Hands-on! Introducción al análisis de datos con R

12 de febrero de 2026

Beatriz Fernández Blanco
Maria Guaita Céspedes

Contents

1. ¿Por qué R?

1.1 Un poco de historia...

Conocer cómo nació R es interesante para comprender sus características. R es un lenguaje de programación creado por Ross Ihaka y Robert Gentleman en los años 90, que eran estadísticos en la Universidad de Auckland (Nueva Zelanda) y querían crear un material mejor para dar el curso de introducción a la estadística a sus alumnos. Así, crearon el lenguaje R, basado en el lenguaje S. R siempre ha sido un proyecto de uso libre desde junio de 1995. El hecho de que fuera creado por estadísticos y no por ingenieros para el desarrollo de software como pasa con otros lenguajes de programación como C y Java, explica que uno de sus puntos fuertes sea la **interactividad** y la **visualización** de los datos.

Figure 1: Logo de R y sus creadores

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. Este equipo mantiene la versión ***base*** de R que, como su nombre indica, es sobre la cual se crean otras implementaciones de R así como los paquetes que expanden su funcionalidad.

Referencias:

- https://bookdown.org/jboscomendoza/r-principiantes4/un-poco-de-historia.html
- http://rafalab.dfci.harvard.edu/dsbook/getting-started.html#fn1
- Ihaka, R., & Gentleman, R. (1996). R: a language for data analysis and graphics. Journal of computational and graphical statistics, 5(3), 299-314.

1.2 Ventajas de usar R

En esta sesión vamos a conocer R como una herramienta para visualizar y analizar datos. Probablemente a estas alturas hayáis empleado algunos programas para el análisis de datos, como puede ser Excel para hacer gráficos y SPSS o Graphpad para la estadística. Veamos algunas ventajas que nos proporciona R:

- Es de uso libre (forma parte del sistema GNU)
- Se puede usar en diferentes sistemas operativos: Windows, Linux, Mac.
- Hay una gran comunidad de usuarios de R activa y en continuo crecimiento y, por lo tanto, hay muchos recursos para aprender y resolver dudas.
- Ofrece muchas utilidades en el análisis de datos, en especial en cuanto a interactividad y visualización.
- En general, conocer un lenguaje de programación os facilitará el análisis de datos, sea cuál sea vuestro ámbito de especialización. Además, mejora otras habilidades transversales, como es el pensamiento lógico.

2. Utilizar R en RStudio

En esta sesión trabajaremos en RStudio en la nube a través de Posit Cloud. Únicamente necesitamos registrarnos con una cuenta de correo electrónico. Si bien es práctico para los objetivos de esta sesión, tened en cuenta que Posit Cloud tiene unos recursos limitados en la versión libre, de los cuales destaca que solo permite 25 horas de uso al mes. Lo ideal es trabajar en local, por lo que os dejamos las instrucciones.

2.1 Instalación de R y RStudio

Todo el código y documentación de R está almacenado en **CRAN** (Comprehensive R Archive Network), que es una red de servidores alrededor del mundo. Es decir, CRAN es el sitio oficial a través del cual descargaremos R.

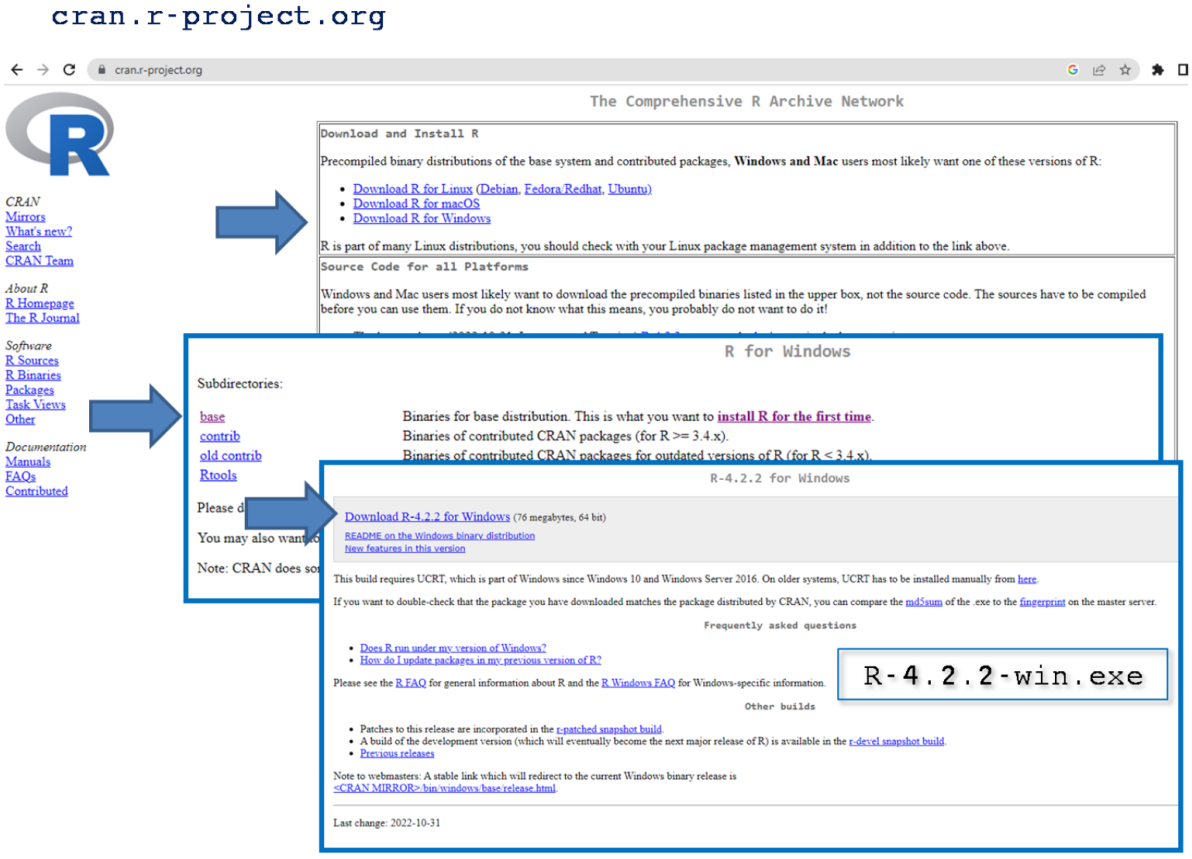


Figure 2: Instalación de R

Una vez instalado R podemos abrir la consola para ver qué apariencia tiene. Aquí ya podríamos trabajar.

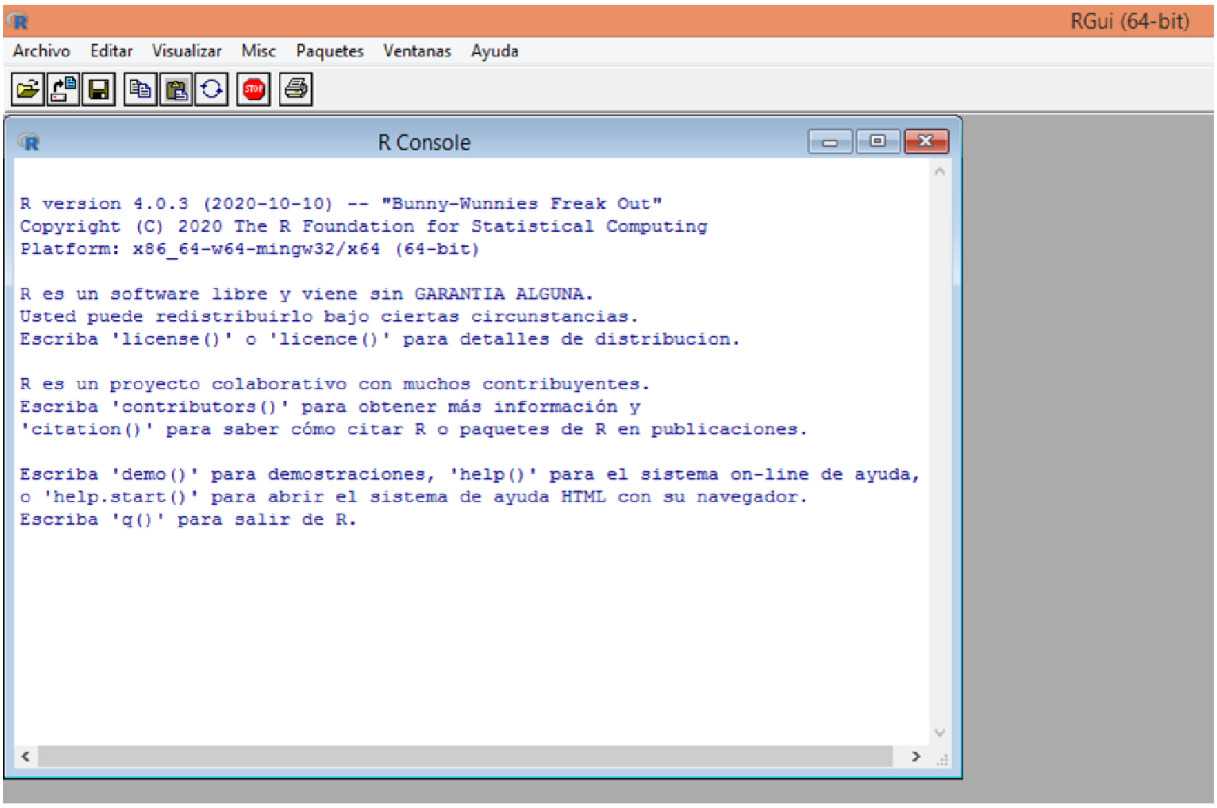


Figure 3: La consola de R

Sin embargo, vamos a aprender a usar R dentro de RStudio, que es una interfaz que nos da muchísimas más funcionalidades que trabajar solo con la consola. La descarga de RStudio se realiza desde Posit.

En Posit, una vez registrados, podremos crear un proyecto de Rstudio (New Project > New RStudio Project). Inmediatamente, se abrirá RStudio. Vamos a subir el material necesario para realizar el taller. En el cuadro de abajo a la derecha seleccionamos *Upload* y subimos la carpeta zip con el material del taller.

2.2 Exploramos RStudio

Cuando abrimos RStudio vemos varios paneles, vamos a ver qué es cada uno de ellos.

