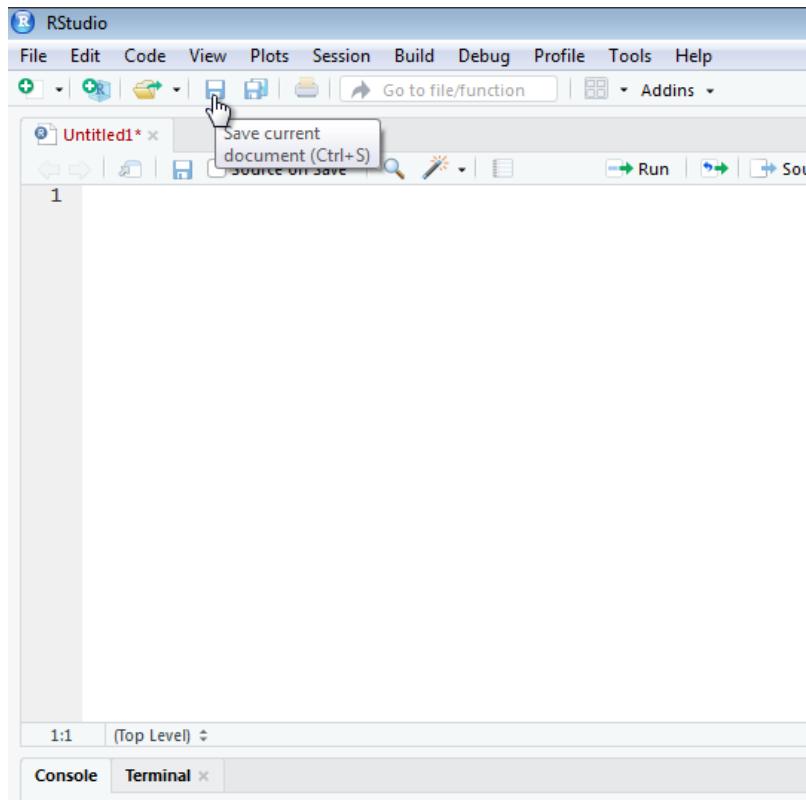
Podemos abrir y ejecutar scripts en R usando la función `source()`, dandole como argumento la ruta del archivo .R en nuestra computadora, entre comillas o yendo a *File*, entonces *Open File* y luego buscando en las carpetas donde tengas el script.

Por ejemplo.

```
source("C:/mi_script.R")
```

1.1.6.1 Cómo ejecutar comandos mientras editan *scripts*

Empezamos por abrir un nuevo *script* y luego nombramos el *script*. Podemos hacer esto a través del editor guardando el nuevo *script* actual sin nombre. Al guardar el script por primera vez use un nombre descriptivo, con letras minúsculas, sin espacios, preferiblemente con “_” para separar palabras, evitar guiones “-” para separar las palabras. Llamaremos a este *script*: *mi_script.R*.



Ahora podemos editar nuestro primer *script*.

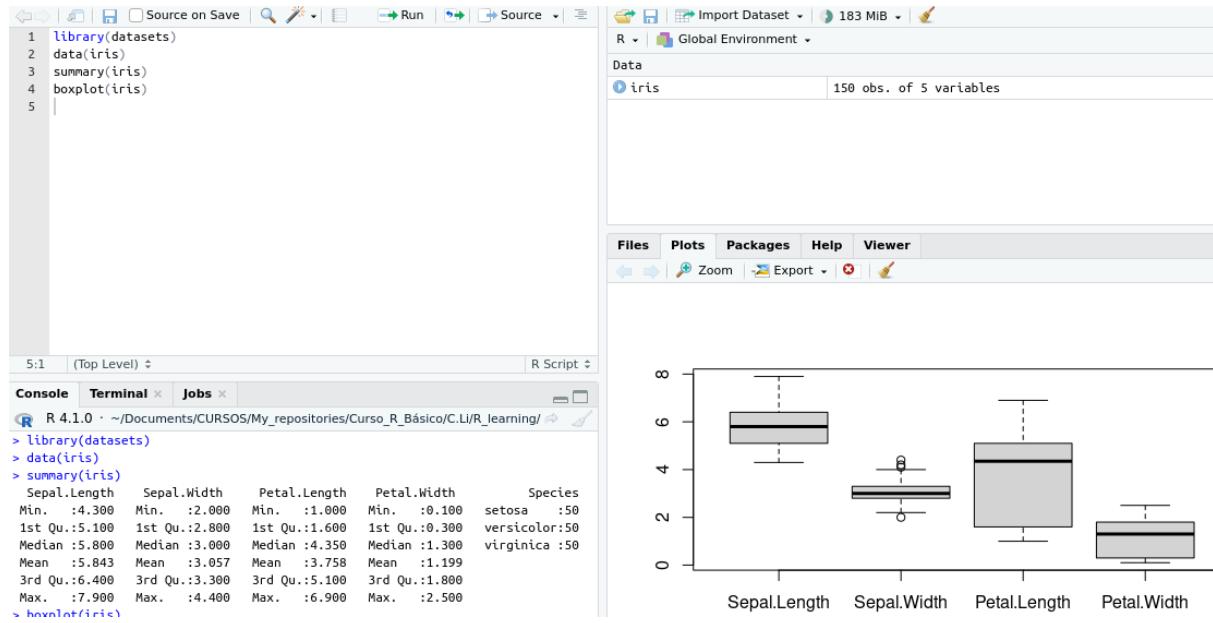
Las primeras líneas de código son títulos o comentarios, para hacerlo siempre debemos poner el símbolo “#” para indicar que no es un código y luego cargamos los paquetes y datos que vamos a utilizar. Para esta parte, luego veremos otra sección donde profundizaremos mejor en esto de cargar datos y paquetes.

Ahora podemos continuar escribiendo código. Como ejemplo, trabajaremos con “iris” dataset. Cargamos la librería “datasets” y cargamos la data, luego veremos el resumen de los datos y por último graficaremos un boxplot. Para hacer esto, escribimos cada línea de código y luego hacemos click en el botón *Run* en la parte derecha superior del panel de edición. Para ejecutar una línea pueden usar Control+Enter en Windows y Linux y Command+Return en Mac.

Estas son las líneas del código:

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

Y así luce al correrlo:



Tan pronto se corra el código, aparece en la consola y el gráfico aparece en en panel de “*Plots*” , este panel permite hacer click hacia delante o hacia atrás en diferentes gráficos, hacer zoom en el gráfico o guardar los gráficos como archivos.

1.1.7 Directorio de trabajo

El *directorio de trabajo* es el lugar en nuestra computadora en el que se encuentran los archivos con los que estamos trabajando en R. Este es el lugar donde R busca los archivos para importarlos y al que serán exportados o guardados, a menos que se indique otra cosa.

Para saber donde está ubicado tu directorio de trabajo, puedes poner el código:

```
getwd()
```

Y para establecer o cambiar este directorio de trabajo, correr el siguiente código:

```
setwd("/home/steph/Desktop/")
```

Por ejemplo en este caso estoy estableciendo mi directorio de trabajo en la carpeta “Escritorio”. También si se quiere conocer el contenido de tu directorio, como archivos o directorios puedes usar los siguientes códigos:

```
list.files()
list.dirs()
```

1.1.8 Sesión

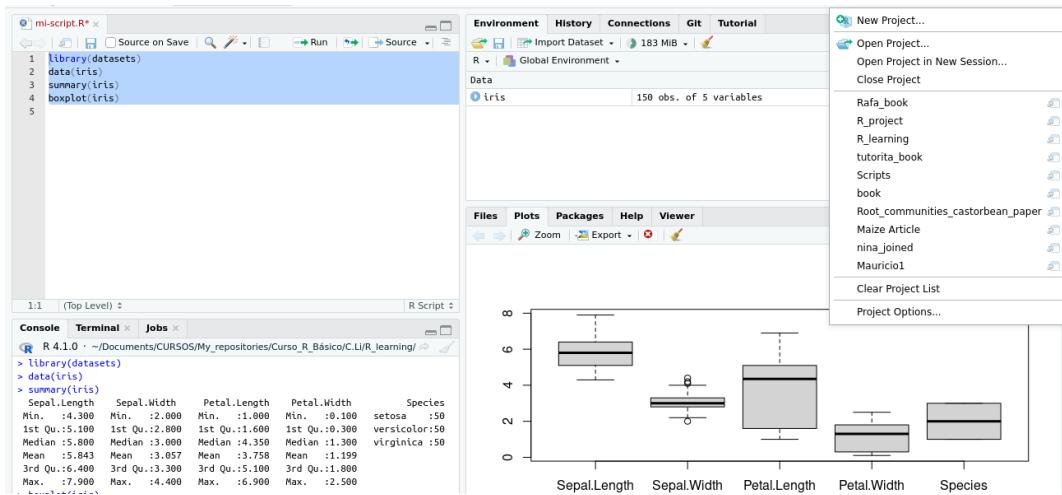
Los objetos y funciones de R son almacenados en la memoria de nuestra computadora. Cuando ejecutamos R, ya estamos creando una instancia del entorno computacional de este lenguaje de programación, cada instancia es una *sesión*.

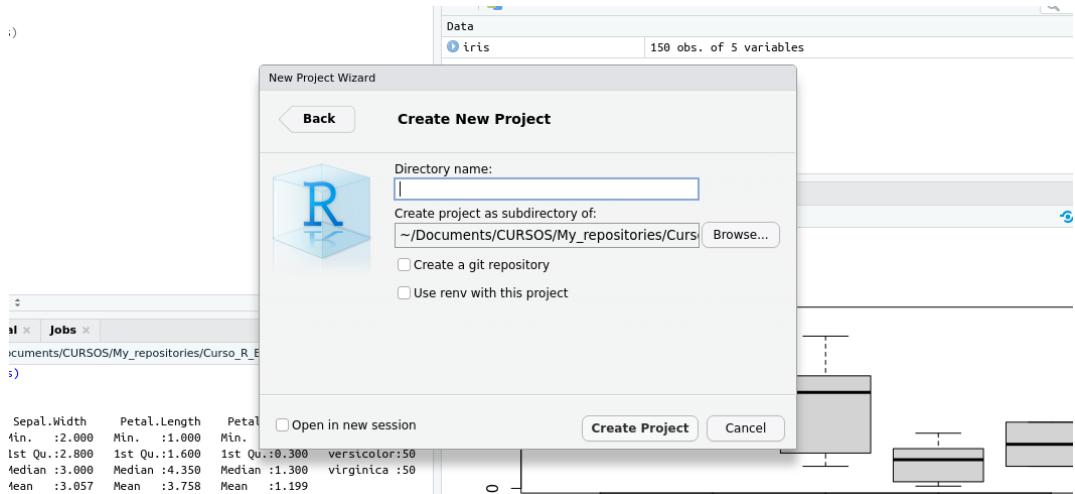
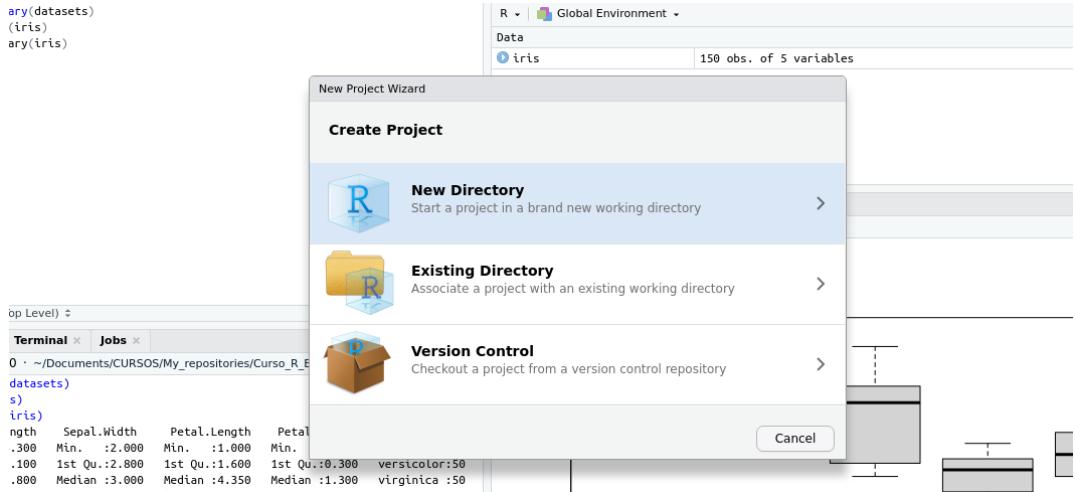
Todos los objetos, tablas, gráficas y funciones creadas en una sesión, permanecen sólo en ella, no son compartidos entre sesiones. Si se quiere guardar toda esta data generada en la sesión antes de cerrar el R hay que indicar que se guarde, sino, aunque se guarden los scripts, los objetos (que algunos pueden ser muy pesados y ocupar memoria) no se guardarán. Lo cual es una opción si no se tiene mucho espacio en el disco. Es posible tener más de una sesión de R activa en la misma computadora. Esto se guarda en un archivo con extensión `**.Rdata*` en tu directorio de trabajo.

Con la función `ls()` conoceremos una lista con los nombres de todo lo guardado en la sesión. En las siguientes imágenes ilustro como crear un nuevo proyecto:

1.1.9 Proyecto

Un proyecto de R (extensión .Rproj) identifica todos los archivos y contenido asociado con él. Ayuda a organizar tu trabajo y así cada curso, artículo o trabajo diferente puede tener un proyecto diferente. Al crear un proyecto todos los ficheros quedan vinculados directamente a él.





1.1.10 Instalación de paquetes de R

Dentro de las ventajas de R está que muchos desarrolladores y programadores elaboran constantemente complementos, aplicaciones o en palabras propias *pa-*

quetes que nos permiten usar en acceso libre y que tienen muchas funcionalidades. Actualmente hay muchos disponibles en CRAN (Comprehensive R Archive Network) que es una red de servidores alrededor del mundo que almacena versiones actualizadas del código de R y su documentación. También hay muchos paquetes desarrollados publicados en GitHub y en Bioconductor. Por ejemplo para instalar el paquete tidyverse, que es de gran utilidad y que veremos en sesiones posteriores, usamos el código:

```
install.packages("tidyverse")
```

En RStudio pueden navegar a la pestaña *Packages* y seleccionar *Install*. Luego, escribir el paquete que queremos siempre y cuando esté en CRAN. Para cargar una librería como lo vimos anteriormente se usa la función, `library()`:

```
library(tidyverse)
```

Una vez que se instalan los paquetes, no deben instalarlo de nuevo, sin embargo, cada vez que cerramos sesión, reiniciamos sesión o abrimos un nuevo proyecto o sesión tenemos que volver a cargarlos.

Debemos tener en cuenta que la instalación de `tidyverse` instala varios paquetes. Esto ocurre comúnmente cuando un paquete tiene *dependencias*, es decir usa funciones de otros paquetes. Cuando cargan un paquete usando `library`, también cargan sus dependencias.

Hay paquetes que no se encuentra en CRAN o que si queremos su versión en desarrollo, se necesitan de otros paquetes para ser instalados, por ejemplo, si queremos instalar la versión en desarrollo del paquete “`rmarkdown`”, que se encuentra en github, se utiliza el paquete `devtools`:

```
devtools::install_github('rstudio/rmarkdown')
```

Los dos puntos “`::`” se utilizan para denotar que llamamos la función de un paquete pero sin llamarla permanentemente en nuestra sesión.

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 *Environment*. Los objetos pueden ser:

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

Más adelante detallaremos este tipo de objetos o datos en R. Aquí unos ejemplos:

<code>a <- 1</code>	<code>#escalar</code>
<code>letra <- "a"</code>	<code>#caracter o letra</code>

```

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

```

Una ventaja de los lenguajes de programación es poder definir variables y escribir expresiones como estas, como se hace en las matemáticas y así almacenar los valores para su uso posterior. Usamos `<-` para asignar valores a las variables. También podemos asignar valores usando `=` en lugar de `<-`, pero recomendamos no usar `=`. Esto, debido a que en R el signo `=` implica igualdad en términos lógicos y no asignación y esto puede evitarnos confusiones en el futuro.

