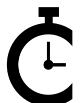
Data cleaning y data wrangling

PID_00264728

Xavi Font

Tiempo mínimo de dedicación recomendado: 5 horas



Universitat Oberta de Catalunya

Xavi Font

El encargo y la creación de este recurso de aprendizaje UOC han sido coordinados por el profesor: Jordi Ayza Graells (2019)

Primera edición: marzo 2019
© Xavi Font
Todos los derechos reservados
© de esta edición, FUOC, 2019
Av. Tibidabo, 39-43, 08035 Barcelona
Diseño: Manel Andreu
Realización editorial: Oberta UOC Publishing, SL

Ninguna parte de esta publicación, incluido el diseño general y la cubierta, puede ser copiada, reproducida, almacenada o transmitida de ninguna forma, ni por ningún medio, sea éste eléctrico, químico, mecánico, óptico, grabación, fotocopia, o cualquier otro, sin la previa autorización escrita de los titulares del copyright.

Índice

Introducción 5

Objectivos 6

1. Preprocesado de datos y calidad de los datos 7
- 1.1. Introducción 7
- 1.2. *Data cleaning* 7
- 1.3. Tipos de problemas en los datos 8
- 1.4. Herramientas para tratar datos 9
- 1.5. Software para el análisis de datos 10
  - 1.5.1. El entorno de desarrollo 10
- 1.6. Breve introducción a R 14
  - 1.6.1. Vectores 18
  - 1.6.2. Matrices y *arrays* 19
  - 1.6.3. *Data frames* 19
  - 1.6.4. Listas 21
  - 1.6.5. Factores 22
- 1.7. Representación gráfica con R 22
  - 1.7.1. Gráficos con R 23
  - 1.7.2. Gráficos con lattice 23
  - 1.7.3. Gráficos con ggplot2 24
  - 1.7.4. Ejemplos de tres sistemas de representación 25
- 1.8. Ayuda 26

2. Selección y filtrado de datos 28
- 2.1. Introducción 28
  - 2.1.1. mtcars Data Set 28
- 2.2. Selección de variables 29
- 2.3. Filtrar observaciones 31
  - 2.3.1. Operaciones relacionales 32
  - 2.3.2. Operaciones lógicas 33
  - 2.3.3. Diferencias con el acceso base de R 33

3. Organizar, crear y agrupar según preferencias 34
- 3.1. Introducción 34
- 3.2. Crear nuevas variables 34
- 3.3. Crear estadísticos (sintetizar) 37
- 3.4. Organización de los datos 38
- 3.5. Agrupación de los datos 39
- 3.6. Uso de *pipes* 40
- 3.7. Remodelación de datos 41

3.7.1. gather ... 42
3.7.2. spread ... 43
3.7.3. separate ... 43
3.7.4. unite ... 44
3.7.5. Síntesis de la remodelación de datos ... 45
3.8. Unión de datos ... 46
3.8.1. Enlazar por filas ... 46
3.8.2. Enlazar por columnas ... 47
3.8.3. Unión, intersección y diferencia ... 47
3.8.4. Joins ... 49
3.8.5. Inner Join ... 49
3.8.6. Left Join ... 49
3.8.7. Right Join ... 50
3.8.8. Full Joins ... 50
3.8.9. Otras operaciones de Join ... 50

4. Identificación de *outliers* (anomalías) ... 53
4.1. Introducción ... 53
4.2. ¿Por qué es importante la detección de *outliers*? ... 53
4.3. Metodos tradicionales ... 54
4.3.1. Métodos avanzados ... 55
4.4. *Missing values* ... 58

Resumen ... 60

Introducción

La importancia en la calidad de los datos es evidente. Sin una garantía de que los datos no están completamente limpios, el análisis posterior carece de todo rigor científico y está en entredicho su capacidad para que sea de utilidad para el fin que buscamos.

Los datos vienen con una serie de posibles anomalías. Algunas de estas están provocadas por la intervención humana, otras por la instrumentación y el uso de sensores, otras por el diseño experimental empleado o por las estrategias de manipulación/transformación de los datos. Cualquier técnica de *machine learning* parte del supuesto de que los datos son de calidad.

Si los datos no son de calidad, tampoco lo serán los resultados que se deriven de su uso, independientemente de los métodos aplicados.

La detección de anomalías o la identificación de observaciones atípicas tiene como objetivo poner al descubierto situaciones raras, extrañas o poco normales, pero que aportan un conocimiento relevante y que pueden inducir a interpretar eventos estratégicos y de interés para nuestros objetivos.

La utilización de procedimientos que permitan preparar los datos con el objetivo de que se garantice su calidad, así como la idoneidad para su uso, es la base de este módulo.

Objectivos

Este módulo describe un conjunto de operaciones que permiten manipular los datos para que cumplan con las especificaciones impuestas. Estos requisitos pueden ser obvios, como la identificación de errores, valores nulos, pero también la preparación de los datos de acuerdo con la técnica que se aplique.

Los principales objetivos los podemos detallar a continuación:

1. Describir diferentes tipologías de errores.
2. Conocer el sistema para la manipulación y el tratamiento automático de los datos (RStudio + R).
3. Conocer las operaciones típicas con los datos:

   a) Selección y filtrado
   b) Creación, ordenación, agrupación y síntesis
   c) Remodelación de los datos
   d) *Joins*

Todas estas operaciones permiten disponer de una batería diversa de herramientas para procesar los datos de una manera ordenada y coherente.

1. Preprocesado de datos y calidad de los datos

1.1. Introducción

En la mayoría de los proyectos donde hay datos y el objetivo es o bien la toma de decisiones, o bien conocer la realidad de un proceso de negocio, debemos entender que no basta con disponer de una cantidad enorme de datos si la calidad de estos está en entredicho. La relevancia y las implicaciones son evidentes si en el proceso de *data analytics* no garantizamos esta característica.

Nos encontramos con la obligación de que, para garantizar la calidad de los datos, es necesaria la manipulación, la transformación o el preprocesado de los datos. En terminología anglosajona aparece el término *data carpentry*, que resume este concepto.

Que los datos hayan sido tratados aplicando *data carpentry* y que parezcan correctos, en el sentido de disponer de nombre correctos, sin valores no disponibles o etiquetas, no garantiza que los valores estén libres de errores. Estas inconsistencias dependerán obviamente del negocio o de las características de donde provienen los datos.

> Ejemplo de esta situación puede encontrarse en el conjunto de datos que proceden de sensores específicos (por ejemplo, los que captan la actividad eléctrica del cerebro EEG (ElectroEncephaloGram). Podemos tener los datos aparentemente correctos pero puede existir en la señal mucho ruido externo que, si no es tratado correctamente, deje los datos como no tratables.

1.2. *Data cleaning*

El correcto (o no) proceso de limpieza de los datos puede afectar de manera muy notable a los resultados finales de la aplicación de técnicas de BA (*business analytics*). Las acciones necesarias para tratar valores perdidos (*missing values*), para la identificación de valores extremos o atípicos (*outliers*) y otras acciones pueden afectar a nuestro análisis. Por esta razón el proceso de *data cleaning* debería considerarse siempre de una manera seria y como una fase ineludible en cualquier proyecto de datos.

Desde una perspectiva ingenieril, todo dato que proviene del mundo real puede no estar limpio. No es una premonición o aplicación del principio de Murphy, sino una constatación.

1.3. Tipos de problemas en los datos

Existen una serie de problemas con los datos que podemos agrupar en una de estas categorías:

- **Datos incompletos**: esto quiere decir que faltan algunos valores en alguna variable, que faltan variables de interés o que por el contrario los datos están de forma agregada y son necesarios de forma desagregada.

- **Datos con ruido**: los datos presentan algún tipo de error. Habitualmente podemos encontrar nombres o etiquetas erróneas, valores sin sentido, valores no fiables u *outliers*, entre otros.

- **Datos inconsistentes**: se aprecian discrepancias que pueden afectar a la codificación utilizada o bien a los nombres.

Figura 1. Tabla de clientes con diferentes problemas

custID	name	birthday	age	gender	civilS	movileP	zip
1732	John Mc Arthur	23 / 10 / 77	30	fem	single	-9	8008
895	Arthur Williams	7 / 6 / 1993	26	male	married	606333309	90134
1732	Amanda Rovert	31/02/98	20	fem		610390904	90134
2705	John Arthur	23 / 10 / 77	42	male	single	677944553	30031

En la figura 1 podemos identificar una serie de problemas que muestran la variedad y tipología de errores posibles en nuestros datos. A continuación enumeramos los errores asociados a esta tabla:

1) Mismo cliente con representaciones distintas. Esta situación se puede dar en nombres de calle, de pueblos o en otros casos.

2) Si la variable custID representa una clave, debería ser única y observamos que hay dos observaciones con el mismo identificador (este error tiene sentido en tablas que provienen del modelo relacional).

3) Inconsistencia entre el nombre del cliente y el género. Si analizamos las variables de forma independiente este tipo de problemas no aparecerá de ningún modo.

4) Contradicción entre la fecha de nacimiento y la edad del cliente. Esta contradicción se podría repetir en otras situaciones, como por ejemplo un cliente menor de 16 años con carné de conducir.

5) Valor incorrecto, o también puede indicar dato no suministrado y esta es la codificación para describir esta situación; también puede indicar que no tiene dispositivos móviles.

6) Dato no suministrado o dato que desconoce.

7) Datos incorrectos. No podemos tener una fecha que no corresponde a una fecha correcta dentro del calendario.

