

Introducción al análisis de datos con R

Índice

1 INTRODUCCIÓN A R Y AL ANÁLISIS DE DATOS	4
1.1 QUÉ ES R Y PARA QUÉ SIRVE	4
1.2 INSTALACIÓN DE R	5
1.3 INSTALACIÓN DE RSTUDIO	7
1.4 INTRODUCCIÓN AL ENTORNO DE PROGRAMACIÓN RSTUDIO	9
1.5 IDEAS CLAVE: INTRODUCCIÓN A R Y AL ANÁLISIS DE DATOS	11
2 PROGRAMACIÓN CON R.....	11
2.1 DIFERENTES TIPOS DE OBJETOS EN R (I)	12
2.2 DIFERENTES TIPOS DE OBJETOS EN R (II).....	14
2.3 CONDICIONALES.....	16
2.4 RESOLUCIÓN DE EJERCICIOS: CONDICIONALES	19
2.5 FUNCIONES (I)	21
2.6 FUNCIONES (II)	23
2.7 RESOLUCIÓN DE EXERCICIOS: FUNCIONES.....	25
2.8 BUCLES (I)	27
2.9 BUCLES (II)	28
2.10 RESOLUCIÓN DE EJERCICIOS: BUCLES.....	30
2.11 IDEAS CLAVE: PROGRAMACIÓN CON R	32
3 TRABAJAR CON BASES DE DATOS	33
3.1 IMPORTAR Y EXPORTAR BASES DE DATOS	33
3.2 FILTRAR BASES DE DATOS	35
3.3 RESOLUCIÓN DE EJERCICIOS: FILTRAR BASES DE DATOS.....	38
3.4 TRANSFORMAR BASES DE DATOS	40
3.5 RESOLUCIÓN DE EJERCICIOS: TRANSFORMAR BASES DE DATOS	43
3.6 IDEAS CLAVE: TRABAJAR CON BASES DE DATOS	44

4 VISUALIZACIÓN DE DATOS	45
4.1 INTRODUCCIÓN A LA VISUALIZACIÓN DE DATOS	45
4.2 GRÁFICOS DE BARRAS Y DIAGRAMAS DE SECTORES.....	48
4.3 HISTOGRAMAS Y DIAGRAMAS DE CAJA	50
4.4 NUBE DE PUNTOS Y RECTA DE REGRESIÓN (I).....	52
4.5 NUBE DE PUNTOS Y RECTA DE REGRESIÓN (II).....	54
4.6 PERSONALIZACIÓN DE GRÁFICOS (I).....	55
4.7 PERSONALIZACIÓN DE GRÁFICOS (II).....	57
4.8 AÑADIR LEYENDAS A NUESTROS GRÁFICOS	59
4.9 IDEAS CLAVE: VISUALIZACIÓN DE DATOS	61

1 INTRODUCCIÓN A R Y AL ANÁLISIS DE DATOS

En este primer módulo explicaremos el proceso de instalación de R y su interfaz gráfica RStudio.

En segundo lugar, presentaremos R como lenguaje de programación, haciendo especial énfasis en la utilidad práctica que podemos darle en nuestros procesos de análisis de datos.

Posteriormente, expondremos las principales opciones que nos ofrece esta interfaz gráfica y exploraremos las maneras más sencillas de sacarle el mayor potencial posible.

Aprenderemos a crear y guardar un script, a aprovechar la cuadricula de RStudio para gestionar la información disponible y hacer un uso eficiente de sus funcionalidades.

1.1 QUÉ ES R Y PARA QUÉ SIRVE

¡Hola a todos y todas!

R es un lenguaje de programación de licencia abierta y totalmente gratuito para la computación estadística y la creación de gráficos. Recientemente ha ido incrementando su popularidad en el ámbito de la ciencia de datos, y junto con Python y Java ocupa las primeras posiciones de este sector.

Como muchos otros lenguajes de programación, se fundamenta en la escritura en línea de comandos. Aún así, está extremadamente extendido el uso de la interfaz gráfica RStudio, por su practicidad y estructura. R también destaca por tener una comunidad muy activa que desarrolla paquetes, especialmente orientados a la modelización, la estadística más clásica y la visualización de datos, que son los puntos fuertes del lenguaje. R dispone de un repositorio de paquetes extensísimo, donde podemos encontrar grupos de prácticamente todas las variantes de la ciencia de datos.

A continuación, haremos una lista de algunas de las principales particularidades de R, que deberíamos conocer como usuarios y usuarias:

1. R es lento. Esto puede parecer curioso como primera característica, pero tiene una justificación. R fue diseñado para hacer fácil al usuario o usuaria la modelización y el análisis estadístico, no por ser el lenguaje más amable para un ordenador. Debemos ser conscientes de que no puede competir en velocidad con otros lenguajes, pero sí ganar en usabilidad.
2. R es extremadamente popular en algunos sectores, y en cambio, prácticamente desconocido en otros. Está prácticamente desaparecido en un entorno más orientado a la ingeniería o la informática, pero domina en la industria farmacéutica, las finanzas, el marketing, los audiovisuales y sobre todo la academia, ya que es donde más se usa la estadística formal, uno de los puntos más fuertes de R. Su importancia como herramienta de *Business Intelligence* es cada vez mayor, principalmente por la

simplicidad de su código y las facilidades que ofrece a la hora de hacer visualizaciones y resúmenes estadísticos.

3. Los modelos en R son increíblemente fáciles de usar. Como R es un lenguaje que prioriza la usabilidad, la modelización y su código asociado son mucho más sencillos que en otros lenguajes y resultan más cercanos para personas no expertas en *data science*.
4. R funciona mucho más rápidamente si utilizamos un tipo de código orientado al uso de estructuras vectoriales y planificamos correctamente las variables que necesitaremos. A diferencia de Python, por ejemplo, algunos bucles y especialmente la concatenación de variables pueden ser muy inefficientes. Pero, como ya hemos comentado, una buena planificación y el uso de alternativas que aprovechen la estructura vectorial del lenguaje para ejecutarse rápidamente pueden ayudar a compensar el punto anterior, que es su baja velocidad.
5. R es más complicado de aprender al principio. Sin embargo, cuando ya hayamos entrado en su lógica y ya hayamos realizado un par de proyectos, su código nos parecerá muy intuitivo y aprenderemos nuevas funcionalidades mucho más rápidamente. Los primeros pasos programando con R son más complejos que otros lenguajes más humanos como Python, donde cada tipo de objeto tiene unas funciones asociadas. En R, estas funciones, por ejemplo, las asociadas a un tipo concreto de modelo, tendremos que conocerlas de antemano o recurrir a la ayuda que nos ofrece el programa, lo que ralentiza el proceso de escritura del código al principio. Afortunadamente, la ayuda que nos ofrece R es muy completa y llena de ejemplos, y en la gran mayoría de paquetes incluyen buenos tutoriales.

En resumen, R es un lenguaje de programación orientado al análisis estadístico y a la visualización de datos. Puede ser más complicado de aprender, al principio, que otros lenguajes, pero, debido a su creciente popularidad y versatilidad, su dominio constituye una competencia de gran valor en el mercado laboral.

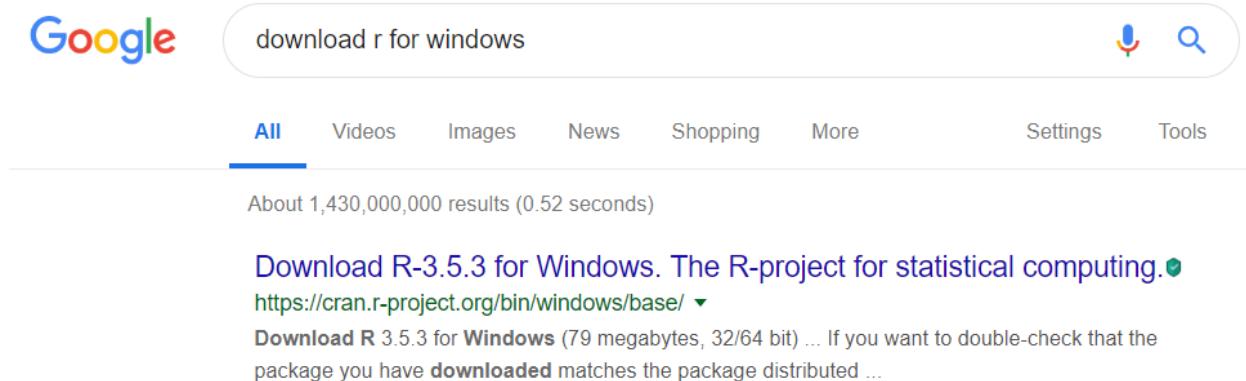
¡Seguimos!

1.2 INSTALACIÓN DE R

¡Hola de nuevo!

En este artículo vamos a avanzar en el proceso de instalación de R, veremos que es muy sencillo, independientemente del sistema operativo con el que estemos trabajando.

El primer paso que realizaremos es buscar "Install R" en Google y entraremos en la primera página que nos aparezca.



download r for windows

All Videos Images News Shopping More Settings Tools

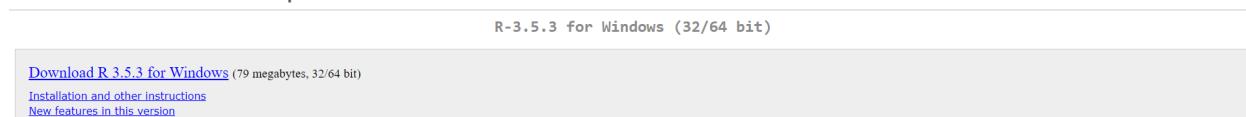
About 1,430,000,000 results (0.52 seconds)

[Download R-3.5.3 for Windows. The R-project for statistical computing. ✓](#)
[https://cran.r-project.org/bin/windows/base/ ▾](https://cran.r-project.org/bin/windows/base/)

Download R 3.5.3 for Windows (79 megabytes, 32/64 bit) ... If you want to double-check that the package you have **downloaded** matches the package distributed ...

Quizás la web no es suficiente amable, ya que se trata de un proyecto libre. Tendremos que hacer clic en "Download R", resaltado en negrita, y nos redirigirá a una página de servidores.

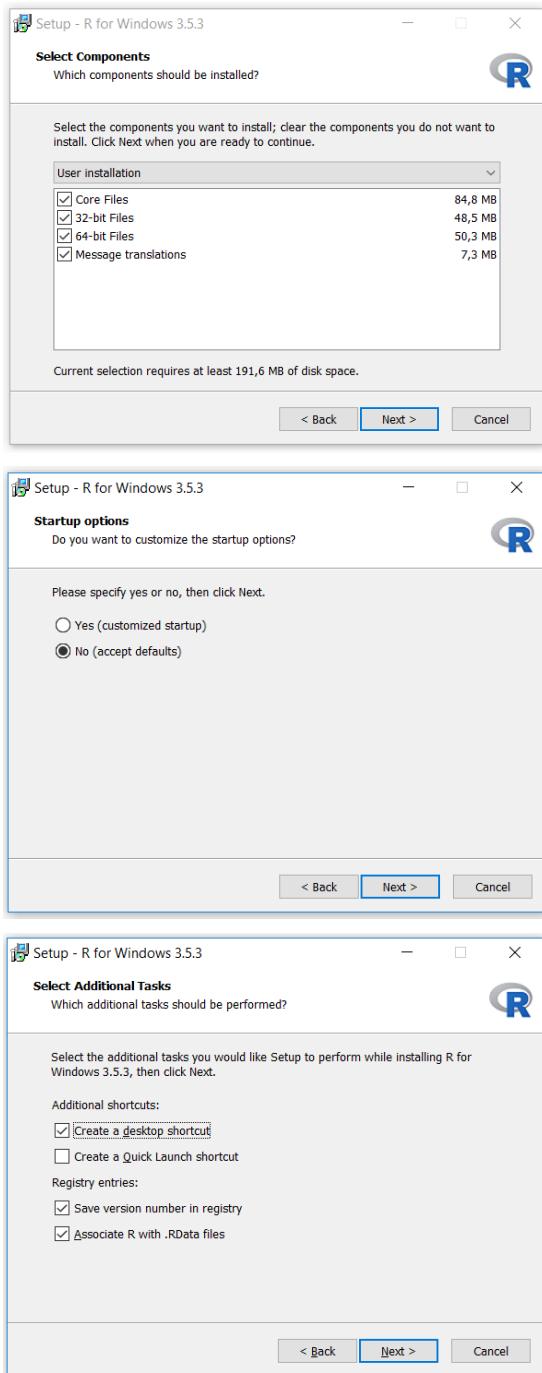
En principio, debería ser indiferente cuál de ellos seleccionamos, así que hacemos clic con el primero. A continuación deberemos seleccionar nuestro sistema operativo. En este caso, utilizaremos Windows para este tutorial, aunque los pasos son prácticamente idénticos para todos los sistemas operativos.



R-3.5.3 for Windows (32/64 bit)

[Download R 3.5.3 for Windows](#) (79 megabytes, 32/64 bit)
[Installation and other instructions](#)
[New features in this version](#)

En la página siguiente, volveremos a elegir la opción resaltada en negrita "Install R for the first time". El siguiente paso es el definitivo y descargaremos el archivo de instalación, que podemos ejecutar para que se nos abra el asistente de instalación.



¡Hasta pronto!

1.3 INSTALACIÓN DE RSTUDIO

¡Hola de nuevo!

En este artículo vamos a avanzar en el proceso de instalación de RStudio, una interfaz gráfica que nos facilitará enormemente el trabajo, a la vez que nos permitirá una visualización más

agradable y estructurada del entorno de programación. Como es de imaginar, debemos tener completamente instalada la última versión disponible de R para llevar a cabo este proceso.

El primer paso que realizaremos será buscar "Install RStudio" en Google y entraremos en la primera página que nos aparezca.

Google

download rstudio for windows

All Videos Images News Shopping More Settings Tools

About 2,750,000 results (0.45 seconds)

[Download RStudio - RStudio](#) https://www.rstudio.com/products/rstudio/download/ ▾

Feb 7, 2019 - Download the RStudio IDE or RStudio Server. ... RStudio 1.1.463 - Windows Vista/7/8/10, 85.8 MB, 2018-10-29 ...

[Download RStudio Server](#) · [Release Notes](#) · [Code Signing](#) · [IDE features](#)

Aquí nos ofrecerá varias opciones; nosotros escogeremos la primera de todas, que es gratuita y completamente funcional.

RStudio Desktop
Open Source License

FREE

DOWNLOAD

[Learn More](#)

Cuando pulsamos el botón "Download", nos llevará más abajo en la misma página, donde podremos elegir nuestro sistema operativo. Es tan sencillo como pulsar en la primera opción, en el caso de Windows.

Installers for Supported Platforms

Installers	Size	Date	MD5
RStudio 1.1.463 - Windows Vista/7/8/10	85.8 MB	2018-10-29	58b3d796d8cf96fb8580c62f46ab64d4

Se nos descargará el archivo que abrirá el instalador. Habrá que ir haciendo clic en siguiente hasta que se complete la instalación.

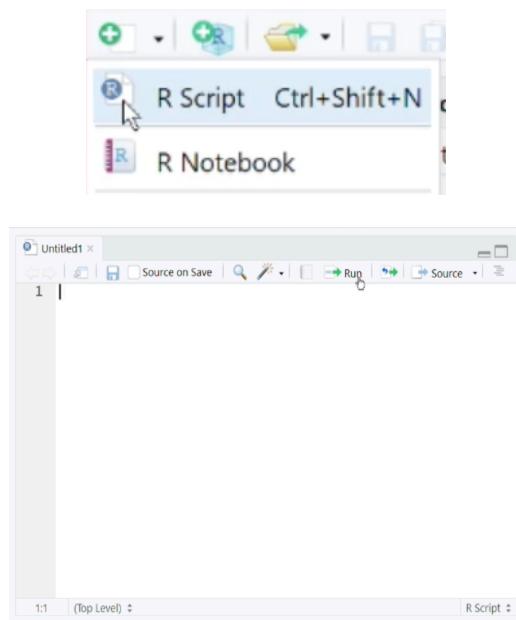
¡Continuamos con el curso!

1.4 INTRODUCCIÓN AL ENTORNO DE PROGRAMACIÓN RSTUDIO

¡Bienvenidos y bienvenidas!