Para ver el valor almacenado en una variable, simplemente le pedimos a R que evalúe `a` y R nos muestra el valor almacenado:

```
a
```

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

Una forma explícita de pedirle a R que nos muestre el valor almacenado en `a` es usar `print` así:

```
print(a)
```

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

Otra forma de examinar los objetos es buscarlos en el *Environment* o ambiente de trabajo y visualizarlos desde allí. Deberíamos ver `a`, `b` y las que ya hemos declarado en el ambiente. Si intentamos imprimir o visualizar el valor de un objeto que no está definido en el ambiente se recibirá un mensaje de error. Por ejemplo, si escriben `f`, verán lo siguiente: `Error: object 'f' not found`.

1.1.11.0.1 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, no nombren una de sus variables `install.packages` escribiendo algo como: `install.packages <- 2`. Esto reescribe la función o causa confusiones para R.
- Que tus nombres sean descriptivos y/o significativos para lo que se está almacenando, usar solo minúsculas y usar guiones bajos (`_`) como sustituto de espacios, evitar caracteres especiales (`- ; . @ ?`) .

1.1.11.1 Guardar los espacios de trabajo y exportar objetos de R

Los objetos evaluados permanecen en el espacio de trabajo hasta que finalicen sus sesiones sin guardar. Pero los espacios de trabajo también se pueden guardar para su uso posterior. De hecho, al salir de R, el programa les pregunta si desean

guardar su espacio de trabajo. Si lo guardan, la próxima vez que inicien R, el programa restaurará el espacio de trabajo con todos los objetos en él.

Sin embargo, no se recomienda guardar el espacio de trabajo porque sino se trabaja con diferentes proyectos, será más difícil darle seguimiento de lo que guardan y ocupara mucho espacio en la memoria de su disco. En cambio, se recomienda en gran medida realizar un proyecto por trabajo o tarea y así tener un espacio de trabajo para cada uno. Decidiendo o no si guardar las sesiones y no entrar en confusiones entre las diferentes sesiones.

Ahora bien R ya posee funciones de exporte, tales como:

```
library(readr)
write_tsv(d, "data.tsv")
write.table(d, "data.txt", sep = "\t")
write_csv(d, "data.csv")
```

Estas funciones guardan tus tablas u objetos con formatos de texto. También hay una forma de guardar objetos de R sin declararlos como texto sino que se queden con la identidad de R:

```
saveRDS(d, "data.RDS")
```

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

```
readRDS("data.RDS")
```

1.1.12 Funciones en R⁶

Una vez que definidos los objetos o las variables, si queremos continuar con el análisis de datos generalmente se usan una serie de funciones específicas que se aplican a las variables o datos. R incluye muchas funciones por *default* o preestablecidas y otras pueden extraerse al cargar los diversos paquetes. A lo largo de este curso ya hemos usado varias funciones tales como las funciones para exportar datos como *write_csv()* o para cargar paquetes tales como *library()*, entre otras. Hay otras funciones más sencillas como por ejemplo la función *log* o *sqrt* con las que podemos obtener el logaritmo o la raíz cuadrada. (*Nótese que las funciones en su gran mayoría están definidas en inglés*).

Muchas otras funciones vienen establecidas en otros paquetes como fue el caso que vimos de *read_csv()* del paquete *readr*. La sintaxis o el lenguaje de R nos indica que necesitas usar paréntesis para evaluar funciones como hemos visto en los ejemplos anteriores que hemos usado funciones. Algunas funciones como *ls()* no requieren argumentos sino que nos da la información que ocupamos sin evaluar nada dentro de los paréntesis. Sin embargo, en otras funciones sí requieren uno o más argumentos. A continuación se muestra un ejemplo de

⁶<https://rafalab.github.io/dslibro/r-basics.html#>

cómo asignamos un objeto al argumento de la función `log`. Recuerden que anteriormente definimos `a` como 1:

```
log(a)
```

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

Podemos explorar funciones en el panel inferior derecho en la pestaña de *Pages* o *Paquetes* podemos explorar los paquetes y al hacer click en alguno nos despliega las funciones que posee y su documento de ayuda también. Se puede saber cuáles son los argumentos de la función o lo que la función espera o las opciones que tiene, con el comando `help` o también anteponiendo un signo `?` antes de la función. También mientras escribimos la función si usamos la tecla `tab` nos indica también que argumentos espera esta función. Pueden averiguar lo que la función espera y lo que hace revisando unos manuales muy útiles incluidos en R. Pueden obtener ayuda utilizando la función `help` así:

```
help("log")
?log
```

La página de ayuda les mostrará qué argumentos espera la función. Por ejemplo, `log` necesita `x` y `base` para correr. Sin embargo, algunos argumentos son obligatorios y otros son opcionales. Pueden determinar cuáles son opcionales notando en el documento de ayuda cuáles valores predeterminados se asignan con `=`. Definir estos es opcional. Por ejemplo, la base de la función `log` por defecto es `base = exp(1)` que hace `log` el logaritmo natural por defecto.

Para echar un vistazo rápido a los argumentos sin abrir el sistema de ayuda, pueden escribir:

```
args(log)
```

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

Pueden cambiar los valores predeterminados simplemente asignando otro objeto:

```
log(x = 8, base = 2)
```

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

El código anterior funciona, pero también podemos ahorrarnos un poco de escritura: si no usan un nombre de argumento, R supone que están ingresando argumentos en el orden en que se muestran en la página de ayuda o por `args`. Entonces, al no usar los nombres, R supone que los argumentos son `x` seguido por `base`:

```
log(8,2)
```

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

Si se usan los nombres de los argumentos, podemos incluirlos en el orden en que queramos:

```
log(base = 2, x = 8)
```

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

Para especificar argumentos, debemos usar `=` y no `<-`. Aquí si aplica la igualdad, en el caso de las funciones.

Hay algunas excepciones a la regla de que las funciones necesitan los paréntesis para ser evaluadas. Entre estas, las más utilizados son los operadores aritméticos y relacionales. Por ejemplo:

```
a + a
```

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

```
a - 2
```

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

```
1 * pi
```

```
## [1] 3.141593
```

```
2 / 3
```

```
## [1] 0.6666667
```

```
4 ^ a
```

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

También es posible declarar una función que a lo mejor no esté definida en R. Por ejemplo, la función `mean()` en Rbase nos da el promedio de un conjunto de datos, pero si quisieramos definirla en caso de no conocerla o hacerlo a nuestra manera, sería algo así:

```
average<- function(x){sum(x)/length(x)}
```

```
x<- 1:100
```

```
average(x)
```

```
## [1] 50.5
```

```
mean(x)
```

```
## [1] 50.5
```

Al compararlas nos arrojan el mismo valor. Lo importante de este punto es que hemos declarado una nueva función, declarando primero la variable (o variables, dependiendo el caso) y luego las operaciones y asignandolos a un objeto de R:

```
nombre <- function(argumentos) {
  operaciones}
```

Para trabajar con R resulta importante conocer los principales tipos de objetos y sus propiedades básicas.

1.2 Tipos de datos

Como vimos en el ejemplo anterior, los objetos en R pueden ser de varios tipos. Por ejemplo, necesitamos distinguir números de los caracteres. La función `class` nos ayuda a determinar qué tipo de objeto tenemos:

```
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
Vacio	null	NULL

Existen otros tipos de datos más complejos que no están en el alcance del presente curso, por ejemplo: números complejos, fechas, entre otros.

1.2.0.1 Para tener en cuenta...

- El tipo *character* representa texto y es fácil reconocerlo porque un dato siempre está rodeado de comillas, simples o dobles. Este es el tipo de datos más flexible de R, pues una cadena de texto puede contener letras, números, espacios, signos de puntuación y símbolos especiales.
- Un factor es un tipo de datos específico a R. Puede ser descrito como un dato numérico representado por una etiqueta. Por último, cada una de las etiquetas o valores que puedes asumir un factor se conoce como **nivel** o **level**. Los niveles tienen un orden diferente al orden de aparición en el factor. En R, por defecto, los niveles se ordenan alfabéticamente. **Advertencia:** Los factores pueden causar confusión ya que a veces se comportan como caracteres y otras veces no. Como resultado, estos son una fuente común de errores. A veces las funciones necesitan a fuerza que se ocupe un vector y a veces lo contrario.
- La diferencia entre las dos es que un dato **NULL** aparece sólo cuando R intenta recuperar un dato y no encuentra nada, mientras que **NA** es usado

para representar explícitamente datos perdidos, omitidos o que por alguna razón son faltantes. `NA` además puede aparecer como resultado de una operación realizada, pero no tuvo éxito en su ejecución.

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
<code>as.integer()</code>	Entero
<code>as.numeric()</code>	Numérico
<code>as.character()</code>	Carácter
<code>as.factor()</code>	Factor
<code>as.logical()</code>	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 función `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
## 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 evalúan 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 resultados.

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

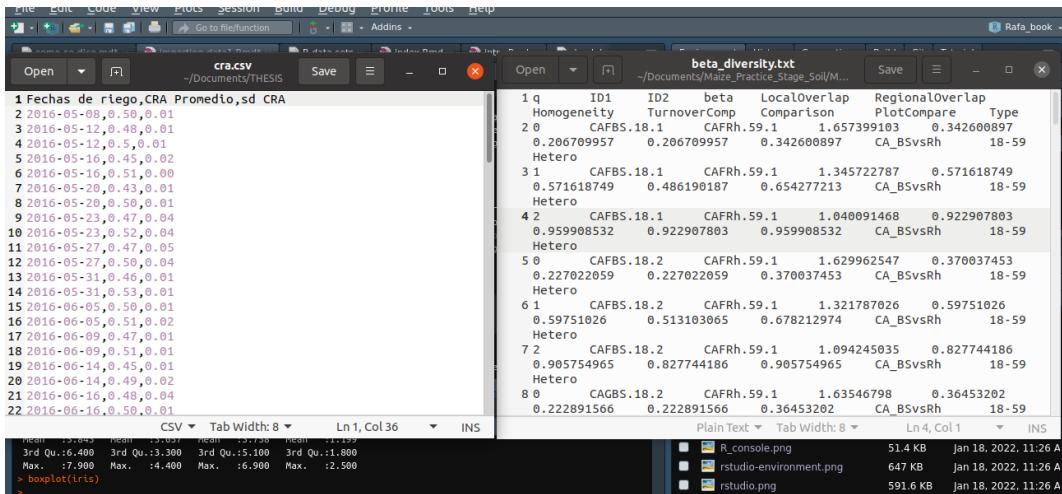
Para el ejemplo que vimos en el capítulo pasado usamos un dataset que está en el ambiente de R por default, si queremos saber cuales son los datasets que tenemos en nuestro ambiente, podemos usar el comando *data()* y nos desplegará la lista:

```
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"
  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-Projects/master/Iris.csv")
```

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 objeto `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" , "Espesura.Petalo" ))
```

En esta función usamos el argumento `col_names` para establecer los *Headers* o nombre de las columnas de esta tabla.

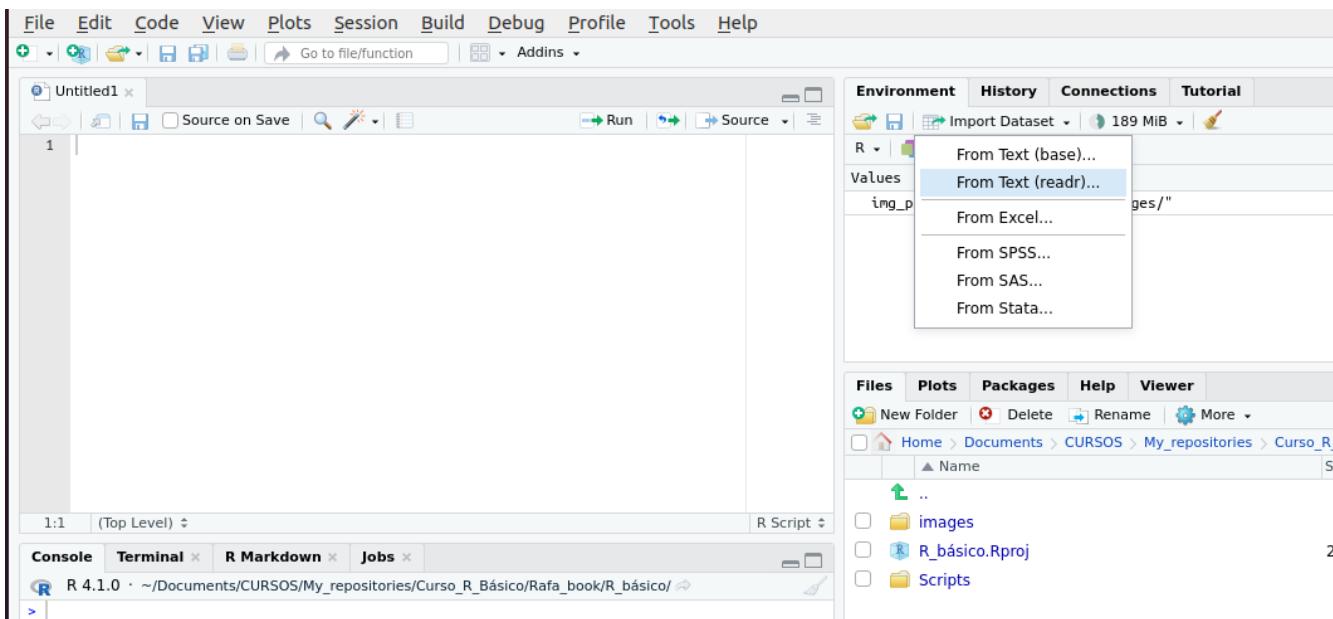
La segunda opción que vimos es utilizar la ruta completa del archivo, por ejemplo:

```
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*⁷

1.6.1.1 *readr*

El paquete **readr** un paquete de tidyverse, 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

⁷<https://bookdown.org/ndphillips/YaRrr/arranging-plots-with-parmfrow-and-layout.html>

Función	Tipo de archivo	Extensión
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

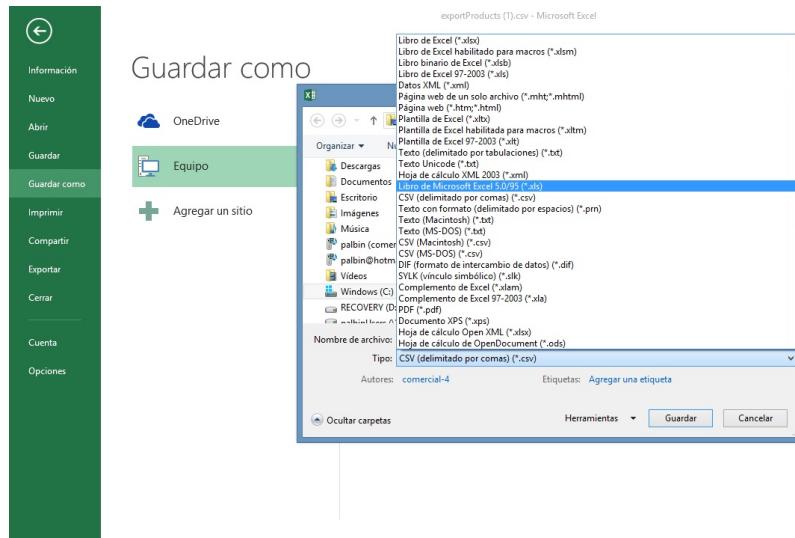
1.6.1.2 readxl

Este paquete ofrece funciones para leer archivos provenientes 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 que 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 columnas.
- **FUN**: La función que aplicaremos a la matriz **X** en su dimensión **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, es `ifelse()` 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 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 citada⁸

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
```

⁸<https://cran.r-project.org/web/packages/cowplot/vignettes/introduction.html>

```
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* (rpart).

Chapter 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 podemos sólo reescribir el nuevo nombre por el viejo, por ejemplo:

```
colnames(promedio_notas)