Nos podemos preguntar si estas situaciones se darán de alguna forma. Pensemos por ejemplo en la adquisición de datos. Imaginad un cliente que no sabe o no quiere facilitar alguno de los datos necesarios para rellenar nuestro formulario. ¿Paramos el proceso de compra (si lo fuese) o dejamos una información sin rellenar? Este tipo de situación se da en multitud de ocasiones. Otra problemática se puede dar con las fechas y su posterior análisis. Se podría continuar con una lista de errores provocados por el factor humano, por la maquinaria utilizada o por un software que se desconoce.

La adquisición de datos puede acarrear una lista considerable de potenciales problemas relativos a la calidad de los datos: sensores que no están correctamente funcionando o que tienen un error inherente (característica del dispositivo), podría ser por el entorno donde opera, por la interacción con otros dispositivos y/o personas o también por el medio donde se produce la transmisión.

Podemos tener datos que hacen referencia al mismo objeto (instancia) con una clara inconsistencia, o podemos tener problemas relativos a las convenciones y/o estándares que son de aplicación en nuestro caso. Existe una lista también larga que puede añadir conceptos relativos a bases de datos como violación de la integridad referencial (concepto de bases de datos relacionales).

1.4. Herramientas para tratar datos

Existen una serie de herramientas profesionales para todas las tareas relacionadas con el *data cleaning*. Estas se pueden encontrar como solución en IBM, ORACLE o SAS, entre otras. Nuestra aproximación se centra en otro conjunto de herramientas que disponen de una serie de principios muy recomendables y que entendemos como lenguajes de programación:

- Tienen licencia GNU.
- Están disponibles en cualquier entorno: Windows, MAC, Unix.
- Tienen una comunidad muy activa y comprometida.
- Disponen de documentación.
- Son utilizadas por universidades de referencia: MIT, Stanford, Harvard.
- Son utilizadas por empresas de referencia: Google, Facebook.

De entre la tres opciones posibles, Julia, Python y R, nos vamos a decantar por la última de ellas. La realidad es que muchos profesionales tienen conocimientos avanzados en R y Python, por ejemplo (ver referencia Airbnb).

1.5. Software para el análisis de datos

La elección de realizar toda la batería de técnicas relacionadas con *data analytics* a través del entorno facilitado por R no debe sorprender. R se creó como un software específicamente para el análisis de datos (Julia, desarrollado por el MIT, da como una de las ventajas el hecho de incorporar la facilidad de modelización que ofrece R).

The R Project for Statistical Computing
Fuente: https://www.r-project.org/

R aparece como una intersección entre el análisis de datos y la computación. Por tanto, debe facilitar la exploración de los datos en todas sus vertientes. Se pueden apreciar sus características en el tratamiento, la manipulación y la representación de datos, así como la simplicidad en la modelización estadística. Debemos pensar en R no ya como un lenguaje de programación, sino como un entorno interactivo que permite realizar proyectos de análisis de datos.

RStudio IDE para un acceso fácil y rápido a R
Fuente: https://www.rstudio.com/

La utilización de un IDE (*integrated development environment*) para R, como RStudio, ofrece un conjunto de soluciones integradas para impulsar proyectos relacionados con el tratamiento de datos. Se pueden citar entre algunas de sus características las siguientes:

- Facilidad para desarrollar proyectos de gran envergadura, en complejidad y/o en dimensión.
- Permite acercar herramientas de colaboración y/o comunicación.
- Gestión y mantenimiento de paquetes (contienen librerías).
- Acceso rápido a la ayuda y/o a las visualizaciones de gráficos.
- Editor de código (en diferentes formatos: .R, .Rmd).
- Rápido acceso al historial de código ejecutado y de los objetos definidos en el sistema (variables y funciones).

RStudio

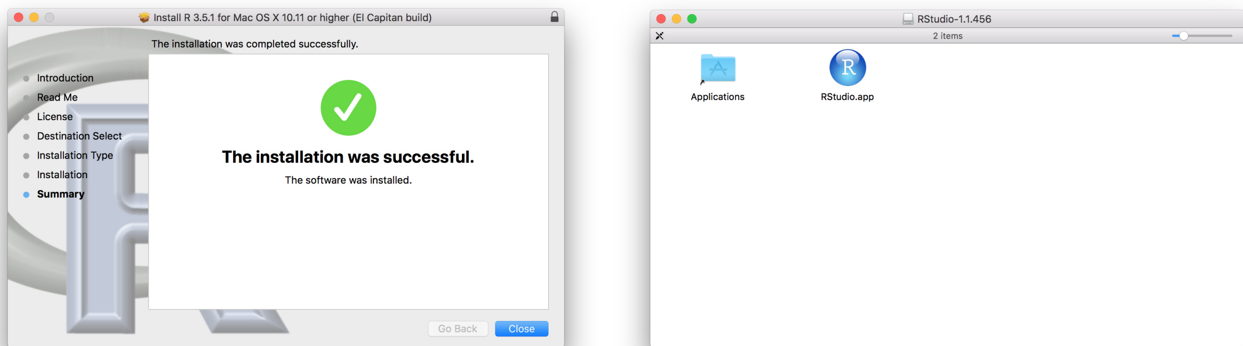
La herramienta de referencia para trabajar con R es RStudio. Este IDE permite realizar las operaciones de una manera más organizada y fácil.

1.5.1. El entorno de desarrollo

Para poder entender y sobre todo practicar los conceptos presentados en este capítulo, recomendamos que se instalen los siguientes productos (en el orden indicado):

- R
- LaTeX (en caso de querer generar *reports* en formato .pdf)
- RStudio
- *Packages* de interés

Figura 2. Proceso de instalación



A la izquierda, proceso de instalación de R completado; a la derecha, instalación de RStudio lista. El orden de instalación sería: primero R como se indica en la figura de la izquierda, a continuación el compilador de LaTeX (MiKTeX para Windows) y finalmente el IDE de RStudio.

La base de toda acción se realizará en el IDE de R, RStudio. A continuación se presenta una rápida descripción de este:

1) Editor de código: el editor de código no aparece visible hasta que abrimos un documento de código. En el momento en el que queramos escribir un *script* de código R (o cualquier otro formato reconocible por el IDE, como R markdown, C++ o javascript, entre otros) aparecerá en la parte superior izquierda una ventana para iniciar nuestro código.

Dentro del editor, RStudio destacará partes del código escrito para facilitar la lectura y comprensión del código. La notación estándar según el lenguaje R es la siguiente:

- Palabras reservadas de R en azul
- Caracteres de texto en verde
- Números en azul oscuro
- Comentarios en color verde

2) Finalización de código: el sistema facilita el nombre de la función que estamos intentando escribir. De esta manera se pueden apreciar los diferentes argumentos necesarios y evitar errores.

3) Ejecución de código: desde el editor de código podemos ejecutar el código de forma parcial o total. Las opciones disponibles son:

- Ejecución de una línea de código (Ctrl + Enter)
- Ejecución repetida de la previa (Ctrl + Shift + P)
- El botón de código ejecuta el fichero entero (Ctrl + Shift + Enter)

4) Pliegue de código: en casos donde el fichero de código sea grande o complejo, RStudio permite plegar regiones de código para facilitar la comprensión y/o lectura. RStudio realiza de manera automática estos pliegues: Regiones con

llaves. Habitualmente asociado a definición de funciones. Bloques de código. Utilizados en la generación de informes (*report generation*). Secciones de código predefinido por el usuario. Cualquiera de estas tres opciones es válida:

```
# Sección Primera ----------
# Sección 1 =========
### Sección A ##########
```

5) El historial y el entorno: esta ventana situada en la parte superior derecha y que comprende dos subventanas (vía tab) facilita información del entorno. Es decir, visualiza todos aquellos objetos que están definidos en el entorno. Se observa el nombre del objeto así como el número de elementos. Facilita el acceso a la totalidad del objeto descrito (clic sobre el símbolo de tabla).

Existe un botón para guiar en la importación de datos (utiliza la función read.csv()).

El historial muestra las instrucciones ejecutadas en la consola. Facilita también el paso de código al editor de código en caso de que sea necesario.

6) La consola: es la ventana situada en la parte inferior derecha y permite la ejecución de código R en el sistema.

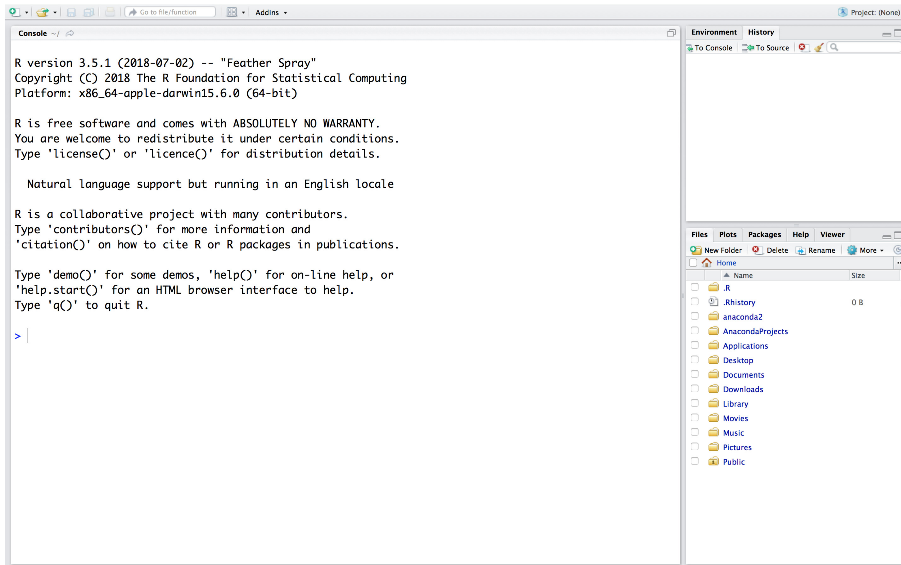
7) Información restante: la ventana situada en la parte inferior derecha está dividida en cuatro subventanas accesibles por (tab), que facilitan información relativa a:

- Ficheros y/o directorios en el directorio activo (función getwd()).
- Gráficos generado por la ejecución de comandos en la consola.
- Paquetes para la gestión y/o activación de librerías adicionales.
- Ayuda. Permite la búsqueda de cualquier función y ofrece ejemplos de uso.
- Panel de vista utilizado para mostrar aplicaciones web locales creadas con Shiny u OpenCPU.