1. La consola A la izquierda tenemos la consola. La consola es el espacio donde R ejecuta las órdenes que le damos. El símbolo de "mayor que" se llama *prompt*, y significa que R está listo para que le demos una orden. En la consola podemos escribir la siguiente operación y pulsamos la tecla INTRO:

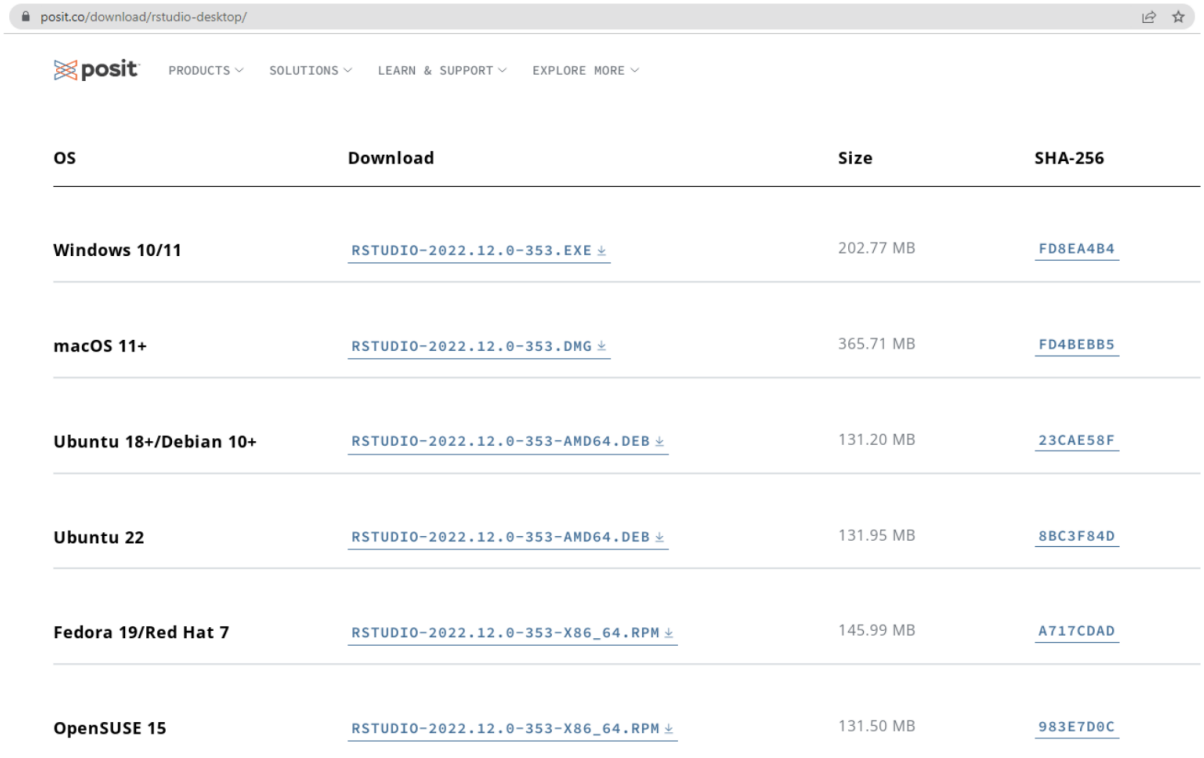


Figure 4: Instalación de RStudio

```
3+2
```

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

2. Scripts Normalmente, en un análisis de datos necesitamos ejecutar varias tareas y finalmente querremos guardarlas para recuperar este análisis en un futuro. Esto no es posible en la consola; ya que las instrucciones que ejecutemos en ella sólo se almacenan durante la sesión de R. La forma de poder guardarlas es generando un **script** de R. Un script es un bloque de código. Vamos a crear nuestro primer script. Para ello, en la barra de herramientas pinchamos en File > New File > R Script. Hay otro tipo de documentos de R, como Markdown, muy útil para realizar informes. Vemos que se abre un nuevo panel como si fuera una hoja en blanco en la que iremos escribiendo nuestras líneas de código. Podemos ejecutar cualquiera de las operaciones anteriores:

```
3+2
```

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

Para ejecutar el código que está escrito en el script pinchamos en "Run" o pulsamos la combinación de teclas CTRL+INTRO. Observad que el resultado se visualiza en la consola: tanto la instrucción como el resultado pero, ¡recordad!, lo que escribimos en la consola no se queda guardado. No tenemos que ejecutar línea por línea, sino que podemos ejecutar bloques de código seleccionando todas las líneas que queramos ejecutar.

3. El entorno En la parte derecha superior hay otro panel con varias pestañas. Destacaremos dos:

- En la pestaña "History" podemos consultar las instrucciones que hemos ido ejecutando. Si pinchamos en alguna de ellas vemos que se copia en la consola. Otra forma de recuperar instrucciones ya ejecutadas es, situándonos en la consola, usar las flechas de hacia arriba y hacia abajo del teclado.
- En la pestaña "Environment" aparecerán todos los elementos que forman parte del entorno de R. El entorno es el conjunto de objetos que están activos durante nuestra sesión de R. Ahora mismo en nuestro entorno únicamente están las funcionalidades de R base, pero veremos cómo va creciendo a lo largo de la sesión.

4. Gráficos y otros En la parte derecha inferior aparece otro panel c on varias pestañas. Destacaremos tres:

- Files. Aquí podemos navegar por las diferentes carpetas de nuestro ordenador.
- Plots. Aquí se visualizan los gráficos que generamos.
- Help. En esta pestaña obtendremos ayuda de R si se la pedimos.

2.3. Instalación de paquetes

Como hemos dicho, R base tiene una serie de funcionalidades que podemos expandir con el uso de paquetes disponibles en CRAN. Estos se instalan con la función install.packages() y se cargan con la función library(). Instalaremos todas las librerías que vamos a usar en el taller.

```
install.packages(c("dplyr","ggplot2","heplots","RColorBrewer",
                   "readxl","swirl"))
library(dplyr)
library(ggplot2)
library(heplots)
library(RColorBrewer)
library(readxl)
library(swirl)
```

3. R como calculadora

El uso más sencillo de R es como calculadora. Aquí tenemos algunas operaciones básicas.

```
3+2 # suma
3-2 # resta
3*2 # multiplicación
3/2 # división con decimales
3%/%2 # división entera
3**2 # potencia
3^2 # potencia
```

4. Objetos en R

Hasta ahora hemos visto el uso de R como calculadora. Sin embargo, las instrucciones que tendremos que ejecutar en un análisis de datos serán muchas otras: importar el conjunto de datos, filtrarlo, hacer operaciones con las variables de nuestro interés, hacer un gráfico, etc. Por ello, debemos conocer cómo interpreta y almacena R la información.

4.1 Concepto de variable

En primer lugar, tenemos que recordar que las instrucciones que se ejecutan en la consola no se guardan, pero necesitamos que R la recuerde y la tengamos accesible para seguir operando sobre ella.

Por ejemplo, hasta ahora hemos realizado operaciones sencillas, pero si pensamos en hacer varias operaciones, seguramente nos interese guardar el resultado de algunas de ellas. Podemos asignar el resultado 3+2 a una variable que se llame "a". La asignación de valores a una variable se puede hacer con la combinación "<-", como si fuera una flecha, o con el signo "=". Por convenio se prefiere la primera. Probad a ejecutar este código y veréis cómo quedan asignado el valor 5 a la variable "a". En R no importa si ponemos espacio entre la variable y su valor, lo que puede ser distinto en otros lenguajes. Lo que sí importa es si escribimos un espacio entre "<" y "-", ya que R interpreta otra cosa... Cada vez que hay un espacio, R interpreta que hay un elemento distinto.

```
a <- 3+2
a=3+2
a < -3 # ¡OJO!
```

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

Definir una variable es la forma en que R se guarda la información para tenerla disponible durante el rato que estamos trabajando en nuestro proyecto. Fijaos en el panel superior derecho: la variable que hemos creado ahora aparece en la pestaña "environment". El "environment" o **entorno** es el conjunto de variables que están activos en R durante nuestra sesión. Podemos eliminar las variables del entorno pinchando en el símbolo de la escoba.

Las variables no tienen por qué ser números, también pueden ser palabras.

```
b <- "Hola"
c <- "Adios"
```

Las variables se pueden sobreescribir y reasignarles otro valor.

```
c<-4
```

Podemos nombrar a las variables como queramos pero teniendo en cuenta algunas reglas:

- **NUNCA** debemos nombrar una variable con un nombre de una variable que ya exista en R. Por ejemplo, no sería apropiado crear la variable sum<-2+3, porque sum() es una función que existe en R y podemos crear conflictos. Ante la duda podemos ejecutar el comando help() para consultar si existe una variable, (help(sum)); si no nos devuelve nada entonces es que esa variable no está predefinida en R.
- Las variables **no contienen espacios** porque si no R las interpretará como dos objetos distintos.
- Por convención se escriben en **minúscula** y donde querríamos poner un espacio pondremos un guión bajo (o un punto). Estas convenciones pueden ser distintas en otros lenguajes de programación.
- Intentad que los nombres de vuestras variables sean **representativos** de su significado, tanto por facilitaros vuestro trabajo como por si lo lee otra persona. Puede que creéis muchas variables y necesitaréis saber qué es cada una de ellas, sobre todo si volvéis a mirar vuestro script pasado un tiempo.

4.2 Concepto de objeto

Las estructuras en las que se almacena la información en R se llaman objetos. Hay diferentes tipos de objetos, desde los que existen en R base (vectores, listas, matrices, conjuntos de datos,...) a otros más complejos que son propios de determinados paquetes, como aquellos dedicados al análisis de secuencias de nucleótidos. Cada tipo de objeto tiene asociadas una serie de operaciones o instrucciones que se le pueden aplicar. Cualquier tipo de objeto se puede asignar a una variable. Es decir, el **objeto** es el tipo de dato y la **variable** es el nombre que le asignamos para poder trabajar con él. Es difícil decidir qué tipo de objeto comenzar a explicar primero porque muchas veces la explicación de uno depende de otro. Normalmente se comienza por los vectores porque son el tipo de objeto más sencillo. Sin embargo, iremos en el orden que nos parece más didáctico a favor de esta sesión.

Funciones (*function*)

En primer lugar, queremos que sepáis que R cuenta con funciones para poder trabajar con otro tipo de objetos. Las funciones son objetos que podemos ver como una máquina a la que damos unos datos y nos devuelve un resultado. Por ejemplo, para obtener la raíz cuadrada de 2 podemos emplear la función sqrt()

```
sqrt(2)
```

```
## [1] 1.414214
```

Iremos viendo poco a poco varias funciones, pero nos gustaría resaltar dos por su utilidad: * La función class() nos dice el tipo de objeto. Como hemos dicho, cada tipo de objeto en R lleva asociadas unas operaciones que se pueden realizar y otras que no. A veces obtenemos errores porque estamos intentando aplicar una operación no permitida. Si sabemos el tipo de objeto con el que estamos trabajando, nos daremos cuenta de qué podemos y qué no podemos aplicarle. Si le preguntamos a R qué tipo de objeto es sqrt, nos dirá que es una función. Quizás ahora no le veáis la utilidad, pero hay clases de objetos más complicados y ocasiones en que tras muchas líneas de código ya no se recuerda de qué tipo son algunas variables.

```
class(sqrt)
```

```
## [1] "function"
```

- La función help() nos da información de las funciones. Si le pedimos ayuda sobre la función sqrt, vemos que nos dice que está implementada en R base y que calcula la raíz cuadrada. Además, nos informa de qué tipo de objeto podemos introducir como entrada y otros parámetros que se puedan modificar, así como ejemplos de cómo usarla.

```
help(sqrt)
```

Nosotros también podríamos crear nuestras propias funciones, aunque esto se escapa del objetivo de esta sesión.