## [1] "estudiante"      "clase"          "notas"          "ponderacion"    "nota_ponderacion"

colnames(promedio_notas)[2] <- "curso"
colnames(promedio_notas)

## [1] "estudiante"      "curso"          "notas"          "ponderacion"    "nota_ponderacion"
```

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 nota_ponderada
## 1          E1 Matemáticas     7       0.7           4.9          E1 Matemáticas     7
## 2          E2 Matemáticas     8       0.7           5.6          E2 Matemáticas     8
## 3          E3 Matemáticas     9       0.7           6.3          E3 Matemáticas     9
## 4          E1 Biología        10      0.3           3.0          E1 Biología        10
## 5          E2 Biología        9       0.3           2.7          E2 Biología        9
## 6          E3 Biología        8       0.3           2.4          E3 Biología        8
##   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 función *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 promedios 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 aplicado¹

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(primer); 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?

¹<https://bookdown.org/ndphillips/YaRrr/arranging-plots-with-parmfrow-and-layout.html>

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 la data llamada `tatuajes.por.áño` 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 superhero favorito es superman?
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?

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 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 superhero 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 utilizando otras funciones del Rbase. Sin

embargo, existe todo un universo llamado *tidyverse* que nos permite 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 x 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", "D2"))
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:

La función `glimpse` nos permite ver los datos como la función `str` pero uniéndolo al pipe.

También podemos seleccionar columnas con criterios, por ejemplo:

```
ToothGrowth %>% select(starts_with("d")) %>% glimpse()
```

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()
```

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:

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  
## $ SUPP <fct> VC, VC  
## $ 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,
```

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

#npositiva

```
ToothGrowth %>% select_at(vars(contains("ose")))) %>% glimpse()
```

Rows: 60

```
## Columns: 1
```

#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  
## $ supp <fct> VC, VC
```

```
ToothGrowth %>% select_at(vars(!contains("ose"))) %>% glimpse()
```

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: 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:

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 0J 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

```
## 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 10.0  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, caracteres, 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 ejemplificar *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 x 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 x 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 x 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 x 2
##   supp    dose
##   <fct> <dbl>
## 1 OJ     1.17
## 2 VC     1.17

ToothGrowth %>% group_by(supp) %>% summarise_all(mean)

## # A tibble: 2 x 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()`:

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

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

Rows: 60

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
```

```
##   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 intersección o las que comparten).

- `left_join()`: incluye todas las filas en x.
- `right_join()`: incluye 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 x 2
##   name   band
##   <chr> <chr>
## 1 Mick   Stones
## 2 John   Beatles
## 3 Paul   Beatles

## # A tibble: 3 x 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 x 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 x 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 x 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 x 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 *tidyverse*

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 x 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 x 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(primer); 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 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 superhero favorito es superman?
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?

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 x 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 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("tatuajespor año"=Tatuajes/Edad)
encuestas$tatuajespor año
```

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

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

```
encuestas %>% filter(tatuajespor año == max(tatuajespor año))
```

```
##           Nombre Sexo Edad Superhéroe Tatuajes tatuajespor año
## 1 Nivedithia     F   32      Maggot      900        28.125
```

7. ¿Cuáles son los nombres de las mujeres a las que su superhero 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
```

Chapter 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*.

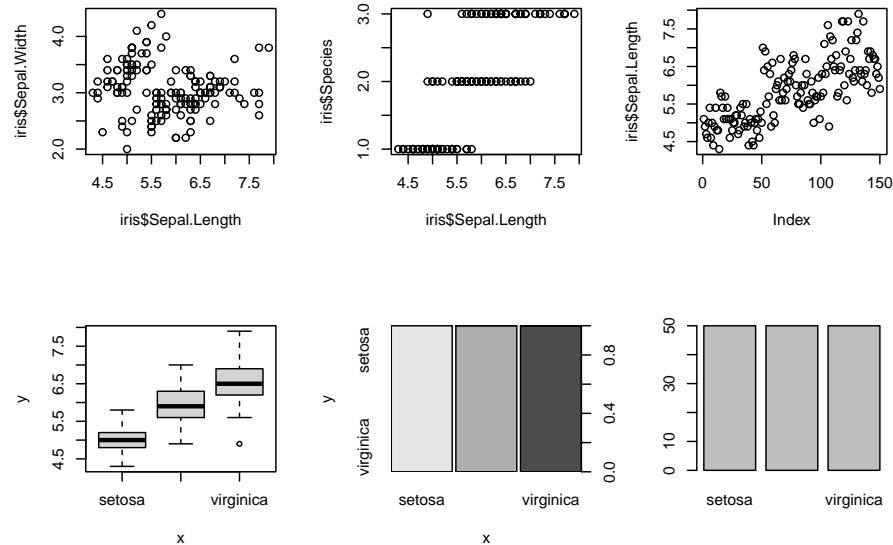
3.1.1 La función *plot()*

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)
```

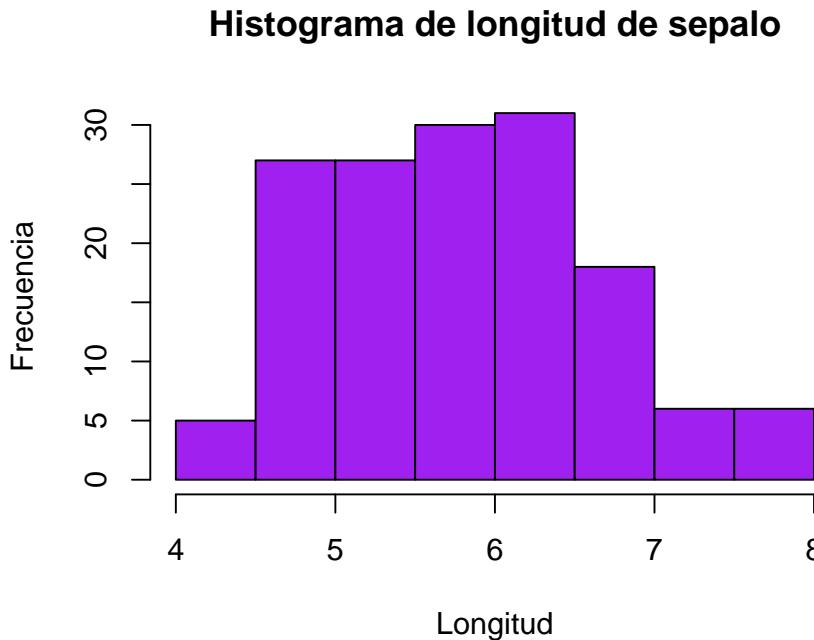


Además de estas, también hay otras funciones de Rbase que nos permiten graficar tipos específicos como son *barplot()*, *boxplot()* o *hist()*.

3.1.2 Histogramas

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 argumentos para darle formato, veamoslo con un ejemplo:

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



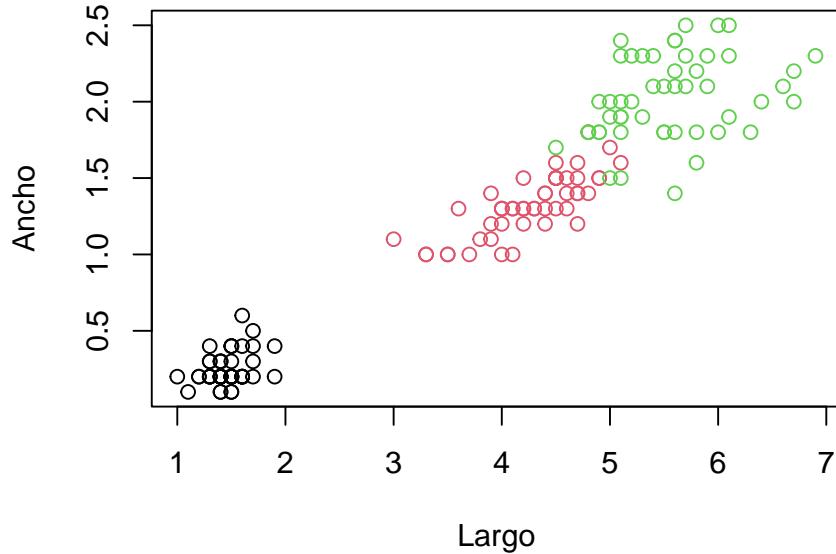
Como vemos en este ejemplo hay argumentos que aplicamos que puede ser aplicados para todos los gráficos de Rbase, tales como:

- *main* : Título 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")
```



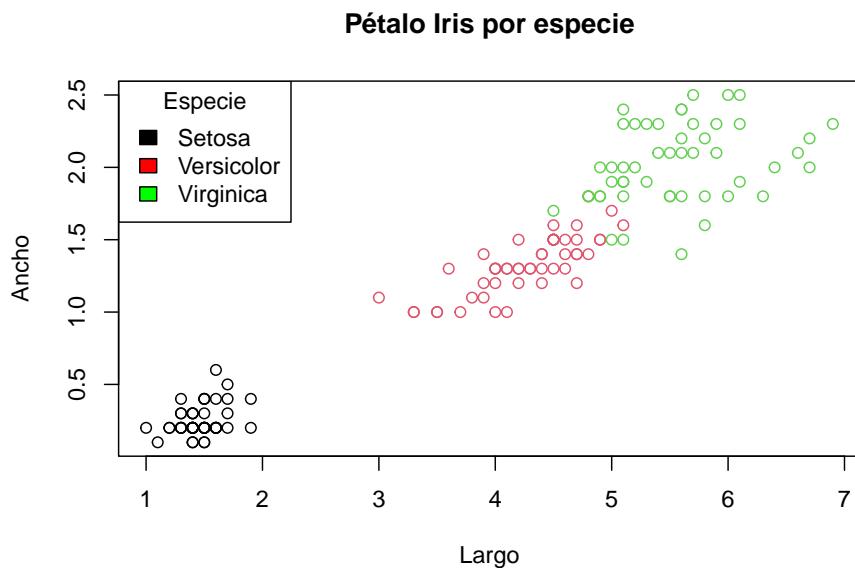
Si queremos agregar una leyenda a alguna de nuestras figuras de Rbase, usamos la función `legend()`:

`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:

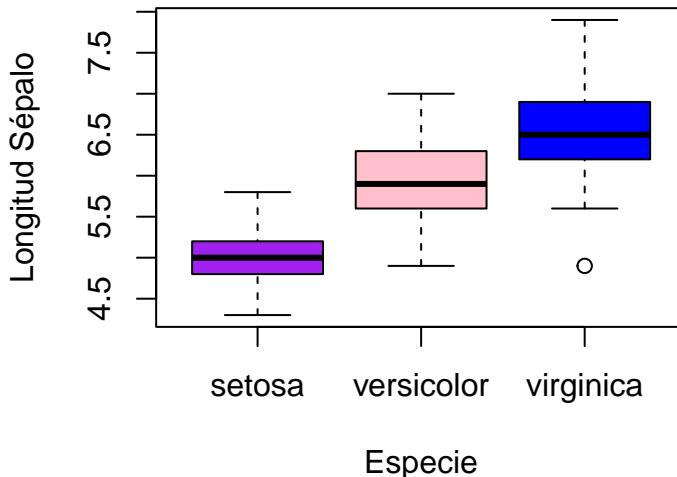
```
plot(x = iris$Petal.Length, y = iris$Petal.Width, col = iris$Species,
      main = "Pétalo Iris por especie", xlab = "Largo", ylab = "Ancho")
legend(x = "topleft", legend = c("Setosa", "Versicolor", "Virginica"),
       fill = c("black", "red", "green"), title = "Especie")
```



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 podemos 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"))
```

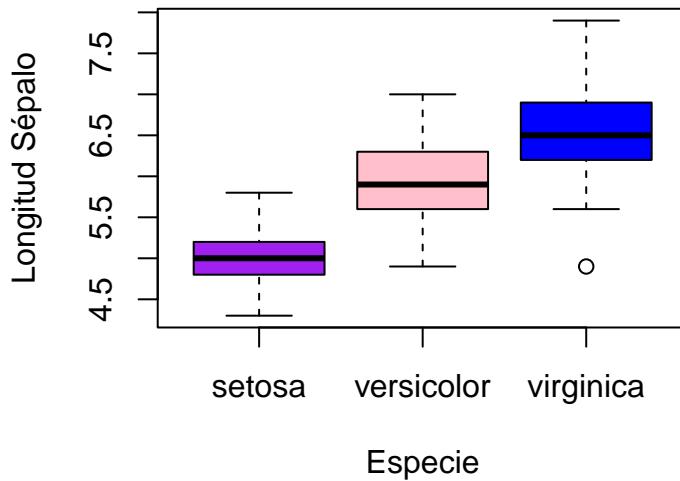


También podemos usar la función `boxplot()`. En esta segunda forma de hacer diagramas de cajas necesitamos declarar dos argumentos principales:

- **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 serán tomadas 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

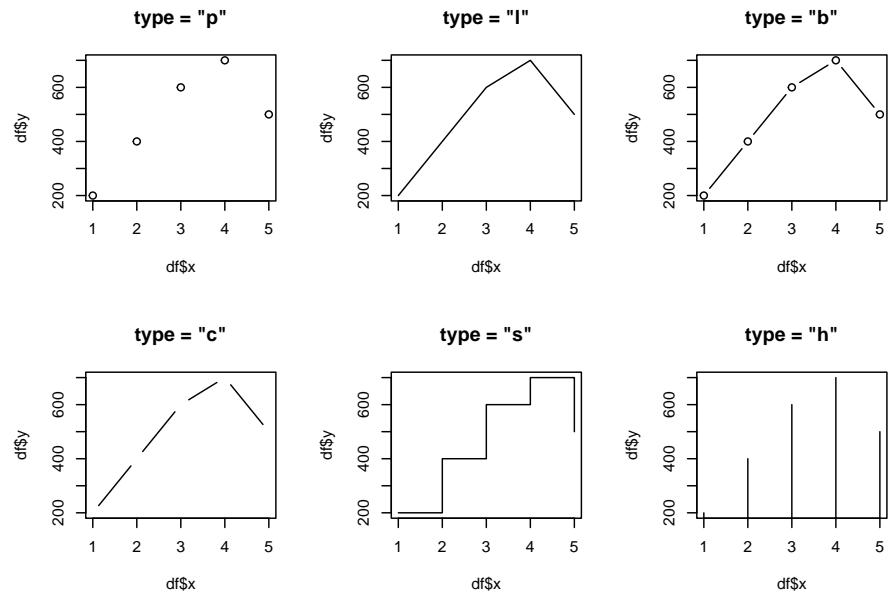
Usando la función `plot()` podemos dibujar más gráficos de los vistos que son los estándares, como fue el caso cuando graficamos modelos lineales. También existe otro argumento de esta función que se denomina `type` y nos permite escoger otros tipos de gráficos. Esto lo podemos ver usando el ‘?’ para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