Como es de esperar, existe la opción de customizar el entorno RStudio para que sea lo más cómodo y atractivo posible para nuestras necesidades. Estas opciones están disponibles a través del menú Herramientas > Opciones globales.

Podemos observar en la figura 3 la organización de las ventanas en el entorno integrado RStudio, que podemos configurar según nuestras necesidades.

Figura 3. Ventana del IDE RStudio



Para las tareas de codificación utilizaremos la ventana superior derecha (ver figura 4).

Figura 4. Ventana del editor de código

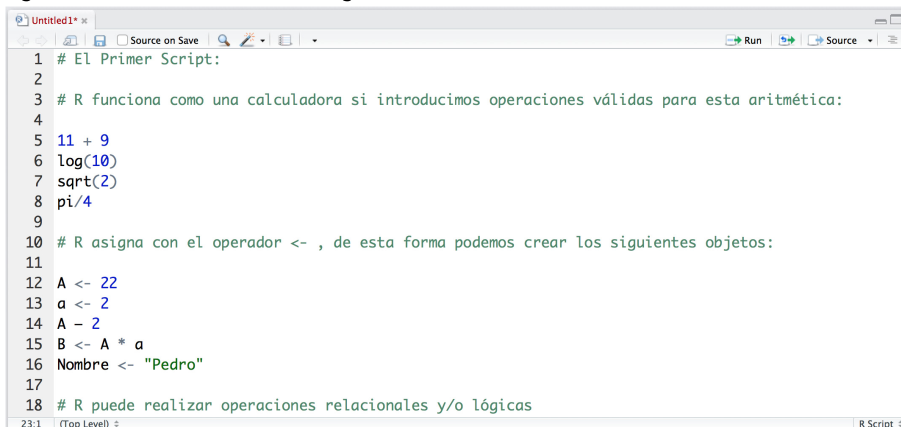
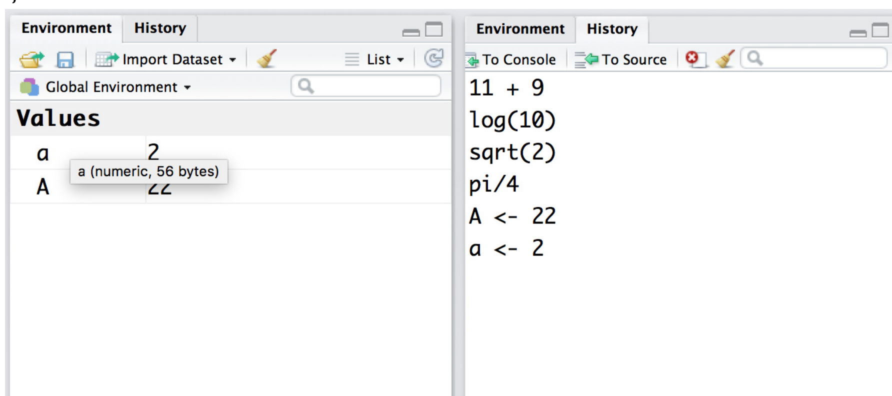


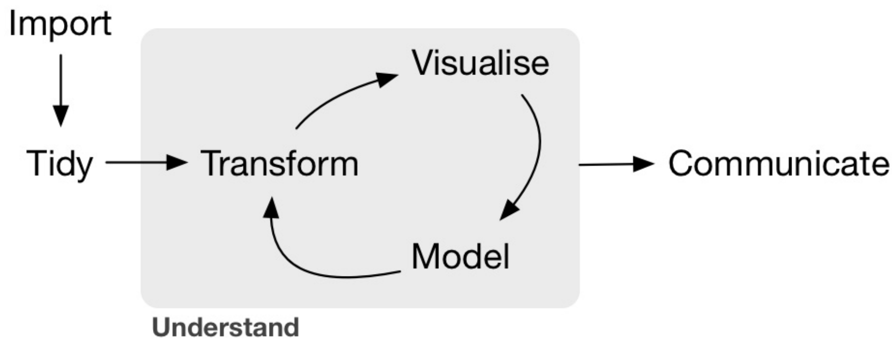
Figura 5. La información contenida en la ventana superior derecha se divide en la relativa al entorno y la que se destina a mantener un histórico de los comandos previamente ejecutados



A la izquierda, ventana de entorno. A la derecha, ventana del historial

El proceso básico que habitualmente vamos a seguir queda descrito en el siguiente diagrama.

Figura 6. Proceso habitual en un proyecto de datos



Fuente: https://r4ds.had.co.nz/introduction.html

1.6. Breve introducción a R

Existen multitud de tutoriales de introducción a R. En este subapartado se describen aquellos aspectos más básicos y esenciales que pueden ayudar a los que empiezan desde cero.

En general, entendemos que tenemos un sistema con el IDE RStudio funcionando correctamente. Es decir, hay una parte superior para editar código y/o texto y otra parte para ejecutar código y obtener resultados (consola). Se debe entender que el propósito de escribir código es hacerlo reproducible. Es decir, que otros colegas puedan replicar el mismo análisis obteniendo los mismos resultados.

El primer *script*:

R funciona como una calculadora si introducimos operaciones válidas para esta aritmética:

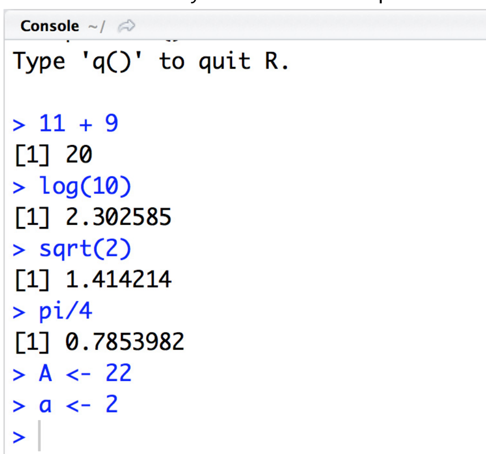
```
11 + 9
log(10)
sqrt(2)
pi/4
```

R asigna con el operador **<-** , de esta forma podemos crear los siguientes objetos:

```
A <- 22
a <- 2
```

```
A – 2
B <- A * a
Nombre <- "Pedro"
```

Figura 7. La consola es el lugar donde se ejecutan las sentencias de R y se obtiene una respuesta.



R puede realizar operaciones relacionales y/o lógicas.

```
A > B
B == A # representa una comparación, no una asignación
```

Algunos comentarios relativos a estas operaciones:

- R asigna automáticamente una clase a cada objeto que se crea, de manera automática.
- R diferencia entre mayúsculas y minúsculas. Por tanto, A y a son dos objetos distintos (observad la ventana de Environment – Entorno).
- R puede utilizar como operador de asignación el =, pero se utiliza <- (no confundir con la comparación ==).
- Para escribir un comentario utilizamos el signo #. Detrás de este símbolo R no lo lee ni interpreta nada.

Rguide

En muchas ocasiones apreciaremos el potencial de R instalando y cargando librerías desarrolladas por la comunidad y que podemos utilizar sin problemas. Escribir código R de forma correcta requiere práctica combinada con conocimiento, y para hacerlo de manera estandarizada se necesita una guía de estilo. Podemos apreciar las indicaciones de Google sobre la guía de estilo utilizada con R.

https://google.github.io/styleguide/Rguide.xml

Esta será la referencia utilizada al desarrollar ejemplos utilizando código R.

R es un lenguaje de programación orientado a objetos. Muchos de estos objetos son vectores, que se pueden crear utilizando las funciones c(), seq() y rep().