En este vídeo presentaremos la interfaz gráfica RStudio, así como las funcionalidades más interesantes que nos puede ofrecer. Antes de comentar un poco más en detalle lo que nos encontraremos, debemos tener en cuenta un consejo: Si tenemos un cierto dominio del inglés, lo más práctico es ver la interfaz en este idioma. De este modo, en el caso de necesitar ayuda en algún momento, será mucho más sencillo encontrarla a través de Google y aplicarla.

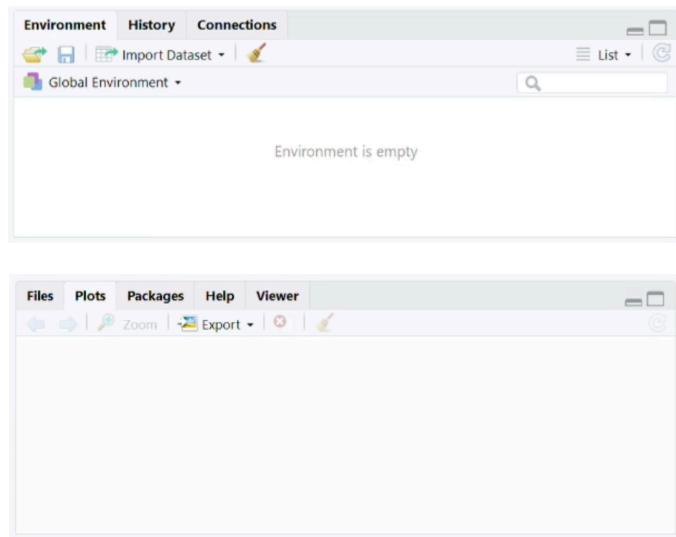
La estructura de RStudio se basa en una parrilla. Es decir, en cuatro elementos. Ahora mismo, sólo vemos tres. Lo que pasa es que todavía no hemos creado un script. El script lo podemos crear utilizando este icono aquí:



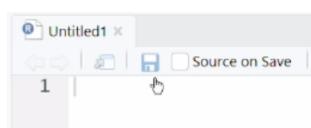
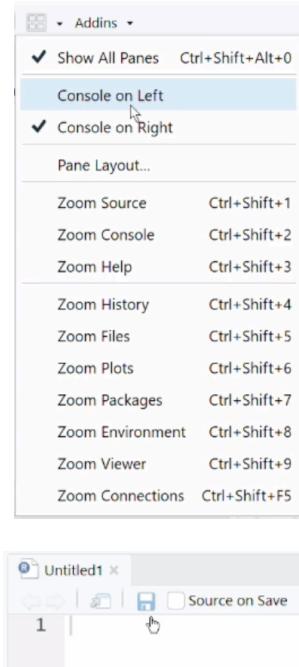
Y abriendo el documento donde escribiremos nuestro código. Todo lo que escribimos aquí y ejecutamos utilizando este botón aquí, lo veremos por la consola.



Aquí, alrededor, veremos los objetos que hemos creado y, por ejemplo, los gráficos que estemos generando.



Los elementos más importantes son estos dos de aquí arriba. Utilizando este botón aquí, podemos mover la consola de banda, por ejemplo. Lo hemos puesto a la izquierda y ahora tendríamos un espacio ajustable, donde podemos ver nuestro script, nuestra consola y, en otra parte, podríamos ver nuestro entorno. Guardar un script es tan sencillo como utilizar este botón de aquí.



Otra función muy importante que necesitamos saber si trabajamos con RStudio es esta aquí: el Working Directory.



Es decir, dónde queremos que se nos guarden los archivos. Si hacemos clic aquí, lo que estamos consiguiendo es que todos los gráficos que generamos y objetos nuevos que queramos guardar se guarden exactamente en la misma carpeta donde tenemos el script que acabamos de generar. También podemos escogerlo utilizando esta opción aquí. Aquí arriba tenemos una gran cantidad de menús que permiten personalizar como vemos nuestro RStudio. Tenemos las opciones globales, donde podemos, por ejemplo, ajustar la visualización de nuestro R.

Estos ajustes son puramente estéticos. Lo que es especialmente útil a la hora de utilizar RStudio, que no nos ofrece R, es esta visualización en formato parrilla, donde podemos ver nuestros gráficos, los paquetes que queramos cargar, ayuda, un entorno donde veremos las variables, un histórico del que hayamos hecho, una consola, el script y tendremos ayuda en el caso de necesitarla.

¡Seguimos!

1.5 IDEAS CLAVE: INTRODUCCIÓN A R I AL ANÁLISIS DE DATOS

En este módulo hemos realizado los primeros pasos con R y RStudio. A continuación, haremos un repaso de las ideas más importantes que han aparecido:

- R es un conocido lenguaje de programación que ha crecido enormemente en los últimos años. Se caracteriza por estar totalmente orientado a la usabilidad, modelización y visualización, y está ampliamente extendido en muchos sectores.
- Hemos instalado R y su interfaz gráfica RStudio, a la vez que hemos explorado las funcionalidades principales de RStudio, como por ejemplo:
 - Visualización de los objetos creados
 - Estructura y legibilidad del código
 - Asistentes de importación
 - Gestión de scripts y de proyectos

2 PROGRAMACIÓN CON R

En este módulo veremos un resumen muy breve de los fundamentos de R como lenguaje de programación.

Comenzaremos presentando los cuatro grandes tipos de objetos con los que podemos trabajar, cuáles son sus particularidades y para qué son útiles.

A continuación, veremos cómo funcionan las estructuras condicionales más sencillas y algunas más complejas, lo mismo con bucles de diferentes tipos, haciendo especial énfasis en sus condiciones de parada.

A lo largo de este módulo veremos, de manera transversal, algunas de las funciones básicas de R.

2.1 DIFERENTES TIPOS DE OBJETOS EN R (I)

¡Hola de nuevo!

En este vídeo veremos los diferentes tipos de objeto que podemos crear en R, cuál es su utilidad principal y cómo podemos tratarlos adecuadamente. Veremos cómo crear variables unitarias, vectores y matrices.

Lo primero que haremos será crear una variable unitaria. Le asignamos un nombre y, utilizando esta flecha, le asignaremos un número.

```
num <- 3
```

Para ejecutar esto que acabo de escribir, puedo apretar Run o puedo pulsar Control + Enter. Lo que podemos ver es que nos aparece en nuestro entorno esta variable que acabamos de crear y la instrucción que hemos ejecutado en nuestra consola.

```
num <- 3.0
```

Otra cosa que podemos hacer es crear un texto. Toda la creación sigue la misma estructura: un nombre y una flecha. Le pongo estas cuatro letras, ejecuto y ahora he creado una variable que se llama "texto" y que contiene estas cuatro letras.

```
texto <- "asdf"
```

Otro tipo de estructura es el vector. Le pondré el nombre "numeros" y con la flecha y esta instrucción aquí: c, abro paréntesis y le puedo añadir los números que quiera.

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

Ejecuto y veo que aquí abajo me dice el nombre, el tipo de variable que es (es decir, numérica), y que tiene tres posiciones, 1, 2 y 3.

Una manera alternativa de crear este mismo objeto podría ser esta de aquí.

```
numeros <- 1:5
```

Hagámoslo del 1 hasta el 5. Ejecuto y lo que veo es que me ha creado un objeto con una secuencia del 1 hasta el 5. Una consideración que hay que hacer sobre los vectores es que todos los elementos que lo componen deben ser del mismo tipo. Me explico: si yo creo este

objeto, pongo 1, 2, 3 y añado un 4 pero en formato texto, que es lo que delimitan estas comillas aquí.

```
numeros <- c(1,2,3, "4")
```

Ejecuto. Lo que habrá pasado es que he creado un objeto con cuatro posiciones donde todas ellas son variables textuales, es decir, el tipo más flexible de variable del texto.

Obtiene este nombre de variable.

```
vector_variado <- c(5, TRUE, "Hola")
```

Los espacios no están permitidos, así que añado esta barra baja. Puedo poner un número, una variable binaria y un texto. Si ejecuto esta instrucción, creo un nuevo vector donde todo es formato texto, y tengo un 5 en formato texto, una variable binaria en formato texto y un texto.

El último que veremos en este vídeo es cómo crear una matriz. Le pondré de nombre "matriz" y la crearé con la instrucción *matrix*, del 1 hasta el 12, y le puedo especificar el número de columnas que quiero que tenga. Le estoy diciendo: "ponme los números del 1 hasta el 12 en tres columnas".

```
matriz <- matrix(1:12, ncol = 3)
```

Ejecutamos y visualizamos el que acabamos de crear. Ejecuto y tengo esta estructura aquí: tres columnas, cuatro filas y los números me ha llenado en vertical. Puedo acceder a los elementos de esta matriz utilizando los corchetes. Por ejemplo, puedo coger el objeto que ocupa la primera fila y la segunda columna.

```
matriz [1,2]
```

En la primera fila y la segunda columna será el número 5.

Lo mismo podríamos hacer diciéndole que nos coja la primera fila y la segunda y tercera columna.

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

También podemos omitir uno de estos dos elementos y, si lo dejamos en blanco, lo que estaríamos haciendo es elegir sólo la primera fila.

```
matriz [1,]
```

2.2 DIFERENTES TIPOS DE OBJETOS EN R (II)

¡Hola!

En este segundo vídeo exploraremos cómo crear y manipular dataframes y listas, los objetos más versátiles y utilizados en R.

Lo primero que haremos es cargar el paquete datasets.

```
require(datasets)
```

Lo que obtendremos dentro de este paquete son un conjunto de bases de datos de ejemplo. Utilizaremos esta, que consiste en un conjunto de coches con sus características.

```
mtcars
```

Guardaremos esta base de datos como df, que es la notación clásica para dataframe.

```
df <- mtcars
```

El ejecutamos y lo que podemos ver es como accedemos a la información de una de las columnas. La sintaxis que se utiliza es el \$ y nos aparece una lista con las columnas que tiene este objeto. Si ejecuto esta instrucción, puedo obtener un vector con cada uno de los elementos de la columna de cilindros para cada uno de estos coches.

```
df $ CYL
```

Una forma alternativa de hacerlo es utilizando los corchetes.

```
df [ "CYL"]
```

El output que obtenemos es ligeramente diferente, ya que aquí nos aparecen los nombres de las filas, lo que es más práctico. Si quiero obtener varias columnas a la vez, puedo utilizar esta sintaxis aquí.

```
df [1: 3]
```

Si, en cambio, lo que hago es añadir una coma (,) y pongo un nombre de columna, como si fuera un texto, ahora lo que obtengo son los valores para las tres primeras filas de la columna "cilindros" (CYL).

```
df [1: 3, "CYL"]
```

Esta misma estructura la puedo utilizar en formato vectorial y lo que puedo conseguir es seleccionar más de una columna a la vez. Por ejemplo seleccionaremos la columna "mpg", tres filas y las dos columnas que le hemos pedido.

```
df[1:3, c("CYL", "mpg")]
```

Vamos a ver otro tipo de estructura, que es la lista. Una lista se crea utilizando esta función.

```
lista <- list()
```

Si la creamos vacía, tenemos una lista de cero elementos. Para añadir elementos a una lista, podemos utilizar también el \$. Por ejemplo, vamos a crear un primer elemento de esta lista y le diremos "objeto1", y aquí guardaremos un vector con tres números.

```
lista$cobjeto1 <- c(1,2,4)
```

Ahora tenemos una lista con un elemento, no con tres, ya que en esta lista el primer elemento es un vector. Lo que puedo hacer es crear otro. Aquí guardaremos "Esto es un texto" y tenemos una lista de dos elementos.

```
lista$cobjeto2 <- "Esto es un texto"
```

Como puedes comprobar, es un formato muy flexible, ya que puedo guardar un vector, un texto y también podemos guardar, por ejemplo, una fracción de nuestro dataframe. Por ejemplo, las tres primeras filas.

```
lista$cobjeto3 <- df[1:3,]
```

Para acceder a la información que hemos guardado, podemos visualizarla así y veríamos el primer elemento, un vector; el segundo elemento, un texto; y el tercer elemento, una base de datos. Podemos acceder o por los nombres, como hacíamos con los dataframes, o utilizando los dobles corchetes.

```
lista[[1]]
```

Podríamos obtener el primer elemento así y, de este primer elemento, podríamos obtener la segunda posición, utilizando esta sintaxis aquí.

```
lista[[1]][2]
```

Así pues, si quisiéramos explorar el dataframe que hemos guardado aquí, deberíamos hacer "lista", "\$", "objeto3"

```
lista $ objeto3
```

que sería esta estructura aquí. Y ahora podríamos hacer, por ejemplo, seleccionar la columna "CYL".

```
lista $ objeto3 [, "CYL"]
```

Y obtendríamos estos tres valores de aquí en formato vectorial.

2.3 CONDICIONALES

¡Hola a todos y todas!

A continuación, veremos qué son las condiciones en un lenguaje de programación y cómo podemos crear nuestras propias con R.

Una condición es aquella estructura de control que utilizamos para decidir, basándose en un cierto criterio, si queremos realizar alguna acción o no. La manera más sencilla de ver cómo funcionan las condiciones es con un ejemplo:

Lo primero que haremos es crear una nueva variable que le diremos "edad", que representará nuestra edad.

```
(Edad <- 26)
```

La estructura condicional que utilizaremos es esta de aquí.

```
if (edad >= 18) {
```

Donde, aquí dentro del paréntesis, especificaremos una condición, por ejemplo, si mi edad es mayor o igual a 18 años, y aquí dentro de estos corchetes qué acción queremos realizar.

```
print ("Mayor de edad")
}
```

Si ejecutamos esta variable y la estructura condicional, obtenemos que el individuo es mayor de edad; si, en cambio, modificamos esta variable, esta estructura no nos devuelve absolutamente nada.

```
Edad <- 16
if (edad >= 18) {
  print ("Mayor de edad")
}
```

Lo que podemos hacer, sin embargo, es mejorar esta estructura, diciéndole que queremos que haga en el caso contrario, "en caso de que esta condición no se cumpla, hazme esto de aquí".

```
else {
  print ("Menor de edad")
}
```

Si ejecutamos toda esta estructura, lo que nos devuelve es que el individuo es menor de edad.

Una manera alternativa de realizar estas estructuras es utilizando la función "ifelse". La función "ifelse" depende de tres parámetros: nuestra condición, que será la edad.

```
ifelse (test = edad >= 18)
```

¿Qué queremos que haga, si se cumple, esto de aquí.

```
ifelse (test = edad >= 18, yes = print ("Mayor de edad"))
```

Y, ¿qué queremos que haga, cuando no se cumple.

```
ifelse (test = edad >= 18, yes = print ("Mayor de edad"), no = print ("Menor de edad"))
```

Si lo ejecutamos, vemos que nos está dando la segunda respuesta.

Esta es una introducción muy breve el funcionamiento de las condiciones con R. Aquí hemos propuesto una estructura basada en una sola condición, que, si se cumplía, imprimía un resultado y, si no se cumplía, imprime otro. Evidentemente, podemos definir condiciones múltiples, como por ejemplo creando una nueva variable que sea nuestra edad.

```
edad <- 20
```

Y si el individuo es hombre o mujer.

```
sexo <- "H".
```

Utilizando la estructura "if", cabe preguntarse si la edad es mayor o igual a 18 años y utilizando "&" el sexo es igual a "Home". Esta estructura de aquí pregunta si se cumple esta condición y esta otra.

```
if (edad >= 18 & sexo == "H")
```

Aquí también hemos introducido la comparación de igualdad. Aquí estabamos haciendo una desigualdad en la que incluíamos igual ($>=$) y aquí estamos haciendo exactamente una igualdad ($==$). Queremos que nos diga "Hombre adulto".

```
if (edad >= 18 & sexo == "H") {
  print ("Hombre adulto")
}
```

Así pues, ejecutamos estas variables y, ahora, cuando ejecutamos esta condición múltiple, nos dirá que las cumple ambas, ya que es lo que le estamos preguntando. Por ejemplo, si cambiáramos "Home" en "Mujer", ahora ejecutamos esta condición y ya no se cumple.

`sexo <- "D".`

Podemos modificar estas condiciones para que se adapten a todo tipo de circunstancias. Por ejemplo, este símbolo aquí, ($|$) la raya en vertical, indica una condición u otra condición.

```
if (edad >= 18 | sexo == "H") {
  print ("Hombre adulto")
}
```

Si lo ejecutamos, nos saldrá "Hombre adulto". ¿Por qué? Porque está evaluando si la edad es mayor que 18, que en este caso es verdad, o si es hombre. En este caso no es verdad, pero como ya se cumplía la primera, nos imprime el que tenemos aquí dentro.