Vectores (*numeric*, *character*)

Los vectores son objetos en los que podemos almacenar más de un valor, tanto números (*numeric*) como caracteres (*character*). Los vectores se crean con la función c(), que significa concatenar, separando los elementos con una coma.

```
edad <- c(21,18,20,21)
carrera<-c("psicología","medicina", "medicina", "biologia")
```

Los valores de un vector son del mismo tipo. Por ejemplo, si creo un vector que tiene números y caracteres, R interpreta los números como caracteres.

```
edad.carrera<-c(21,18,"psicología","medicina")
class(edad.carrera)
```

```
## [1] "character"
```

Para acceder a los elementos de un vector empleamos los corchetes. Los elementos en R empiezan a contarse desde el 1 (otros lenguajes empiezan en 0). Así accedemos al segundo elemento:

```
edad[2]
```

```
## [1] 18
```

Este índice que R reconoce lo podemos usar para ordenar los datos. Vamos a decirle que queremos ver el vector "arboles" pero que nos muestre primero el tercer elemento, luego el primero y finalmente el segundo.

```
edad[c(3,1,2)]
```

```
## [1] 20 21 18
```

También tenemos algunas funciones en R para ordenar vectores. La función sort() nos devuelve los números ordenados de menor a mayor. Si indicamos el parámetro decreasing=TRUE, entonces los devuelve de mayor a menor.

```r
sort(edad)
```

```
## [1] 18 20 21 21
```

```r
sort(edad, decreasing = TRUE)
```

```
## [1] 21 21 20 18
```

La función order() funciona igual que sort(), pero en vez de devolvernos los números directamente, nos devuelve el puesto que ocupan en el vector.

```r
order(edad)
```

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

```r
order(edad, decreasing = TRUE)
```

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

Curiosamente, las funciones order() y sort() también se pueden aplicar en vectores de caracteres, que nos devuelven ordenados alfabéticamente.

```r
sort(carrera)
```

```
## [1] "biologia"   "medicina"   "medicina"   "psicología"
```

```r
order(carrera)
```

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

Conjuntos de datos (*data.frame*)

Ya hemos visto los dos tipos de objetos básicos para poder explicar el tipo de objeto que más nos interesa hoy: los conjuntos de datos, Los conjuntos de datos son tablas con un número de filas y de columnas. Se crean con la función data.frame(), indicando el nombre de cada variable (cada columna) y los valores que contiene (es decir, las observaciones para cada una).

```
df<-data.frame(meses=c("enero","febrero","marzo","abril","mayo","junio",
                       "julio","agosto","septiembre","octubre",
                       "noviembre","diciembre"),
               dias=c(31,28,31,30,31,30,31,31,30,31,30,31))
```

Podemos acceder a las columnas del data.frame con el signo del dólar.

```
df$meses
```

```
##  [1] "enero"      "febrero"    "marzo"      "abril"      "mayo"
##  [6] "junio"      "julio"      "agosto"     "septiembre" "octubre"
## [11] "noviembre"  "diciembre"
```

```
df$dias
```

```
##  [1] 31 28 31 30 31 30 31 31 30 31 30 31
```

También podemos acceder con los corchetes, teniendo en cuenta que los data.frame tienen dos dimensiones: filas y columnas. Por lo tanto, indicamos primero las filas que queremos seleccionar y luego las columnas, separando los números por una coma. Si no ponemos ningún número, R interpretará que lo queremos todo.

```
df[,] # nos devuelve todas las filas y todas las columnas
```

```
##         meses dias
## 1       enero   31
## 2     febrero   28
## 3       marzo   31
## 4       abril   30
## 5        mayo   31
## 6       junio   30
## 7       julio   31
## 8      agosto   31
## 9  septiembre   30
## 10    octubre   31
## 11  noviembre   30
## 12  diciembre   31
```

```
df[,1] # nos devuelve todas las filas y la primera columna
```

```
##  [1] "enero"      "febrero"    "marzo"      "abril"      "mayo"
##  [6] "junio"      "julio"      "agosto"     "septiembre" "octubre"
## [11] "noviembre"  "diciembre"
```

```r
df[1,] # nos devuelve la primera fila y todas las columnas
```

```
##   meses dias
## 1 enero   31
```

```r
df[,"dias"] # nos devuelve todas las filas y la columna "dias"
```

```
##  [1] 31 28 31 30 31 30 31 31 30 31 30 31
```

5. Rutas y directorios

Una cosa importante para importar y exportar datos en R es saber indicarle en qué lugar de nuestro ordenador están o queremos guardar nuestros datos.

Directorio de trabajo
El directorio de trabajo es lugar en nuestro ordenador (la carpeta) donde estamos trabajando y donde, si no indicamos otra cosa, R buscará y escribirá información cuando se lo pidamos. Para saber cuál es nuestro directorio de trabajo usamos la función getwd() (*get working directory*).

```r
getwd()
```

```
## [1] "D:/CIB2026"
```

Rutas completas y relativas
La función getwd() nos devuelve la ruta de carpetas que tendríamos que ir abriendo desde la carpeta origen hasta nuestra carpeta de trabajo. Esto se llama una **ruta completa**. Podemos saber qué archivos hay en nuestro directorio con la función list.files()

```r
list.files()
```

Por defecto, esta función nos devuelve los archivos solo con su nombre, en forma de **ruta relativa** al directorio en el que estamos. Las rutas relativas se expresan respecto a un directorio, que puede ser el actual u otro, pero no desde la carpeta de origen. La función list.files() también nos puede devolver al ruta completa.

```r
list.files(full.names = TRUE)
```

Podemos cambiar nuestro directorio de trabajo con la función setwd(), indicando la ruta de la carpeta deseada, sea en forma completa o relativa. Aquí es muy útil el tabulador para ir autocompletando sin necesidad de que nos acordemos del orden de las carpetas.

6. Operaciones con conjuntos de datos

Ahora que sabemos un poco de cómo funciona R vamos a seguir profundizando en los conjuntos de datos, que creemos que es lo que más podéis usar en esta etapa (TFG, TFM). Vamos a ver ejemplos de las operaciones más comunes que nos puede interesar hacer.

6.1. Importar un conjunto de datos

1. Importar datos de Excel (.xlsx) Las hojas de cálculo como Excel son muy prácticas para guardar datos y seguramente así lo hagáis. Para leer datos de R en Excel usamos el paquete readxl. Si no está instalado hay que instalarlo y luego cargar el paquete al entorno. Os proporcionamos unos datos de absorbancia de un ensayo BCA para medir concentración de proteínas. Tenemos las funciones read_xls() y read_xlsx(). En este caso usamos la función read_xlsx() porque nuestro archivo es .xlsx. COn el parámetro *sheet* indicamos la hoja que queremos leer. El parámetro *skip* lo usamos si nos queremos saltar alguna línea, normalmente algún título o comentario.

```
bca<-read_xlsx("BCA_assay.xlsx",sheet=1,skip = 1)
```

2. Importar datos delimitados por separadores En general, cualquier archivo que contenga datos tiene los valores organizados en filas y dentro de cada fila los valores separados por algún caracter (separador) para indicar que pertenecen a diferentes columnas. Este separador puede ser un espacio (.txt), una coma (.csv) o un tabulador (.tsv) En general, la función read.delim() detecta bien el tipo de separador que se ha empleado para delimitar las columnas. También podemos emplear la función read.table(), aunque en esta sí tenemos que indicarle cuál es el separador.

```
tmb<-read.delim("tmb_mskcc_2018_clinical_data.tsv")
tmb<-read.table("tmb_mskcc_2018_clinical_data.tsv",header=TRUE,sep="\t")

# read.csv para archivos separados por ","
```

Además hay paquetes fuera de R base que también sirven para leer archivos, como **readr** y **openslsx**.

6.2. Conocer la estructura de un conjunto de datos

Vamos a seguir trabajando con el conjunto de datos *"tmb_mskcc_2018_clinical_data.tsv"*. Este conjunto de datos lo hemos obtenido a través del portal **cBioPortal**. cBioPortal contiene datos de genómica en cáncer y su creación y mantenimiento es del Memorial Sloan Kettering Cancer Center. El conjunto de datos está disponible como "TMB and immunotherapy (MSK,Nat Genet 2019)" (https://www.cbioportal.org/study/summary?id=

tmb_mskcc_2018). Podéis descargarlo desde el repositorio de github o en cBioPortal en la pestaña "Clinical Data" seleccionando todas las variables y descargando los datos en formato ".tsv".

Cuando vamos a analizar un conjunto de datos debemos conocer el tipo de información que contiene; con más razón cuando los datos no son nuestros, por ejemplo los que descargamos de repositorios públicos. Lo primero que tenemos que conocer de nuestro conjunto de datos es el número de observaciones (filas) y variables (columnas), así como el tipo de datos que contiene. Tenemos una serie de funciones que nos ayudan a obtener una primera visión de nuestro conjunto de datos.

```
str(tmb)
summary(tmb)
dim(tmb)
nrow(tmb)
ncol(tmb)
```

```
rownames(tmb) # nombres de las filas
colnames(tmb) # nombres de las columnas
head(tmb) # primeras líneas (por defecto 6)
tail(tmb) # últimas líneas (por defecto 6)
```

Recordamos cómo acceder a los datos (data.frame[fila, columna])

```
# Acceder a las columnas:
tmb$Patient.ID
tmb[,"Patient.ID"]

# Acceder a la primera fila
tmb[1,]

# Acceder a un valor específico:
tmb[3,2]
```

6.3. Filtrar un conjunto de datos

Podemos filtrar un conjunto de datos a las filas o columnas que nos interesen,e incluso almacenarlo en una nueva variable. Por ejemplo, vamos a quedarnos solo con los 10 primeros pacientes.

```
tmb2<-tmb[1:10,]
```

Aunque seguramente nos interese filtrar el conjunto de datos por alguna variable de interés. Y aquí es donde explicaremos los operadores lógicos. Los **operadores lógicos** nos sirven para hacerle preguntas a R y que nos responda con verdadero o falso.

```
tmb2<-tmb[tmb$Sex=="Female",]
tmb2<-tmb[tmb$TMB..nonsynonymous.>5,]
tmb2<-tmb[tmb$Cancer.Type!="Glioma",]
```

También podemos aplicar más de un filtro. Por ejemplo, vamos a seleccionar las muestras de metástasis de cáncer de mama.

```
tmb_mama_metastasis<-tmb[tmb$Cancer.Type == "Breast Cancer" &
                           tmb$Sample.Type == "Metastasis",]
```

6.4. Manejar valores perdidos (NA)

Puede haber ocasiones en que tenemos algún dato que nos falta. En R se llaman NA. Miremos por ejemplo la columna Mutation.count. Esto nos puede dar problemas.

```
mean(tmb$Mutation.Count)
```

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

```
mean(tmb$Mutation.Count, na.rm = TRUE)
```

```
## [1] 12.61242
```