Tipos de datos básicos:

R dispone de los siguientes tipos básicos:

- carácter
- numérico (real o decimal)
- entero
- lógico
- complejo
- character: “R”, “Enjoy”
- numeric: 15, 3.14
- integer: 7L (the L tells R to store this as an integer)
- logical: TRUE, FALSE
- complex: 2-3i (complex numbers with real and imaginary parts)

R permite examinar el contenido de los objetos definidos en nuestro entorno. Esta es una posible lista de comandos que permiten visualizar esta información:

- class() – TIPO DE OBJETOt (high-level)?
- typeof() – tipo de objeto (low-level)?
- length() – dimensión o longitud What about two dimensional objects?
- attributes() – información adicional (metadata). Para un conjunto de datos obtendremos:
- – nombre de las columnas
- – nombre de las filas
- – estructura de los datos

Funciones para identificar si un objeto es de un tipo particular:

- *is.numeric*
- *is.integer*
- *is.character*
- *is.factor*
- *is.na*

Si bien no vamos a tratar con todos los tipos que R puede tratar, esta lista da idea de los diferentes objetos que R soporta y que son susceptibles de ser utilizados si los necesitamos:

spoken.type	class	mode	typeof	storage.mode
logical vector	logical	logical	logical	logical
integer vector	integer	numeric	integer	integer
numeric vector	numeric	numeric	double	double
complex vector	complex	complex	complex	complex
character vector	character	character	character	character
raw vector	raw	raw	raw	raw
factor	factor	numeric	integer	integer
logical matrix	matrix	logical	logical	logical
numeric matrix	matrix	numeric	double	double
logical array	array	logical	logical	logical
numeric array	array	numeric	double	double
list	list	list	list	list
pairlist	pairlist	pairlist	pairlist	pairlist
data frame	data.frame	list	list	list
closure function	function	function	closure	function
builtin function	function	function	builtin	function
special function	function	function	special	function
environment	environment	environment	environment	environment
null	NULL	NULL	NULL	NULL
formula	formula	call	language	language
expression	expression	expression	expression	expression
call	call	call	language	language
name	name	name	symbol	symbol
paren in expression	(	(	language	language
brace in expression	{	call	language	language
S3 lm object	lm	list	list	list
S4 dummy object	dummy	S4	S4	S4
external pointer	externalptr	externalptr	externalptr	externalptr

Si nos encontramos con alguna situación que no sabemos resolver, se recomienda dar información de las características de nuestro entorno así como la versión utilizada. Esta información de nuestra sesión se puede obtener con la siguiente sentencia:

```
print(sessionInfo(), local = FALSE)
```

La inclusión de esta información en foros es, en ocasiones, obligatoria para que los usuarios entiendan el entorno de configuración de nuestro sistema:

```
> print(sessionInfo(), local = FALSE)
R version 3.5.1 (2018-07-02)
Platform: x86_64-w64-mingw32/x64 (64-bit)
Running under: Windows >= 8 x64 (build 9200)

Matrix products: default
```

```
attached base packages:
[1] stats     graphics  grDevices utils     datasets
[6] methods   base

loaded via a namespace (and not attached):
[1] compiler_3.5.1 tools_3.5.1    yaml_2.2.0
```

1.6.1. Vectores

Uno de los tipos de datos más utilizados es el vector. Un vector es una estructura compuesta por un conjunto de elementos del mismo tipo (enteros, reales o caracteres, por ejemplo).

```
# Vectores
notasPosibles <- c(0:10) # vector de enteros con 11 elementos
marcaCalidad <- LETTERS[1:4] # vector de caracteres con 4 valores
peso <- seq(from=50,to=100,by=0.1) # vector de reales con 501 valores
ordenAlu <- rep(1:2,each=3,times=5) # vector de enteros con 30 valores
# : 2 x 3 x 5
```

El contenido del objeto utilizado para asignar no se visualiza en la consola. Únicamente se especifica una asignación. Si se quiere visualizar el contenido, podemos escribir el nombre del objeto, con lo que visualizaremos su contenido:

```
> ordenAlu
 [1] 1 1 1 2 2 2 1 1 1 2 2 2 1 1 1 2 2 2 1 1 1 2 2 2 1 1 1 2 2 2
```

La forma de acceder al vector es utilizando el operador de indexación, que es el corchete: [ ]. El primer elemento se sitúa en la posición 1 (no en la posición 0, como ocurre en otros lenguajes de programación). Así podemos especificar el elemento que interesa.

Se puede observar cómo R asigna de manera automática el tipo de dato de cada objeto. Los tipos básicos de R son los reales (numeric), los números complejos (complex), los lógicos (logical) y los caracteres (caracter).

Con estos tipos básicos se pueden construir estructuras de datos más complejas y útiles, como los vectores introducidos brevemente en el anterior punto.

Enlaces de interés

En caso de querer profundizar y practicar con este tipo de objetos, se recomienda realizar los ejemplos que figuran en los siguientes enlaces:
https://sejdemyr.github.io/r-tutorials/basics/vectors/
https://bit.ly/2MW3fGX

1.6.2. Matrices y *arrays*

Una fácil generalización de los vectores son las matrices (objeto de dos dimensiones) y los *arrays* (objeto de tres o más dimensiones). En caso de realizar al mismo tiempo la asignación y la visualización del objeto, hay que utilizar paréntesis sobre la instrucción que queremos ejecutar.

```
# Matrices y visualización
> (mx <- matrix(1:20,nrow=4,ncol = 5,byrow = TRUE))
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    6    7    8    9   10
[3,]   11   12   13   14   15
[4,]   16   17   18   19   20

(m3x <- array(1:20,dim=c(2,5,2)))
, , 1

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

, , 2

     [,1] [,2] [,3] [,4] [,5]
[1,]   11   13   15   17   19
[2,]   12   14   16   18   20
```

1.6.3. *Data frames*

Los objetos más utilizados para analizar datos son los *data frames*. Conceptualmente es como una matriz, pero con la característica de que las columnas pueden ser de tipo diferente.

```
# Data frames
col1 <- c(1054,2342,3302,7914)
col2 <- c("azul", "rojo", "blanco","verde")
col3 <- c(TRUE,TRUE,TRUE,FALSE)
df <- data.frame(col1,col2,col3)
names(df) <- c("identif","color","verific") # nombre de las variables
```

```
# Data frames
col1 <- c(1054,2342,3302,7914)
```

```
col2 <- c("azul", "rojo", "blanco", "verde")
col3 <- c(TRUE,TRUE,TRUE,FALSE)
df <- data.frame(col1,col2,col3)
names(df) <- c("identif","color","verific") # nombre de las variables
```

Si queremos acceder al *data frame* creado con nombre df:

```
df[2:3] # columnas 2 y 3 del data frame
   color verific
1   azul    TRUE
2   rojo    TRUE
3 blanco    TRUE
4  verde   FALSE

> df[c("identif","verific")] # columnas identif and verific del data frame
  identif verific
1    1054    TRUE
2    2342    TRUE
3    3302    TRUE
4    7914   FALSE

> df$color # variable Color del data frame
[1] azul   rojo   blanco verde
Levels: azul blanco rojo verde

> df[3:4,c(1,3)] # fila 3 y 4 con las variables / columnas 1 y 3
  identif verific
3    3302    TRUE
4    7914   FALSE
```

Es importante recalcar que la mayor parte de los datos ya introducidos en algunas de las librerías de R están en formato *data frame*. Por ejemplo, si queremos trabajar con el fichero mtcar podemos introducir el siguiente comando:

```
# Lectura datos
>data("mtcars")

> str(mtcars)
'data.frame': 32 obs. of  11 variables:
 $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
 $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
 $ disp: num  160 160 108 258 360 ...
 $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
 $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
```

```
 $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
 $ qsec: num  16.5 17 18.6 19.4 17 ...
 $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
 $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
 $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
 $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

Se aprecia la estructura interna del objeto mtcars como un *data frame* con 11 columnas y 32 filas. O dicho en lenguaje más propio de análisis de datos: 11 variables y 32 observaciones.

A continuación mostramos un listado de los métodos que podemos aplicar a los *data frames* para obtener la siguiente información:

- *dim()* - dimensión del *data frame* (número de filas y número de columnas).
- *head()* - muestra las primeras filas.
- *tail()* - muestra las últimas filas.
- *nrow()* - número de filas.
- *ncol()* - número de columnas.
- *names()* - muestra el atributo *names*, que permite obtener el nombre de las columnas.

1.6.4. Listas

El último objeto que se describe brevemente son las listas (*list*), que son la estructura más flexible de todas las mostradas. Una lista es un conjunto de objetos de cualquier tipo. Es decir, una lista puede contener un *data frame*, un vector, un escalar, una lista o cualquier otro objeto que deseemos añadir.

```
# ejemplo de una lista con 4 componentes -
# a string, a numeric vector, a matrix, and a scaler
milista <- list(nombre=Nombre, misNotas=notasPosibles, miMat3D=m3x,mydf=df)
class(milista) # tipo de objeto
length(milista) # número de elementos (u otros objetos) en la lista
attributes(milista) # información - metadata del objeto

# ejemplo de una lista con dos listas
ml2 <- c(milista,milista)
length(ml2)
attributes(ml2)

# ¿Cómo se accede a los elementos de una lista? Utilizando el operador [[]]
milista[[2]] # componente 2 de la lista
milista[["nombre"]] # componente de la lista con el nombre nombre
```

```
milista[c(1,2)] # componente 1 y 2
milista[[2]][2:3] # elementos 2 y 3 de la componente 2 de la lista (vector misNotas)
```

1.6.5. Factores

Existen otros tipos de datos que pueden ser de gran utilidad si se quieren realizar análisis estadísticos y según qué representación gráfica. Estos dos tipos de datos a los que nos referimos son las variables categóricas. R permite declarar una variable como factor o variable nominal. Si se necesita especificar una variable categórica con orden, por ejemplo notas donde podemos entender que hay un orden implícito (bien, notable o excelente), se puede definir en R como un factor ordenado (*ordered*).