Una última cosa es que podemos modificar esta doble igualdad por una desigualdad. Aquí, lo que le estamos diciendo a R es que nos compruebe si el sexo no es igual, ($"! ="$) A hombre.

```
if (edad >= 18 | sexo != "H") {
  print ("Hombre adulto")
}
```

Y hasta aquí esta introducción a las condiciones. ¡Continuamos!

A continuación os proponemos unos ejercicios donde tendremos que utilizar múltiples veces la condicional () para generar estructuras condicionales más complejas. La dificultad de estas estructuras radica en donde definimos las diferentes condiciones, y en qué orden lo hacemos!

Ejercicios:

1. Crea el código que, facilitando un año de nacimiento concreto, te diga si has nacido antes del 80, entre 1980 y 1999, o a partir del año 2000.

2. La instrucción `ifelse()` también funciona cuando la aplicamos en vectores. Crea vector `c` `(-5, 4, 8, -1)` y, utilizándola, consigue el vector `c ("Negativo", "Positivo", "Positivo", "Negativo")`.
3. Utilizando los paréntesis adecuados, que determinan la importancia de las operaciones, y una sola condición, encuentra una manera de detectar si un número es inferior a 18 o superior a 99, y al mismo tiempo es par. Tendrás que utilizar la siguiente condición: `numero %% 2 == 0`. Explora el funcionamiento antes de comenzar el ejercicio. Qué hace `numero %% 2?`

Para resolver los ejercicios, encontraremos la solución a continuación.

2.4 RESOLUCIÓN DE EJERCICIOS: CONDICIONALES

¡Hola de nuevo!

A continuación veremos cómo resolver los ejercicios planteados.

Lo primero que pedíamos era hacer un filtro que clasificara un año en función de otras categorías.

`Año <- 1993`

Partiendo de este año, miraremos si es mayor o igual al año 2000. Si nos encontramos en los 80 o 90 o antes de los 80. Lo primero que hacemos es comprobar si este año es mayor o igual que el año 2000. En caso afirmativo, imprimimos esta instrucción.

```
if (año >= 2000) {
  print ("A partir de los 2000")
```

En caso contrario, realizamos toda esta estructura de aquí, que es un condicional en sí.

En caso de que esta condición no se haya cumplido, miraremos si el año es mayor o igual que en 1980. En caso afirmativo, nos encontramos en los 80 y 90.

```
} Else {
  if (año >= 1980) {
    print ("80s-90s")}
```

En caso contrario, por fuerza, nos encontraremos antes de los 80.

```
} Else {
  print ("Antes de los 80")
}
```

Si ejecutamos toda esta instrucción, obtenemos el resultado deseado.

```
if (año >= 2000) {
  print ("A partir de los 2000")
} Else {
  if (año >= 1980) {
    print ("80s-90s")
  } Else {
    print ("Antes de los 80")
  }
}
```

A continuación, mostraremos una solución para el segundo ejercicio, que básicamente te pedíamos que utilizando este vector aquí "c (-5,4,8, -1)" clasificaras los números según si son negativos o positivos. Esto es tan sencillo como utilizar la función ifelse aplicando la condición de que, si cada uno de los números es más pequeño que 0, ejecute esta parte de aquí ("Negativo") y, en caso contrario, esta aquí ("Positivo").

```
numeros <- c (-5,4,8, -1)
ifelse (numeros <0, "Negativo", "Positivo")
```

Elijo el vector, ejecuto y obtengo exactamente lo que me interesaba. Este resultado de ahí puedo guardar como un nuevo vector y así tengo los resultados en un objeto independiente.

```
nouvector <- ifelse (numeros <0, "Negativo", "Positivo")
```

Aquí he omitido la parte yes y no, ya que, por posición, la primera siempre ocupa la parte del yes y la segunda la parte del no.

```
nouvector <- ifelse (numeros <0, yes = "Negativo", no = "Positivo")
```

Y, en este último ejercicio, vemos como aplicar una condición múltiple. Lo que necesitamos es crear una variable, que será nuestro número.

```
numero <- 12
```

Y, dentro de la condición, le pedimos dos cosas, en realidad tres. Por un lado, que se cumpla una de estas dos (numero <18 | numero > 99) y por otra parte, que se cumpla esta (numero %% 2 == 0).

```
if ((numero <18 | numero > 99) & numero %% 2 == 0) {
```

Lo primero que hacemos es decirle que mi número sea menor que 18 o mayor que 99. Esto aquí se cumplirá, es decir será true, cuando mi número cumpla o ésta (numero <18) o esta

(numero > 99). Por otra parte, buscaré que el resto de la división de mi número por 2 sea igual a 0. También es cierto.

Si miramos las dos condiciones conjuntamente, debe cumplirse tanto esta (numero < 18 | numero > 99) como ésta (numero %% 2 == 0), ya que lo hemos especificado con el símbolo &.

Si ejecutamos toda esta instrucción:

```
numero <- 12
if ((numero < 18 | numero > 99) & numero %% 2 == 0) {
  print ("Cumple todas las condiciones")
}
```

Nos imprime por pantalla que el número cumple todas las condiciones.

2.5 FUNCIONES (I)

¡Bienvenidas y bienvenidos de nuevo!

En este vídeo presentaremos qué son y cómo se utilizan las funciones en R. Podemos simplificar la definición de qué es una función con la siguiente frase: "es toda aquella instrucción en R que va seguida de dos paréntesis". Normalmente, las funciones se caracterizan por llevar a cabo un proceso muy determinado sobre alguno de los objetos con los que estamos trabajando, aunque no siempre es así.

Los ejemplos más clásicos de funciones son aquellos que calculan estadísticos sobre un vector. Por ejemplo, si tenemos estas valoraciones de un producto, por ejemplo, y los aplicamos una función.

```
valoraciones <- c (7,8,6,10,5,7,4,6,10)
```

La más famosa de todas es la media.

```
mean(valoraciones)
```

La media, sencillamente, suma todos estos valores y los divide entre la longitud. Otra función sería, por ejemplo, la suma.

```
sum(valoraciones)
```

Y otra, la longitud.

```
length(valoraciones)
```

Utilizando la combinación de estas dos funciones,

`sum(valoraciones) / length(valoraciones)`

podríamos obtener exactamente lo mismo que en la función `mean`.

Otras funciones muy interesantes son, por ejemplo, la varianza;

`var(valoraciones)`

la desviación estándar, que básicamente es la raíz de la varianza;

`sd(valoraciones)`

el mínimo;

`min(valoraciones)`

el máximo;

`max(valoraciones)`

la mediana, que separa el 50% de las observaciones superiores del 50% de las observaciones inferiores;

`median(valoraciones)`

y, una de mis preferidas, la función `summary`,

`summary(valoraciones)`

que según el objeto sobre el que la aplicamos, obtenemos un resultado u otro. Sobre un vector, nos dará el mínimo, el primer cuartil, la mediana, la media, el tercer cuartil y el máximo.

El primer y el tercer cuartil son los puntos que separan los 25% de las observaciones por debajo y el 25% de las observaciones por encima. Si quisieramos obtener información más detallada de los cuantiles, podríamos utilizar la función `quantile`.

`quantile(valoraciones)`

Si lo ejecutamos, nos dará el mínimo, el 25, el 50, el 75 y el 100. Exactamente los mismos que nos estaba dando la función `summary`. Pero podemos especificarle, por ejemplo, qué probabilidades queremos. Si queremos el cuantiles 40%, lo podemos hacer con esta instrucción.

```
quantile(valoraciones, probs = .4)
```

2.6 FUNCIONES (II)

Hemos empezado explicando cómo aplicar funciones ya existentes en R, ya que con la paciencia suficiente y con la ayuda de R o Google, seguramente seamos capaces de encontrar una función que realice exactamente el cálculo o proceso que queremos en prácticamente todas las situaciones .

Aún así, siempre podremos definir nuestras propias funciones. ¿De qué nos servirá esto? Pues permitirá guardar un código que queramos aplicar muchas veces de una manera muy compacta y definir el tipo de salida o resultado que deseamos.

Veamos un par de ejemplos muy sencillos:

Primero, vamos a crear nuestra propia función para calcular la media. Definimos un nombre de función, le diremos "media", y, utilizando la instrucción function, que dependerá de unos parámetros que pondremos aquí dentro, realizaremos un cálculo.

```
media <- function () {  
}
```

El cálculo que realizaremos es lo que os hemos mostrado antes, la suma de X dividido por la longitud. Así pues, mi función media dependerá de un valor X, un parámetro X, y lo que hará es sumar todo y dividirlo por su longitud, exactamente el cálculo de su media.

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

Si ejecuto esta función, digamos que me aparece como una nueva función, la que acabo de definir ahora mismo. Si creo un objeto y le digo "números", del 1 al 5, por ejemplo.

```
numeros <- c (1,2,3,4,5)
```

Puedo ejecutar directamente y me calcula la media de estos números.

```
media (numeros)
```

Esto que acabo de escribir es equivalente a utilizar la instrucción return.

```
media <- function (x) {  
  return (sum (x) / length (x))
```

}

Esto aquí, básicamente, lo que especifica es lo que quiero que me devuelva la función. Si no lo especificamos, sencillamente devuelve el último que encuentra, y en este caso era equivalente.

En segundo lugar, utilizando las condiciones que ya hemos visto anteriormente, detectaremos si nos encontramos en un año bisiesto (básicamente si el año es divisible por 4, para simplificar) o no.

Así pues, creamos esta función, que dependerá de un año.

```
anytraspas <- function (año)
```

Y ahora, aquí dentro, vamos a crear una estructura condicional: si el año dividido por 4 da exacto, nos devolverá "El año tiene 366 días".

```
if (año %% 4 == 0) {
  return ( "El año tiene 366 días")
}
En caso contrario, nos devolverá "El año tiene 365 días".
else {
  return ( "El año tiene 365 días")}
```

Como puedes comprobar, podemos utilizar la instrucción return tantas veces como queramos. Y siempre termina lo que hace la función. Así pues, si le añadiéramos un return aquí abajo, como que habría encontrado uno de estos dos, este que hubiéramos puesto aquí ya no se ejecutaría.

```
anytraspas <- function (año) {
  if (año %% 4 == 0) {
    return ( "El año tiene 366 días")
  } Else {
    return ( "El año tiene 365 días")
  }}
```

Ejecutamos la función y ahora veremos lo que nos devuelve si le ponemos 2004, por ejemplo, era un año bisiesto.

```
anytraspas (2004)
```

O en 2006, que no lo era.

```
anytraspas (2006)
```

Al ser una función, podemos guardar este resultado en una variable. "Resultado 2006", por ejemplo.

```
resultat2006 <- anytraspas (2006)
```

Ejecutamos y ahora vemos que se nos ha ejecutado una variable textual que nos dice que el año tiene 365 días.

A continuación, proponemos algunos ejercicios sobre funciones. Encontraremos la solución de cómo hacerlo más adelante:

1. Define una función que imprima en pantalla la diferencia entre el máximo y el mínimo de un vector.
2. Modifica la función anterior para que, en el caso de recibir un solo número en lugar de un vector de número, imprima en pantalla una advertencia. Puedes hacerlo con la función length () .
3. Si definimos una función como multiplicar <- function (a, b) {return (a * b)} y lo llamamos como multiplicar (2,3), obtendremos el número 6. Así, podemos especificar funciones que dependan de más de una variable. Crea una función que dependa de dos números. La función debe devolver el número más grande e imprimir en pantalla si alguno de los números es negativo.

2.7 RESOLUCIÓN DE EXERCICIOS: FUNCIONES

¡Hola de nuevo!

Veamos cómo resolver los ejercicios de funciones.

Lo primero que he hecho es definir un vector y un número, que es el que utilizaremos para ver cómo funcionan nuestras funciones.

```
vector <- c (1,4,5,7,4,2,4,6,8,9,10,2)
numero <- 7
```

La primera que hemos pedido es crear es la función rango, es decir, la diferencia entre el máximo y el mínimo.

```
rango <- function (x) {
  max (x) - min (x)
}
```

Aquí, como sencillamente estamos aplicando esta diferencia, no hace falta utilizar la función return. Si ejecutamos esta función y ahora la llamamos sobre el vector, nos dirá la diferencia entre el número mayor, 10, y el número más pequeño, 1.

rango (vector)

Si ejecutamos esta misma función sobre el número, nos da rango 0.

rango (numero)

El segundo ejercicio busca corregir esto, es decir, busca que nuestra función entienda que, si no le estamos entrando un vector, es decir, si la longitud del objeto que entra es igual a 1, es decir, un número, no un vector, nos dirá: "eh, has introducido un número, no un vector". En caso contrario, será exactamente lo que le hemos pedido. Observa que en esta primera condición no devuelve nada. Sencillamente avisa por pantalla.

```
rango <- function (x) {
  if (length (x) == 1) {
    print ("Has introducido un número, no un vector")
  } Else {
    return (max (x) - min (x))
  }
}
```

Si ejecutamos esta función, que estamos sobrescribiendo el anterior, ahora podemos probar qué hace cuando ejecutamos el rango del número. Ya no nos da 0, como anteriormente, sino que nos dice: "Has introducido un número, no un vector".

En el último ejercicio te pedía una función que te dijera cuál de los dos números que le dabas es más pequeño que el otro, y que te avisa si al menos uno de los dos es negativo.

```
mesg <- function (numero1, numero2) {
  if (numero1 < 0 | numero2 < 0) {
    print ("Al menos uno de los números es negativo")
  }
  if (numero1 > numero2) {
    return (numero1)
  }
  return (numero2)
}
```

Estas dos condiciones no son excluyentes, es decir, primero lo que haremos es comprobar si uno de los dos números es menor que 0, utilizando esta condición (`numero1 < 0`) o esta otra (`numero2 < 0`); que nos avisará que al menos uno de los dos es negativo y, independientemente de lo que haya pasado hasta ahora, mirará si el número 1 es mayor que el número 2: nos devolverá el número 1, es decir, el mayor; y, en caso contrario, nos devolverá el número 2. Aquí no es necesaria la estructura `else`, ya que, si utilizamos este `return` (return

(numero1)), salimos de la función y, si esta condición no se ha cumplido, llegamos aquí (return (numero2)) y devolverá el número 2.

Vamos a ver cómo funciona esta función. La ejecutamos y la aplicamos sobre el número 7 y el -8, por ejemplo.

```
mesg (7, -8)
```

Ejecutamos y nos da dos resultados: por un lado, lo que nos está devolviendo, es decir el número más grande y, por otro lado, nos avisa de que al menos uno de los dos números es negativo.

Si guardáramos el resultado de esta función en un objeto, por ejemplo, le diremos "obj",

```
obj <- mesg (7, -8)
```

si miramos lo que estamos haciendo es mostrar un resultado por pantalla que no se guarda

```
obj <- mesg (7, -8)
obj
```

y, dentro del objeto, hemos guardado el número 7, como podíamos ver en nuestro entorno.

2.8 BUCLES (I)

¡Bienvenidas y bienvenidos de nuevo al vídeo!

A continuación expondremos de manera muy breve qué son los bucles y su utilidad práctica, ya que son una de las estructuras más importantes de los lenguajes de programación, junto con los condicionales y los objetos.

Un bucle, básicamente, es un proceso que repetiremos tantas veces como definimos, y que normalmente se utiliza para explorar los diferentes elementos de un objeto, o realizar alguna operación hasta que se satisfaga alguna condición predefinida.

Veamos un ejemplo práctico. Si queremos explorar este vector, esta lista de números del 1 hasta el 20, lo que hará el bucle FOR que acabamos de definir, que es seguramente el más popular de todos, es ir elemento por elemento y aplicar el proceso que hayamos definido en su interior.

```
listanumeros <- 1:20
```

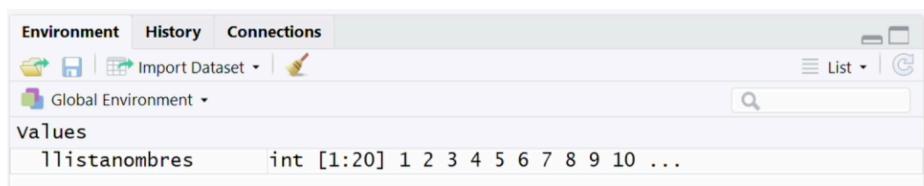
Así pues, defino el bucle con unos paréntesis.