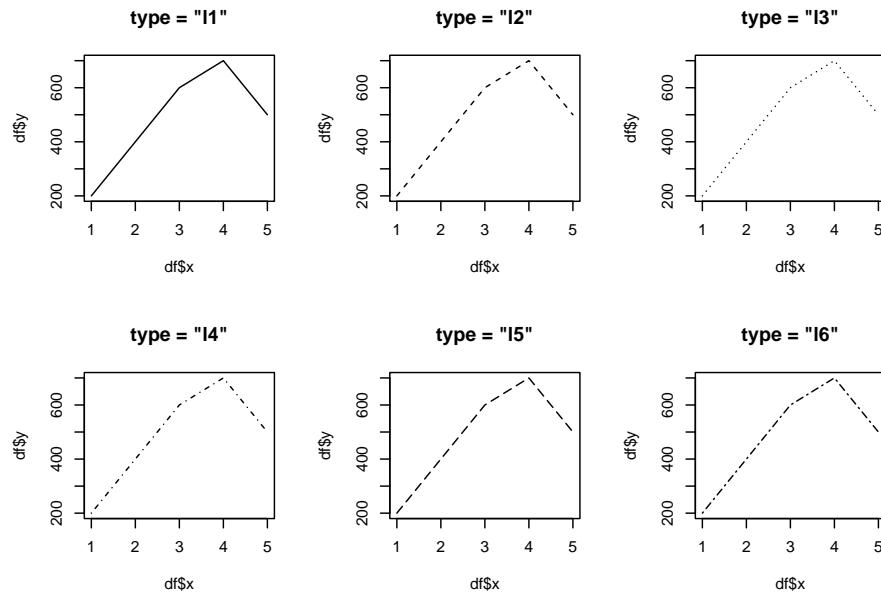
```
#dataset de ensayo
df<- data.frame(x= c(1:5),
                 y= c(200, 400, 600, 700, 500))
par(mfrow = c(2, 3))
plot(df$x, df$y, type = "p", main = 'type = "p"')
plot(df$x, df$y, type = "l", main = 'type = "l"')
plot(df$x, df$y, type = "b", main = 'type = "b"')
```

```
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*:

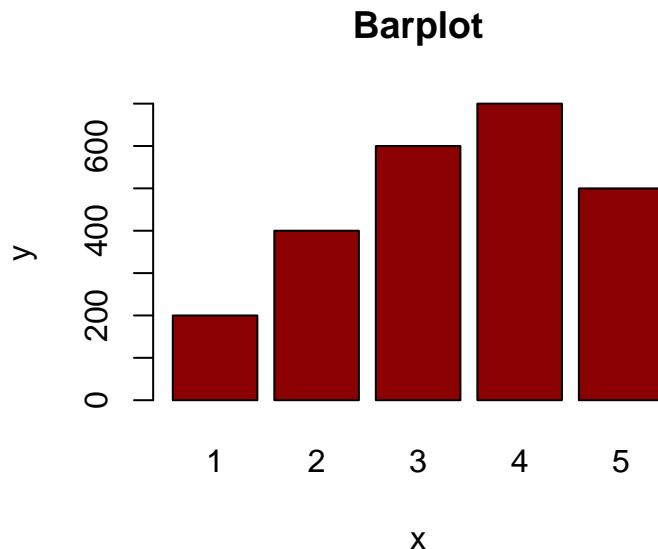
```
par(mfrow = c(2, 3))
plot(df$x, df$y, type = "l", lty=1, main = 'type = "l1"')
plot(df$x, df$y, type = "l", lty=2, main = 'type = "l2"')
plot(df$x, df$y, type = "l", lty=3, main = 'type = "l3"')
plot(df$x, df$y, type = "l", lty=4, main = 'type = "l4"')
plot(df$x, df$y, type = "l", lty=5, main = 'type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



3.1.6 Barplots o gráficas de barras

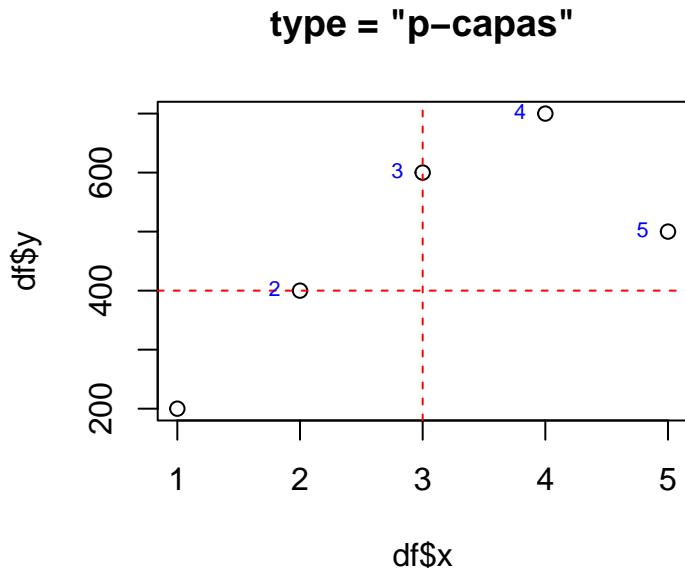
Las gráficas de barras nos permiten ver los valores de una manera diferente a las líneas:

```
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*](#), 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 repasarán 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.

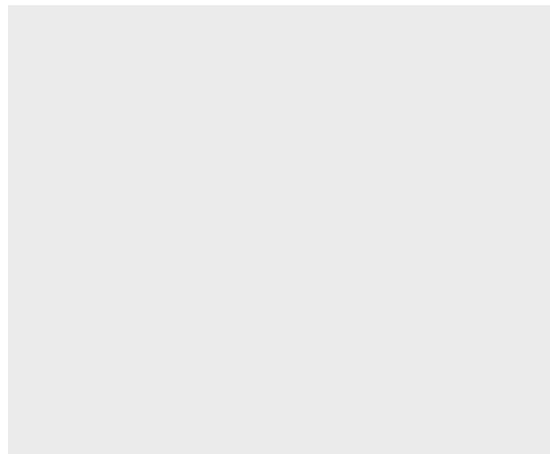
Para esta parte usaremos como ejemplo el set de Datos de “ToothGrowth” de nuevo y contruiremos por partes la gráfica.

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:

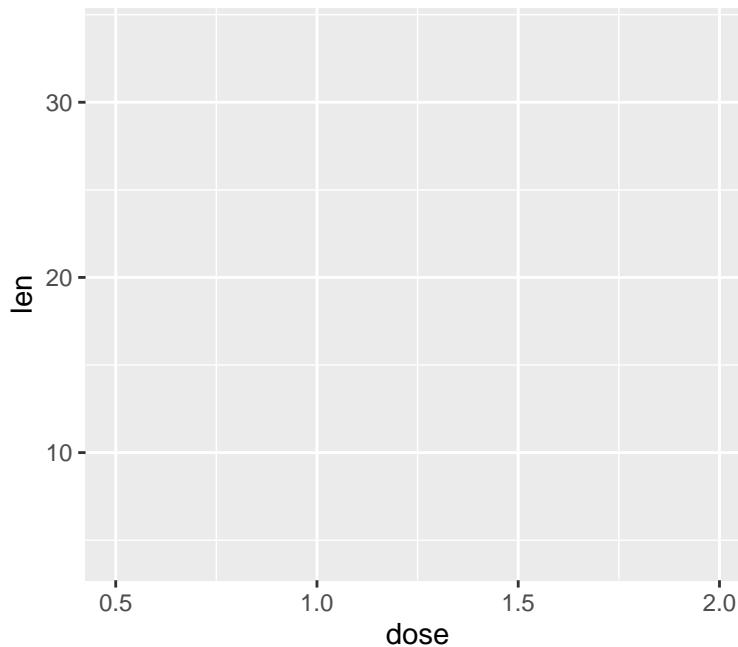
```
data("ToothGrowth")
library(dplyr)
library(ggplot2)
ggplot(data = ToothGrowth)
ToothGrowth %>% ggplot()
```



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))
```



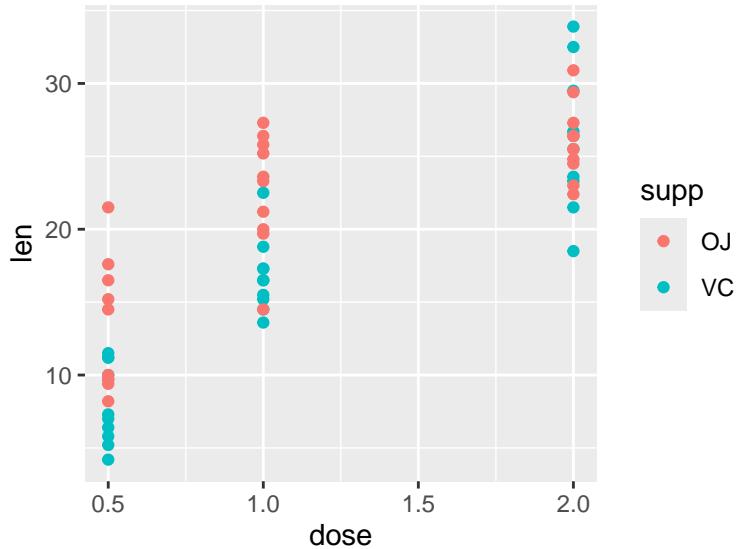
Aquí ya vemos que el eje x y el eje y ya aparecen con nombres de las variables, pero aún no nos muestra ningún gráfico, esto es porque no le indicamos aún qué geometría o qué tipo de gráfico queremos.

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`.

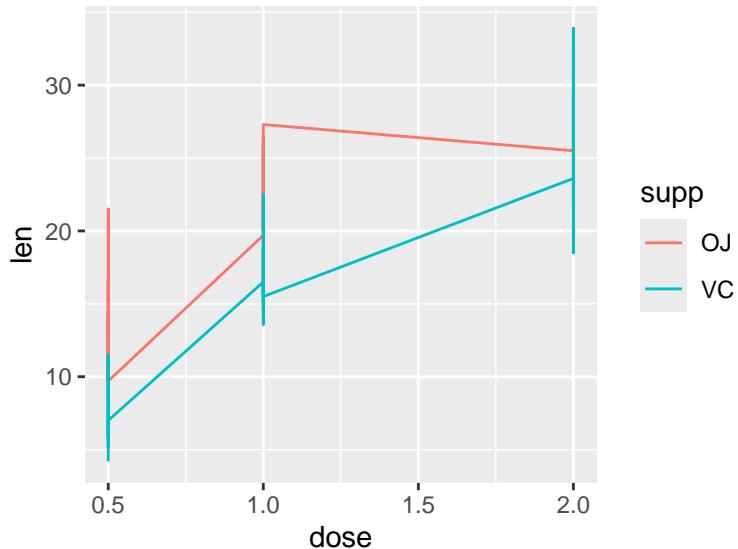
Digamos que queremos ahora sí dibujar la gráfica del paso anterior pero usando puntos, para esto usamos la geometría `geom_point`. Luego de cada capa de *ggplot2* en vez de usar pipe usamos el signo `+` y así vamos agregando cada capa.

```
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:

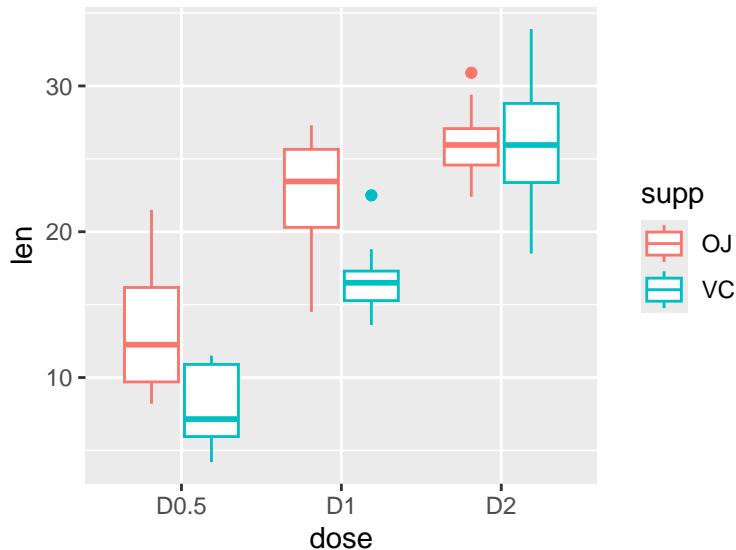
```
ToothGrowth %>% ggplot(aes(x = dose, y = len, color=supp)) +
  geom_line()
```



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

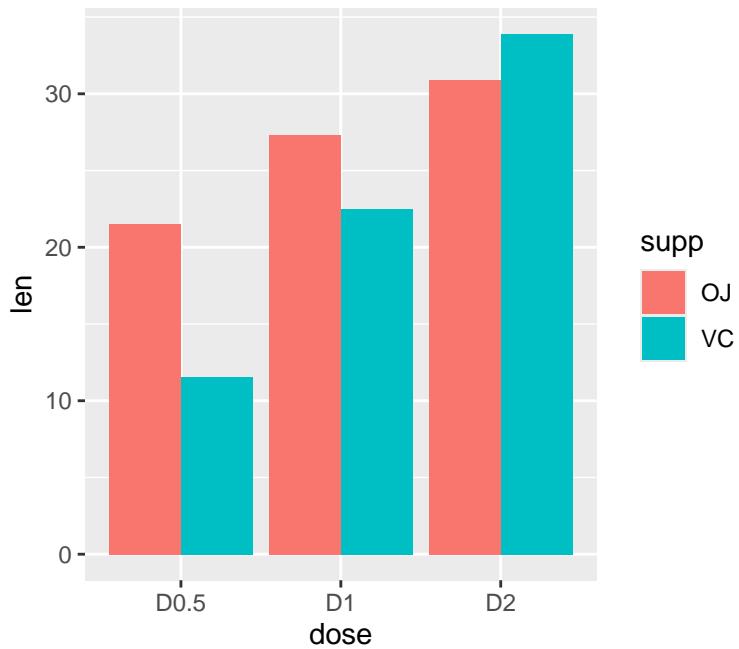
hacerla discreta en vez de numérica o continua.

```
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()
```



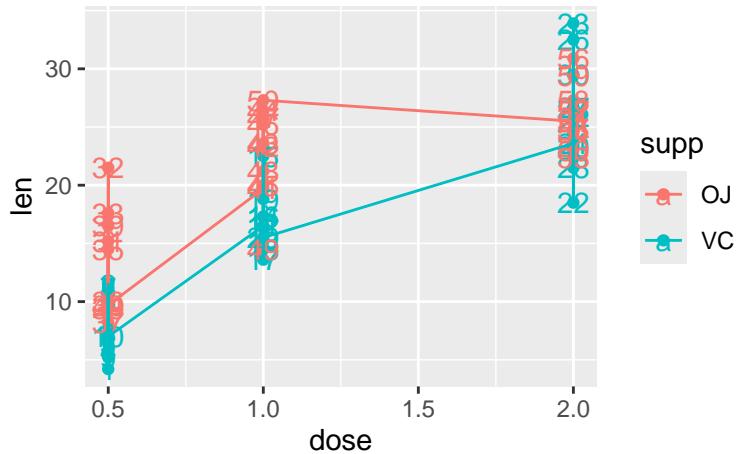
En el caso de *geom_bar* debemos cambiar y poner más argumentos, por ejemplo, como las barras son huecas en vez de usar ‘color’ usamos ‘fill’ y dentro de la geomtría escogemos una **stat** y una **position**, esto es para indicar cómo estarán las barras ubicadas y representadas. Los argumentos más usados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

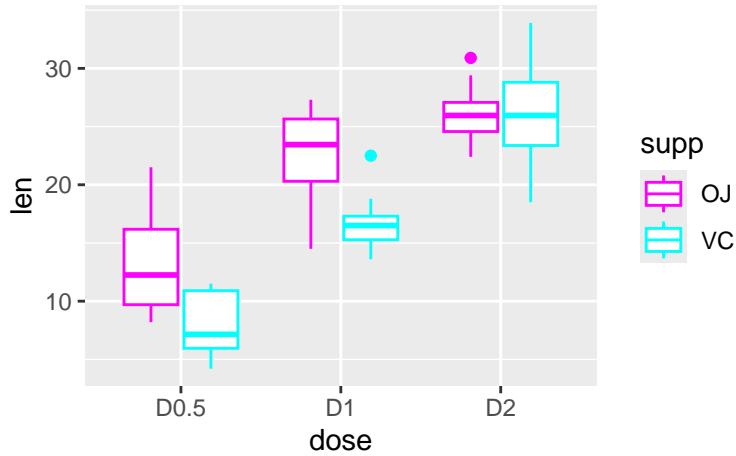
Todas estas opciones son depende de nuestro tipo de datos o del tipo de escala que queremos utilizar. Para declarar los colores en R podemos utilizar los nombres de los colores inglés como lo hicimos en la parte de R base o usando el formato de colores de html. También hay muchas paletas de varios paquetes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

Para buscar los colores en formato html podemos usar el siguiente link: [html-colors](#). El código html se ve de esta manera:

HTML HEX COLOR CODES				
Maroon #800000	Red #FF0000	Orange #FFA500	Yellow #FFFF00	Olive #808000
Purple #800080	Fuchsia #FF00FF	White #FFFFFF	Lime #00FF00	Green #008000
Navy #000080	Blue #0000FF	Aqua #00FFFF	Teal #008080	
Black #000000	Silver #C0C0C0	Gray #808080		

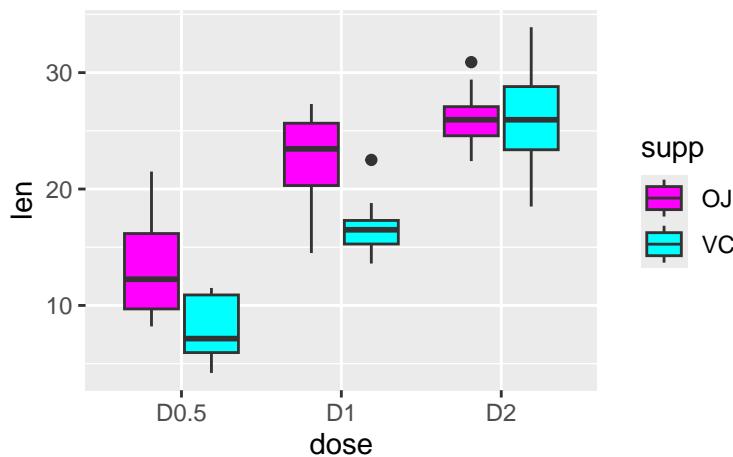
Y de esta manera los declaramos en nuestra gráfica usando `scale_color_manual()`:

```
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"))
```

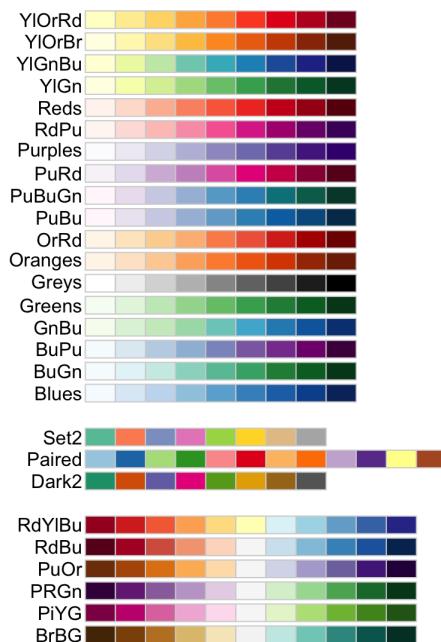


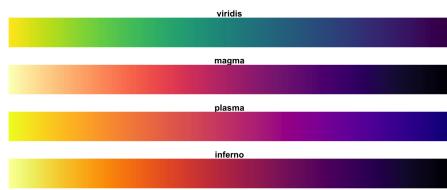
Ahora bien, como les mencioné anteriormente los diagramas de cajas y barras son elementos huecos entonces cuando indicados *col* lo que realmente coloreamos son los bordes, pero si queremos colorear las cajas adentro debemos usar *fill* como usamos en el *geom_bar()*, pero, en este caso en vez de usar *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"))
```



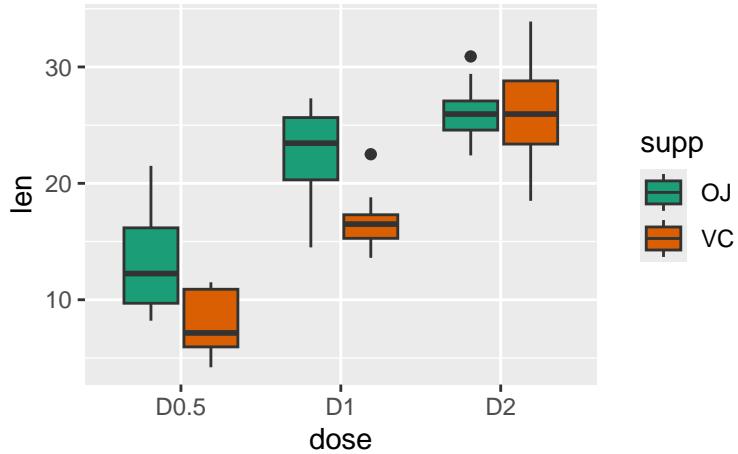
Ahora, si dudamos mucho o no estamos seguros de cómo escoger los colores, podemos utilizar paletas de colores ya determinadas como las que mencioné `RColorBrewer` y `viridis`. Estas son las paletas de `RColorBrewer` y `viridis`:



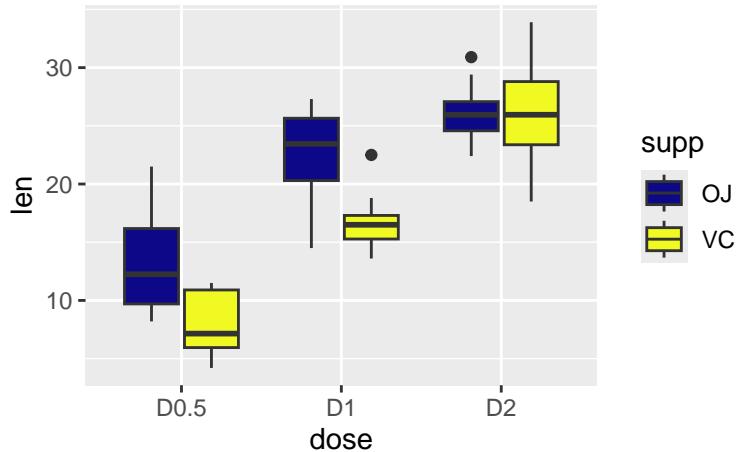


Y así las podemos usar:

```
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")
```



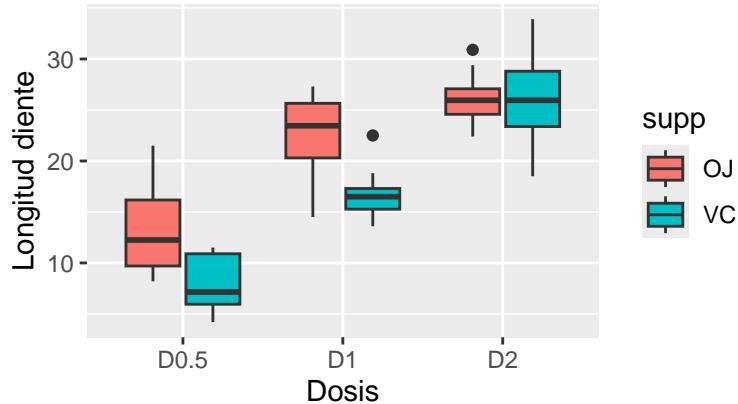
Entonces para *brewer* ponemos el nombre de la paleta que queremos usar y el *viridis* ponemos la opción de paleta que queremos: “magma” (o “A”), “inferno” (“B”), “plasma” (“C”), “viridis” (“D”, default) and “cividis” (o “E”).

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")
```

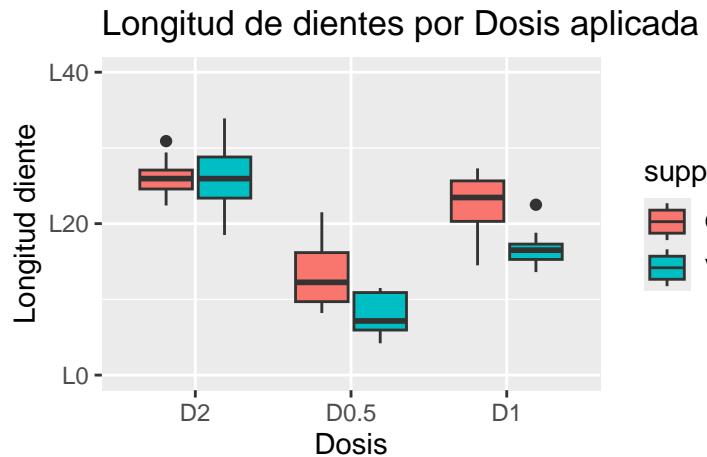
Longitud de dientes por Dosis aplicada



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 (continuos o numéricos y discretos o categóricos) en cada axis (x o y) y.

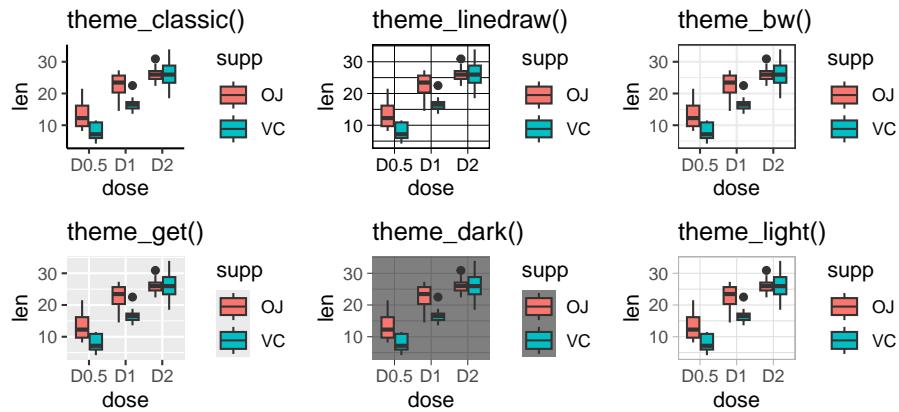
```
ToothGrowth %>% mutate(dose=case_when(
  dose==0.5~"D0.5",dose==1~"D1", dose==2~"D2")) %>% ggplot(
  aes(x = dose, y = len, fill=supp)) + geom_boxplot() +
  ylab("Longitud diente") +
  xlab("Dosis") +
  ggtitle("Longitud de dientes por Dosis aplicada") +
  scale_x_discrete(limits = c("D2", "D0.5", "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

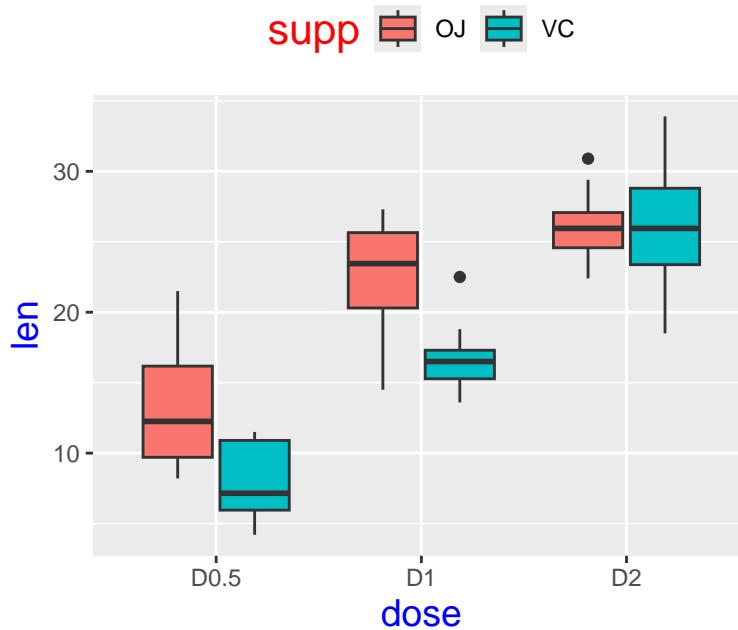
Hay otros ajustes más geneales o específicos que podemos realizar a nuestros gráficos. Por ejemplo, los temas. `ggplot2()` usa como *default* el tema `theme_grey()` pero hay otros que podemos usar para cambiar el formarto de nuestras figuras. Veamos algunos ejemplos:



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 `theme()` es todo un capítulo muy extenso para entrar en detalles, si queremos conocer muchos de estos detalles hacemos `?theme()` y nos despliega 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)¹
- cowplot (ggplot2)²
- grid.arrange(ggplot2)³
- 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.

¹<https://bookdown.org/ndphillips/YaRrr/arranging-plots-with-parmfrow-and-layout.html>

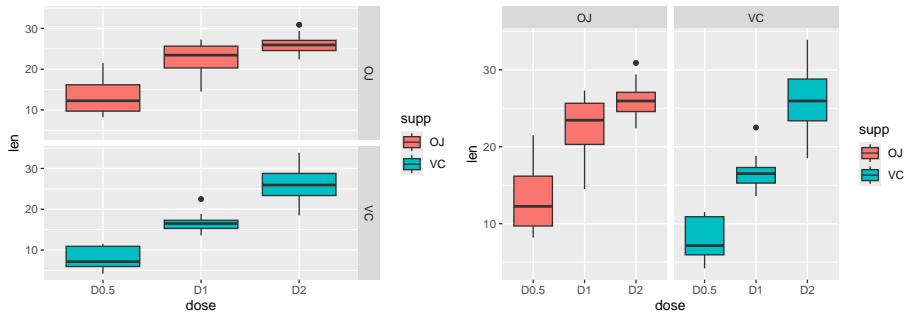
²<https://cran.r-project.org/web/packages/cowplot/vignettes/introduction.html>

³<https://cran.r-project.org/web/packages/egg/vignettes/Ecosystem.html>

Para facet_grid() hay dos opciones, presentarla por columnas o por filas:

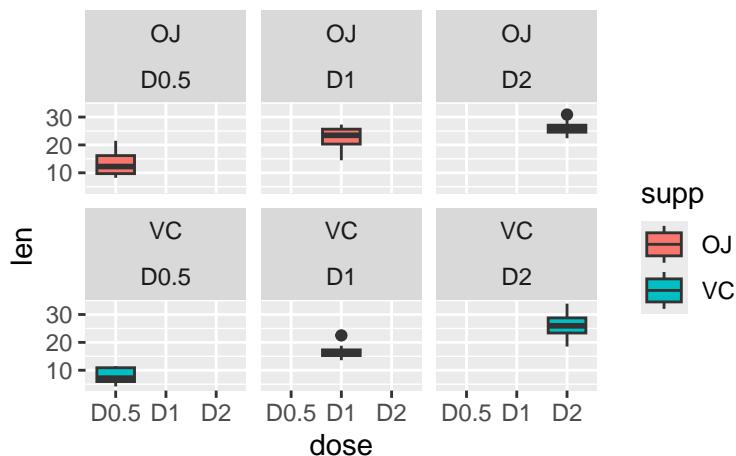
```
ToothGrowth %>% mutate(dose=case_when(
  dose==0.5~"D0.5",dose==1~"D1", dose==2~"D2")) %>% ggplot(
  aes(x = dose, y = len, fill=supp)) + geom_boxplot()+
  facet_grid(supp~.)
```

```
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()`

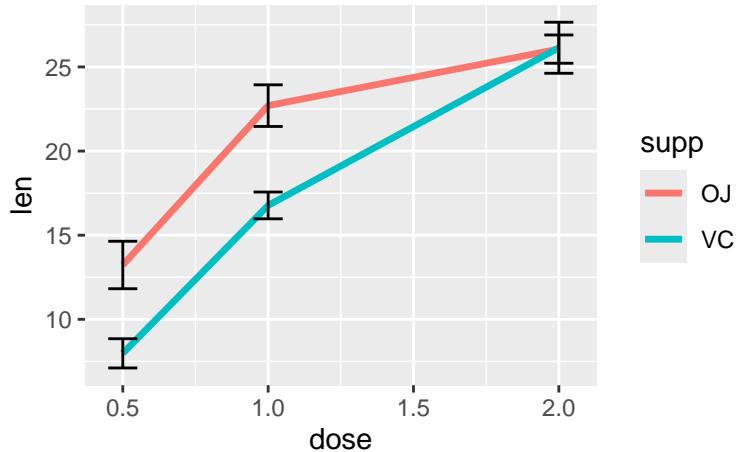
```
ggsave(filename = "plot.png", plot = a1, dpi = 300, width = 4, height = 3.5)
```

3.4 Extras

3.4.1 stat_summary() y stat_smooth()

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

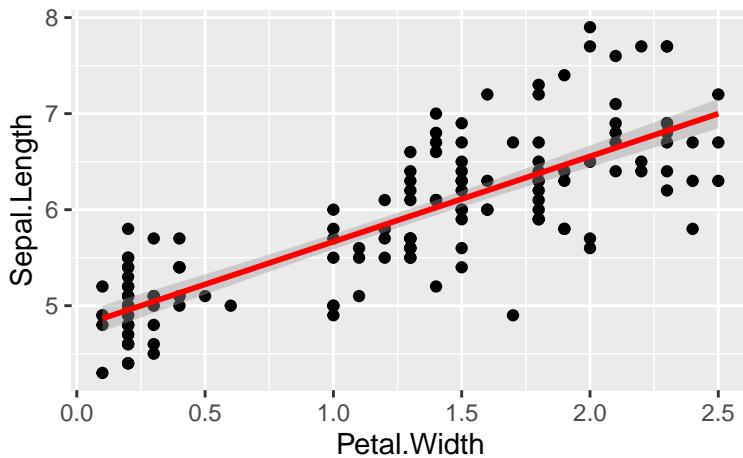
```
ToothGrowth %>%
  ggplot(aes(x = dose, y = len)) +
  stat_summary(geom = "line", fun = mean, aes(group = supp, color = supp), size = 1.2) +
  stat_summary(geom = "errorbar", fun.data = mean_se, aes(group = supp), width = 0.1)
```



`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'
```