```
# Factores
genero <- c(rep("hombre",3), rep("mujer", 35))
genero <- factor(genero)
str(genero)

# Notas "bien", "notable", "excelente"
notaOpt <- c("bien", "notable", "excelente")
notaQ <- sample(notaOpt,size=5,replace=TRUE)
str(notaQ)
notaQ <- ordered(notaQ)
str(notaQ)

# recodifica las notas a 1,2,3 y las asocia
# 1:"bien", 2:"notable", 3:"excelente" internamente
# R lo tratará como un factor
```

Uno de los aspectos más valorados de R es su facilidad para visualizar datos de manera fácil. Si bien R dispone de una serie de comandos base para realizar representaciones gráficas, como histogram, boxplot o plot, nuestra aproximación será a través del *package* ggplot2.

Aunque un módulo posterior tratará con más detenimiento los procesos de visualización, en esta pequeña introducción mostramos cómo realizar representaciones gráficas básicas.

1.7. Representación gráfica con R

R tiene tres sistemas de representación gráfica. El primero es el conocido como sistema de representación básico; es poco atractivo pero es fácil de utilizar. El segundo es lattice (del paquete lattice). Una propuesta mejor es utilizar

el *package* **ggplot2** desarrollado por Hadley Wickham en 2005 y que ofrece una estructura en la especificación de los gráficos intuitiva, coherente y clara, llamada gramática gráfica.

1.7.1. Gráficos con R

Es la aproximación más habitual cuando uno quiere representar gráficos de sus datos. La forma de realizarlos es conceptualmente simple y se construye el gráfico paso a paso utilizando las diferentes funciones que pone R a nuestra disposición.

Por ejemplo, la creación de un histograma, *boxplot* o diagrama de dispersión se lleva a cabo fácilmente con las siguientes llamadas:

```
hist(mpg)
boxplot(mpg~cyl)
plot(hp,mpg)
```

El principal problema es que si queremos mejorar la salida, debemos hacer uso de una batería de afinaciones que no siempre son directas. Algunas opciones del gráfico se consiguen con el comando: *par*

- pch: símbolo utilizado para representar el gráfico.
- lty: indica el tipo de línea (*line type*).
- lwd: la anchura de la línea (*line width*).
- col: define el color, especificado como un número, *string* o código hexadecimal.
- bg: color del fondo (*backgroud*).
- mar: las dimensiones del margen.
- oma: dimensiones de los margenes exteriores.
- mfrow: número de gráficos por fila.
- mfcol: número de gráficos por columna.

1.7.2. Gráficos con lattice

Únicamente se pueden realizar gráficos en este formato si previamente instalamos y cargamos la librería lattice.

Esta librería se caracteriza por el uso de funciones simples especificando los parámetros directamente en la llamada a la función. Es especialmente recomendada para gráficos condicionales. Es decir, cómo cambia el consumo en función de la potencia según el número de cilindros. La sintaxis es muy parecida a la formulación estadística utilizada para los modelos ANOVA (análisis de la varianza).

A continuación se exponen algunas de las funciones más utilizadas con la librería lattice:

- xyplot: creación de diagramas de dispersión (*scatterplots*).
- bwplot: *box-and-whiskers* (*boxplots*).
- histogram: histogramas.
- stripplot: como un *boxplot* pero con los puntos (representado las observaciones).
- dotplot: diagrama de puntos.
- splom: matriz de diagramas de dispersión (similar a *pairs* en R).
- levelplot, contourplot: para representar imágenes.

Las llamadas a las funciones típicas serían:

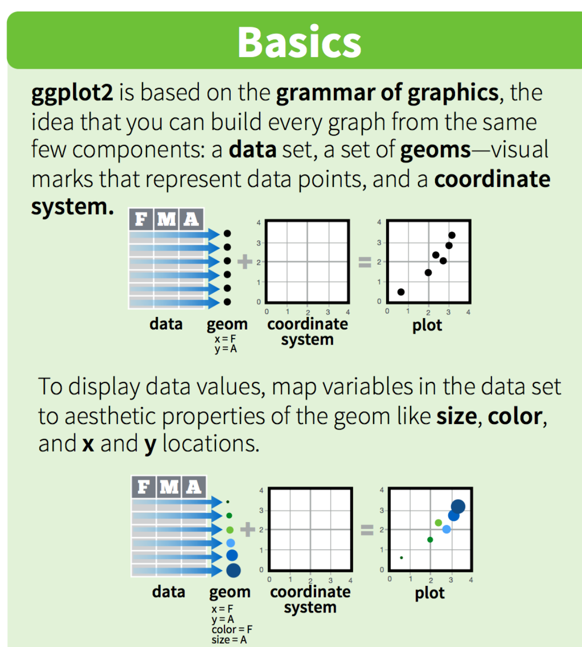
```
histogram(~mpg)
bwplot(~mpg | cyl)
xyplot(mpg~hp)
```

1.7.3. Gráficos con ggplot2

Se caracteriza por la facilidad en la creación de gráficos más atractivos y con una notación coherente. La idea de ggplot2 es construir el gráfico a partir de capas que podemos añadir según nuestras necesidades.

La filosofía de uso de este sistema se resume en el gráfico de la figura 8.

Figura 8. Enfoque de la gramática gráfica



Fuente: https://github.com/rstudio/cheatsheets/blob/master/data-visualization-2.1.pdf

Para realizar los mismos gráficos que en los anteriores ejemplos utilizando ggplot2, haríamos:

```
ggplot(mtcars, aes(mpg)) + geom_histogram(binwidth = 7)
 ggplot(data = mtcars, aes(factor(cyl), mpg)) +
        geom_boxplot()
ggplot(mtcars, aes(hp,mpg)) + geom_point()
```

Enlace de interés

Para más información sobre la sintaxis de la gramática gráfica basada en capas, podéis consultar este enlace: https://bit.ly/2b1gp5j

1.7.4. Ejemplos de tres sistemas de representación

La representación gráfica más utilizada es el histograma. Esta representación trata de describir la distribución de los datos (numéricos) de una manera visual y sintética.

La segunda representación gráfica que se muestra es el *boxplot*, también llamada diagrama de caja. No es más que un histograma simplificado que de una manera muy fácil transmite la división de los datos en sus cuartiles (Q1, Q2 y Q3).

La última representación es la de un gráfico en 2D. En este caso utilizamos el eje de las abscisas para representar la potencia del vehículo (eje x) contra el consumo (kilómetros recorridos por litro de gasolina) en el eje de ordenadas (eje y).

Figura 9. Representación con el sistema gráfico base de R

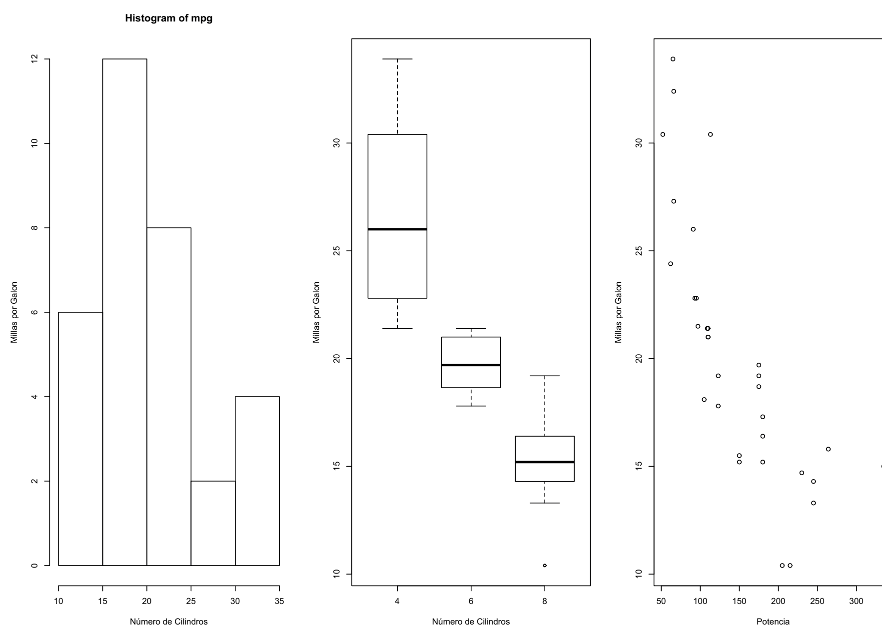


Figura 10. Representación con el sistema gráfico lattice

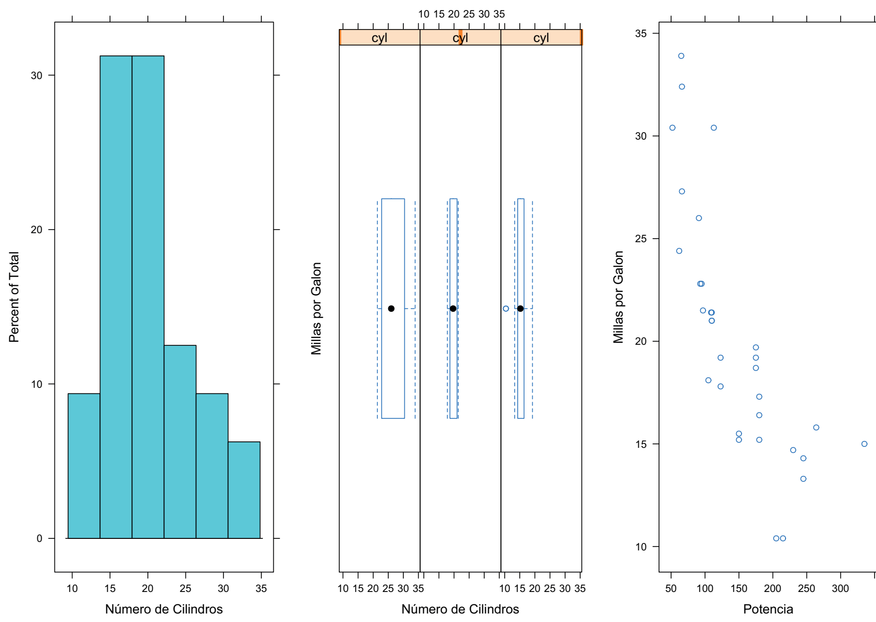
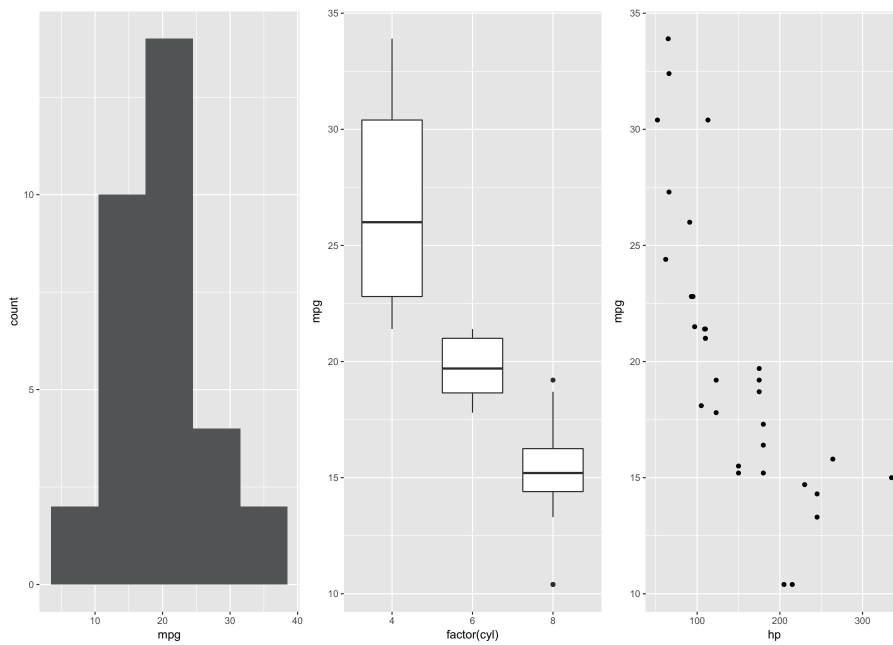


Figura 11. Representación con el sistema gráfico ggplot2



1.8. Ayuda

Cómo obtener ayuda con R es relativamente sencillo: las dos aproximaciones más utilizadas son a través de la ayuda local por medio de los siguientes comandos:

- *?help*
- *help.search("variance")*

- *help(nombre del comando) #si sabes exactamente el nombre del comando*
- *¿&&' #para operadores o palabras como if y for*
- *help.start() for R online documentation*

y a través de URL de interés:

- RSeek: http://www.rseek.org/
- Revolutions: http://blog.revolutionanalytics.com/
- R-bloggers: http: //www.r-bloggers.com/

2. Selección y filtrado de datos

2.1. Introducción

El conjunto de las operaciones que realizamos antes de poder aplicar un modelo representará un tiempo que oscila entre el 50 y el 80 % de nuestra dedicación. Para realizar la manipulación de los datos, vamos a utilizar una librería de R llamada dplyr. Si bien podemos utilizar las instrucciones por defecto que ofrece R, la facilidad con que la librería dplyr permite realizar *data carpentry* merece su atención. Entre sus mejores características cabe destacar:

- Una clara sintaxis.
- Correspondencia entre el comando y la acción.
- Permite una rápida comprensión y lectura del código.

En este subapartado y en la siguiente vamos a simplificar los conceptos para que sea fácil el seguimiento de los ejemplos mostrados. El conjunto de los datos que se van a utilizar se denomina mtcars (*motor trend car road tests*). Obviamente, muchas de las facilidades que ofrece la librería dplyr solo pueden observarse si el conjunto de los datos analizado es suficientemente grande.

2.1.1. mtcars Data Set

Los datos que vamos a utilizar como ejemplo en este capítulo y los siguientes están ya introducidos en R con el nombre de *mtcars*. Este es un fichero de datos muy sencillo pero suficientemente real para entender las operaciones que vamos a realizar con él.

Para hacer uso de este conjunto de datos únicamente se debe introducir en R el comando data y el nombre del conjunto de datos. Si adicionalmente se quiere obtener información de estos datos, con el comando de ayuda obtendremos más información.

```
data(mtcars)
help(mtcars)
```

Los datos provienen de la revista *Motor Trend* de Estados Unidos del año 1974. La información contenida se centra en el consumo de combustible y en diez aspectos adicionales del diseño y rendimiento de 32 automóviles (modelos 1973-1974).

Descripción de variables y unidades

var Name	var Description	var Unit
mpg	Miles/(US) gallon	mpg
cyl	Number of cylinders	n
disp	Displacement in cubic inches(cu.in.)	CID
hp	Gross horsepower	hp
drat	Rear axle ratio	ratio
wt	Weight (1000 lbs)	lbs
qsec	1/4 mile time	s
vs	Engine (0 = V-shaped, 1 = straight)	logic
am	Transmission (0 = automatic, 1 = manual)	logic
gear	Number of forward gears	n
carb	Number of carburetors	n

2.2. Selección de variables

La selección de variables de nuestro *data frame* se realiza mediante la siguiente función:

select

```
Sintaxis:
select(.data, ...)
```

Con este comando se puede extraer cualquiera de las variables que estemos tratando.

Figura 12. Selección de variables

Fuente: https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf

Por ejemplo, ¿cómo seleccionar las variables mpg y hp?

```
select(mtcars,c("mpg","hp"))
```

n	r labels	mpg	hp
1	Mazda RX4	21.0	110
2	Mazda RX4 Wag	21.0	110
3	Datsun 710	22.8	93
...	...	...	...
31	Maserati Bora	15.0	335
32	Volvo 142E	21.4	109

Podemos seleccionar todas las variables excepto algunas que no queremos y señalemos esto con el signo negativo -. ¿Cómo seleccionamos todas las variables excepto las de esta lista: mpg, hp, disp y wt?

```
select(mtcars,-c("mpg","hp","disp","wt"))
```

El resultado de esta selección corresponde al siguiente conjunto de datos:

n	r labels	cyl	drat	qsec	vs	am	gear	carb
1	Mazda RX4	6	3.90	16.46	0	1	4	4
2	Mazda RX4 Wag	6	3.90	17.02	0	1	4	4
3	Datsun 710	4	3.85	18.61	1	1	4	1
...	...	...	...					
31	Maserati Bora	8	3.54	14.60	0	1	5	8
32	Volvo 142E	4	4.11	18.60	1	1	4	2

Podemos seleccionar el conjunto de las variables que van desde una variable a otra, seleccionando estas dos variables. ¿Cómo preparar los datos para incluir las variables que van de *wt* a *gear*?

```
select(mtcars,wt:gear)
```

El resultado es el esperado:

n	r labels	wt	qsec	vs	am	gear
1	Mazda RX4	2.620	16.46	0	1	4
2	Mazda RX4 Wag	2.875	17.02	0	1	4
3	Datsun 710	2.320	18.61	1	1	4
...	...	...	...			
31	Maserati Bora	3.570	14.60	0	1	5
32	Volvo 142E	2.780	18.60	1	1	4

Existen algunas opciones adicionales para seleccionar columnas según algunos criterios, que pueden ser:

- *ends_with()* = Selecciona las columnas que terminan con el *string* pedido.
- *contains()* = Selecciona las columnas que contienen el *string* pedido.

- *matches()* = Selecciona las columnas que cumplen la expresión regular facilitada.
- *one_of()* = Selecciona los nombres de las columnas del grupo de nombres facilitado.
- *num_range()* = Selecciona las columnas dentro del rango.

Con conjuntos de datos poco extensos estas funciones no tienen mucho sentido, pero si tratamos con datos con un número grande de variables, puede ser muy aconsejable:

Por ejemplo, ¿cómo obtener todas las variables que no contienen la letra a?

```
select(mtcars,-contains("m"))
```

El resultado es:

n	r labels	cyl	disp	hp	drat	wt	qsec	vs	gear	carb
1	Mazda RX4	6	160	110	3.90	2.620	16.46	0	4	4
2	Mazda RX4 Wag	6	160	110	3.90	2.875	17.02	0	4	4
3	Datsun 710	4	108	93	3.85	2.320	18.61	1	4	1
...	...	...	...							
31	Maserati Bora	8	301	335	3.54	3.570	14.60	0	5	8
32	Volvo 142E	4	121	109	4.11	2.780	18.60	1	4	2

2.3. Filtrar observaciones

El filtrado de observaciones de nuestro *data frame* se realiza mediante la siguiente función:

filter

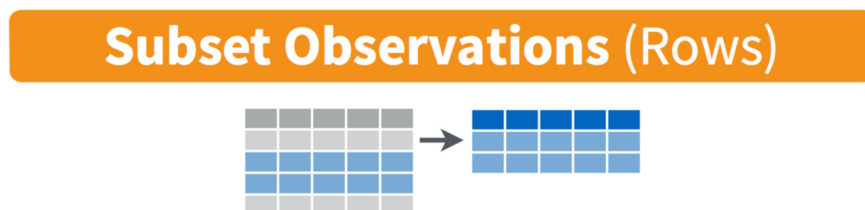
```
Sintaxis:
filter(.data, ...)
```

En esta situación la operación es sobre las filas o sobre las observaciones que cumplen con la condición o expresión lógica pedida.

Por ejemplo, filtrar las observaciones que tienen 4 cilindros (cyl==4) y que el número de marchas no es 4 (gear != 4)

```
filter(mtcars, cyl == 4 & gear != 4)
```

Figura 13. Filtrado de observaciones



Fuente: https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf

El resultado de esta operación es un conjunto de datos con 3 filas, las únicas que cumplen las condiciones impuestas.

mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
21.5	4	120.1	97	3.70	2.465	20.01	1	0	3	1
26.0	4	120.3	91	4.43	2.140	16.70	0	1	5	2
30.4	4	95.1	113	3.77	1.513	16.90	1	1	5	2

Obtener el subconjunto de filas que tienen 8 cilindros y el tiempo en recorrer un cuarto de milla es superior a 17.4.