Dentro de los paréntesis, le digo un nombre de una variable, que es arbitrario. Aquí hubiera podido ponerle cualquier nombre que yo hubiera querido y, por cada elemento que encuentre

dentro de este vector o lista, irá llamándole y y aplicará esta operación que le estoy pidiendo. En este caso, le estoy que me imprima cada uno de estos elementos que encuentre en la lista de números elevado al cuadrado. Esta es la sintaxis para elevar un número al cuadrado.

```
for (i in listanumeros) {
  print (y ^ 2)
}
```

Así pues, antes de ejecutar, lo que interesa ver es que tomará el primer número, es decir el 1. Este de aquí.



Este número lo guardará como y y la elevará al cuadrado y la imprimirá por pantalla. Llegará aquí y seguirá, ya que aún quedan números en la lista para explorar. El siguiente número es el 2. Guardará con el nombre de y este valor y lo imprimirá elevado al cuadrado. Llegará aquí y, como todavía quedarán números para explorar, cogerá el 3, el 4, etc. hasta llegar al 20, para cada uno de los elementos que haya encontrado en esta lista. Si ejecutamos este vector, vemos que ejecuta 20 veces la instrucción print.

¡Y seguimos!

2.9 BUCLES (II)

Hola de nou!

Otra manera muy clásica de definir los loops o bucles es, en vez de iterar por cada uno de los elementos de una lista, como hemos hecho anteriormente, hacerlo para cada uno de los índices.

```
listanumeros <- c (7,8,9,2,4,5,6,1,7,8,9,10)
for (i in listanumeros) {
  print (y ^ 2)
}
```

Así pues, podemos transformar el bucle que teníamos, utilizando no los valores en sí, es decir 7, 8, 9, 2 ... sino la posición que ocupan en el vector, es decir, 1, 2, 3, 4, 5 , etc. Esto es tan sencillo como transformar la lista sobre la que iter en esta lista de posiciones, del 1 hasta la longitud de nuestro vector.

```
listanumeros <- c (7,8,9,2,4,5,6,1,7,8,9,10)
```

```
for (i in 1: length (listanumeros)) {
print (y ^ 2)
}
```

Si ejecutamos este bucle, obtendremos los números del 1 al 12 elevados al cuadrado.

Pero eso no es lo que nos interesa, sino que nos interesa que el número que ocupa la primera posición del elevamos al cuadrado. Así pues, tenemos que repensar esta instrucción de aquí y transformarla.

```
print (listanumeros [y] ^ 2)
```

De la lista de números, tomaremos, a cada vuelta, la posición "y", y acabamos de conseguir el mismo bucle que teníamos antes.

```
listanumeros <- c (7,8,9,2,4,5,6,1,7,8,9,10)
for (i in 1: length (listanumeros)) {
print (listanumeros [y] ^ 2)
}
```

Finalmente, otra estructura muy útil es el while, que funciona de manera bastante diferente, ya que el criterio de parada es más complejo. Por ejemplo, podemos definir un número:

```
numero <- 1
```

El 1, por ejemplo, y, con esta función aquí, que tendrá una condición de parada aquí dentro, y las acciones que queramos hacer, y luego pondremos que, mientras el número sea menor que 1000, mientras esto se cumpla, nos imprima el número y, cada vez que lo haga, este número se guarde como él mismo multiplicado por 2.

```
while (numero < 1000) {
print (numero)
numero <- numero * 2
}
```

Así pues, empezamos con un 1 y, en cada vuelta, vamos doblando este resultado. Cuando este resultado excede el número que hemos fijado aquí, es decir el 1000, saldremos del while y seguiríamos con nuestro programa.

Ejecutamos y lo que vemos es que nos ha hecho 1, 2, 4, 8 ... Nos lo imprimiendo, porque le hemos pedido, y nos imprime hasta el 512, ya que tenemos un 256, que multiplicado por 2 es 512. El 512 cumple esta propiedad. Imprime el 512, lo multiplica por 2 y el guarda. Este valor aquí ahora vale 1024. Como a 1024 ya no cumple esta condición, no entra dentro de este espacio de aquí dentro y sale del bucle.

¡Continuamos!

A continuación proponemos algunos ejercicios sobre bucles. Encontrarás la solución de cómo hacerlo más adelante:

1. Explora el uso de la instrucción break. Programa un bucle que recorra los números del 1 al 100 mientras los imprime en pantalla. Mediante una condición, dile que ejecute break cuando el número en el que se encuentre sea el 30. ¿Qué ha pasado?
2. Crea un bucle que itere sobre un vector que hayas definido tú, que tenga tanto números positivos como negativos. Para los números positivos, queremos que diga si son pares o no. Para los negativos, que los imprima en pantalla en sentido positivo, si son superiores a -10. Si el número es menor que -10, queremos que el bucle pare.
3. Crea dos números, los que quieras, y, mediante un bucle while, ve doblando a ambos en cada iteración. Detener el bucle, cuando el mayor sea menos 1000 unidades mayor que el pequeño. Piensa detenidamente en cómo puedes definir esta condición de una manera sencilla, para que el bucle se ejecute mientras no se cumpla.

2.10 RESOLUCIÓN DE EJERCICIOS: BUCLES

¡Bienvenidos de nuevo!

Veamos cómo se resuelven los ejercicios de bucles.

El primero de todos, que es el más sencillo, se basa en crear una estructura de bucle for simple, en la que le pedimos que, cuando encuentre el valor $y = 30$, salga del bucle. ¿Qué pasará cuando ejecutamos esto? Que irá imprimiendo todos los números del 1 al 100 hasta que encuentre el 30. A partir de ahí, saldrá del bucle. Esto implica que imprimirá el 1 hasta el 29 y el 30 no llegará a imprimirllo, porque saldrá en este momento aquí.

```
for (i in 1: 100) {
  if (y == 30) {
    break
  }
  print (i)
}
```

Si lo ejecutamos, podremos ver efectivamente que llegamos hasta el número 29.

El segundo ejercicio es más rebuscado. Lo que buscamos es definir un vector de números positivos y negativos, y aplicarles una cierta acción o una cierta otra, en función de su valor. Lo primero que buscamos es si el número es positivo. En este caso, realizamos todas estas acciones de aquí dentro.

```
vectordefinido <- c (1, -1, 3, 4, -5.20, -12, 4, 5)
for (i in vectordefinido) {
```

```

if (i >= 0) {
  if (y %% 2 == 0) {
    print ("Par")
  } Else {
    print ("Impar")
  }
} Else {
  if (y < (-10)) {
    break
  }
  print (-y)
}
}

```

En función de si es par, utilizando esta sintaxis aquí ($y \% \% 2 == 0$), o si no lo es, es decir, de lo contrario. Aquí lo único que hacemos es imprimir, si es par, si cumple la primera condición, estando dentro de los números positivos; o, de lo contrario, imprimimos que es impar.

Si no estamos en el caso de números positivos, entramos esta parte de aquí al final.

```

else {
  if (y < (-10)) {
    break
  }
  print (-y)
}

```

Aquí, lo que hacemos es comprobar si el número es menor que -10. Si el número es menor que -10, es decir, -11, -12, etc., salimos del bucle. En caso contrario, mostramos el número cambiado de signo, que se puede hacer sencillamente añadiendo un menos ("-") frente al valor sobre el que ITER. Ejecutamos el vector, el bucle, y vemos que se ejecuta correctamente, ya que obtenemos "Impar", el número en positivo, "Impar", "Par", el número en positivo, "Par"; y este número de aquí y los subsecuentes (-12,4,5) ya no aparecen, ya que hemos accedido aquí (break) y hemos roto el bucle.

Y, ya para terminar, lo que buscamos es crear dos números

```

a <- 2
b <- 1

```

y, utilizando un bucle while, buscamos si la diferencia entre estos números (yo lo he hecho utilizando la función "absoluto" (abs), que lo que hace es que nos calcula la diferencia sin importar si el mayor es el primero o el segundo), si esta diferencia es menor que 1000.

```
while (abs (a-b) <1000)
```

Mientras esto se cumpla, es decir, mientras la diferencia entre a y b sea menor que 1000, vamos doblando los números utilizando estas instrucciones aquí.

```
a <- a * 2
b <- b * 2
```

Una vez se cumpla, salimos del bucle y imprimimos los resultados.

```
print (a)
print (b)
```

lo ejecutamos

```
while (abs (a-b) <1000) {
  a <- a * 2
  b <- b * 2
}
print (a)
print (b)
```

y vemos que, al ser potencias de 2, lo que estamos obteniendo es 2048 y 1024.

Podemos hacer las pruebas con otros valores y veríamos cuando se cumple que esta diferencia, es decir, que el resto entre a y b sea mayor que 1000.

La ventaja de utilizar esto aquí ($\text{abs} (\text{a}-\text{b})$) es que, si yo pongo un 1 y pongo un 5, ejecuto estas variables, obtengo los mismos resultados que no hubiera obtenido si hubiera hecho $\text{a} - \text{b}$ directamente.

Y eso es todo, ¡seguimos!

2.11 IDEAS CLAVE: PROGRAMACIÓN CON R

Ya estamos en el final del segundo módulo. A continuación veremos las ideas y los conceptos más importantes que hemos desarrollado:

- Los diferentes tipos de objeto que se pueden crear en R son:
 - Variables unitarias: utilizadas para guardar valores únicos del tipo que sea.
 - Vectores: conjuntos de variables unitarias del mismo tipo.
 - *Dataframes*: estructuras similares a las hojas de cálculo de Excel.
 - Listas: parecidos a los vectores, pero más flexibles.

- Hemos aprendido cómo funcionan las estructuras condicionales y las comparaciones en R.
 - Por un lado, podemos utilizar *if* y *else*, los condicionales más clásicos para definir procesos diferenciados.
 - Por otro lado, la función compacta *ifelse*, que permite condensar este mismo proceso binario en una única función.
- Hemos visto cómo utilizar las funciones ya existentes en R y qué estructura general siguen, así como una breve introducción a la creación de funciones propias, lo que nos permitirá reproducir procesos repetitivos de análisis muy fácilmente.
- Por último, hemos aprendido a diseñar y aprovechar estructuras en bucle, es decir, procesos iterativos (como el *for* o el *while*) sobre nuestros objetos (listas, dataframes o vectores) y cómo podemos especificar sus condiciones de parada.

3 TRABAJAR CON BASES DE DATOS

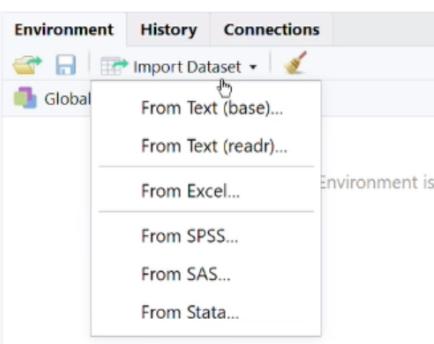
En este módulo veremos cómo utilizar el importador de bases de datos de RStudio para diferentes formatos, y cómo podemos guardar los objetos que hayamos creado, como por ejemplo, las bases de datos en formatos propios de R o Excel. También veremos los principales tipos de filtro y transformación que podemos aplicar a nuestros dataframes, para obtener aquellos datos que nos interesan en el formato deseado.

3.1 IMPORTAR Y EXPORTAR BASES DE DATOS

¡Hola de nuevo!

A continuación veremos cómo utilizar el importador de bases de datos de RStudio, que simplifica enormemente las tareas de importación en el formato correcto, y cómo podemos guardar los objetos con los que trabajamos, ya sea en un formato legible por otro programa como puede ser Excel u OpenOffice, o en el formato nativo de R.

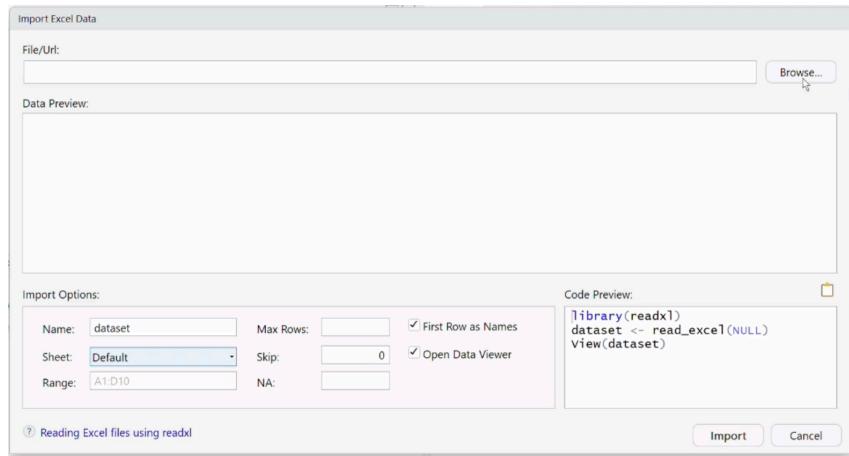
El importador de bases de datos se encuentra en el entorno. Debemos hacer clic aquí y tenemos varias opciones.



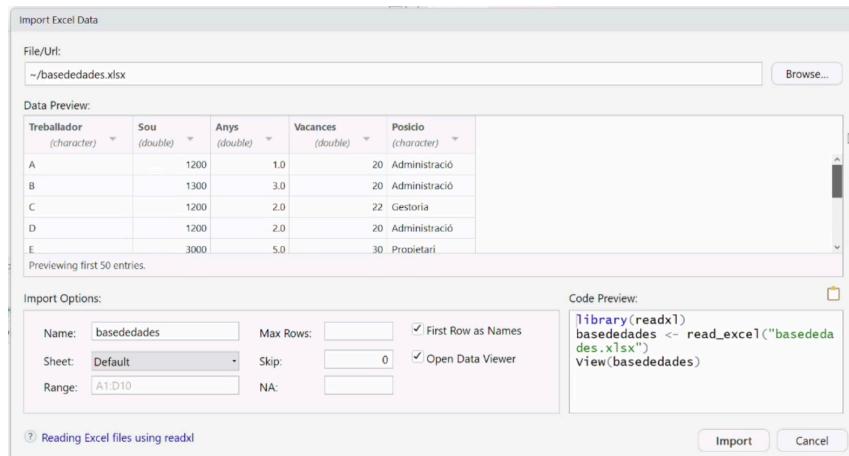
Estos tres aquí son software estadístico.

Tenemos Excel y dos importadores de texto. Mi preferido es este segundo (From Text (readr)). Nosotros lo que haremos es utilizar el importador de Excel.

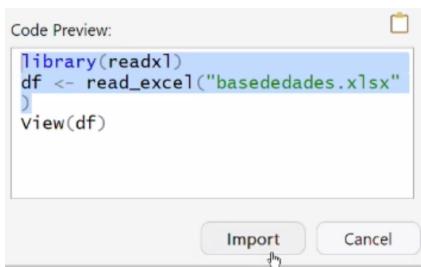
Tendremos que seleccionar el archivo que nos interesa, haciendo clic aquí.



Aquí, ya tengo varios ficheros. Cogeré el primero y nos mostrará, en este recuadro aquí, una primera visualización.



Puedo modificar su nombre, por ejemplo, ponerle la notación estándar `df`; seleccionar el número máximo de filas que quiero importar, y aquí me da un código muy útil que podré copiar y pegar, para luego ahorrarme hacer este proceso.



Se nos abre una visualización de la base de datos, se ha ejecutado el código automáticamente y ya podemos ir a nuestro script y pegar el código que acabamos de copiar.

Guardar objetos

Si lo que queremos es guardar un objeto (normalmente un dataframe, aunque no siempre sea así) para usarlo en I posteriormente, podemos utilizar la siguiente instrucción.

```
save (df, file = "datos.Robj")
```

Donde tendremos que especificar el objeto que queremos guardar, en este caso nuestro dataframe, y donde queremos que lo guarde.

El formato Robject es un formato nativo de R, que permite la importación y la exportación de manera muy sencilla. Para ver cómo funciona, lo ejecutamos. Se nos guarda donde tengamos especificado nuestro directorio de trabajo. Borramos el objeto que nosotros teníamos y, ahora, utilizando la función load y el nombre del archivo con el que estemos trabajando, volvemos a recuperar el objeto que teníamos inicialmente.

```
load ( "datos.Robj")
```

Si lo que queremos es exportarlo en formato .csv, que es perfectamente legible con Excel, sencillamente tendremos que utilizar la función write_csv, que forma parte del paquete readr.

```
require (readr)
```

Cargamos este paquete y, utilizando esta función, donde le tendremos que especificar el objeto que queremos guardar otra vez, y donde lo queremos guardar.

```
write_csv (df, path = "df.csv")
```

Estamos guardando este archivo en un archivo legible con procesadores de cálculo como Excel.