La función is.na() nos permite saber si hay valores perdidos. Podemos rastrear rápidamente qué columnas tienen valores perdidos.

```
colSums(is.na(tmb)) # Nulos en la columna de concentracion
```

```
##                                     Study.ID
##                                            0
##                                   Patient.ID
##                                            0
##                                    Sample.ID
##                                            0
## Age.at.Which.Sequencing.was.Reported..Days.
##                                            1
##             Age.Group.at.Diagnosis.in.Years
##                                            0
##                                  Cancer.Type
##                                            0
##                          Cancer.Type.Detailed
```

```
##                                         0
##                                 Drug.Type
##                                         0
##                                Gene.Panel
##                                         0
##                           Institute.Source
##                                         0
##                            Metastatic.Site
##                                       760
##                             Mutation.Count
##                                        51
##                              Oncotree.Code
##                                         0
##                  Overall.Survival..Months.
##                                         0
##                    Overall.Survival.Status
##                                         0
##                         Primary.Tumor.Site
##                                         1
##                               Sample.Class
##                                         0
##               Number.of.Samples.Per.Patient
##                                         0
##                           Sample.coverage
##                                         0
##                               Sample.Type
##                                         0
##                                       Sex
##                                         0
##                             Somatic.Status
##                                         0
##                       TMB..nonsynonymous.
##                                         0
##                               Tumor.Purity
##                                        26
```

```
tmb.noNA <- na.omit(tmb) # Elimina todas las filas con algún NA
```

6.5. El paquete dplyr

El paquete dplyr es muy útil para trabajar con conjuntos de datos. Es una cuestión personal emplear R base u otros paquetes. Podéis consultar las *cheat sheet*: https://nyu-cdsc.github.io/learningr/assets/data-transformation.pdf y https://iqss.github.io/dss-workshops/

R/Rintro/base-r-cheat-sheet.pdf. Hay *cheat sheets* disponibles para muchos temas: Rbase, colores, paquetes, etc. Veamos algunos ejemplos en R base y con funciones del paquete dplyr.

```
tmb %>%
  group_by(Sex) %>%
  count() # Similar a la función table

# Aparece un NA
tmb %>%
  group_by(Cancer.Type) %>%
  summarise(Promedio_tiempo = mean(Age.at.Which.Sequencing.was.Reported..Days.))

tmb %>%
  group_by(Cancer.Type) %>%
  summarise(Promedio_tiempo = mean(Age.at.Which.Sequencing.was.Reported..Days.,
                                   na.rm = TRUE))

tmb %>%
  filter(Cancer.Type == "Melanoma") %>%
  group_by(Sex) %>%
  summarise(Promedio_tiempo = mean(Age.at.Which.Sequencing.was.Reported..Days.,
                                   na.rm = TRUE))
```

7. Breve análisis estadístico

No vamos a entrar mucho en el terreno de la estadística, ya que es complejo y todo lo infinito que queramos. Sin embargo, hay una parte muy importante a la hora de hacer cualquier análisis, que es conocer el comportamiento de las variables (estadística descriptiva) antes de poder analizar la significancia de la relación entre ellas (estadística inferencial).

Este apartado servirá como ejemplo de cómo proceder a analizar un conjunto de datos.

1. Exploramos la estructura del conjunto de datos con la función str().

```
tmb<-read.delim("tmb_mskcc_2018_clinical_data.tsv")
str(tmb)
```

```
## 'data.frame':    1661 obs. of  24 variables:
##  $ Study.ID                                  : chr  "tmb_mskcc_2018" "tmb_mskcc_2018
##  $ Patient.ID                                : chr  "P-0000057" "P-0000062" "P-00000
##  $ Sample.ID                                 : chr  "P-0000057-T01-IM3" "P-0000062-T
##  $ Age.at.Which.Sequencing.was.Reported..Days.: int  41 80 62 66 61 63 47 44 67 60 ..
##  $ Age.Group.at.Diagnosis.in.Years           : chr  "31-50" ">71" "61-70" "61-70" ..
```

```
##  $ Cancer.Type                                : chr  "Breast Cancer" "Esophagogastric
##  $ Cancer.Type.Detailed                       : chr  "Breast Mixed Ductal and Lobular
##  $ Drug.Type                                  : chr  "PD-1/PDL-1" "PD-1/PDL-1" "PD-1/
##  $ Gene.Panel                                 : chr  "IMPACT341" "IMPACT341" "IMPACT3
##  $ Institute.Source                           : chr  "MSKCC" "MSKCC" "MSKCC" "MSKCC"
##  $ Metastatic.Site                            : chr  NA NA NA NA ...
##  $ Mutation.Count                             : int  5 6 13 10 12 12 6 3 5 8 ...
##  $ Oncotree.Code                              : chr  "MDLC" "GEJ" "BLCA" "BLCA" ...
##  $ Overall.Survival..Months.                  : int  0 1 42 43 57 12 18 4 1 8 ...
##  $ Overall.Survival.Status                    : chr  "1:DECEASED" "1:DECEASED" "0:LIV
##  $ Primary.Tumor.Site                         : chr  "Breast" "Esophagus" "Bladder" "
##  $ Sample.Class                               : chr  "Tumor" "Tumor" "Tumor" "Tumor"
##  $ Number.of.Samples.Per.Patient              : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ Sample.coverage                            : int  835 1176 900 795 905 783 997 506
##  $ Sample.Type                                : chr  "Primary" "Primary" "Primary" "P
##  $ Sex                                        : chr  "Female" "Male" "Male" "Male" ..
##  $ Somatic.Status                             : chr  "Matched" "Matched" "Matched" "M
##  $ TMB..nonsynonymous.                        : num  5.55 6.65 15.53 9.98 13.31 ...
##  $ Tumor.Purity                               : chr  "25" "30" "70" "30" ...
```

Vemos que tenemos 1661 observaciones para 24 variables, de las cuales unas son de tipo caracter y otras son de tipo numérico. Me llama la atención que hay una variable que identifica al paciente y otra la muestra del paciente, por lo que me oregunto, ¿hay algún paciente con más de una muestra? Esto lo puedo comprobar viendo si hay algún identificador del paciente repetido, con la función duplicated().

```
id.duplicados<-duplicated(tmb$Patient.ID)
sum(id.duplicados)
```

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

Una buena práctica es comprobar si hay algún valor faltante. Ya hemos visto cómo evaluar si hay valores perdidos en R. Vamos a quedarnos únicamente con aquellas observaciones que están completas. A este nuevo conjunto de datos le vamos a llamar tmb.complete.

```
na.count<-rowSums(is.na(tmb))
tmb.complete<-tmb[na.count==0,]
```

7.1. Estadística descriptiva

En primer lugar, debemos caracterizar cada variable, describir qué valores tiene en nuestro estudio. La descripción de cada variable dependerá de si es de tipo numérico o categórico.

1. Variables de tipo numérico. Las variables de tipo numérico se describen con medidas de centralidad, como son la media y la mediana. Otros valores que podemos obtener son los valores mínimo y máximo. La función summary() nos permite obtener esta información para todas las variables, excepto para las de tipo caracter.

```
summary(tmb.complete)
```

También podemos obtener estos valores para alguna variable de interés. Por ejemplo, vamos a explorar el número de mutaciones.

```
mean(tmb$Mutation.Count)
```

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

```
mean(tmb$Mutation.Count, na.rm = TRUE)
```

```
## [1] 12.61242
```

```
mean(tmb.complete$Mutation.Count)
```

```
## [1] 13.12588
```

También podemos caracterizar los valores mínimo y máximo.

```
min(tmb.complete$Mutation.Count)
```

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

```
max(tmb.complete$Mutation.Count)
```

```
## [1] 163
```

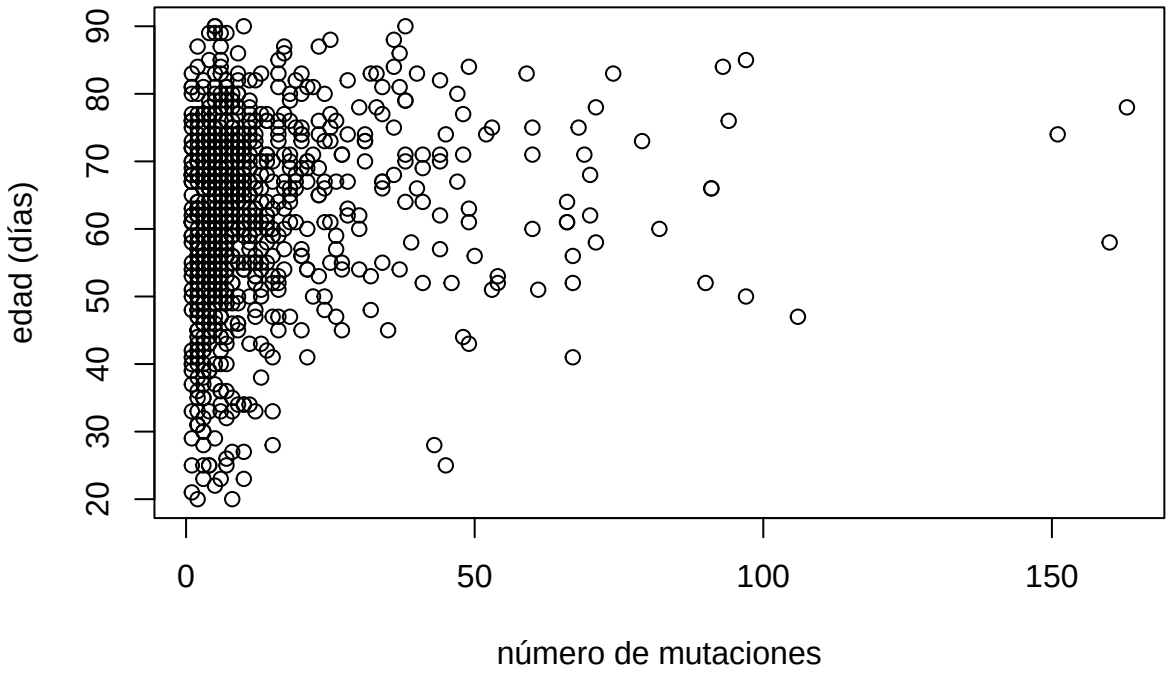
2. Variables de tipo categórico. Las variables de tipo categórico se definen por la frecuencia de sus valores. Podemos explorar la variable "Cancer.Type" con la función table(). Vemos que solo hay un paciente con cáncer de piel, por lo que probablemente eliminaremos este paciente en las pruebas estadísticas.

```
table(tmb.complete$Cancer.Type)
```

3. Gráficos Representar la información de forma gráfica es un recurso muy útil para visualizar de forma inmediata su comportamiento