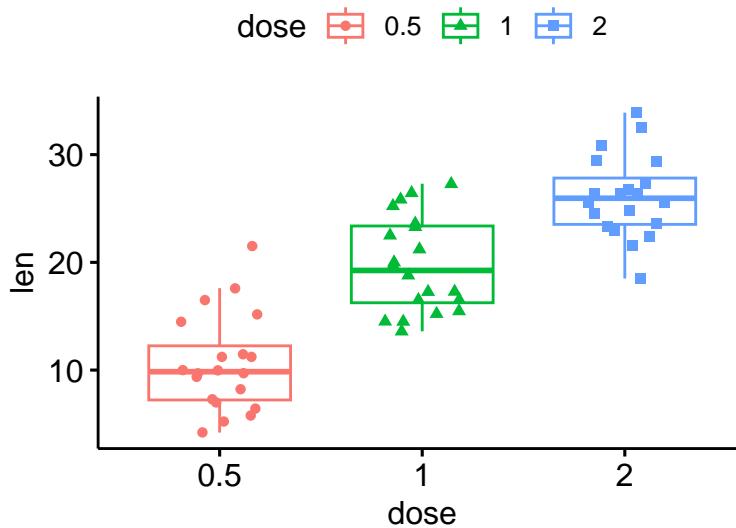


3.4.2 *ggbubr()*

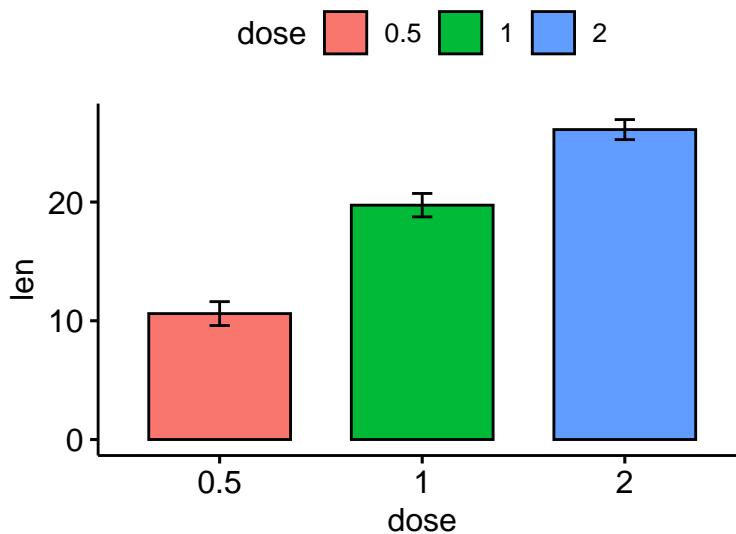
`ggbubr()` es una paquetería tipo compatible con `ggplot2()` pero un poco más fácil de declarar y más intuitivo.

Veamos algunos pocos ejemplos de sus utilidades:

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



```
ggbarpot(ToothGrowth, x = "dose", y = "len", fill = "dose", add = "mean_se")
```

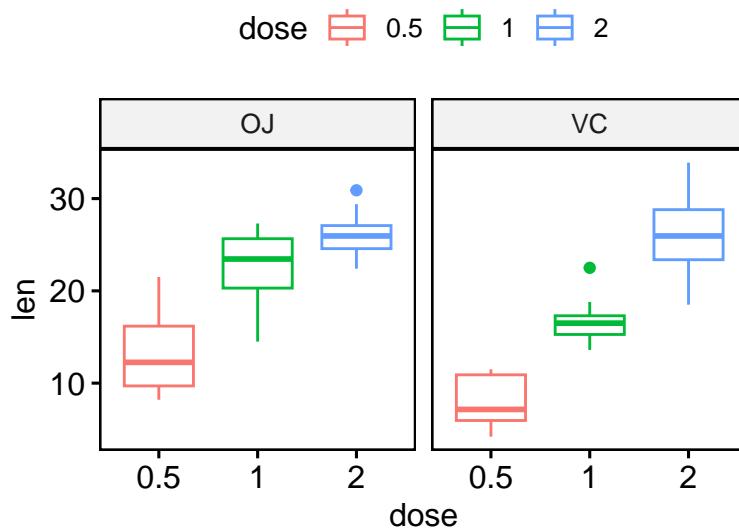


Vemos que los parámetros se definen similarmente pero con algunas diferencias, además que nos permite agregarles más elementos sin agregar más capas. Y el

'output' o tipo de gráfica es igual del tipo `ggplot2()`.

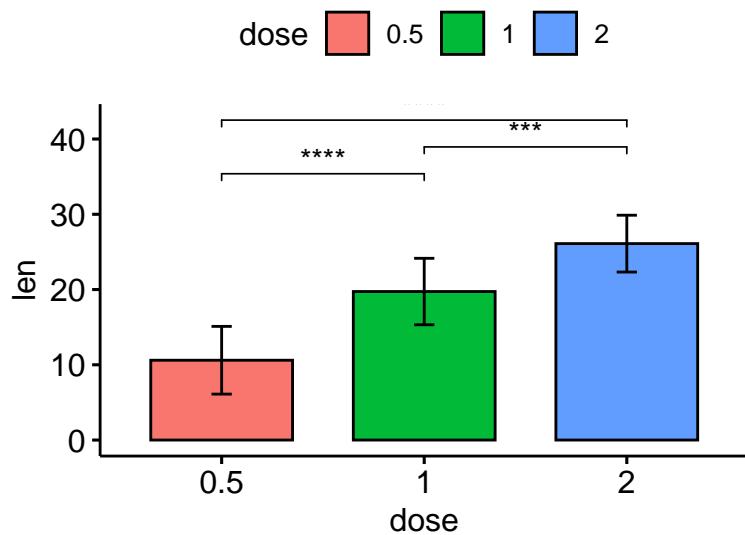
Otras funciones interesantes de este paquete son `facet.by` que es un argumento que puede ser declarado dentro de la función principal y nos permite hacer las gráficas multi-panel y otro que es muy útil es el de `stat_compare_means()` que nos permite agregar una capa que hace análisis estadísticos como `t.test`, `wilcoxon`, `kruskal.wallis`, `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") )

ggbargraph(ToothGrowth, x = "dose", y = "len", fill = "dose", add = "mean_sd") +
  stat_compare_means(comparisons = comparaciones, label = "p.signif")
```



3.4.3 Misceláneos: *ggcats()*⁴

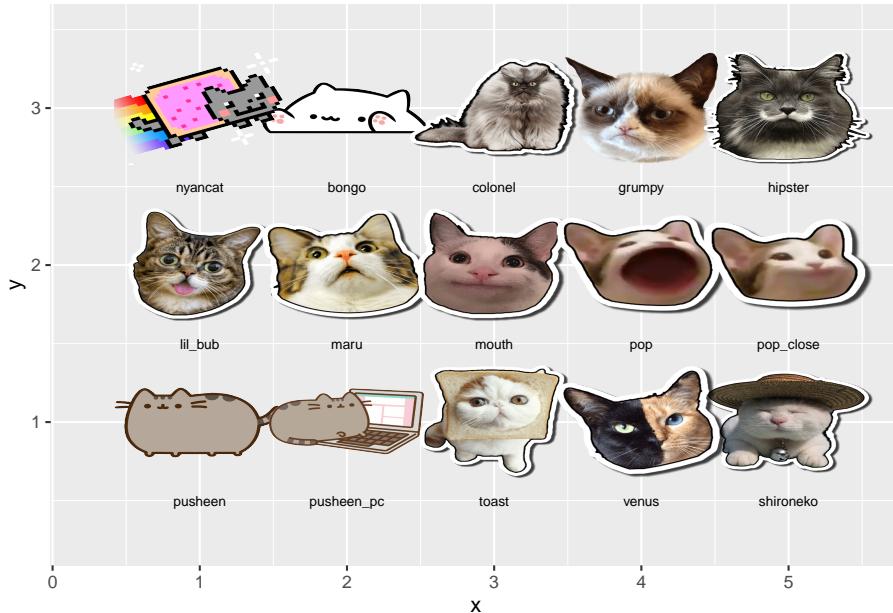
ggplot2() nos permite usar diversos paquetes desarrollados por muchos usuarios con fines divertidos o misceláneos, podemos verlo en el pie de página el link para explorar todos los que hay. Veremos *ggcats()*. Estas son las opciones que podemos usar:

```
# 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))
```

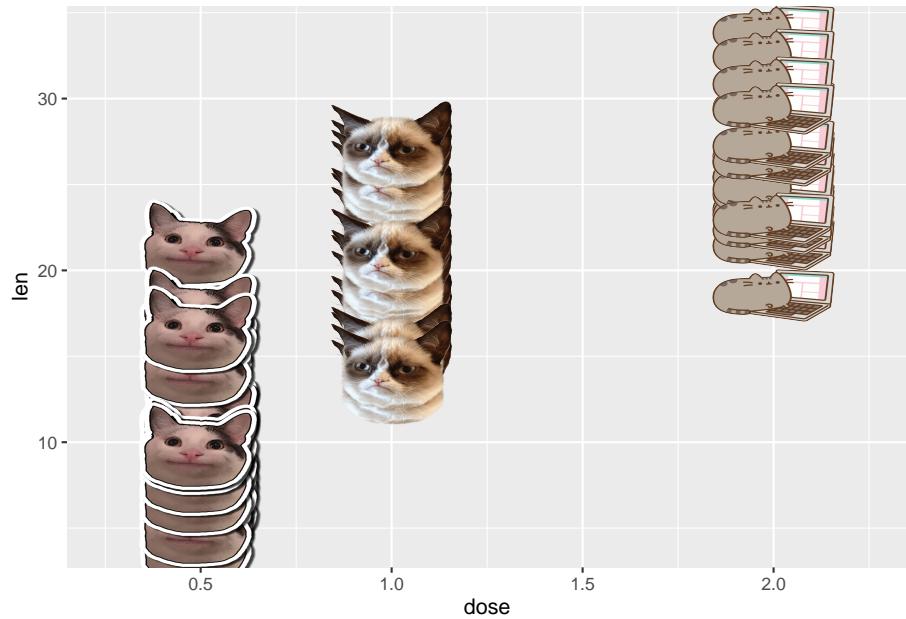
⁴<https://r-charts.com/es/misclanea/>



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))
ggttexttable(df, rows = NULL)
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa

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

```
ggttexttable(df, rows = NULL, theme = ttheme("blank"))
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa

```
ggttexttable(df, rows = NULL, theme = ttheme("light"))
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa

```
ggttexttable(df, rows = NULL, theme = ttheme("classic"))
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa

```
ggttexttable(df, rows = NULL, theme = ttheme("minimal"))
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa

```
ggttexttable(df, rows = NULL, theme = ttheme("lVioletWhite"))
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa

```
ggttexttable(df, rows = NULL, theme = ttheme("mVioletWhite"))
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa

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

Otros formateos:

- Poner en negrita una celda:

```
ggttexttable(df, rows = NULL, theme = ttheme("classic")) %>%
  table_cell_font(row = 3, column = 2, face = "bold", color = "red")
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa

- Resaltar una columna:

```
ggtexttable(df, rows = NULL, theme = ttheme("classic")) %>%  
  table_cell_bg(row = 2:5, column = 3, fill="yellow")
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa

- 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")
```

Data iris

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa

*Alguna nota

Chapter 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**

Chapter 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**
- 5.5 Estadística inferencial no paramétrica: supuestos principales de las pruebas no paramétricas**
- 5.6 ¿Cómo saber si mis datos presentan una distribución normal y homogeneidad de las variancias?:**
 - 5.6.1 Prueba de Shapiro-Wilk**
 - 5.6.2 Pruebas de ajuste o distribución: Prueba de Kolmogorov-Smirnov, Prueba de Anderson-Darling**
 - 5.6.3 Prueba de Leven (varianzas).**

Data

Para la revisión de los estadísticos básicos en R trabajaremos con el dataset `iris`.

```
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.



5.7 Estadísticos descriptivos

Utilizando la función `summary()` podemos obtener información sobre nuestra data, como el valor mínimo, máximo, el promedio, la mediana y el rango intercuantil.

```
summary(iris)
```

```

##   Largo_Sepalo    Ancho_Sepalo    Largo_Petalo    Ancho_Petalo      Especies
##  Min.   :4.300    Min.   :2.000    Min.   :1.000    Min.   :0.100    setosa     :50
##  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300    versicolor:50
##  Median :5.800   Median :3.000   Median :4.350   Median :1.300    virginica :50
##  Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
##  3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
##  Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500

```

Si queremos estos datos por aparte o solo nos interesa estos y otros datos de la variable “Largo_Sepalo”, entonces usamos las funciones establecidas en R:

```

mean(iris$Largo_Sepalo)

## [1] 5.843333

min(iris$Largo_Sepalo)

## [1] 4.3

max(iris$Largo_Sepalo)

## [1] 7.9

median(iris$Largo_Sepalo)

## [1] 5.8

quantile(iris$Largo_Sepalo, 0.25) # primer cuantil

## 25%
## 5.1

quantile(iris$Largo_Sepalo, 0.75) # tercer cuantil

## 75%
## 6.4

```

Otros estadísticos descriptivos...

```

sd(iris$Largo_Sepalo) #desviación estándar

## [1] 0.8280661

range(iris$Largo_Sepalo) #min y max

## [1] 4.3 7.9

IQR(iris$Largo_Sepalo) #diferencia entre el tercer y primer cuantil

```

```
## [1] 1.3
var(iris$Largo_Sepalo) #varianza
## [1] 0.6856935
sd(iris$Largo_Sepalo) / mean(iris$Largo_Sepalo) #Coeficiente variación
## [1] 0.1417113
```

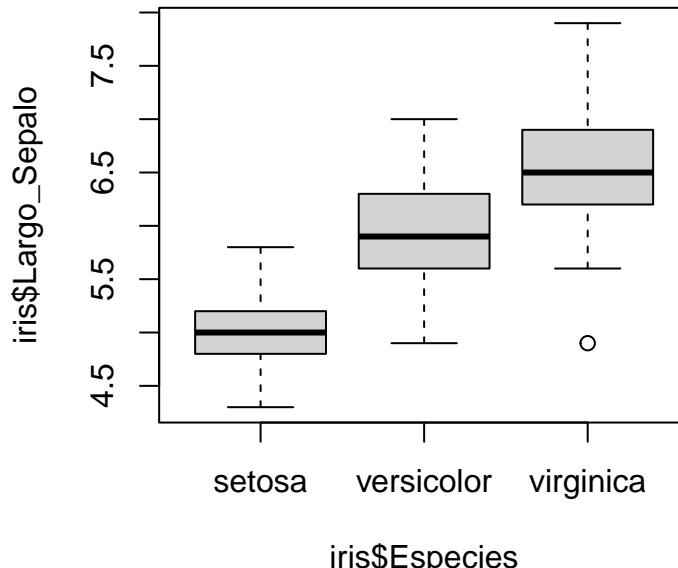
Para conocer la desviación estándar de todas las columnas numéricas, usamos la función apply como anteriormente vimos:

```
lapply(iris[, 1:4], sd)
## $Largo_Sepalo
## [1] 0.8280661
##
## $Ancho_Sepalo
## [1] 0.4358663
##
## $Largo_Petalo
## [1] 1.765298
##
## $Ancho_Petalo
## [1] 0.7622377
```

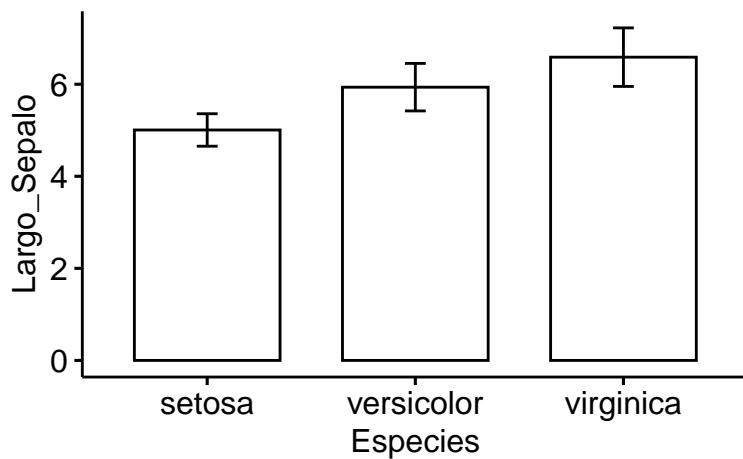
5.8 Gráficos descriptivos

Si queremos explorar cómo es la variación de la longitud del sepalo por cada especie:

```
boxplot(iris$Largo_Sepalo ~ iris$Especies)
```



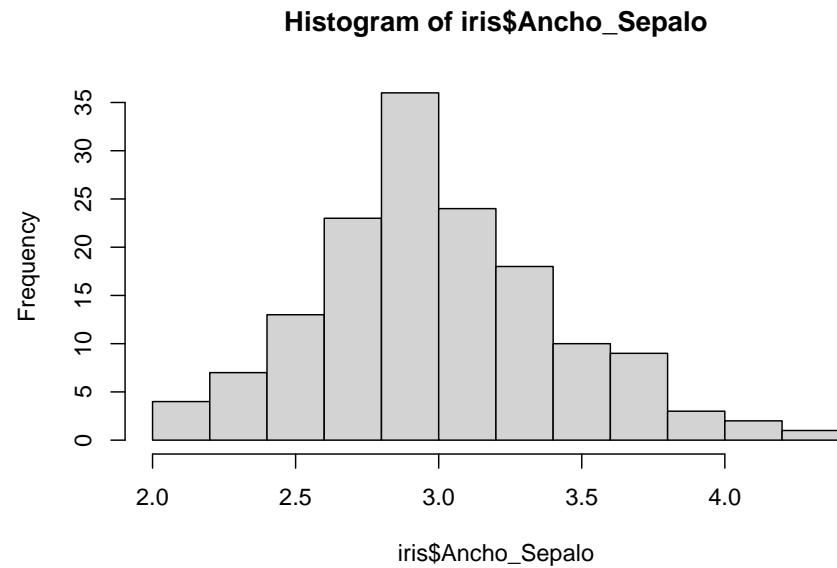
```
library(ggpubr)
ggbarplot(data = iris, x = "Especies", y = "Largo_Sepalo", add = "mean_sd")
```



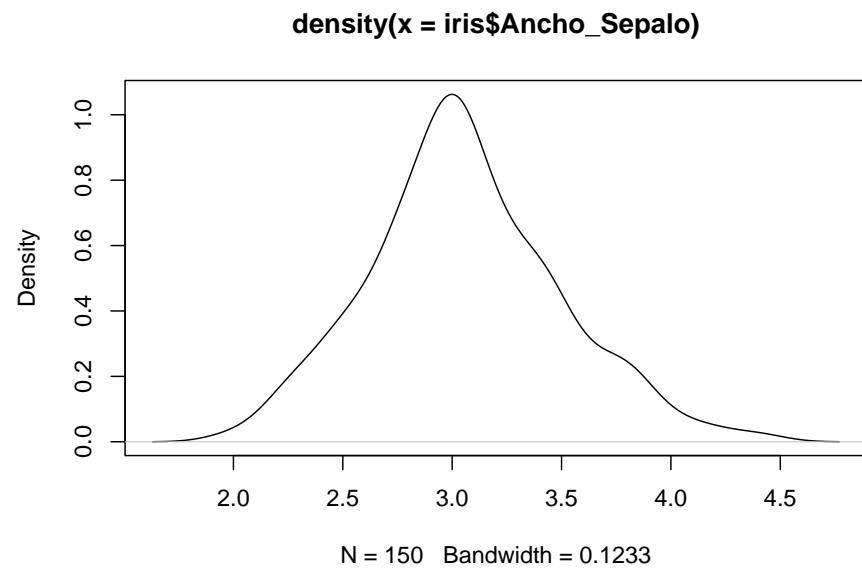
5.9 Explorando normalidad en los datos

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 si 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. “

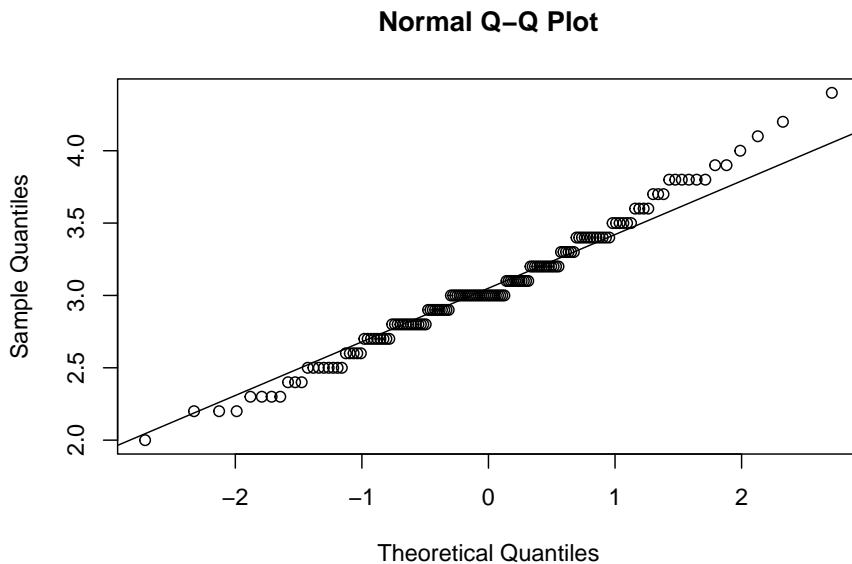
```
hist(iris$Ancho_Sepalo)
```



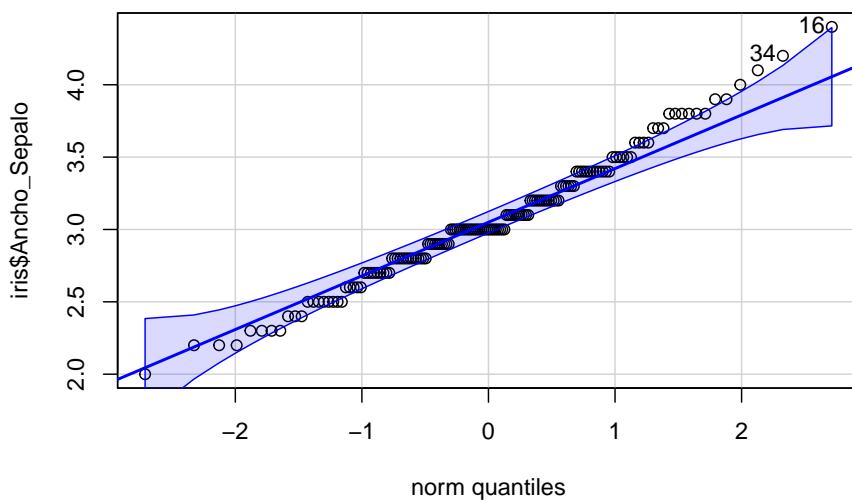
```
plot(density(iris$Ancho_Sepalo))
```



```
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.

5.10 Correlación y Regresión

Correlación: Describe cómo dos variables están relacionadas. Es una herramienta común para describir relaciones simples sin hacer afirmaciones sobre causa y efecto. Estas relaciones pueden ser o no lineales, pero usualmente se busca o se desea saber si esta relación es lineal.

Por ejemplo queremos saber si existe una relación entre el largo y el ancho del pétalo de estas flores sin importar la especie:

```
cor.test(iris$Largo_Petalo, iris$Ancho_Petalo)
```

```
##
## Pearson's product-moment correlation
##
## data: iris$Largo_Petalo and iris$Ancho_Petalo
## t = 43.387, df = 148, p-value < 2.2e-16
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
## 0.9490525 0.9729853
## sample estimates:
##      cor
## 0.9628654
```

Los valores que más nos interesan aquí son el *cor* y el *p-value*: Para un valor de cor de +1 quiere decir que dos variables están perfectamente correlacionadas

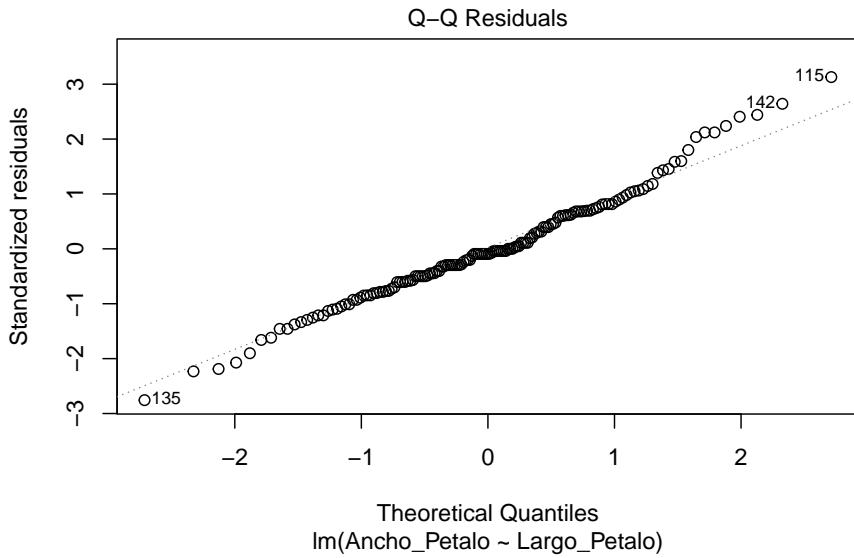
positivamente. Es decir, al aumentar una, aumenta la otra. Un valor de -1 significa que las dos variables están perfectamente relacionadas negativamente, es decir, mientras una aumenta, la otra disminuye en la misma medida. Un valor 0 significa que no hay correlación en las dos variables. El valor p es la probabilidad de obtener un valor de cor más que extremo que el cor observado, dado los grados de libertad y si cor fuera 0. Igualmente valores menores a 0.05 son significativos. Hay otra función que nos permite obtener solo el coeficiente de correlación, de manera más práctica:

```
cor(iris$Largo_Petalo, iris$Ancho_Petalo)          # si los datos son normales
## [1] 0.9628654

cor(iris$Largo_Petalo, iris$Ancho_Petalo, method = "spearman")    #si los datos no son normales
## [1] 0.9376668
```

Ahora bien, ya sabemos que estas dos variables están correlacionadas positivamente, así que si quiero construir un modelo que permite predecir valores en base a otros no medidos podemos aplicar una regresión lineal. Una regresión lineal es un modelo lineal que describe la ecuación de dos variables de interés definidas en una función lineal: $y = ax + b$, donde a es la pendiente y b el intercepto. La regresión lineal debe aplicarse sobre datos normales. Así que chequemos la normalidad. Primero, construimos el modelo y luego graficamos. El modelo se construye con la variable de respuesta al lado izquierdo de la ecuación y la variable que la explique a la derecha divididos por un signo de “~”.

```
modelo <- lm(Ancho_Petalo ~ Largo_Petalo, data = iris)
plot(modelo, which = 2)
```



Son solo pocos puntos que se salen de la gráfica, así que asumimos normalidad.

Exploraremos el modelo:

```
summary(modelo)
```

```
## 
## Call:
## lm(formula = Ancho_Petalo ~ Largo_Petalo, data = iris)
## 
## Residuals:
##      Min       1Q   Median       3Q      Max 
## -0.56515 -0.12358 -0.01898  0.13288  0.64272 
## 
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept) -0.363076  0.039762 -9.131  4.7e-16 ***
## Largo_Petalo  0.415755  0.009582 43.387 < 2e-16 ***
## ---      
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1 
## 
## Residual standard error: 0.2065 on 148 degrees of freedom
## Multiple R-squared:  0.9271, Adjusted R-squared:  0.9266 
## F-statistic: 1882 on 1 and 148 DF,  p-value: < 2.2e-16
```

Con `summary()` podemos ver los coeficientes de la ecuación, en este caso son: para el intercepto -0.36 y para la pendiente es 0.41. De nuevo los valores p están

por debajo de 0.05. Los coeficientes son la pendiente y el intercepto. Así que la ecuación queda -> Ancho_Petalo = Largo_Petalo*0.4157 - 0.3630

Otro resultado importante es el R cuadrado que nos dice la bondad del ajuste del modelo, esto es la fracción de mis datos que es explicado por el modelo en este caso si miramos el valor ajustado, el modelo explica el 92% de mis datos.

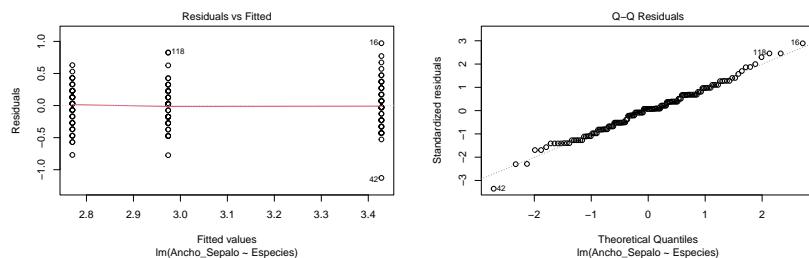
5.11 ANOVA (1 vía)

ANOVA o análisis de varianza es un método estadístico que nos permite comparar las varianzas entre las medias (promedios) de diferentes grupos.

El ANOVA tiene varios supuestos: 1. Independencia: cada observación es independiente de otra. (Por ejemplo, si tenemos mediciones del mismo individuo a lo largo del tiempo, esta medida es dependiente al individuo). 2. Normalidad : Que los datos siguen una distribución normal (como verificamos anteriormente). 3. Homocedasticidad: varianzas equivalentes entre grupos.

Para la condición 1, en ningún lado nos dice que son muestras longitudinales, es decir, del mismo individuo a lo largo del tiempo, así que asumimos independencia. Vamos a ver con estas dos gráficas si efectivamente se cumplen las condiciones 2 y 3.

```
modelo_ancho <- lm(Ancho_Sepalo ~ Especies, data = iris)
plot(modelo_ancho, which = c(1,2) )
```



Este gráfico muestra si los residuos tienen patrones no lineales. Si encuentra residuos igualmente distribuidos alrededor de una línea horizontal sin patrones distintos, es una buena indicación de que no tiene relaciones no lineales. La linea roja debe ser más o menos recta, no debe estar curvada, entre más recta mejor.

```
anova(modelo_ancho)
```

```
## Analysis of Variance Table
##
## Response: Ancho_Sepalo
##              Df Sum Sq Mean Sq F value    Pr(>F)
## Especies     2 11.345  5.6725  49.16 < 2.2e-16 ***
##
```

```
## Residuals 147 16.962 0.1154
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

En un sentido aplicado el número que más nos interesa es el valor F, sin embargo, se ha extendido la importancia del valor p que se define como la probabilidad de encontrar valores F más extremos que el observado y en este sentido la probabilidad es muy baja, mucho menor que el valor establecido como umbral que suele ser 0.05. Así que rechazamos la hipótesis nula, lo que quiere decir que las medias de las especies son diferentes para el ancho del sépalo.

5.12 Prueba de Tukey

ANOVA nos dice que hay diferencias en el ancho del sepalo por especie, pero no nos dice cual es más grande o cuales menor, o cual es diferente a cual. Para esto hacemos una prueba de Tukey. La función `aov()` realiza lo mismo que la de `anova()`.