3.2 FILTRAR BASES DE DATOS

¡Hola a todos y todas!

En este vídeo veremos como podemos tratar nuestros dataframes de manera que obtengamos sólo aquellas datos que deseamos, ya sea desde un punto de vista puramente estético, como puede ser ordenándolas o visualizando una pequeña parte, o aplicando filtros a valores de nuestras variables.

Una primera función que veremos es la función head.

head (df)

Esta función permite seleccionar sólo las primeras filas de nuestra base de datos. Del mismo modo, podemos utilizar la función tail, para ver las últimas.

tail (df)

Si lo que queremos es seleccionar sólo las filas que cumplan una cierta propiedad, por ejemplo, los trabajadores y trabajadoras con más de dos años en la empresa, podríamos hacer esto: aplicar un filtro donde queremos que todas las filas cumplan una propiedad de una de las columnas; que el valor de esta columna sea mayor o igual a dos años.

```
df [, df $ Años >= 2,]
```

Así pues, lo que obtendremos es toda nuestra base de datos, donde esta columna tenga un valor 2 o superior. Podríamos guardar este objeto con el nombre "trabajoantiguo".

```
trabajoantiguo <- df [, df $ Años >= 2,]
```

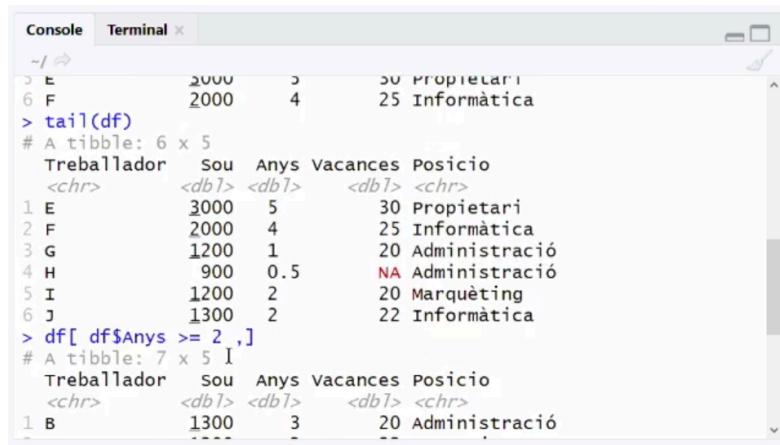
Aquí vemos que tres de las observaciones que teníamos, ahora ya no disponemos de ellas.

Los filtros que podemos aplicar a una base de datos son de todo tipo. Por ejemplo, lo que podría hacer es buscar únicamente el trabajador con cierto nombre, por ejemplo, vamos a buscar el trabajador "F".

Podríamos hacerlo utilizando la base de datos original, seleccionando la fila que cumpla que el nombre del trabajador, que es una columna (estoy utilizando el tabulador para autocompletar, cuando me sale la ayuda en poner el dólar), sea igual a exactamente lo valor que aparece aquí. Una buena práctica es copiarlo exactamente igual.

```
df [, df $ Trabajador == "F",]
```

Obtengo la información, es decir, todas las columnas, de este trabajador. Si os habéis fijado, en esta base de datos, tengo un dato que falta, que aparece resaltada aquí.



```

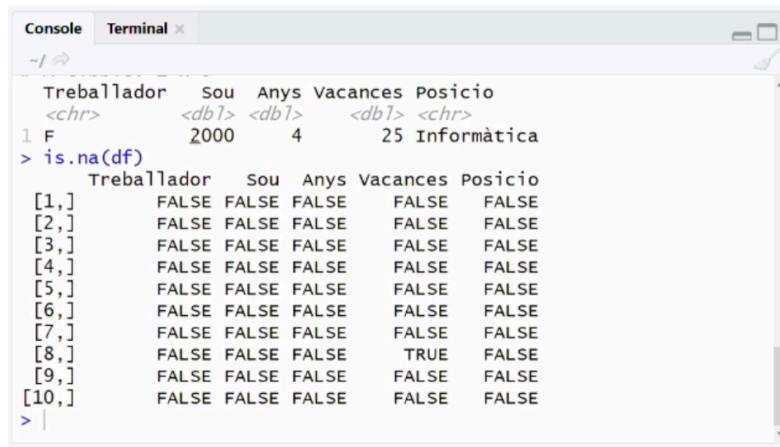
Console Terminal ×
~/🔗
> E      3000  5    30 Propietari
6 F      2000  4    25 Informàtica
> tail(df)
# A tibble: 6 x 5
  Treballador Sou Anys Vacances Posicio
  <chr>       <dbl> <dbl>   <dbl> <chr>
1 E           3000     5       30 Propietari
2 F           2000     4       25 Informàtica
3 G           1200     1       20 Administració
4 H            900     0.5      NA Administració
5 I           1200     2       20 Marquèting
6 J           1300     2       22 Informàtica
> df[ df$Anys >= 2 ,]
# A tibble: 7 x 5
  Treballador Sou Anys Vacances Posicio
  <chr>       <dbl> <dbl>   <dbl> <chr>
1 B           1300     3       20 Administració

```

Puedo detectarla, utilizando esta función.

`is.na(df)`

Si la aplico, sobre todo en la base de datos, lo que obtengo es esto aquí, una matriz de falsos y verdades, cuando encuentra una dato faltante.



```

Console Terminal ×
~/🔗
> F      2000  4    25 Informàtica
> is.na(df)
  Treballador Sou Anys Vacances Posicio
  [1,] FALSE FALSE FALSE FALSE FALSE
  [2,] FALSE FALSE FALSE FALSE FALSE
  [3,] FALSE FALSE FALSE FALSE FALSE
  [4,] FALSE FALSE FALSE FALSE FALSE
  [5,] FALSE FALSE FALSE FALSE FALSE
  [6,] FALSE FALSE FALSE FALSE FALSE
  [7,] FALSE FALSE FALSE FALSE FALSE
  [8,] FALSE FALSE FALSE TRUE FALSE
  [9,] FALSE FALSE FALSE FALSE FALSE
  [10,] FALSE FALSE FALSE FALSE FALSE
> |

```

Te animo a que explores como utilizar este tipo de estructura para eliminar las filas que tengan valores faltantes.

Otra cosa que podemos hacer es aplicar filtros múltiples. Por ejemplo, podríamos filtrar los trabajadores y trabajadoras, a su sueldo, mayor a 1200, por ejemplo, y el número de días de vacaciones, superior a 20. Si no añadimos esta como aquí, nos daría un error.

`df[, df$Sueldo > 1200 & df$Vacaciones > 20,]`

Para terminar este vídeo, vamos a ver cómo podemos ordenar una base de datos. Lo que haremos es utilizar la función `order` y le debemos especificar una columna de nuestra base de datos que sea numérica, por ejemplo el sueldo.

`df[order(df$Sueldo),]`

Si ejecutamos esta instrucción, obtenemos el sueldo ordenado de menor a mayor. Esto lo podemos corregir, utilizando uno de los parámetros de la función. Queremos que sea decreciente? Sí. Le especificamos que queremos que el orden sea decreciente.

```
df [order (df $ Sueldo, decreasing = TRUE),]
```

Ejecutamos y ya tenemos nuestros trabajadores ordenados en función de su salario.

¡Seguimos!

A continuación proponemos algunos ejercicios sobre cómo filtrar bases de datos.

Encontraremos la solución de cómo hacerlo en este vídeo:

1. ¿Cómo utilizarías las funciones order () y tail () para obtener únicamente el nombre de la persona trabajadora con un sueldo más alto? Prueba de conseguir toda su información con la función max () .
2. ¿Cómo obtendrías todos y todas las trabajadoras con un sueldo de entre 1.250 y 2.500 €?
3. ¿Cómo encontrarías si la persona trabajadora con un sueldo más bajo también es quien tiene menos vacaciones?

3.3 RESOLUCIÓN DE EJERCICIOS: FILTRAR BASES DE DATOS

¡Hola de nuevo!

Vamos a ver cómo podemos resolver estos ejercicios.

Lo primero que pedía era, utilizando la función tail, obtener el trabajador con un sueldo más elevado. Para empezar, lo que tenemos que hacer es ordenar la base de datos.

```
tail (df [order (df $ Sueldo),])
```

Lo que estamos haciendo es aplicar un filtro que lo que nos da es toda la base de datos con el sueldo ordenado de manera creciente. Utilizando la función tail, lo que obtendríamos son los últimos registros de esta tabla. Podemos especificarle que sólo queremos el último de todos, con este número aquí.

```
tail (df [order (df $ Sueldo),], 1)
```

Si ejecutamos esto, obtenemos efectivamente el último registro de todos, es decir, el propietario.

Una forma alternativa de hacerlo es utilizando este filtro por filas, es decir, seleccionando todas las filas donde el sueldo sea exactamente igual al máximo del sueldo.

```
df [ , df $ Sueldo == max (df $ Sueldo),]
```

Esta sintaxis aquí, podemos simplificarla utilizando esta función de R, que básicamente nos da la posición donde se encuentra el máximo.

```
df [which.max (df $ Sueldo),]
```

Mirémoslo y, si lo ejecutamos, este valor aquí no es el sueldo máximo, evidentemente, sino que es, en la ordenación original, qué fila contiene el sueldo máximo. Si ejecutamos, vemos que obtenemos el mismo resultado que al principio.

El segundo ejercicio que propongo es buscar todos aquellos trabajadores que cumplan que su sueldo se encuentre entre 1.250 y 2.500 euros. Lo que tenemos que aplicar aquí, pues, es un filtro múltiple. Hay otro software donde esto se puede hacer más directamente. R también contempla algunas opciones para hacerlo con un intervalo directamente, aunque, para empezar, es recomendable utilizar estas condiciones múltiples.

```
df [df $ Sueldo > 1250 & df $ Sueldo < 2500,]
```

Si ejecuto esta parte de aquí (df \$ Sueldo > 1250 & df \$ Sueldo < 2500), lo que vemos es que nos da un conjunto de falsos y verdades en función de si cada uno de los trabajadores y trabajadoras cumplen estas propiedades.

Si ejecuto, obtengo toda la información de la base de datos que cumple las dos propiedades simultáneamente, es decir, si estamos dentro de este intervalo.

Si quisiéramos seleccionar algunas columnas en concreto, podríamos hacerlo de la siguiente manera.

```
df [df $ Sueldo > 1250 & df $ Sueldo < 2500, c ( "Posicion", "Trabajador")]
```

Así pues, obtendríamos los mismos tres registros pero ahora sólo de la información que nos interesa, por ejemplo: la posición y su nombre.

Y, para terminar, el último ejercicio, preguntaba si el trabajador con un menor sueldo coincidía con el trabajador con menos días de vacaciones. Aquí, mi propuesta es utilizar la función which.min, que básicamente encuentra la posición que ocupa la persona con el sueldo menor.

```
df [which.min (df $ Sueldo),] $ Trabajador
```

Y hacer lo mismo con la persona que ocupa la última posición en número de días de

vacaciones, y seleccionando la columna relevante utilizando la sintaxis típica para los dataframes, es decir, el \$ (dólar).

```
df [which.min (df $ Vacaciones),] $ Trabajador
```

Aquí, lo que estamos haciendo es obtener, por un lado, el nombre del trabajador; por otra parte, el segundo nombre, que ya vemos que no coincide.

```
df [which.min (df $ Sueldo),] $ Trabajador == df [which.min (df $ Vacaciones),] $ Trabajador
```

Y, si ejecutamos las dos instrucciones conjuntamente, lo que obtenemos es si coinciden o no. En este caso, no coinciden.

Una manera más rápida de hacer esto es sencillamente utilizando las funciones which.min. Ya que, si la posición no coincide, es que no son lo mismo trabajador.

```
which.min (df $ Sueldo) == which.min (df $ Vacaciones)
```

Y hasta aquí el vídeo. ¡Nos vemos en la próxima!

3.4 TRANSFORMAR BASES DE DATOS

¡Hola!

La transformación de una base de datos puede ir desde la ordenación de sus columnas, como ya sabemos, hasta la creación de nuevas filas o columnas. Aquí nos centraremos en estos aspectos. Cuando decidimos añadir nueva información a las bases de datos que ya teníamos, algo que debemos tener siempre presente es las dimensiones del objeto que tenemos. Si queremos añadir una nueva fila, por ejemplo, un trabajador, las nuevas datos que incorporamos deberán estar exactamente en el mismo formato original, o podríamos obtener errores o datos erróneos. Lo mismo ocurre con las columnas nuevas, tenemos que vigilar siempre que tengan la misma longitud que las previas. Una buena manera de proceder es generándose a partir de las ya existentes.

Vamos a ver un ejemplo.

Lo primero que haremos es seleccionar un solo trabajador de nuestra base de datos. El guardaremos con ese nombre.

```
nuevotrabajador <- df []
```

Y, utilizando un filtro por filas, seleccionaremos el trabajador número o nombre "C". Acabamos de crear un objeto de tipo dataframe con una observación y 5 columnas.

```
nuevotrabajador <- df [, df $ Trabajador == "C"]
```

Ahora lo que podemos hacer es modificar las características de este objeto. Por ejemplo, le cambiaremos el nombre y le diremos trabajador "k".

```
nuevotrabajador $ Trabajador <- "K"
```

Y, por ejemplo, también le podemos cambiar el número de días de vacaciones que tiene assignnas. Y los asignaremos 20.

```
nuevotrabajador $ Vacaciones <- 20
```

Si ahora miramos el nuevo trabajador, vemos que tiene el nombre que le hemos dado y el número de días de vacaciones que tenía. Lo que podemos hacer es añadir este nuevo trabajador, que tiene las dimensiones que nos interesan en la base de datos original que teníamos.

Esto lo podemos hacer, utilizando esta instrucción aquí: raw bind (rbind). Estamos añadiendo filas nuevas en la base de datos. Ponemos: el nombre de la base de datos y la nueva fila.

```
rbind (df, nuevotrabajador)
```

Esta instrucción evidentemente funciona también si queremos añadir muchas filas. Estamos obteniendo la concatenación de esta fila.

Si ahora miramos el objeto df, veremos que las datos no se han añadido.

```
rbind (df, nuevotrabajador)
```

```
df
```

No se han añadido para que esta instrucción de aquí no se añade directamente, sino que lo que tenemos que hacer es guardarlo.

Ahora hemos sobreescrito nuestro objeto df, añadiéndole esta nueva fila. Podemos comprobar que ahora sí ha funcionado.

Si lo que queremos, por otra parte, es añadir nuevas columnas; lo podemos hacer de dos maneras. Veamos primero la más rudimentaria. Podemos utilizar una instrucción muy similar a esta, será colum bind (cbind) y un vector de tamaño correcto.

Lo que haremos es crear una nueva variable y le diremos "sexo", que deberá tener las dimensiones de nuestra base de datos, es decir, 11. Lo haré de manera más o menos automática. Le diré que me repita 5 hombres y 6 mujeres. Estas instrucciones de ahí que me crean son un vector de 5 H y estas aquí, un vector de 6 D.

```
sexo <- c (recibe ( "H, 5), RPE (" D ", 6))
```

Si ejecutamos, lo que acabamos de obtener es un vector de caracteres con 5 hombres y 6 mujeres. Ahora, este objeto aquí tiene las mismas dimensiones que el número de filas que tiene nuestra base de datos. Así que lo que podemos hacer es crear una nueva columna utilizando el mismo que acabábamos de ver.

```
df <- cbind (df, sexo)
```

Si ahora miramos nuestra base de datos, veremos que tenemos la nueva columna que le acabamos de pedir.

Por otra parte, y esto es lo más interesante, podemos crear una columna a partir de una o más columnas. Por ejemplo, si pagamos las vacaciones a 50 € el día + 5 € por cada año de antigüedad, podríamos hacer la siguiente operación.

```
df $ Costevacaciones <- df $ Vacaciones * (50 + 5 * df $ Años)
```

Creo una nueva columna, la llamo "coste de las vacaciones" y le digo que es el número de días de vacaciones, que está guardado en esta columna, multiplicado por 50, el costo base, más 5 euros adicionales por cada año de antigüedad. Si ejecuto esta instrucción, no aparece nada.