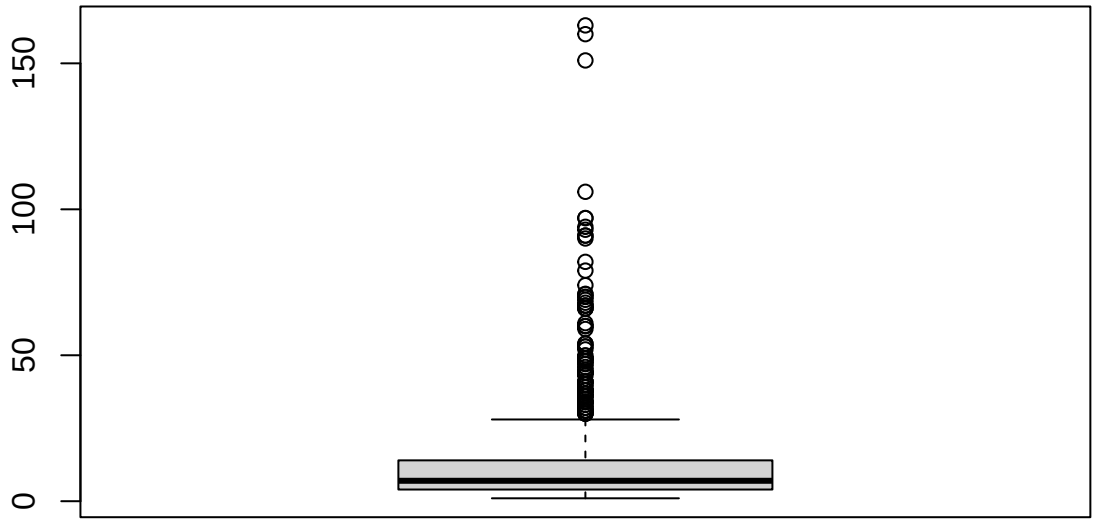
- Gráfico de dispersión. Es útil para explorar si existe alguna relación entre dos variables. Nos podemos preguntar si hay alguna relación entre el número de mutaciones y la edad del paciente. Fijaos que podemos cambiar el título de los ejes para que nos quede más presentable.

```
plot(tmb.complete$Mutation.Count,
     tmb.complete$Age.at.Which.Sequencing.was.Reported..Days.,
     xlab="número de mutaciones",
     ylab = "edad (días)")
```



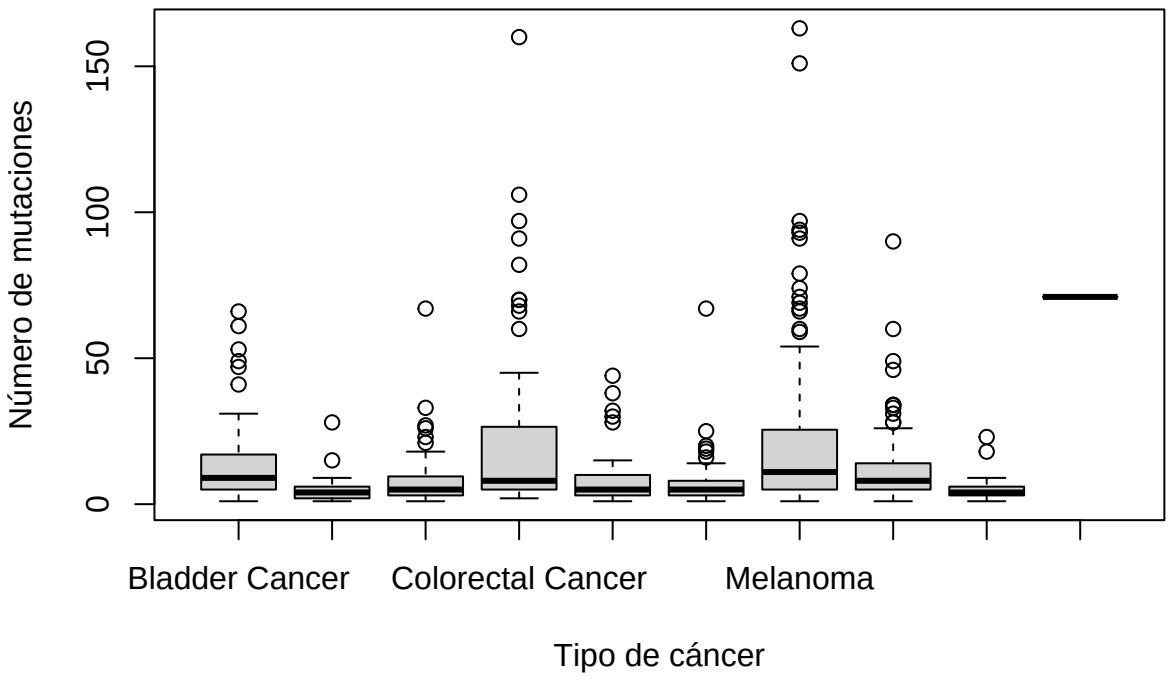
- Gráfico de cajas y bigotes Los gráficos de cajas y bigotes nos sirven para ver la dispersión y simetría de nuestros datos. Venos que la variable Mutation.Count tiene valores extremos.

```
boxplot(tmb.complete$Mutation.Count)
```



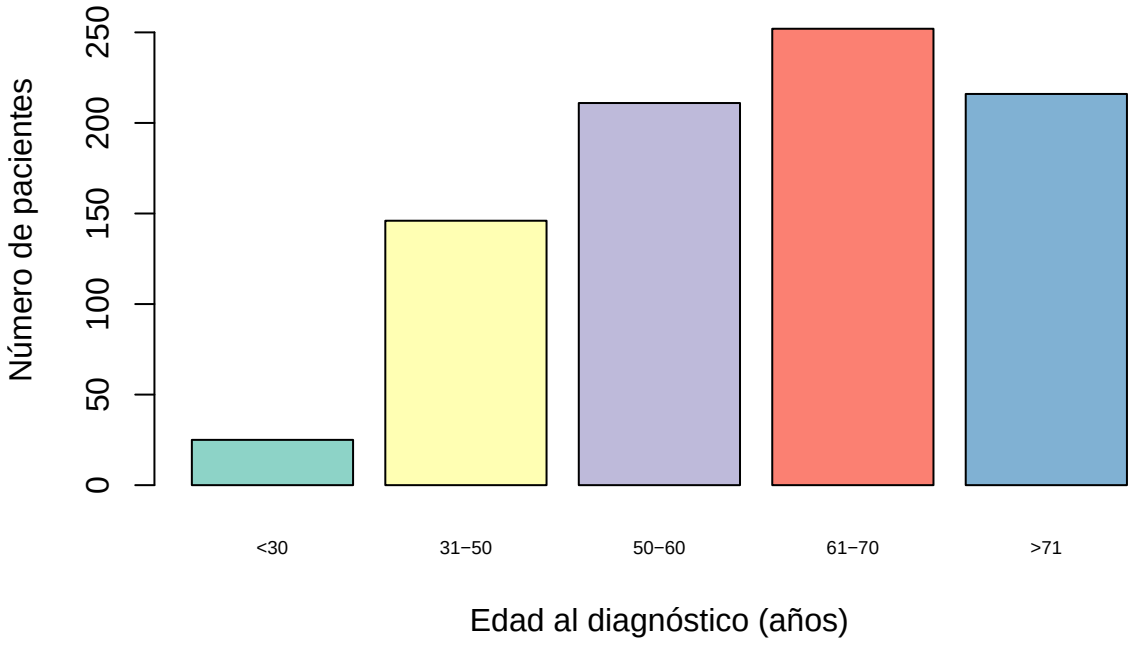
También lo podemos representar respecto a otra variable; por ejemplo, el número de mutaciones en los diferentes tipos de cáncer.

```
boxplot(tmb.complete$Mutation.Count ~tmb.complete$Cancer.Type,
        xlab="Tipo de cáncer",
        ylab="Número de mutaciones")
```



- Gráfico de barras. Nos sirve para representar la frecuencia de los valores de una variable. Podemos representar el grupo de edad de los pacientes. Vemos que el grupo más frecuente es el de 61-70 años.

```
age.freq<-table(tmb.complete$Age.Group.at.Diagnosis.in.Years)[c(1,3,4,5,2)]
age.bar<-barplot(age.freq,
        col=brewer.pal(5, "Set3"),
        xlab="Edad al diagnóstico (años)",
        ylab="Número de pacientes",
        ylim = c(0,max(age.freq)+30),
        cex.names=0.6
        )
```



7.2. Estadística inferencial

En un análisis de datos biológicos normalmente no querremos quedarnos en describir las variables que hemos estudiado, sino responder a una pregunta biológica, es decir, tenemos una hipótesis. Para contrastar nuestra hipótesis y ver si se cumple o no en base a los datos que hemos colectado, recurrimos a las pruebas estadísticas. Hay dos premisas que se tienen en cuenta para las variables numéricas, que son la normalidad y la igualdad de varianzas entre las variables a comparar.

1. Test de normalidad Una cuestión que tenemos que tener en cuenta para aplicar casi cualquier prueba estadística es saber si nuestros datos tienen una distribución normal, ya que las condiciones en que son válidas ciertas pruebas solo se aplican si los datos son normales. Para ello, podemos emplear pruebas como el test de Shapiro Wilks y pruebas gráficas.

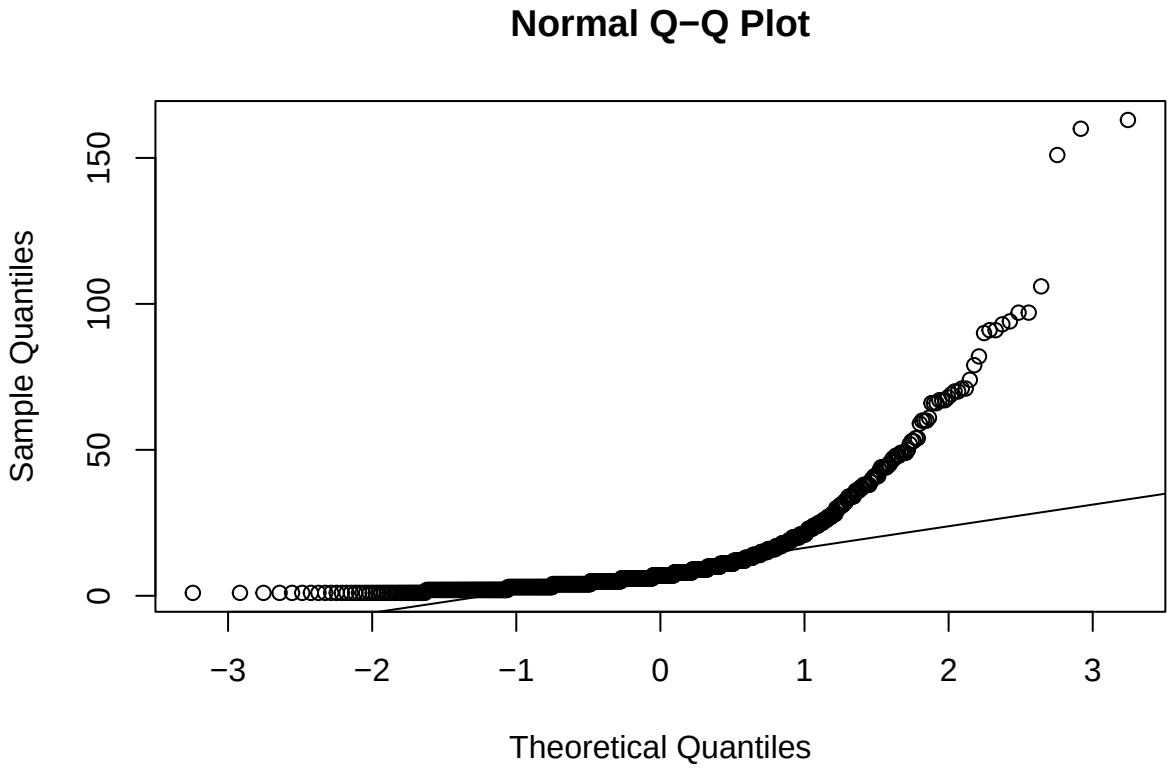
En el caso del test Shapiro Wilks, la hipótesis nula es que no hay diferencia entre la distribución de nuestros datos y una distribución normal. Por lo tanto, si el p valor está por encima del valor de significancia que hayamos establecido (que normalmente es del 5%), aceptaremos la hipótesis nula. En caso contrario, la rechazaremos y diremos que nuestros datos no siguen una distribución normal.

```
shapiro.test(tmb.complete$Mutation.Count)
```

```
##
##  Shapiro-Wilk normality test
##
## data:  tmb.complete$Mutation.Count
## W = 0.60519, p-value < 2.2e-16
```

También podemos recurrir a los gráficos de cuantiles, que son muy informativos. En él se representan los datos frente a los de una distribución normal, con una línea que los correlaciona. Solo rechazaremos la normalidad si los puntos se alejan mucho de la línea que relaciona los cuantiles con los cuantiles teóricos.

```
qqnorm(tmb.complete$Mutation.Count)
qqline(tmb.complete$Mutation.Count)
```



En este caso, ambas pruebas nos indican que la variable Mutation.Count no sigue una distribución normal.