```
filter(mtcars, cyl == 8, qsec > 17.4)
```

O también obtenemos el mismo resultado aplicando el operador lógico AND:

```
filter(mtcars, cyl == 8 & qsec > 17.4)
```

Veamos el resultado de la operación de *filter*.

mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
21.5	4	120.1	97	3.70	2.465	20.01	1	0	3	1
26.0	4	120.3	91	4.43	2.140	16.70	0	1	5	2
30.4	4	95.1	113	3.77	1.513	16.90	1	1	5	2

La facilidad en aplicar este filtrado depende básicamente de las comparaciones y la lógica de R.

2.3.1. Operaciones relacionales

- $x < y$: menor que
- $x > y$: mayor que
- $x <= y$: menor o igual que
- $x >= y$: mayor o igual que
- $x == y$: igual que

- *x*! = *y* : diferente que
- *%in%* : pertenece a un grupo
- *is.na* : es un NA
- *!is.na* : no es un NA

2.3.2. Operaciones lógicas

- & : AND : Y
- | : OR : O
- *xor* : exactamente O
- ! : NOT : negación
- *any* : alguno TRUE o cierto
- *all* : todos TRUE

2.3.3. Diferencias con el acceso base de R

El acceso a subconjuntos de un *data frame* se realiza con la indexación utilizando corchetes []. Así, acceder a las observaciones con 4 cilindros sería:

```
mtcars[mtcars$cyl == 4, ]
```

Comparadlo con la sintaxis de dplyr:

```
filter(mtcars, cyl == 4)
```

El mismo procedimiento se utiliza con la selección de variables. Seleccionar la variable wt se realiza en R base:

```
mtcars[ ,"wt"]
```

Mientras que en sintaxis de dplyr sería:

```
select(mtcars,wt)
```

3. Organizar, crear y agrupar según preferencias

3.1. Introducción

Adicionalmente a las operaciones de selección de variables y de filtrado de observaciones, muchas de las operaciones que se realizan tienen que ver con otros conceptos. Por ejemplo, con cómo se visualiza esta información, si los datos se presentan ordenadamente o no, si realizamos algún cálculo en función de una variable de tipo categórica o si creamos nuevas variables a partir de las ya existentes. En este subapartado presentamos este conjunto de operaciones adicionales que son utilizadas con frecuencia para obtener una primera aproximación a la naturaleza de los datos que estamos investigando.

3.2. Crear nuevas variables

En ocasiones se quiere crear una nueva variable a partir de las que ya tenemos disponibles en nuestro conjunto de datos. Esta operación tiene dos vertientes: manteniendo las variables previas o eliminándolas y obteniendo únicamente las que estamos creando.

Estas tareas se realizan con:

mutate() : creamos nuevas variables y preservamos las existentes.
transmute() : creamos nuevas variables y eliminamos las existentes.

Gráficamente se puede visualizar como muestra la figura:

Figura 14. Creación de nuevas variables

Fuente: https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf

En nuestro ejemplo podemos crear una nueva variable correspondiente a peso por caballos de potencia. Esta variable se interpreta como la relación potencia a peso y se usa habitualmente como medida de rendimiento de un vehículo.

```
mutate(mtcars,powerToWratio = hp/wt)
```

Obtenemos el siguiente resultado:

n	r labels	mpg	cyl	...	carb	powerToWratio
1	1	21.0	6	...	4	41.98473
2	2	21.0	6	...	4	38.26087
3	3	22.8	4	...	1	40.08621
...	...	...	...			
31	31	15.0	8	...	8	93.83754
32	32	21.4	4	...	2	39.20863

Otra variable que podríamos crear corresponde a una transformación de las unidades para hacer más interpretable nuestro conjunto de datos. Así, podemos convertir la variable mpg a km/l. Si sabemos la regla que hay que aplicar, esta operación es muy sencilla:

```
mutate(mtcars,kml = mpg* 0.425144)
```

n	r labels	mpg	cyl	...	carb	kml
1	1	21.0	6	...	4	8.928024
2	2	21.0	6	...	4	8.928024
3	3	22.8	4	...	1	9.693283
...	...	...	...			
31	31	15.0	8	...	8	6.377160
32	32	21.4	4	...	2	9.098082

Observad que para manipular los datos de manera que permanezcan permanentemente necesitamos actualizar la operación. O, dicho en otras palabras, necesitamos asignar el resultado a un objeto/variable:

```
mtcars <- mutate(mtcars,powerToWratio = hp/wt)
mtcars <- mutate(mtcars,kml = mpg* 0.425144)
```

O bien:

```
mtcars <- mutate(mtcars,powerToWratio = hp/wt, kml = mpg* 0.425144)
```

Si únicamente queremos quedarnos con las variables creadas, se utiliza el verbo *transmute* en lugar de *mutate*:

```
miNuevoDF <- transmute(mtcars,powerToWratio = hp/wt, kml = mpg* 0.425144)
```

Así quedaría nuestro nuevo conjunto de datos:

n	r labels	powerToWratio	kml
1	1	41.98473	8.928024
2	2	38.26087	8.928024
3	3	40.08621	9.693283
...	...	...	...
31	31	93.83754	6.377160
32	32	39.20863	9.098082

Una de las variantes que podemos encontrar de interés al querer aplicar técnicas de *machine learning* sobre nuestro conjunto de datos puede ser la de aplicar una transformación a todas las columnas. Una de estas transformaciones es la estandarización, es decir, que toda variable tenga una media de 0 y una desviación de 1.

Vamos a aplicar esta transformación sobre nuestro objeto miNuevoDF:

```
miNuevoDF <- mutate_all(miNuevoDF,funs(scale(.),.-mean(.),./sd(.)))
```

Observad que utilizamos el . para hacer referencia a la variable que utiliza la función, como si fuera la x.

n	rLab	pWrat	kml	pWratScal	kmlScal	pWrat_-	kml_-	pWrat_/	kml_/
1	1	41.98	8.93	-0.21	0.15	-3.35	0.39	2.58	3.48
2	2	38.26	8.93	-0.43	0.15	-7.07	0.39	2.35	3.48
3	3	40.09	9.69	-0.32	0.45	-5.25	1.15	2.46	3.78
...	...	...	...						
31	5	93.84	6.38	2.98	-0.84	48.50	-2.16	5.76	2.49
32	6	39.21	9.10	-0.38	0.22	-6.13	0.56	2.41	3.55

Existen una serie de operaciones que podemos aplicar sin ningún problema y que nos pueden ayudar en ocasiones para obtener las variables correctas para el propósito que necesitemos.

Operaciones aritméticas modulares: básicamente la división entera o el resto (%/% o %%). Este tipo de operación puede tener su uso para obtener las horas y/o minutos.

Operaciones matemáticas: para facilitar el uso de técnicas como las descritas anteriormente, normalización (scale) o logarítimicas (log) u otras aplicando los habituales operadores aritméticos.

Operaciones con rangos: en ocasiones queremos obtener un rango. Dplyr proporciona una serie de métodos que facilitan obtener rangos:

```
dense_rank() : mutate(miNuevoDF, rank = dense_rank(kml))
min_rank() : mutate(miNuevoDF, rank = min_rank(kml))
percent_rank() : mutate(miNuevoDF, rank = percent_rank(kml))
row_number()
```

Enlace de interés

Para obtener información adicional, podéis consultar las especificaciones en el siguiente enlace: https://bit.ly/2Ii75vp

Operaciones con retardos: en casos en los que necesitemos calcular diferencias entre observaciones, como puede ser variables que representen cotizaciones en bolsa y necesitamos trabajar con los incrementos/decrementos de la cotización. Esta operación puede no tener sentido si la columna no representa un eje temporal.