Pero, si voy a ver qué ha pasado con mi base de datos, obtengo una nueva columna, que es "coste de las vacaciones", que, ha aplicado exactamente este cálculo aquí.

```
df $ Costevacaciones <- df $ Vacaciones * (50 + 5 * df $ Años)
df
```

Tengo un dato que falta, que no es más que producto de que estoy haciendo una operación con un dato que no existe. Y, por tanto, no me lo puede calcular.

¡Seguimos!

A continuación proponemos algunos ejercicios sobre cómo transformar bases de datos.

Encontraremos la solución de cómo hacerlo en este vídeo:

1. Inventa una columna que dé valores numéricos al rendimiento de los trabajadores y trabajadoras. Posteriormente, crea una segunda columna donde se describa el rendimiento en relación con su sueldo.
2. Descubre cómo podemos borrar filas de un dataframe. Tendrás que utilizar el signo "-". ¿Funciona en el caso de las columnas?
3. Explora cómo funciona la instrucción table () sobre una columna cualitativa del dataframe.

3.5 RESOLUCIÓN DE EJERCICIOS: TRANSFORMAR BASES DE DATOS

¡Bienvenidas y bienvenidos de nuevo!

A continuación resolveremos los tres ejercicios propuestos.

Lo primero que hacemos es crear una nueva columna en nuestra base de datos que se basa en el rendimiento de nuestros trabajadores y trabajadoras. Así pues, lo hacemos con esta instrucción y lo que podemos ver es que efectivamente hemos añadido una nueva columna en la base de datos.

```
df $ Rendimiento <- c (7,8,9,10,7,6,8,9,5,9)
```

A partir de esta, vamos a crear una nueva columna que llamaremos "PrecioRendimiento", que básicamente será el rendimiento que estamos suponiendo que tienen nuestros trabajadores y trabajadoras, dividido por el sueldo, para ver cuáles son los trabajadores que más rinden en función del que ganan.

```
df $ PrecioRendimiento <- df $ Rendimiento / df $ Sueldo
```

Ejecutamos esta instrucción y, si exploramos el objeto, vemos que no es especialmente legible.

Así pues, lo que yo propongo es multiplicar este resultado por 1000, por ejemplo, y sobrescribir esta columna (df \$ PrecioRendimiento) y ver qué queda.

```
df $ PrecioRendimiento <- df $ Rendimiento / df $ Sueldo * 1000
```

Queda una variable bastante más legible, pero aún podemos mejorar, utilizando la instrucción round, que redondea nuestros números, por ejemplo a dos decimales.

```
df $ PrecioRendimiento <- round (df $ Rendimiento / df $ Sueldo * 1000,2)
```

Si ahora miramos los resultados, efectivamente las puntuaciones son más atractivas.

Luego, proponía utilizar el signo menos (-) para borrar columnas y filas de nuestra base de datos. Borrar una fila es tan sencillo como utilizar esta instrucción aquí.

```
df <- df [-5,]
df $ PrecioRendimiento <- NULL
```

Si lo ejecutamos, lo que acabamos de hacer es borrar la fila número 5 de nuestra base de datos. Vemos que ahora sólo llega hasta el 9. Si queremos utilizar esta misma estructura para

las columnas, también lo podemos hacer. Ya que es tan sencillo como borrar, por ejemplo, el nombre.

```
df <- df [-5,]
df <- df [, - 1]
df $ PrecioRendimiento <- NULL
```

Aún así, una mejor práctica es borrar las columnas por su nombre. Lo que se muestra aquí, por ejemplo, es cómo borrar la columna nueva que acabábamos de crear. Si lo ejecutamos, en esta línea de aquí le estamos diciendo que la columna precio-rendimiento la borre, la haga desaparecer.

```
df <- df [-5,]
df <- df [, - 1]
df $ PrecioRendimiento <- NULL
df
```

Debemos tener cuidado ya que lo que estamos haciendo es borrar y guardar el objeto con el nombre original. Podría ser interesante que los objetos en los que los borramos una columna o unas filas los guardáramos con un nombre diferente.

Y ya para acabar proponíamos explorar la función *table* para ver un resumen cualitativo de algunas de las columnas de nuestra base de datos. Por ejemplo, la posición.

```
table (df $ Posicion)
```

Esta es una función muy útil que lo que nos permite es generar un resumen de cuantos elementos encontramos de cada categoría. Esto de aquí que podremos guardar, por ejemplo, en un objeto llamado mesa, será muy útil para hacer diagramas de barras y diagramas de sectores.

```
tabla <- table (df $ Posicion)
```

¡Seguimos!

3.6 IDEAS CLAVE: TRABAJAR CON BASES DE DATOS

Veamos los conceptos más importantes del módulo, que tienen que ver con la realización de varios procesos con nuestras propias bases de datos:

- Importar los datos a R, en diferentes formatos, mediante el asistente de importación de RStudio.
- Guardar nuestros objetos, ya sea en el formato nativo de R o formatos legibles para Excel.

- Obtener **datos** de nuestros *dataframes* mediante filtros simples y múltiples, ya sea por columnas o filas.
- Modificar las **dimensiones** de nuestra base de datos, añadiéndole o borrándole datos, así como crear nuevas variables a partir de la información de la que disponíamos al principio.

4 VISUALIZACIÓN DE DATOS

En este último módulo veremos cómo realizar algunos de los gráficos más populares con R, y cómo podemos personalizarlos y mejorarlos para hacerlos más atractivos. Dado que la comunicación de resultados en formato gráfico es una de las competencias más necesarias actualmente, pondremos especial énfasis en la interpretación de los resultados. Los gráficos que exploraremos en este módulo son el diagrama de sectores, el gráfico de barras, el histograma, el diagrama de caja y la nube de puntos, así como algunas herramientas para añadir información adicional.

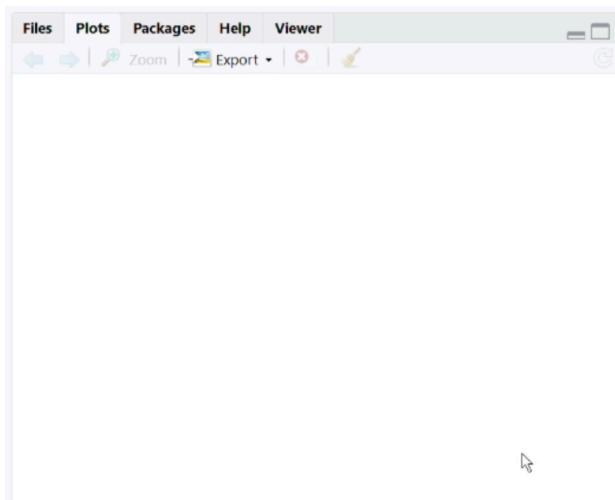
4.1 INTRODUCCIÓN A LA VISUALIZACIÓN DE DATOS

¡Hola a todas y todos!

En este vídeo exploraremos las bases de la visualización de datos en R, veremos cómo podemos aprovechar RStudio para tal finalidad y algunas consideraciones previas a la hora de trabajar con visualizaciones.

En primer lugar, hemos de entender el espacio que nos ofrece RStudio para la visualización:

Los gráficos en RStudio aparecen en la pestaña Plots, en este espacio de aquí.

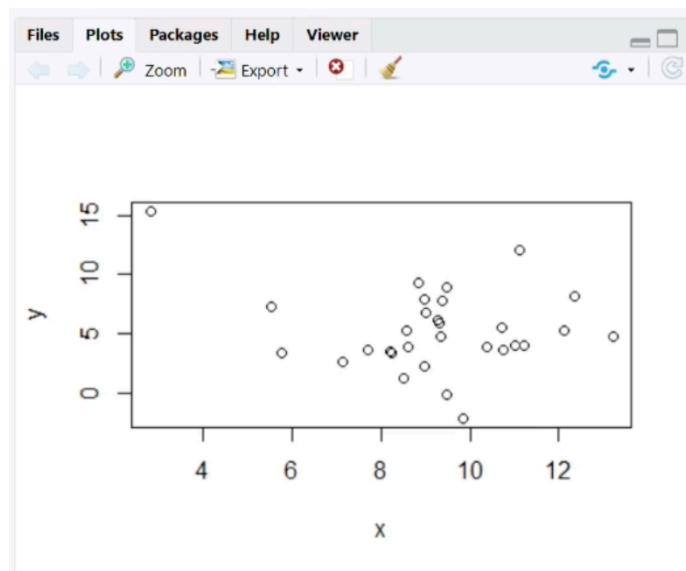


Es importante que siempre lo tengamos con un tamaño suficientemente grande para que podamos acomodar un gráfico. Si tengo la pestaña en una dimensión similar a esta, cuando

ejecutamos una función gráfica, por ejemplo *plot*, R se quejará. Así pues, la mantenemos más o menos maximizada.

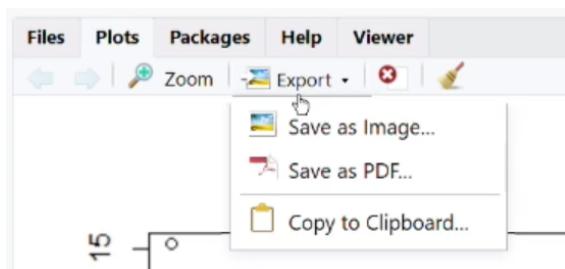
Aquí, he generado dos vectores de datos aleatorios, es decir 30 puntos, de media 10 y desviación 2, de una variable aleatoria normal. Y, utilizando una instrucción *plot x, y*, obtengo una nube de puntos por x y por y.

```
x <- rnorm (30,10,2)
y <- rnorm (30,5,3)
plot (x, y)
```



Es importante entender que cuando estamos generando un gráfico, estamos creando algo así como una hoja de papel, y todo lo que queramos añadir, leyendas, puntos, líneas ... lo estamos añadiendo encima. Y esto implicará que podemos haber superposiciones.

En segundo lugar, debemos destacar la importancia de la exportación.



Los gráficos se exportan utilizando esta pestaña aquí.

Podemos copiarlo directamente utilizando esta instrucción, pero lo más habitual es guardarlo como una imagen. Aquí, podemos elegir el formato que queramos.



Los más habituales son los dos primeros y el directorio donde lo queremos guardar, así como el nombre. Podemos fijar la anchura y la altura de la imagen, y es importante considerar que, como lo hayamos generado, influirá en cómo la estamos guardando.

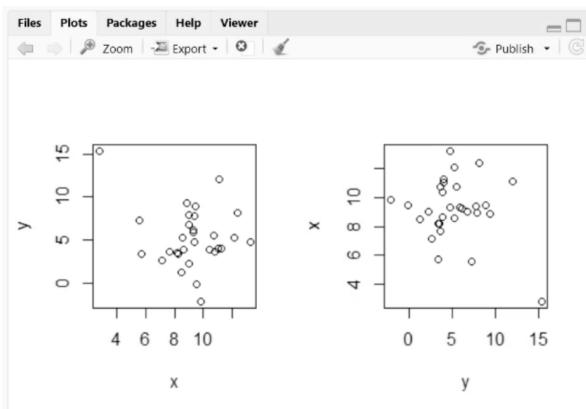
En tercer lugar, mostraremos cómo juntar dos gráficos en una sola visualización y la importancia que tiene su disposición en la comunicación de información.

Así pues, utilizando la instrucción `par` y este parámetro aquí, `mfrow`, estamos consiguiendo partir nuestro espacio de gráficos en una fila, dos columnas.

```
par(mfrow = c(1,2))
```

Por ejemplo, si ahora ejecutamos este gráfico aquí, nos lo coloca en la primera posición, dejándonos espacio para otro gráfico. Por ejemplo, podríamos imprimir el gráfico inverso, es decir, y con x, y obtendríamos esta disposición aquí.

```
plot(x, y)
```



```
plot(y,x)
```

La comparación que podríamos hacer sería bastante práctica. Por otro lado, si lo que hacemos es crear un espacio donde tenemos dos filas y una columna.

```
par(mfrow = c(2,1))
```

Lo ejecutamos y ahora volvemos a ejecutar estos dos gráficos de aquí. Se queja de que el espacio es demasiado pequeño, porque los dos gráficos no caben muy bien en vertical. Lo amplificamos un poco más, ejecutamos y vemos que la visualización no es nada atractiva. Así pues, es muy importante que, a la hora de partir los gráficos en dos espacios, pensemos, antes de hacerlo, cuál es la disposición que comunicará mejor nuestra información.

Es importante destacar que en este módulo veremos cómo generar y modificar gráficos utilizando las funciones básicas de R, por su sencillez y potencial. Si nos interesan visualizaciones más avanzadas, atractivas o interactivas, deberíamos considerar paquetes como "ggplot2" o "Plotly".

Así que ya sabemos algo más de RStudio.

Antes de terminar este vídeo, para recuperar la visualización en un solo gráfico, lo que necesitamos es volver a ejecutar esta instrucción diciéndole que queremos una sola fila y una sola columna.

```
par(mfrow = c(1,1))
```

4.2 GRÁFICOS DE BARRAS Y DIAGRAMAS DE SECTORES

¡Hola de nuevo!

A continuación, veremos unos ejemplos de código muy sencillos de cómo podemos hacer dos de las visualizaciones más populares con R.

El código, como verás, para cada una de las visualizaciones es muy simple, pero sus opciones de personalización son muy interesantes. Debemos destacar que a la hora de crear un gráfico o visualización, lo más importante siempre es el ensayo y el error, ya que muchos de los parámetros que podremos modificar no tienen una fórmula que nos diga qué es un buen valor en todos los escenarios, sino que debemos explorar cuál es el más adecuado para cada una de las circunstancias en las que nos encontramos.

Empezamos creando una tabla. Crearemos este objeto, utilizando la función `table`, de una de las columnas categóricas de nuestra base de datos. Por ejemplo, la posición.

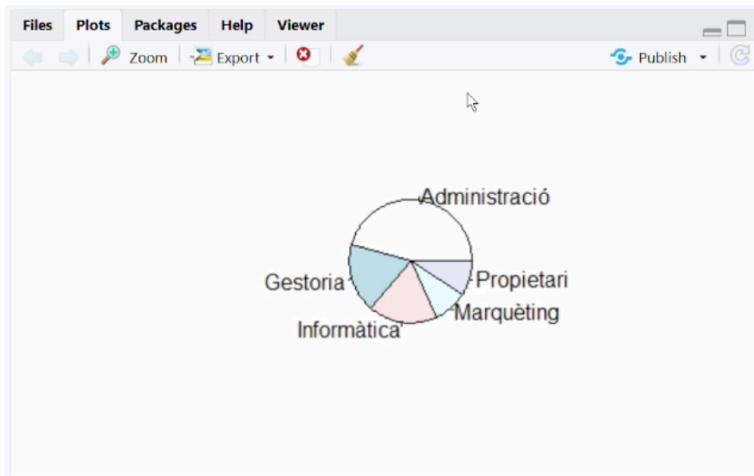
```
tabla <- table(df$Posicion)
```

Ejecutamos. Se nos crea aquí un objeto de tipo mesa y vamos a ver qué contiene.

Contiene un recuento de las posiciones que tenemos para cada uno de los individuos. Una vez tenemos este tipo de objeto, crear un diagrama circular, un diagrama de quesito, es tan sencillo como utilizar la función pie.

pie (tabla)

Esta es la vista predeterminada, pero modificarlo, como ya he comentado, tiene mucho potencial.



Lo primero que podemos hacer es añadirle un título, utilizando la función *main*.

pie (tabla, main = "Título")

Otra propiedad que podemos modificar es esta aquí, que lo que permitirá, como veremos, es que comience exactamente a las 12 del mediodía.

pie (tabla, main = "Título", clockwise = TRUE)

Con ello, lo que conseguimos es que sea más interpretable esta primera categoría. Ya que percibimos mejor las áreas, comenzando exactamente desde el centro.

También podemos modificar los colores, por ejemplo, utilizando una función muy interesante, que es la función *rainbow* y le debemos especificar la longitud de la tabla.

pie (tabla, main = "Título", clockwise = TRUE, col = rainbow (length (tabla)))

Si ejecutamos sólo esta parte de aquí, podemos ver que lo que obtenemos son 5 colores diferentes en la nomenclatura técnica. Si ejecutamos todo el gráfico, obtenemos la misma estructura pero con colores del arco iris. Una última modificación sería modificar el radio de

nuestro gráfico. Aquí sí que tendremos que hacer ensayo y error, y podemos probar con radio 1. Radio 1 es mayor, podríamos probar con radio 5, por ejemplo; y ver que nos hemos pasado.

```
pie(tabla, main = "Título", clockwise = TRUE, col = rainbow(length(tabla), radius = 1)
```

Esto, como ya he comentado, hay que ir ajustando en función del tamaño que tengamos de nuestra pestaña.