2. Test de igualdad de varianzas Otra de las premisas que se tienen que cumplir para aplicar ciertas pruebas es que las varianzas de los grupos sean iguales. Para ello podemos emplear el test de Levene usando la función leveneTests() del paquete heplots.

Por ejemplo, imaginad que quiero comparar si la varianza del número de mutaciones es igua entre hombres y mujeres. El test nos indica que las varianzas no son iguales.

```
leveneTests(tmb.complete[,"Mutation.Count", drop=FALSE],
            tmb.complete$Sex)
```

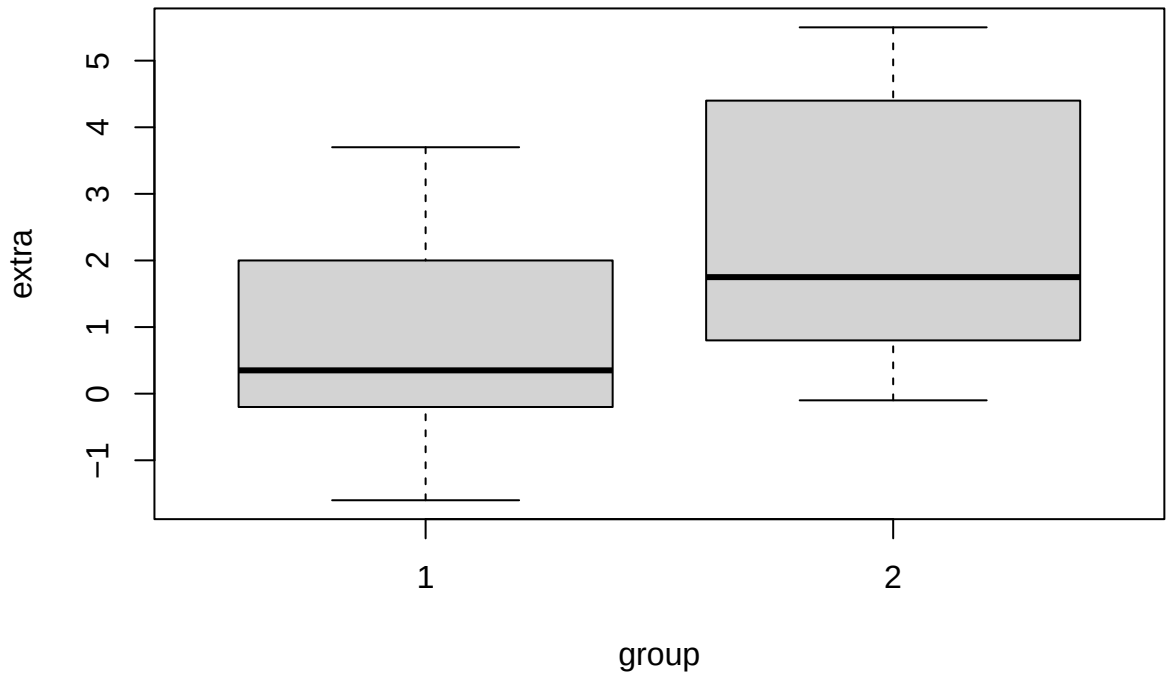
```
## Warning in leveneTest.default(x, group = group, center = center, ...): group
## coerced to factor.

## Levene's Tests for Homogeneity of Variance (center = median)
##
##                 df1 df2 F value Pr(>F)
## Mutation.Count   1 848  5.2262 0.0225 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

3. Comparación de medias para distribuciones normales (pruebas paramétricas)

- Comparación de dos grupos: T test. Como en nuestro conjunto de datos no tenemos variables que sigan una distribución normal, vamos a usar otros datos de ejemplo.

```
plot(extra ~ group, data = sleep)
```



```
leveneTests(sleep[,1, drop=FALSE], sleep$group)
```

```
## Levene's Tests for Homogeneity of Variance (center = median)
##
##       df1 df2 F value Pr(>F)
## extra   1  18  0.2482 0.6244
```

```
t.test(extra ~ group, data = sleep, var.equal=TRUE)
```

```
##
##  Two Sample t-test
##
## data:  extra by group
## t = -1.8608, df = 18, p-value = 0.07919
## alternative hypothesis: true difference in means between group 1 and group 2 is not e
## 95 percent confidence interval:
##  -3.363874  0.203874
## sample estimates:
## mean in group 1 mean in group 2
##            0.75            2.33
```

- Comparación de más de dos grupos: ANOVA. Nos podemos preguntar si hay diferencias en el número de mutaciones entre los diferentes tipos de cáncer. El test de ANOVA nos dice que sí hay diferencias entre los distintos tipos de cáncer. En un sentido estricto, este test estaría mal aplicado porque la distribución de la variable no es normal, aunque hay autores que dicen que con más de 30 observaciones se puede aplicar un test paramétrico.

```
anova(lm(Mutation.Count ~ Cancer.Type, data = tmb.complete))
```

```
## Analysis of Variance Table
##
## Response: Mutation.Count
##              Df Sum Sq Mean Sq F value    Pr(>F)
## Cancer.Type   9  31094  3454.9  11.883 < 2.2e-16 ***
## Residuals   840 244219   290.7
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

4. Comparación de medias para distribuciones no normales (pruebas no paramétricas)

- Comparación de dos grupos: test de Wilcoxon. Nos podemos preguntar si el número de mutaciones es diferentes entre hombres y mujeres.

```
wilcox.test(tmb.complete$Mutation.Count[tmb.complete$Sex=="Female"],
            tmb.complete$Mutation.Count[tmb.complete$Sex=="Male"])
```

```
##
##  Wilcoxon rank sum test with continuity correction
##
## data:  tmb.complete$Mutation.Count[tmb.complete$Sex == "Female"] and tmb.complete$Mut
## W = 84643, p-value = 0.7397
## alternative hypothesis: true location shift is not equal to 0
```

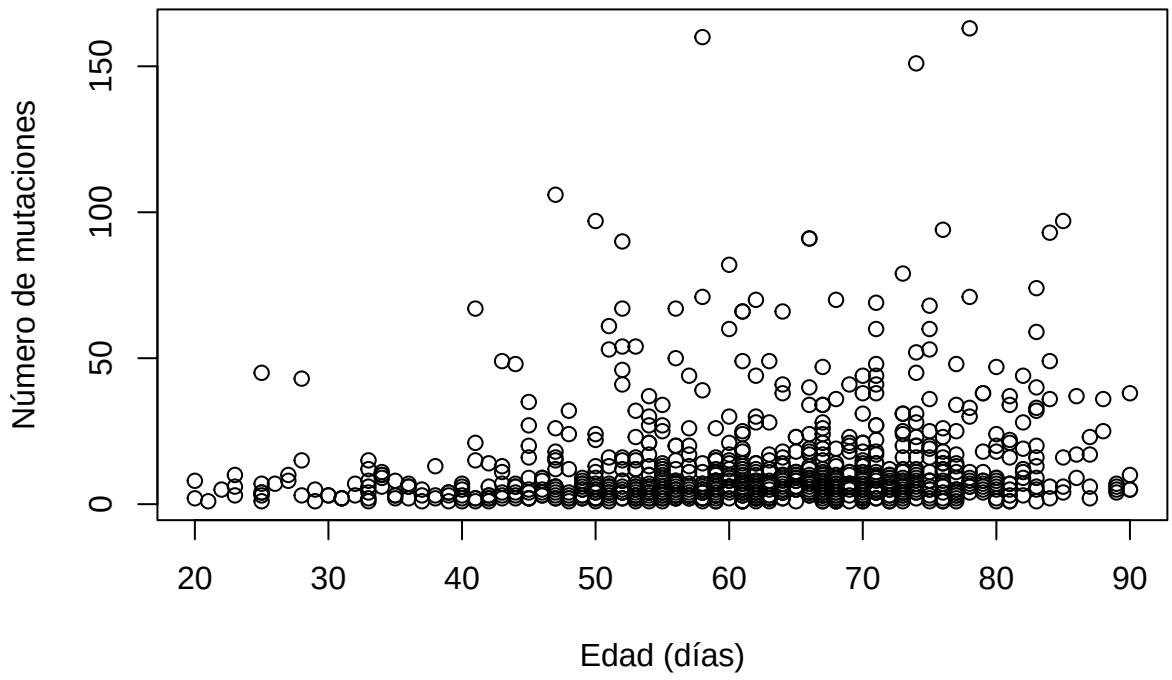
- Comparación de más de dos grupos: test de Kruskal Wallis. Vemos que el resultado del test no paramétrico nos lleva a la misma conclusión que el paramétrico: hay diferencias en el número de mutaciones entre los distintos tipos de cáncer.

```
kruskal.test(Mutation.Count ~ Cancer.Type, data = tmb.complete)
```

```
##
##  Kruskal-Wallis rank sum test
##
## data:  Mutation.Count by Cancer.Type
## Kruskal-Wallis chi-squared = 102.16, df = 9, p-value < 2.2e-16
```

5. Correlación entre variables. Podemos emplear la función plots() para ver la relación entre variables y la función cor() para obtener el coeficiente de correlación. Vemos que el número de mutaciones y la edad no parecen tener correlación, y esto lo confirmamos con que el coeficiente de correlación es muy bajo.

```
plot(tmb.complete$Age.at.Which.Sequencing.was.Reported..Days.,
     tmb.complete$Mutation.Count,
     xlab="Edad (días)",
     ylab = "Número de mutaciones")
```