```
fm1<- aov(modelo_ancho)
TukeyHSD(fm1, "Especies", ordered = TRUE)

## Tukey multiple comparisons of means
## 95% family-wise confidence level
## factor levels have been ordered
##
## Fit: aov(formula = modelo_ancho)
##
## $Especies
##          diff      lwr      upr      p adj
## virginica-versicolor 0.204 0.04314472 0.3648553 0.0087802
## setosa-versicolor     0.658 0.49714472 0.8188553 0.0000000
## setosa-virginica      0.454 0.29314472 0.6148553 0.0000000
```

```
library(agricolae)
HSD.test(fm1, "Especies", group = TRUE, console = TRUE)

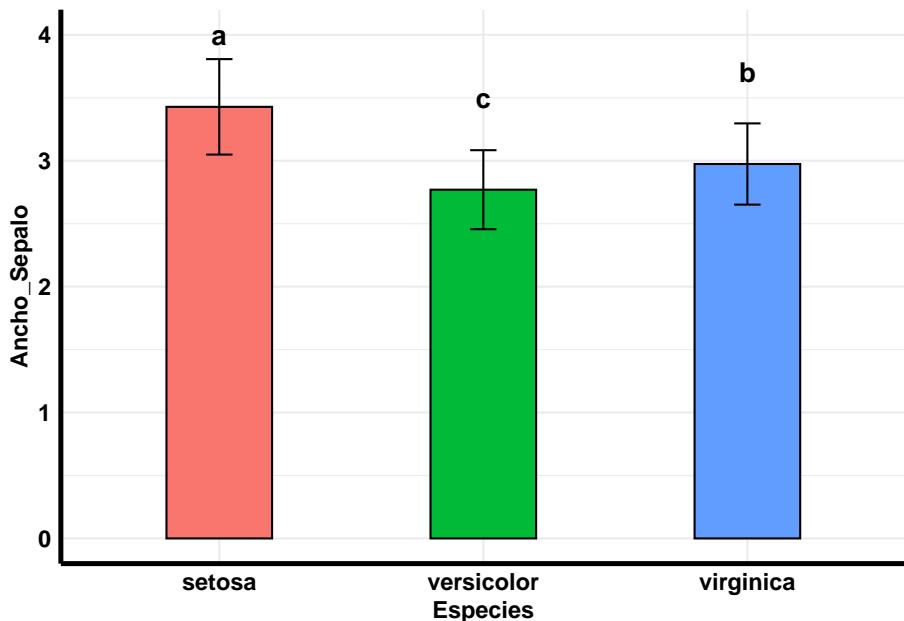
##
## Study: fm1 ~ "Especies"
##
## HSD Test for Ancho_Sepalo
##
## Mean Square Error: 0.1153878
##
```

```

## Especies, means
##
##          Ancho_Sepalo      std   r       se Min Max   Q25 Q50   Q75
## setosa      3.428 0.3790644 50 0.0480391 2.3 4.4 3.200 3.4 3.675
## versicolor  2.770 0.3137983 50 0.0480391 2.0 3.4 2.525 2.8 3.000
## virginica   2.974 0.3224966 50 0.0480391 2.2 3.8 2.800 3.0 3.175
##
## Alpha: 0.05 ; DF Error: 147
## Critical Value of Studentized Range: 3.348424
##
## Minimun Significant Difference: 0.1608553
##
## Treatments with the same letter are not significantly different.
##
##          Ancho_Sepalo groups
## setosa      3.428     a
## virginica   2.974     b
## versicolor  2.770     c

```

Esta función aparte de ver las diferencias de medias nos ordena con letras cual es la mayor y cual es la menor, veamoslo mejor en una gráfica de barras.



5.13 ANOVA (2 vías)

El análisis de varianza de dos vías nos ayuda a estudiar la relación entre una variable dependiente cuantitativa y dos variables independientes cualitativas (factores) cada uno con varios niveles. Este método hace todas las asunciones o supuestos que el ANOVA de 1 vía sobre normalidad y demás. También es importante recalcar que para este tipo los grupos deben tener el mismo número de muestras o réplicas. El ANOVA de dos vías permite estudiar cómo influyen por si solos cada uno de los factores sobre la variable dependiente (modelo aditivo) así como la influencia de las combinaciones que se pueden dar entre ellas (modelo con interacción).

- Modelo aditivo: `aov(variable_respuesta ~ factor1 + factor2, data)`
- Modelo con interacción: `aov(variable_respuesta ~ factor1 x factor2, data)`
Para este ejemplo usaremos el set de datos ‘ToothGrowth’ con el que trabajamos la clase pasada.

```
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 ...
```

Como podemos observar la variable de respuesta aquí sería la longitud de los dientes y los factores a evaluar son ‘*supp*’ y ‘*dose*’, *supp* es la forma en que le dieron la vitamina C a los cerdos, si como OJ (jugo de naranja) o AS (ácido ascórbico) a diferentes dosis (*dose*) de 0.5, 1 y 2 mg/día. El factor ‘*dose*’ o dosis no aparece como factor sino como variable numérica. Esto puede ser un inconveniente al correr el ANOVA así que modificaremos esto en la data.

```
ToothGrowth$dose <- factor(ToothGrowth$dose,
                           levels = c(0.5, 1, 2),
                           labels = c("D0.5", "D1", "D2"))

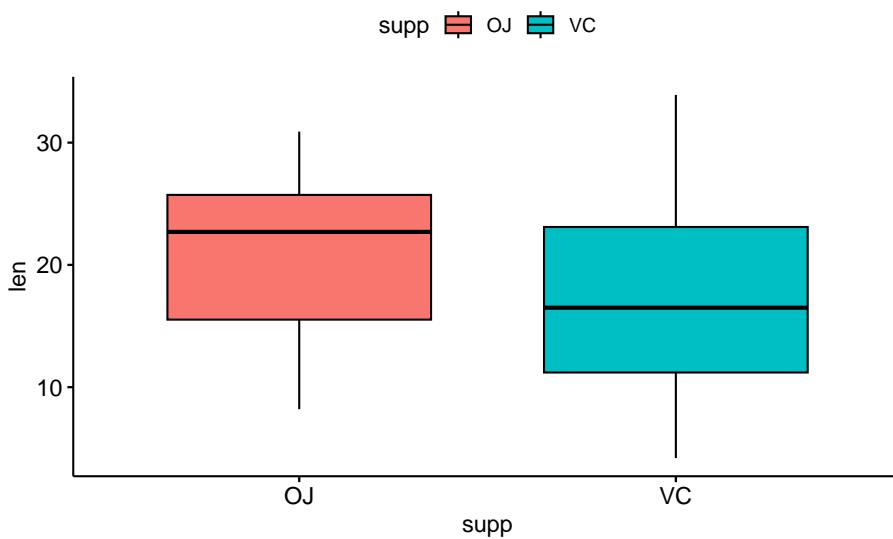
head(ToothGrowth)

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

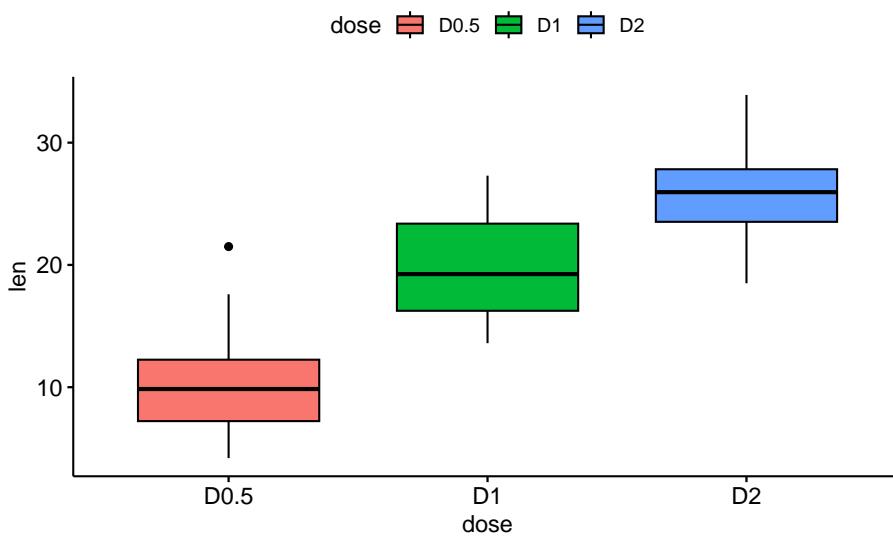
Bien, visualizaremos nuestros datos para ver las tendencias de nuestros factores sobre nuestra variable de respuesta:

Boxplots :

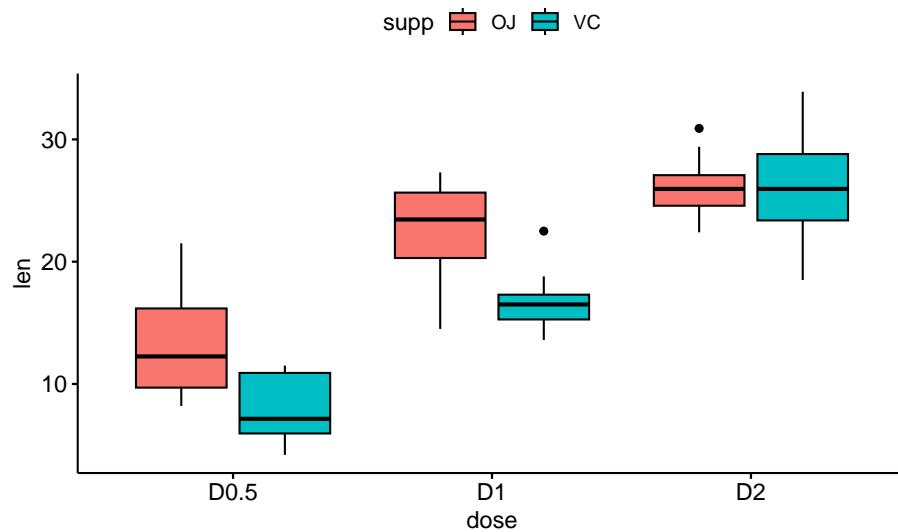
```
library(ggpubr)
ggboxplot(data = ToothGrowth, x = "supp", y = "len", fill = "supp")
```



```
ggboxplot(data = ToothGrowth, x = "dose", y = "len", fill = "dose")
```

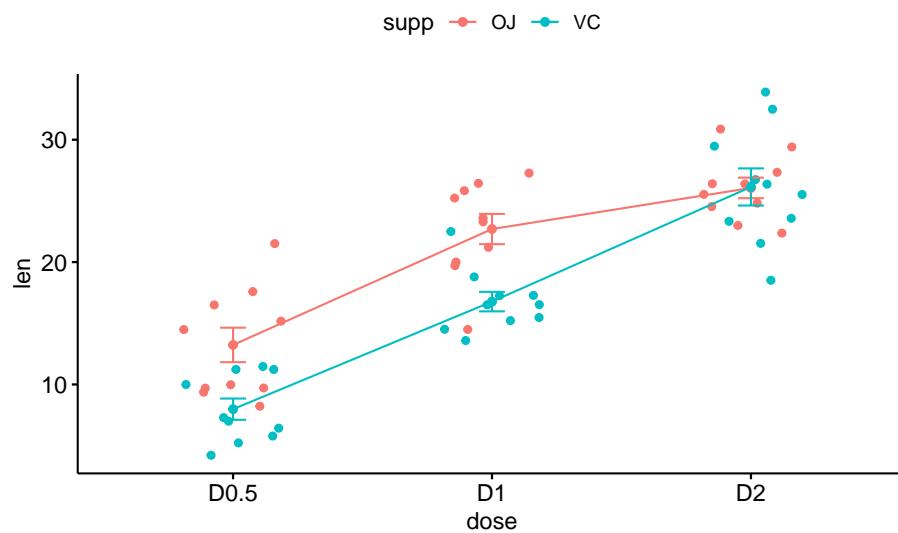


```
ggboxplot(data = ToothGrowth, x = "dose", y = "len", fill = "supp")
```



Líneas:

```
gglime(ToothGrowth, x = "dose", y = "len", color = "supp", add = c("mean_se", "jitter"))
```



Como pudimos ver aparentemente los factores podrían tener una interacción aunque muy leve, en este caso podríamos correr una ANOVA dos vías o bien

aditivo o bien multiplicativo si queremos confirmar esta pequeña interacción.

```
anova1<- aov(len ~ supp + dose, data = ToothGrowth)
anova2<- aov(len ~ supp * dose, data = ToothGrowth)

summary(anova1)

##           Df Sum Sq Mean Sq F value    Pr(>F)
## supp        1 205.4   205.4  14.02 0.000429 ***
## dose        2 2426.4  1213.2  82.81 < 2e-16 ***
## Residuals  56  820.4    14.7
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

summary(anova2)

##           Df Sum Sq Mean Sq F value    Pr(>F)
## supp        1 205.4   205.4  15.572 0.000231 ***
## dose        2 2426.4  1213.2  92.000 < 2e-16 ***
## supp:dose   2 108.3    54.2   4.107 0.021860 *
## Residuals  54  712.1    13.2
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Como notamos tanto el modelo aditivo como multiplicativo, los factores explican de buena manera la varianza de nuestra variable. Sin embargo en el modelo multiplicativo, que es el que nos muestra la interacción, nos dice que efectivamente hay una interacción entre nuestros factores (aunque muy pequeña), es decir, que el efecto de un factor depende del otro factor.

Las pruebas que realizamos anteriormente son de las comunes aplicadas para los conjuntos de datos, pero hay otras que podemos realizar también, por ejemplo en el caso de datos normales para hacer pruebas pareadas (dos niveles) podemos utilizar tambien `t.test` en caso de no ser normales `wilcoxon.test`, en el caso de mas niveles para no parametricas podemos usar `kruskal.test`. Por ejemplo:

```
#dos niveles
t.test(len ~ supp, data = ToothGrowth, paired = TRUE)
wilcox.test(len ~ supp, data = ToothGrowth, paired = TRUE)

#más de dos niveles
kruskal.test(len ~ dose, data = ToothGrowth)
```

Hay otras pruebas que se aplican dependiendo de los datos que tengamos y el objetivo de nuestro pregunta de investigación por ejemplo Dunn.test, Dun-

can.test, Welch.test, entre otros. Hay muchos paquetes además de los que trae por default R stats (que viene por default con R) tales como **agricolae**, **vegan**, **emmeans**, entre otros, que vienen con más funciones y pruebas aplicadas a datos biológicos, ecológicos, entre otros.

Chapter 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.**

Chapter 7

Tipos de pruebas no paramétricas

- 7.1 Prueba de Mann-Whitney U (Wilcoxon rank-sum test)**
- 7.2 Prueba de Wilcoxon para muestras relacionadas**
- 7.3 Prueba de Kruskal-Wallis**
- 7.4 Prueba de Friedman**
- 7.5 Prueba de Chi-cuadrado**
- 7.6 Prueba de McNemar**
- 7.7 Correlación de rango de Spearman**
- 7.8 Correlación de Kendall**

Chapter 8

Análisis de riqueza y diversidad alfa y beta

Chapter 9

Análisis multivariados

- 9.1 PCA
- 9.2 PCoA
- 9.3 NMDS
- 9.4 Permanova
- 9.5 permdis
- 9.6 Anosim
- 9.7 RDA
- 9.8 CCA

Chapter 10

Modelización estadística:

- 10.1 Modelo lineal general
- 10.2 GLM con distribución Poisson
- 10.3 GLM con distribución quasi-Poisson
- 10.4 GLM con distribución de Bernoulli (Binomial)
- 10.5 GLMM: modelos lineales generales mixtos