Para terminar este vídeo, lo que haremos es un diagrama de barras. La función es barplot y podemos utilizar otra vez el mismo objeto "mesa".

```
barplot(tabla)
```

Si lo ejecutamos, vemos esta visualización aquí. Parece que no haya funcionado correctamente. Lo único que pasa es que no tiene espacio suficiente para mostrarnos todas las etiquetas correctamente. Y, si desplazamos nuestro espacio, podemos visualizarlas. Algunos de los parámetros que podemos modificar son, por ejemplo, main. Le añade un título. También podemos añadirle colores, de manera idéntica a como hemos hecho en el ejemplo anterior. Copiaré directamente.

```
barplot(tabla, main = "Título", col = rainbow(length(tabla)))
```



Y podríamos comprobar qué parámetros podemos mover, mirando toda esta lista aquí.

Te animo a que lo explores y mires qué es lo que necesitas para mejorar o adecuar tu gráfico en un contexto concreto.

¡Seguimos con el curso!

4.3 HISTOGRAMAS Y DIAGRAMAS DE CAJA

¡Hola!

En este vídeo veremos cómo se generan los dos tipos de gráficos más extendidos a la hora de estudiar la distribución y dispersión de nuestros datos. Los diagramas de caja o boxplots y los histogramas. Pondremos especial énfasis en su interpretación correcta.

Un boxplot sirve fundamentalmente para describir los cuartiles y los *outliers* de una variable numérica y, sobre todo, para comparar varias categorías que comparten una misma escala.

Lo que podemos hacer, por ejemplo, es cargar nuestra base de datos mtcars y hacer un boxplot de una de las columnas de esta base de datos. Por ejemplo, esta de aquí.¶

```
fecha (mtcars)
boxplot (mtcars $ wt)
```

Si ejecutamos, podemos ver un diagrama de caja muy sencillo, pero que ya contiene toda la información que nos interesa. Un diagrama de caja nos muestra, en primer lugar, la mediana, esta recta aquí, que separa entre el 50% inferior de los coches y el 50% superior de los coches. La caja incluye desde el 75% superior de los coches, es decir, el tercer cuartil que va abajo y el 25% de los coches hacia abajo, que separa esta línea de aquí, que es el primer cuartil.

Este valor aquí es el mínimo; y el último punto que encontramos es el máximo. ¿Qué significa que aparezcan en este formato? Que están bastante lejos de la caja como para considerarlas datos anómalos, es decir, outliers. Estos outliers en esta base de datos aparecen por encima, pero también podrían aparecer por debajo.

Así repasando, podemos ver los extremos de nuestras datos, la mediana, el tercer cuartil y el primer cuartil.

Como he comentado, una de las utilidades principales de los boxplots es comparar dos grupos. Lo que podemos hacer, por ejemplo, es comparar dos veces esta variable.

```
boxplot (mtcars $ wt, mtcars $ wt)
```

Si lo ejecutáramos, veríamos dos veces el mismo boxplot, pero cumpliendo una propiedad en concreto. Por ejemplo, que la variable *vs* sea igual a 1, y que, para el segundo grupo, sea igual a 0.

```
boxplot (mtcars $ wt [mtcars $ vs == 1], mtcars $ wt [mtcars $ vs == 0])
```

Ahora ejecutamos y podemos ver cómo se distribuye la variable *wt*, en función de la categoría 1 o la categoría 0. Uno de los parámetros que podemos modificar, en este caso, es el color. Tendremos que especificar dos colores, por ejemplo, verde y rojo.

```
col = c ("green", "red")
```

Y podríamos especificarle también, este es un parámetro muy importante, los nombres. En este caso le especificaríem grupo 1 y grupo 0.

```
names = c ("Grupo 1", "Grupo 0")
```

Ejecutamos y vemos que hemos conseguido el grupo 1, el grupo 0, por orden de aparición, y los colores en el mismo orden en el que los hemos ido introduciendo.

Y, si hacemos referencia a los histogramas, estos nos permiten visualizar, con mucho más detalle y de una manera más intuitiva, como se distribuyen las datos.

Así pues, la función que utilizamos es *hist*. Utilizamos la misma variable.

```
hist (mtcars $ wt)
```

Y lo que podemos ver es el perfil de cómo se distribuye esta variable aquí, *wt*, y cuántas veces aparece, es decir, su frecuencia, para cada uno de los valores de la variable.

Una manera muy interesante de mejorar este gráfico es añadiéndole nuevos cortes. Por ejemplo, podemos añadir otros 15.

```
hist (mtcars $ wt, breaks = 15)
```

Lo que permite comprobar que tenemos una categoría mucho más poblada que las otras y ver un perfil mucho más detallado de cómo se distribuye. Si lo que nos interesa es una frecuencia relativa, lo que podríamos hacer es decirle la frecuencia *FALSE*.

```
hist (mtcars $ wt, breaks = 15, freq = FALSE)
```

Otra opción que puede ser interesante es decirle *plot = FALSE*.

```
hist (mtcars $ wt, breaks = 15, freq = FALSE, plot = FALSE)
```

Con lo cual estamos obteniendo *warning*, que no debe preocuparnos porque no es un error. Estaríamos obteniendo todos los cálculos que ha realizado y todas las datos numéricos que se extraen de un histograma, y esto puede ser especialmente interesante si lo que queremos ver es cuáles son estos cortes y cuál es su densidad.

4.4 NUBE DE PUNTOS Y RECTA DE REGRESIÓN (I)

¡Hola!

Ha llegado el momento de introducir uno de los modelos fundamentales de la estadística y el análisis de datos: y este es el modelo de regresión lineal, también llamado "recta de regresión". Para ello, utilizaremos la función *plot*, la más básica de todas, para dibujar la nube de puntos en el formato que deseamos, y veremos cómo añadirle, de manera gráfica, un modelo encima.

La función `plot` requiere dos coordenadas: la coordenada x y la coordenada y.

```
plot (x, y)
```

Utilizaremos dos columnas de nuestra base de datos: `mtcars`, `wt` y `mtcars mpg`.

```
plot (x = mtcars $ wt, y = mtcars $ mpg)
```

Con esta instrucción, podemos ver directamente una nube de puntos muy sencilla. Lo que veremos ahora es cómo mejorarla, colorear estos puntos basándonos en alguna característica y cómo añadirle la recta de regresión.

Lo primero que haremos es crear un filtro. Un filtro que decidirá si estos coches aquí pertenecen a un grupo o a otro. Utilizaremos la columna `vs` y tomaremos los valores que sean igual a 0 para una categoría e igual a 1 para otra.

```
filtro <- mtcars $ vs == 0
```

Obtenemos este tipo de filtro, que utilizaremos para seleccionar en qué casos pintamos de un color y en qué casos pintamos de otro. Así pues, lo que queremos es añadirle unos puntos encima de este gráfico. Estos puntos los podemos añadir con la función `points` y lo que haremos es decirle "coge estos valores aquí, pero ahora píntame de un color especial los que tengan un valor *TRUE* del filtro que acabamos de crear".

```
points (x = mtcars $ wt [filtro], y = mtcars $ mpg [filtro])
```

Hay muchas maneras de hacer esto que estoy haciendo, pero esta es una manera muy interesante porque permite aprender a crear filtros y manipularlos posteriormente. Estos puntos de aquí los queremos de otro color, por ejemplo, de color rojo.

```
points (x = mtcars $ wt [filtro], y = mtcars $ mpg [filtro], col = "red")
```

Así pues, lo que hemos hecho es superponer puntos de color rojo en los puntos que teníamos antes.

Añadirle la recta de regresión será tan sencillo como definir un modelo de regresión lineal. La función es "lineal modelo" (`lm`) y tenemos que definir mi variable del eje vertical, una tilde (~) y "wt". Debemos especificarle también de qué base de datos lo hemos extraído, en este caso "`mtcars`".

```
modelo <- lm (mpg ~ wt, fecha = mtcars)
```

Y eso que acabamos de ejecutar es un modelo de regresión lineal. No entraremos a interpretarlo, y lo que haremos es añadirle una recta. La recta se añade con esta función

aquí, `abline()`, que permite añadir rectas verticales, horizontales o rectas con pendiente, como es este caso.

`abline (modelo)`

Si ejecutamos, vemos que acabamos de añadir la recta encima de nuestros puntos. Podemos mejorar un poco este gráfico, modificando algunos parámetros. Por ejemplo, cambiando el tipo de punto, haciéndolo más grueso o añadiéndole un título.

```
plot (x = mtcars $ wt, y = mtcars $ mpg, PCH = 4, LWD = 1.5, main = "Recta de regresión")
```

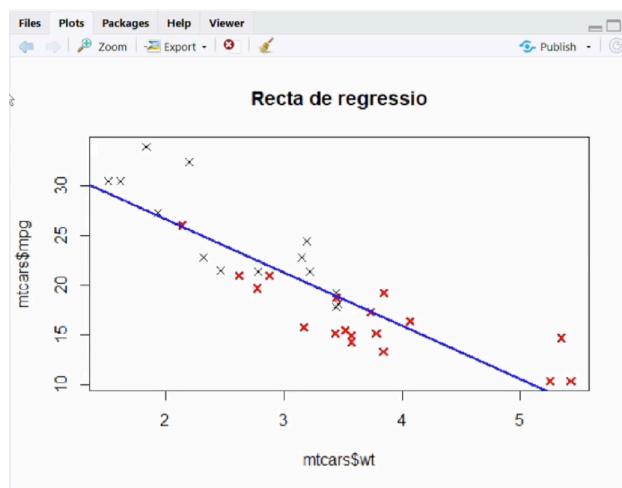
Lo mismo podemos hacer con la función `points`. Ya hemos visto que le hemos cambiado el color. Ahora, para que no desentoné, los pondremos de un grosor algo superior a los demás y del mismo tipo.

```
points (x = mtcars $ wt [filtro], y = mtcars $ mpg [filtro], col = "red", LWD = 2, PCH = 4)
```

Y esta recta que estamos añadiendo, la cambiaremos de color también, y también la haremos más gruesa. Podríamos cambiarle el tipo, pero eso lo veremos más adelante.

`abline (modelo, col = "blue", LWD = 2)`

Si ahora ejecutamos todas estas instrucciones, obtenemos la recta de regresión sobre nuestras datos filtradas basándonos en dos categorías y, al final, acaba quedando con este formato.



Y ya lo tendríamos. ¡Continuamos!

4.5 NUBE DE PUNTOS Y RECTA DE REGRESIÓN (II)

¡Hola de nuevo!

En este segundo vídeo veremos cómo interpretar el output que nos genera la función *lm*, cuáles son los elementos más importantes a considerar y cómo podemos incorporarlos a la comunicación de nuestros procesos de análisis.

Aquí vemos el código que genera el modelo de regresión simple, es decir, estamos ajustando el consumo por galón, utilizando el peso de todos estos coches.

```
modelo <- lm (mpg ~ wt, data = mtcars)
```

Si ahora observamos qué hay dentro del objeto *modelo*, tenemos esta fórmula aquí, es decir, un coeficiente que intercepta uno de los ejes y un coeficiente asociado a la variable.

Este valor aquí es la pendiente de la recta. Un valor negativo indica que cuanto más valor de esta variable, es decir, cuanto más peso, menor consumo.

Si queremos ver más información, lo que podemos hacer es *summary* de nuestro modelo.

```
summary (modelo)
```

Una primera ojeada a este resumen nos permite ver que tenemos mucha más información de la que seguramente necesitamos a nivel de usuario. Tenemos un estudio de los residuos, un estudio estadístico de los coeficientes, que, como podemos ver, son los mismos que teníamos ejecutando sólo el modelo, y un resumen en texto de las especificaciones del modelo.

Probablemente el elemento más importante de todos es este de aquí. Este número nos indica en tanto por 1, es decir, muy fácilmente convertible en un tanto por ciento, ¿qué porcentaje de variabilidad explica esta variable con relación a esta?. Dicho de otro modo, ¿en qué porcentaje estas variables están relacionadas?. En este caso, en un 74%, si movemos la coma dos unidades hacia allá. Un 74% de relación lineal entre las variables es una relación muy elevada. Normalmente las datos no presentan una relación tan fuerte, pero en este caso, al tratarse de coches, y variables tan relacionadas como el peso y el consumo, podemos obtener una interpretación muy fuerte de cómo se relacionan.

En términos estadísticos, también es muy importante esta región aquí. Si estos valores aquí son muy pequeños, en estos casos lo son, ya que es $1,29 \times 10^{-10}$, un número muy pequeño, lo que estamos diciendo es que esta variable aquí es muy relevante. Va muy asociado a esta variable aquí, pero si estamos ajustando un modelo de regresión múltiple, por ejemplo añadiéndole aquí más variables, podríamos ver cuáles de estas son más importantes.

¡Continuamos!

4.6 PERSONALIZACIÓN DE GRÁFICOS (I)

¡Hola de nuevo!

En esta lección veremos cómo podemos añadir elementos adicionales a nuestros gráficos, que tanto pueden aportar nueva información al lector como hacerlo más atractivo. Para verlo, trabajaremos en los gráficos que hemos generado anteriormente.

Una primera mejora que podemos aplicar, y es recomendable hacerlo siempre, es modificar los nombres de los ejes con los parámetros *xlab* y *ylab*.

Así pues, lo que utilizaremos son estas dos instrucciones aquí. Por ejemplo, peso y consumo.

```
plot (mtcars $ wt, mtcars $ mpg, xlab = "Peso"), ylab = "Consumo")
```

Ya habíamos visto anteriormente que podemos modificar el título de un gráfico utilizando la instrucción *main*.

```
main = "Peso vs Consumo"
```

Si lo que nos interesa es centrarnos en una región concreta de este gráfico, lo que podríamos hacer es decir, por ejemplo, que queremos mirar sólo los vehículos que están entre 3 y 4 toneladas de peso. Esto lo podríamos hacer utilizando esta instrucción. Esto dependerá de un vector, donde le tendremos que especificar el mínimo y el máximo.

```
XLIM = c (3,4)
```

Lo que acabamos de hacer es aplicar un zoom sobre nuestro gráfico.

Ahora estamos viendo que no necesitamos todo este espacio de aquí arriba, así que podríamos aplicar exactamente el mismo para el eje vertical.

```
ylim = c (10,25)
```

Y sugerimos añadir un código de colores más atractivo. A diferentes paquetes podemos encontrar gamas cromáticas bastante más atractivas que los colores básicos de R o la instrucción *rainbow*, que ya hemos visto antes, que genera gráficos muy infantiles. Buenos ejemplos son los que veremos a continuación.

Utilizaremos la función, por ejemplo, *terrain.colors*, y le tendremos que especificar un número, que será el número de colores que nos estará generando la función.

```
col = terrain.colors (20)
```

Por ejemplo, podemos utilizar 20 y, si lo ejecutamos, veremos esta secuencia de colores. ¿Qué tienen en común? Que pertenecen todos a una gama cromática concreta, que hace que el estilo sea mucho más atractivo.

Vamos a hacer los puntos algo más visibles y lo que vemos es que todos los colores mantienen una cierta tonalidad.

LWD = 4

Otra gama cromática bastante interesante es *heat.colors*. Es una gama más basada en los rojos y, por la impresión en blanco y negro, una muy interesante es *gray.colors*, que genera una impresión donde está utilizando sólo la gama de blanco a negro. Es importante considerar que aquí lo que estamos haciendo es colorear nuestros puntos, basándonos en un criterio aleatorio.

Lo más interesante sería darle una cierta interpretación a este tipo de colores, por ejemplo haciendo que los grises más oscuros fueran los coches de una cierta categoría y los grises más claros los de otra.

4.7 PERSONALITZACIÓN DE GRÁFICOS (II)

En este segundo vídeo de personalización de gráficos, veremos como añadir elementos ligeramente más sofisticados en nuestras visualizaciones. Nos centraremos en la superposición de información estadística básica y de elementos textuales.

Anteriormente hemos visto cómo añadir puntos mediante la instrucción *points*. Podemos hacer lo mismo con la instrucción *abline()* para líneas rectas o *lines()* para añadir figuras más complejas.

Vamos a añadir, en el gráfico que tenemos aquí, los promedios para el eje vertical y para el eje horizontal.