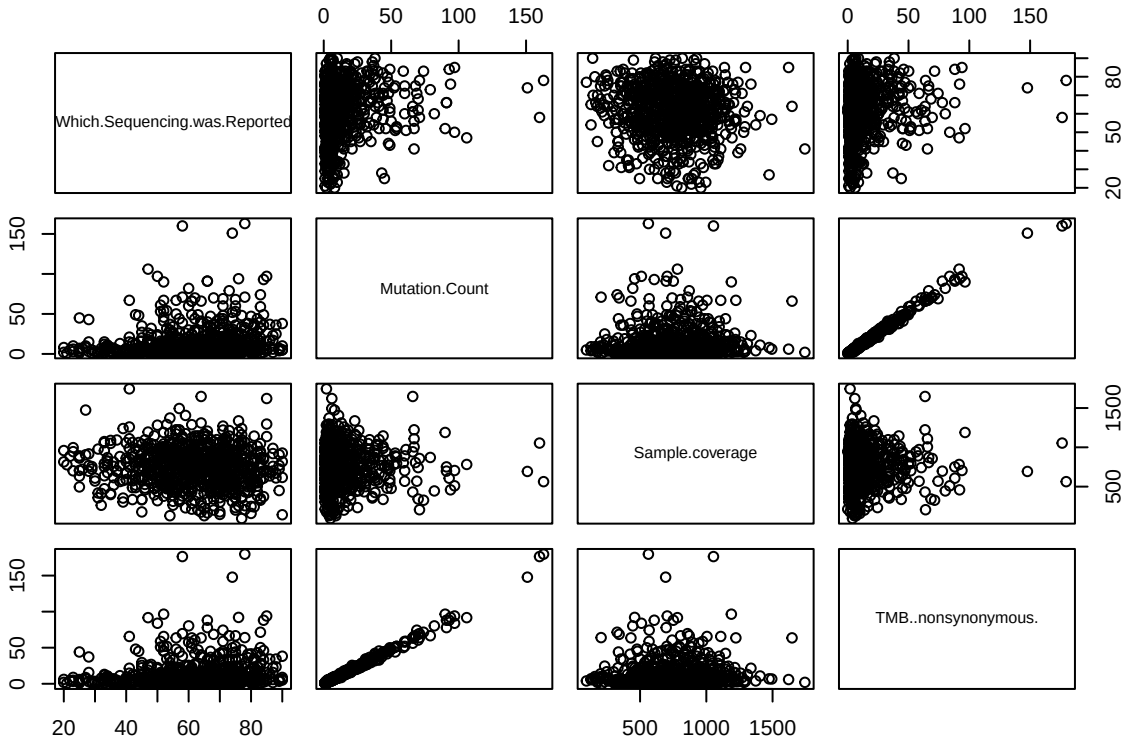



```
cor(tmb.complete$Age.at.Which.Sequencing.was.Reported..Days.,
    tmb.complete$Mutation.Count)
```

```
## [1] 0.1465951
```

La función pairs() nos permite obtener gráficos de dispersión para todas las variables dos a dos.

```
pairs(tmb.complete[,c("Age.at.Which.Sequencing.was.Reported..Days.",
                      "Mutation.Count",
                      "Sample.coverage",
                      "TMB..nonsynonymous.")]
      )
```



6. Test para variables categóricas Nos podemos preguntar si el cáncer de vejiga (bladder) es más frecuente en hombres o en mujeres. Para ello emplearemos el test de chi cuadrado con la función chisq.test(), que nos devuelve que sí es significativa la diferencia en el número de casos de cáncer de vejiga entre hombres y mujeres.

```
table(tmb.complete$Cancer.Type,tmb.complete$Sex)
```

```
##
##                                Female Male
##   Bladder Cancer                   19   68
##   Breast Cancer                    30    0
##   Cancer of Unknown Primary        32   32
##   Colorectal Cancer                19   36
##   Esophagogastric Cancer           14   29
##   Head and Neck Cancer             16   71
##   Melanoma                         87  161
##   Non-Small Cell Lung Cancer       96   74
##   Renal Cell Carcinoma             17   48
##   Skin Cancer, Non-Melanoma         0    1
```

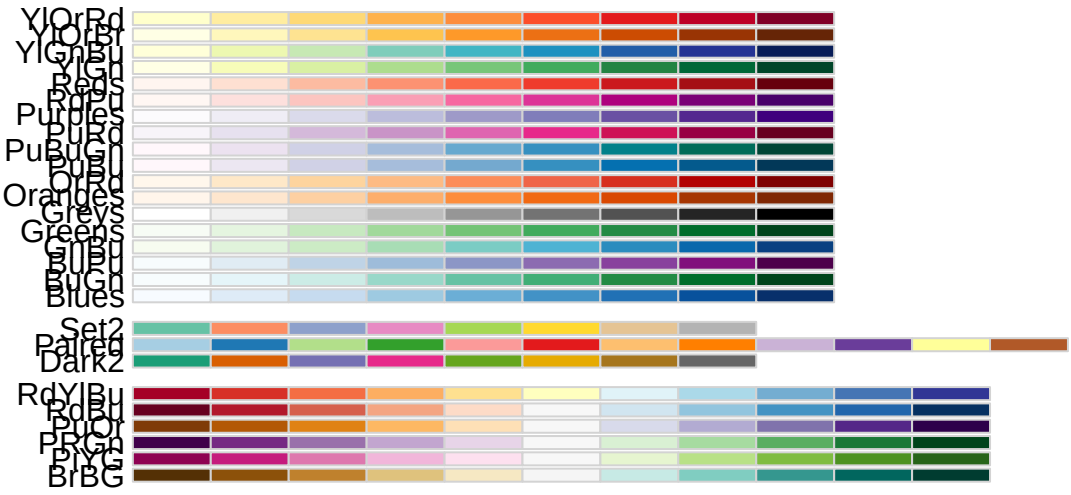
```
chisq.test(table(tmb.complete$Sex[tmb.complete$Cancer.Type=="Bladder Cancer"]))
```

```
##
##  Chi-squared test for given probabilities
##
## data:  table(tmb.complete$Sex[tmb.complete$Cancer.Type == "Bladder Cancer"])
## X-squared = 27.598, df = 1, p-value = 1.494e-07
```

7.3. Creación de gráficos con el paquete ggplot2

El paquete ggplto2 nos permite hacer gráficos muy personalizables. Para entender su funcionamiento, pensemos en un lienzo en blanco sobre el que vamos añadiendo capas: pintar los puntos, los ejes, cambiar el color, etc. Aprovechamos para señalar que es importante usar paletas de colores que sean accesibles para todos. Dentro de la librería RColorBrewer podemos ver qué paletas son adecuadas para personas daltónicas.

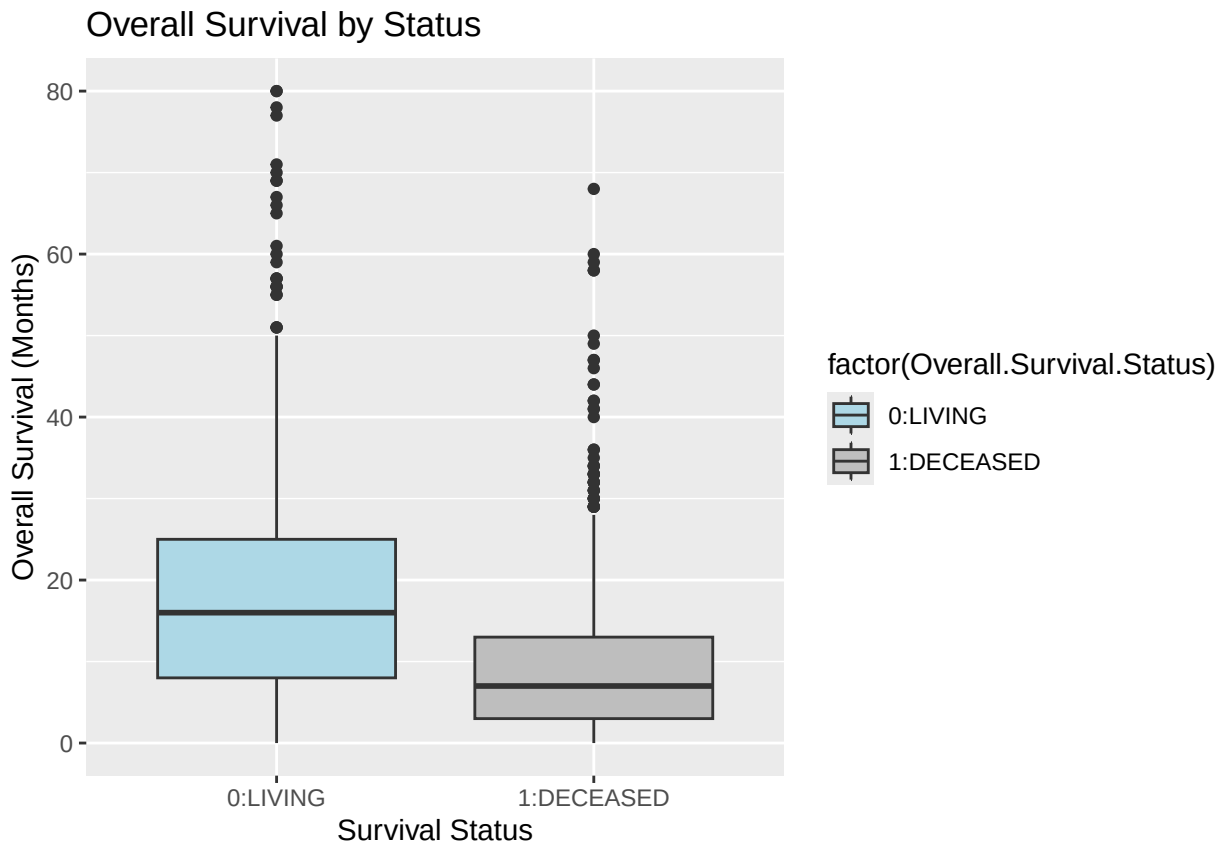
```
display.brewer.all(colorblindFriendly = TRUE)
```



Con la librería colorBlindness podéis convertir los colores de un gráfico a cómo los vería una persona con alteraciones en la visión y comprobar que se vería bien. Podéis consultar esta página para más información.