Ya he comentado anteriormente que podíamos utilizar esta función aquí, que es la que utilizamos para añadir el modelo de regresión lineal, para añadir líneas horizontales y verticales. Ahora, añadiremos una línea horizontal. Por lo tanto, lo que haremos es calcular la media de la variable consumo. Utilizamos la función *mean* de la variable consumo, que es *mpg*. Lo mismo podemos hacer para añadir la media para la variable peso.

```
abline (h = mean (mtcars $ mpg))
```

Ahora tendremos que modificar el parámetro horizontal y decirle que la queremos en vertical.

```
abline (v = mean (mtcars $ mpg))
```

Si lo ejecutamos ahora, veremos que no hace nada, ya que no hemos modificado esta variable aquí. No es que esta instrucción no haya funcionado, sencillamente nos ha dibujado una recta que cae fuera del espacio que tenemos aquí.

Si ahora selecciono la variable correcta, veré que acabo de generar un eje de coordenadas y de ordenadas que permite situar como de dispersas están las datos en base a las medias de sus categorías.

```
abline(v = mean(mtcars $ wt))
```

Para hacerlo más atractivo, podemos modificar los colores de estas líneas, haciéndolas más apagadas.

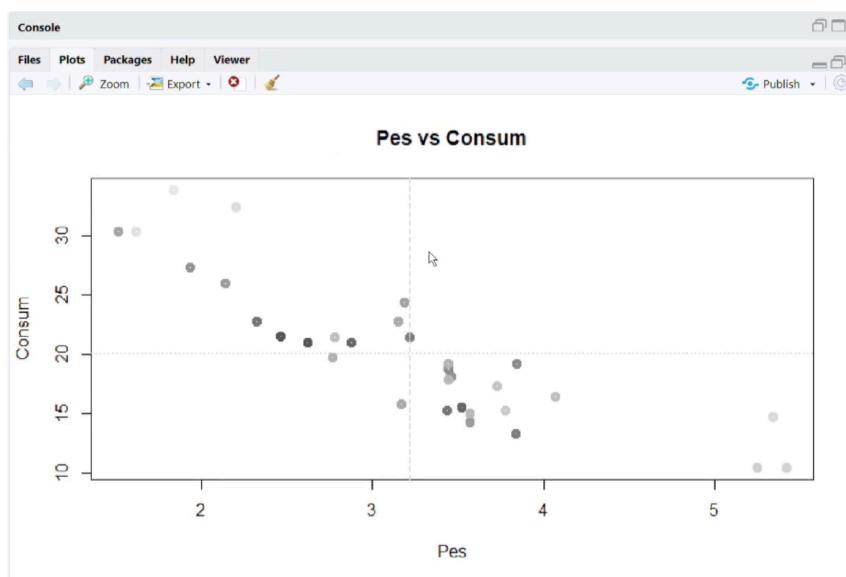
```
col = "lightgray"
```

Y podríamos cambiarles el tipo. Por ejemplo, con esta instrucción.

```
lty = 3
```

Escogemos el 2 y el 3, para ver cómo funcionan y, si lo ejecutamos, veremos que no hace prácticamente nada, al menos no perceptiblemente.

Lo que ha pasado es que nos ha superpuesto una línea de un tipo diferente a la línea que ya teníamos. Lo que tenemos que hacer es volver a ejecutar nuestro gráfico y ahora veremos correctamente lo que estamos haciendo; es decir, estamos añadiendo unas rectas discontinuas con una frecuencia de cortes diferente entre ellas.



Para terminar este vídeo, veremos cómo podemos añadir texto relevante a nuestros gráficos. Ya hemos visto como añadirle un título, nombre a los ejes y ahora lo que vamos a ver es cómo podemos añadir texto en todo este espacio aquí.

La función es la función *text* y depende de tres parámetros. En primer lugar, le debemos especificar en qué coordenada en la escala del eje X queremos que seleccione, por ejemplo 4 (seleccionará esta línea vertical), y qué coordenada en el eje vertical, por ejemplo podría ser

el 25 (y seleccionará esta línea horizontal).

texto (x = 4, y = 25)

El tercer parámetro que le especificamos es el texto deseado, por ejemplo "Mi coche".

texto (x = 4, y = 25, "Mi coche")

Lo que ha hecho es añadir un texto centrado exactamente en el punto que le he pedido. A mí me interesaría, por ejemplo, destacar que el punto interesante es este de aquí. Este tipo de modificaciones se realizan normalmente haciendo ensayo y error. Por ejemplo, yo aquí lo que haría es seleccionar un punto, por ejemplo el 3, y una altura, por ejemplo el 32. Y ver qué tal queda.

texto (x = 3, y = 32, "Mi coche")

Debería ser un poquito más arriba y un poquito más hacia la izquierda. 32,5 (no olvides que los decimales se escriben con un punto) y un poquito hacia la izquierda, 2,8, por ejemplo.

texto (x = 2.8, y = 32.5, "Mi coche")

Como vemos, se está superponiendo a mis textos anteriores, ya que R entiende añadir nuevos elementos a un gráfico como si fuera una etiqueta encima del gráfico que ya teníamos. Así pues, si vuelvo a ejecutar todo, obtendré sólo la etiqueta definitiva.

¡Seguimos!

4.8 AÑADIR LEYENDAS A NUESTROS GRÁFICOS

¡Hola de nuevo!

El proceso de añadir una leyenda a un gráfico en R se realiza manualmente, lo que permite un nivel de personalización muy elevado, pero se le debe dedicar tiempo. Para ver un ejemplo de código, vamos a crear un gráfico de puntos filtrado con tres categorías y le daremos color.

Es importante notar el proceso con el que hemos construido este gráfico.

```
plot (mtcars $ wt, mtcars $ mpg, PCH = 4, LWD = 1.2, col = "orange", main = "Recta de Regresión", ylab = "Consumo", xlab = "Peso")
```

En primer lugar, dibujamos todos los puntos y añadimos título y nombre de los ejes. Después, creamos un filtro. Este filtro servirá para colorear aquellos coches con un cilindro de 4 unidades. Estos puntos los pintaré de color rojo. Unos ciertos parámetros.

```
filtro <- mtcars $ CYL == 4
points (mtcars $ wt [filtro], mtcars $ mpg [filtro], col = "darkred", LWD = 2, PCH = 4)
```

Utilizando el mismo nombre, es decir, sobreescribiendo este filtro, volveré a utilizar la misma estructura para llamar los puntos pintados de color verde.

```
filtro <- mtcars $ CYL == 8
points (mtcars $ wt [filtro], mtcars $ mpg [filtro], col = "darkgreen", LWD = 2, PCH = 4)
```

La función para crear leyendas se llama *legend*.

```
legend ( "topright", legend = c ( "4 CYL", "6 CYL", "8 CYL"), col = c ( "darkred", "orange",
"darkgreen"), PCH = 4, cex = . 8, title = "Leyenda", horiz = TRUE)
```

Esta función depende de muchas parámetros. El primero de todos es la posición de la leyenda. La posición de la leyenda la podemos especificar numéricamente, es decir, con los valores de las coordenadas; por ejemplo, 4 y 25. Nos pondría la leyenda aquí. Pero yo recomiendo, al menos para empezar, utilizar el formato actual, por ejemplo "topright".

Lo que estamos haciendo, especificando le "topright", es que queremos que nos ponga la leyenda en la parte superior de la derecha. Si ejecutamos esta instrucción, se quejará; ya que no le estamos diciendo que queremos que nos dibuje. Nos falta el parámetro *legend*. Aquí, en formato vectorial, le tenemos que poner todos los elementos que queremos que nos muestre. Así pues, queremos que nos muestre 4 cilindros, 6 cilindros y 8 cilindros (*legend* = c ("4 CYL", "6 CYL", "8 CYL")). Si ahora ejecutamos, vemos que nos ha añadido un recuadro con los tres nombres que le hemos especificado. Esto, obviamente, lo podemos mejorar.

Una primera mejora a considerar es añadirle color. Ahora veremos cómo funciona. Le debemos especificar el parámetro *color* y, en un vector, le asignaremos un color a este elemento, un color a este y un color a este (*col* = c ("darkred", "orange", "darkgreen")). Ya veremos qué pasa.

Acabamos de ejecutar y no ha cambiado absolutamente nada. Básicamente, no ha cambiado porque no le hemos dicho qué tipo de elemento queremos que asocie a estos nombres. Lo que tenemos que hacer, por ejemplo, es decirle que sean puntos del mismo tipo que los que estábamos dibujando, es decir, tipo 4 (*PCH* = 4).

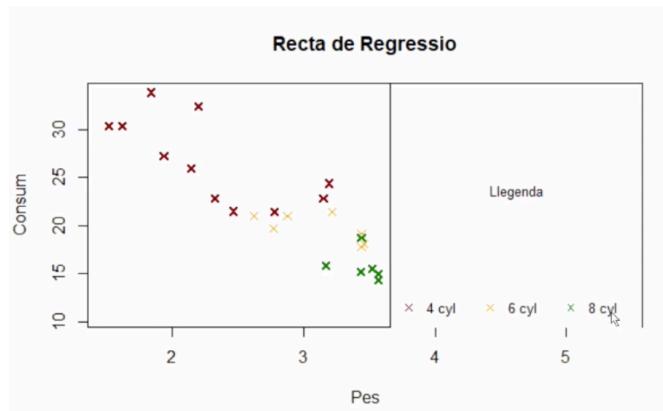
Ahora ya aparecen estos elementos. Si pudiéramos mostrar diferentes tipos de punto (no es el caso en este gráfico), pero yo lo muestro, podríamos hacerlo así (*PCH* = c (2,4,6)). Y ahora tendríamos tres puntos diferentes: 2, 4 y 6, en función del primer elemento, el segundo elemento y el tercer elemento. Dejémoslo en 4. Modificamos el tamaño del texto, utilizando esta instrucción aquí (*cex* = .8). Vemos que se ha superpuesto sobre la leyenda anterior. Esto es lo que estaba ocurriendo hasta ahora, pero ahora como que hemos variado el tamaño, aparece superpuesta y se ve esta superposición.

Podemos añadirle también un título. Ahora, en vez de *main*, es *title*. Le diremos "Leyenda" (*title = "Leyenda"*). Y, por último, dado que es una leyenda que quizás ocupa un poco demasiado de espacio, la pondremos en horizontal, utilizando este parámetro, que será "verdad" o "falso" en función de cómo la queramos (*horiz = TRUE*). Ejecutamos y tenemos esta leyenda aquí.



Para recuperar el gráfico y que se borren estas leyendas que no necesitamos, tenemos que volver a ejecutar todo el código.

Es importante notar que, cuando tenemos un gráfico y estamos añadiéndole elementos, importa el tamaño de la pestaña que tenemos. Así pues, si yo tengo un gráfico así, y le añado la leyenda; volviendo a ejecutar, una vez lo haga grande, la leyenda me quedará en este formato.



Obviamente, esto no es una buena visualización, así que se debe considerar que el tamaño de la pestaña que tengamos a la hora de ejecutar es muy relevante, ya que define también el tamaño de la leyenda que le estamos añadiendo al gráfico .

¡Continuamos!

4.9 IDEAS CLAVE: VISUALIZACIÓN DE DATOS

Ya estamos al final del módulo. A continuación, faremos un repaso de las ideas más importantes que han aparecido.

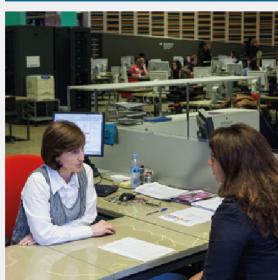
En este módulo final hemos visto una breve introducción a la generación y edición de gráficos en R, su exportación, la creación de gráficos múltiples, así como maneras sencillas de enriquecer nuestras visualizaciones para hacerlas más atractivas y útiles para los lectores u oyentes. Los diferentes tipos de gráficos que hemos explorado son:

- Diagrama de sectores
- Diagrama de barras

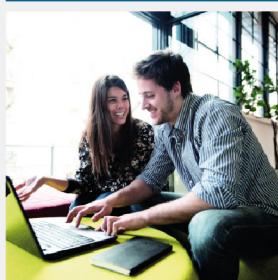
- Histograma
- Diagrama de caja
- Nube de puntos y recta de regresión

La elección del tipo de gráfico y las modificaciones que hacemos deben venir siempre dados como una respuesta a los resultados que queramos comunicar, por lo que es necesario explorar previamente y en profundidad las datos con las que estemos trabajando.

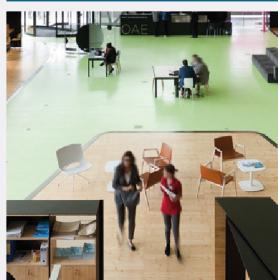
Descubre todo lo que Barcelona Activa puede hacer por ti



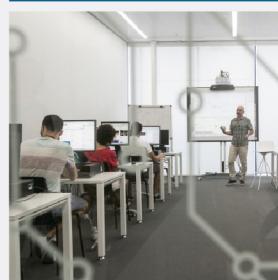
Acompañamiento durante todo el proceso de búsqueda de empleo
barcelonactiva.cat/treball



Apoyo en la puesta en marcha de tu idea de negocio
barcelonactiva.cat/emprendoria



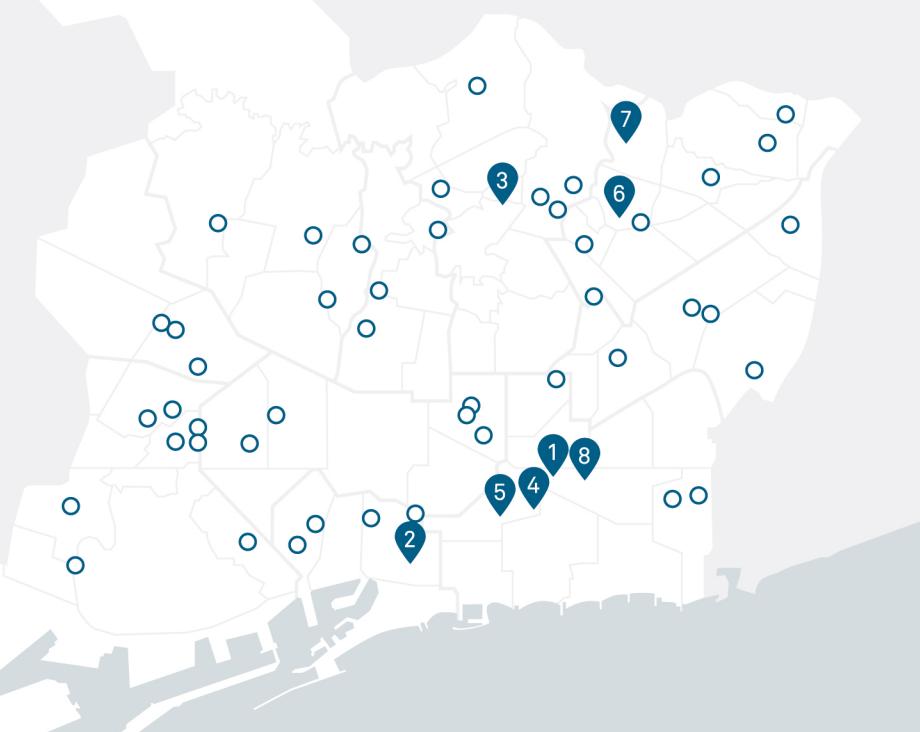
Servicios a las empresas e iniciativas socioempresariales
barcelonactiva.cat/empresas



Formación tecnológica y gratuita para la ciudadanía
barcelonactiva.cat/cibernarium

Red de equipamientos de Barcelona Activa

- 1 Sede Central Barcelona Activa
Porta 22
Centro para la Iniciativa Emprendedora Glòries
Incubadora Glòries
- 2 Convent de Sant Agustí
- 3 Ca n'Andalet
- 4 Oficina de Atención a las Empresas Cibernàrium
Incubadora MediaTIC
- 5 Incubadora Almogàvers
- 6 Parque Tecnológico
- 7 Nou Barris Activa
- 8 innoBA
- Puntos de atención en la ciudad



© Barcelona Activa
Darrera actualització 2018

Cofinanciado por:



Síguenos en las redes sociales:

- barcelonactiva.cat/empreses
- [barcelonactiva](https://www.facebook.com/barcelonactiva)
- [barcelonactiva](https://twitter.com/barcelonactiva)
- [company/barcelona-activa](https://www.linkedin.com/company/barcelona-activa)