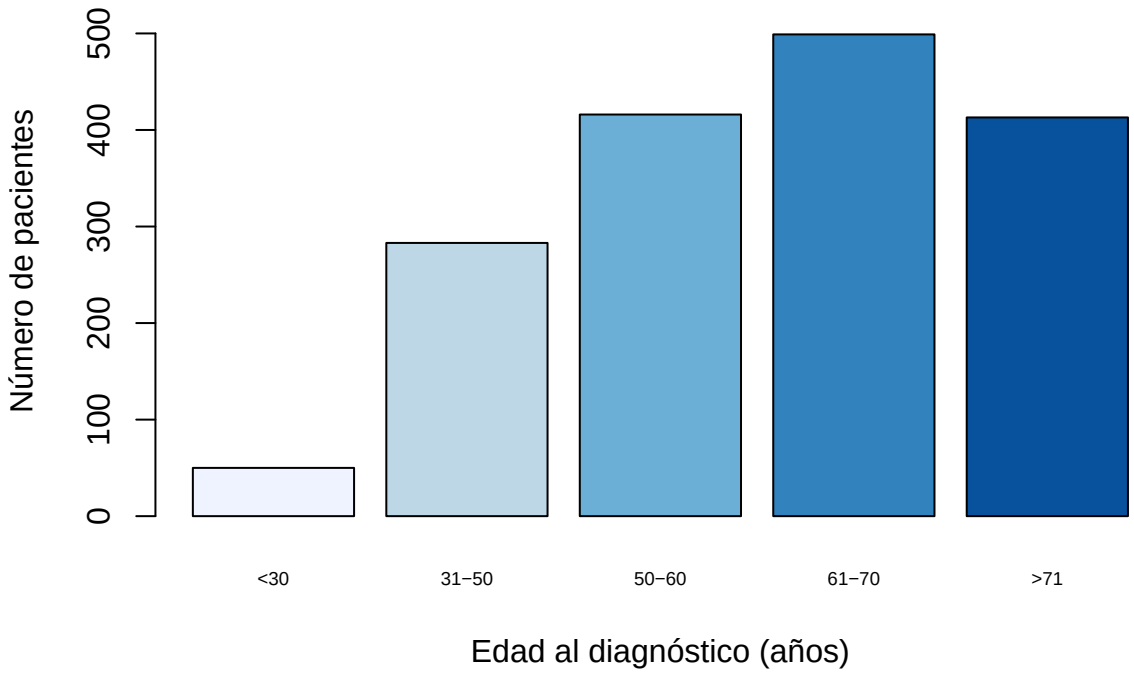
Veamos cómo crear un gráfico de cajas y bigotes con ggplot:

```
ggplot(data = tmb, aes(x = Overall.Survival.Status, y = Overall.Survival..Months., fill
  geom_boxplot() +
  scale_fill_manual(values = c("0:LIVING" = "lightblue", "1:DECEASED" = "grey")) +
  labs(title = "Overall Survival by Status",
       x = "Survival Status",
       y = "Overall Survival (Months)")
```



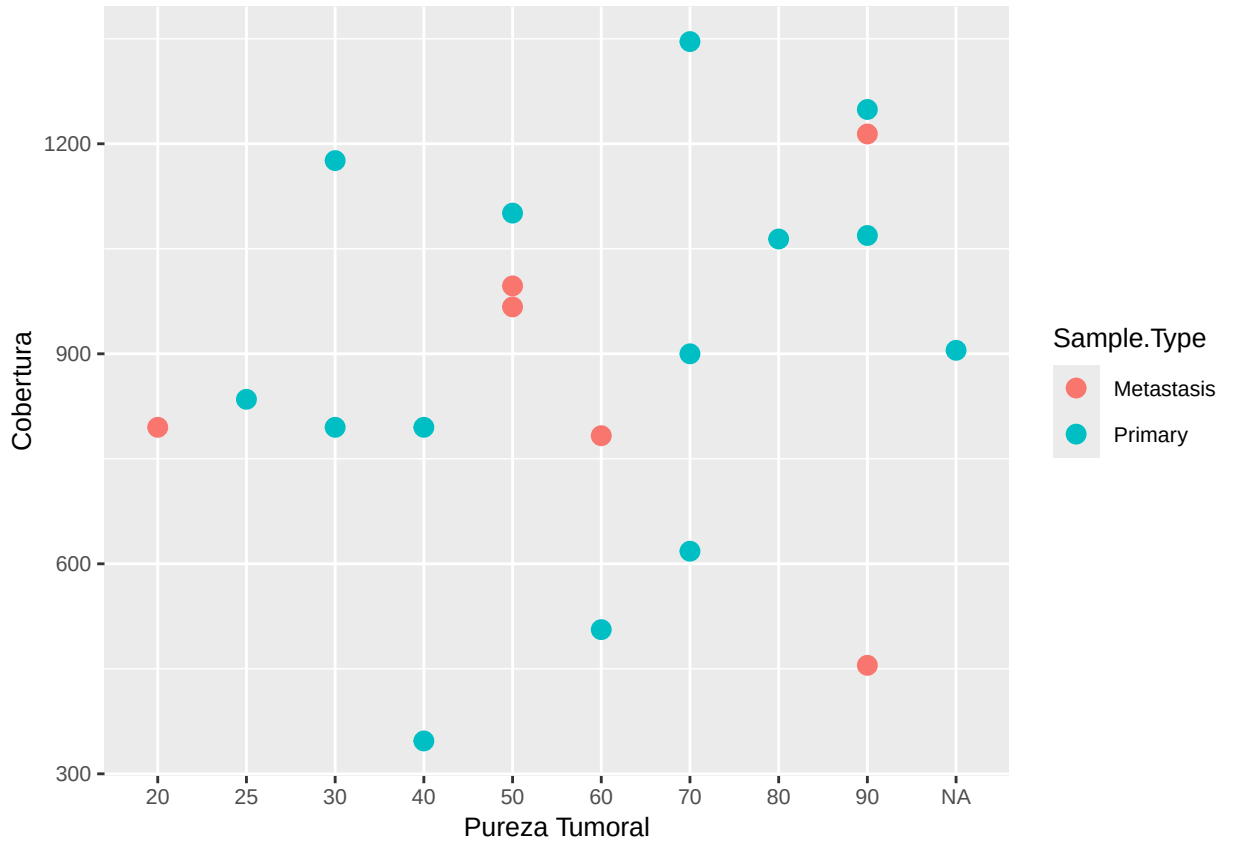
Así creamos un gráfico de barras.

```
age.freq<-table(tmb$Age.Group.at.Diagnosis.in.Years)[c(1,3,4,5,2)]
age.bar<-barplot(age.freq,
                 col=brewer.pal(5, "Blues"),
                 xlab="Edad al diagnóstico (años)",
                 ylab="Número de pacientes",
                 ylim = c(0,max(age.freq)+30),
                 cex.names=0.6
)
```



Y así se representa un gráfico de dispersión:

```
ggplot(data=tmb[1:20,],
       aes(y=Sample.coverage, x=Tumor.Purity, color=Sample.Type)) +
  geom_point(size=3) + # capa de puntos tamaño 3
  theme(text=element_text(size=10))+
  labs(y="Cobertura",x="Pureza Tumoral")
```



Por cierto, los gráficos también se pueden almacenar en variables:

```
p <- ggplot(data=tmb[1:20,],
       aes(y=Sample.coverage, x=Tumor.Purity, color=Sample.Type)) +
  geom_point(size=3) + # capa de puntos tamaño 3
  theme(text=element_text(size=10))+
  labs(y="Cobertura",x="Pureza Tumoral")
```

8. Guardar la información

Probablemente durante nuestro análisis hayamos modificado nuestro conjunto de datos, generado gráficos, etc. y queramos guardarlos.

Para guardar un conjunto de datos podemos emplear la función write.table, indicando el conjunto de datos que queremos guardar y la ruta donde lo queremos escribir, incluyendo el nombre y extensión del fichero.

```
write.table(x = df,
            filename = "/path/to/file.tsv",
            sep ="\t",
```

```
            quote = FALSE,
            row.names = TRUE)
```

Los gráficos generados se pueden guardar con la función ggsave(). También podemos pinchar en el desplegable "Export" y guardarlo en el formato que queramos o copiarlo en el portapapeles.

```
ruta_file_name = ""
ggsave(filename = "path/to/plot.png",
       plot=last_plot(),
       device = "png")
```

9. Buenas prácticas

- Escribir comentarios en el código. Los comentarios son líneas que van precedidas con la almohadilla "#" y que R no ejecuta. Sirven para dejar notas sobre lo que hace nuestro código o para dejar como borrador líneas de código que aún no queremos eliminar Lo hacemos con Alt Gr + 3 o con la combinación de teclas Ctrl + MAYUS + C si sobre una selección de líneas.
- No usar espacios al nombrar archivos (mejor Mi_archivo.R que Mi archivo.R). También para las carpetas sería una buena costumbre.
- Usar nombres de variables representativos.
- Guardar el trabajo frecuentemente. Ctrl + S.

10. Ejercicios para practicar

1. Importa el conjunto de datos "mel_ucla_2016_clinical_data_excel2.xlsx", que contiene datos sobre un estudio de melanoma. 1.1. Dibuja un gráfico de barras para los valores de la columna "M stage" con R base. 1.2. Haz lo mismo en ggplot. 1.3. Busca información para cambiar algunos aspectos que te interesen, como el color de las barras, el título de los ejes, etc. Recuerda que puedes usar la función help() para obtener ayuda de las funciones de R.

2. Carga el archivo "BCA_assay.xlsx". 2.1. Filtra el conjunto de datos a las filas 1 a 9 para quedarte con las medidas para hacer la recta patrón. 2.2. Calcula la media de las dos medidas de absorbancia (medida1 y medida2) y guárdalo en una variable. 2.3. Haz un gráfico representando los valores de concentración en el eje x y los de la media que acabas de calcular en el eje y.

3. Importa el conjunto de datos "mel_ucla_2016_clinical_data_excel2.xlsx". 3.1. Describe el número de pacientes y variables clínicas. 3.2. Elige una variable cualitativa y represéntala con el gráfico que te parezca adecuado. 3.3. Elige una variable cuantitativa y represéntala con el gráfico que te parezca adecuado.

4.

Cómo seguir aprendiendo por tu cuenta

Un recurso muy sencillo, gratis y eficaz es swirl, que es un paquete para aprender R dentro de R. La instalación de swirl se realiza con install.packages(). Podéis encontrar las instrucciones en este enlace.

```
library(swirl)
```

También hay mucho material disponible en internet. Algunas recomendaciones:

- *An introduction to R" (31/10/2022). https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf
- Los libros de Rafael Irizarry, profesor de bioestadística en la Universidad de Harvard. Uno de sus libros es Introduction to Data Science, disponible gratuitamente en Leanpub (https://leanpub.com/datasciencebook). También hay la versión en español (http://rafalab.dfci.harvard.edu/dslibro/). Echad también un ojo a sus cursos en edX (https://www.edx.org/es/bio/rafael-irizarry). Aunque son de pago si queréis el certificado, el material está accesible gratuitamente durante bastante tiempo.
- Cursos gratuitos en Datacamp: https://www.datacamp.com/courses/free-introduction-to-r
- Esta página es muy resolutiva en temas de estadística: http://www.sthda.com/english/wiki/what-is-r-and-why-learning-r-programming

La forma de aprender a usar R es usándolo. Buscad algo que os motive. Un buen ejercicio puede ser usar datos de vuestros proyectos que sean sencillos y podáis comparar vuestros análisis con R y con la herramienta que hayáis empleado hasta ahora. En vuestra cuenta de Posit Cloud ahora tenéis guardado lo que habéis hecho en esta sesión. Os animamos a que instaléis R y RStudio en casa, a que os entretengáis un rato a repasar lo que hemos visto, a que os salgan errores, a intentar solucionarlo,... y a que poco a poco os vayáis animando a utilizarlo para analizar vuestros propios datos. Veréis que cuando os deis cuenta de que R es como un folio en blanco en el que podéis pintar con absoluta libertad, ¡no querréis volver a usar otra cosa!