```
mutate(miNuevoDF, offSets = lag(kml),offSets2 = lag(kml,2))
mutate(miNuevoDF, offSets = lead(kml),offSets2 = lead(kml,2))
```

3.3. Crear estadísticos (sintetizar)

Otra de las operaciones que en ocasiones se deben realizar consiste en generar resúmenes de las variables que tenemos. Entre estos estadísticos tenemos algunos muy conocidos, como la media, la desviación y el rango, entre otros.

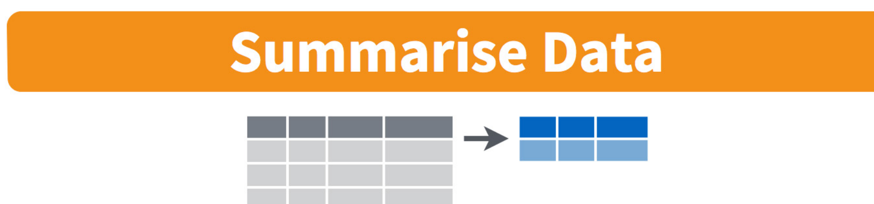
Si bien es cierto que en ocasiones las preguntas que nos podemos formular implican realizar una agrupación previa, por ejemplo cuál es el peso medio de los vehículos con 4 cilindros, en el siguiente apartado trataremos de añadir el concepto de agrupación.

La sintaxis para realizar este tipo de funciones es:

summarize()

La idea es la de sintetizar un amplio conjunto de observaciones en un único valor. Observad la idea a través de la figura mostrada en:

Figura 15. Obtención de medidas resumen (estadísticos)



Fuente: https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf

Por ejemplo, si queremos obtener una media y una desviación de nuestro objeto creado para la variable powerToWratio y llamado miNuevoDF:

```
summarize(miNuevoDF,mean = mean(powerToWratio), stdDev = sd(powerToWratio))
```

Obtendremos:

mean	stdDev
45.33466	16.28767

Podemos indicar otras funciones de interés:

```
summarize(miNuevoDF,meanKMl = mean(kml), sumKMl = sum(kml),nObs=n())
```

meanKMl	sumKMl	nObs
8.541409	273.3251	32

Observad que el objetivo de este tipo de operación es terminar con un valor numérico útil para interpretar o resolver una cuestión específica planteada.

Una lista de los diferentes métodos que podemos realizar sería la siguiente:

- Medidas de centralidad: mean(), median()
- Medidas de dispersión: sd(), IQR(), mad()
- Medidas de rango: min(), max(), quantile()
- Medidas de posición: first(), last(), nth(),
- Medidas de contaje: n(), n_distinct()
- Medidas lógicas: any(), all()

Una lista de ejemplos muy triviales sería:

```
summarize(miNuevoDF,first = first(kml))
summarize(miNuevoDF,th5 = nth(kml,5))
summarize(miNuevoDF,ndist = n_distinct(kml))
summarize(miNuevoDF,anyGT9 = any(kml>9))
summarize(miNuevoDF,allGT4 = all(kml>4.5))
```

3.4. Organización de los datos

La organización de los datos es especialmente relevante para inspeccionar de manera adecuada nuestros datos. En ocasiones necesitaremos visualizar los datos de forma creciente en función de una variable de interés. Cuando es necesaria la organización de los datos para definir un orden, utilizaremos la siguiente sintaxis:

arrange

```
arrange(miNuevoDF,kml)
```

n	r labels	powerToWratio	kml
1	1	39.05	4.42
2	2	39.64	4.42
3	3	63.80	5.65
...	...	...	...
31	31	30.00	13.77
32	32	35.42	14.41

Es posible ordenar de forma descendiente utilizando desc, y también es útil indicar dos variables en caso de que deseemos evitar empates al realizar la ordenación:

```
arrange(miNuevoDF,desc(kml),powerToWratio)
```

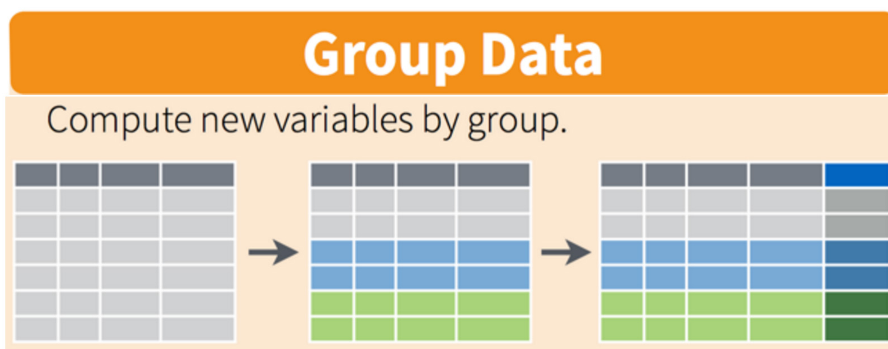
n	r labels	powerToWratio	kml
1	1	35.42	14.41
2	2	30.00	13.77
3	3	32.20	12.92
...	...	...	...
31	31	39.05	4.42
32	32	39.64	4.42

3.5. Agrupación de los datos

Una de las operaciones más habituales realizadas con los datos es la de agrupar en función de otra variable. Esta operación se puede combinar con otras de las ya descritas anteriormente. Por ejemplo, la utilidad del método summarize se ve incrementada si se combina con la agrupación de datos utilizando la sintaxis:

group_by

Figura 16. Operación de agrupar datos



Fuente: https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf

Para ver la relevancia de estas operaciones, volvamos a nuestro ejemplo con los datos de mtcars para visualizar la media de consumo en función del número de cilindros.

```
## agrupación con mtcars
byCyl <- group_by(mtcars,cyl)
summarize(byCyl,perf=mean(kml,na.rm = TRUE))
```

cyl	perf
4	11.335885
6	8.393557
8	6.419674

Quizá nos gustaría visualizar el resultado de forma ordenada según el número de cilindros o según la media. Estas operaciones son muy fáciles si utilizamos el verbo *arrange*.

Así, podemos obtener la siguiente vista (observación: el objeto resAux es el resultado de aplicar la operación anterior):

```
resAux <- summarize(byCyl,perf=mean(kml,na.rm = TRUE))
arrange(resAux,cyl)
```

cyl	perf
4	11.335885
6	8.393557
8	6.419674

3.6. Uso de *pipes*

En la mayoría de las ocasiones se va a realizar una combinación de operaciones de forma iterativa. El uso de *pipes* permite centrarse en el proceso de transformación de una manera natural e intuitiva. En el ejemplo previo obtendríamos lo siguiente:

```
mtcars %>% group_by(cyl) %>% summarize(perf=mean(kml,na.rm = TRUE)) %>% arrange(cyl)
```

En una única línea de código y de forma natural indicamos el orden en el que se realizan las operaciones. Es decir, tomamos los datos (mtcars), luego los agrupamos por el número de cilindros, a continuación calculamos la media para cada uno de estos grupos (definidos en el paso previo) y entonces ordenamos los datos en función de la variable cyl.

Para entender el funcionamiento de una *pipe*, basta con observar que en realidad la operación:

```
mtcars %>% group_by(cyl)
```

no es más que

```
group_by(mtcars,cyl)
```

y si extendemos este concepto a

```
mtcars %>% group_by(cyl) %>% summarize(perf=mean(kml,na.rm = TRUE))
```

es equivalente a la expresión siguiente:

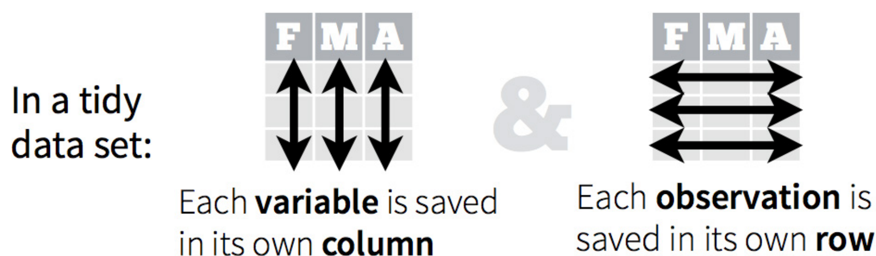
```
summarize(group_by(mtcars,cyl),perf=mean(kml,na.rm = TRUE))
```

3.7. Remodelación de datos

Muchas de las técnicas que se aplican en los entornos de *business analytics* requieren que los datos tengan un formato llamado tidy.

Entendemos por tidy un conjunto de datos donde cada observación está representada por una fila y cada variable, por una columna.

Figura 17. Formato deseable de los datos: tidy (por filas y columnas)



Fuente: https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf

En ocasiones este formato no está disponible por el modo como se adquirieron los datos originales. La librería dplyr ofrece una forma sencilla de remodelar el conjunto de datos según nuestros intereses.

3.7.1. gather

Cuando los datos presentan un formato donde una variable está representada en el eje de las filas y la otra variable en el eje de las columnas, siendo la intersección de estas el valor de interés (por ejemplo, el presupuesto de un departamento), no se cumple el principio de datos en formato tidy.

Imaginad datos relativos a nuestra empresa, donde los años figuran en la cabecera de las columnas y el departamento, en la de las filas. Esta situación se muestra en la siguiente tabla:

	2015	2016	2017	2018	2019
Sales	88800	101500	109300	91100	108100
IT	95400	102600	74700	124500	102300
Marketing	131200	134400	86300	107200	88900

En realidad, si cada columna representa una variable, no podemos tener cinco columnas para representar la misma variable.

La siguiente operación permite ensamblar los datos en el formato típico. La función descrita toma las columnas y contrae los pares clave-valor duplicando columnas según se necesite. Se utiliza la siguiente sintaxis:

gather

```
Sintaxis:
gather(.data, ...)


df2 <- df %>% rownames_to_column("depart")
df2.tidy <- gather(df2,key=years,"budget",2:6)
```

El resultado es la siguiente tabla:

depart	years	budget
Sales	2015	88800
IT	2015	95400
Marketing	2015	131200
Sales	2016	101500
IT	2016	102600
Marketing	2016	134400
Sales	2017	109300
IT	2017	74700
Marketing	2017	86300
Sales	2018	91100
IT	2018	124500
Marketing	2018	107200
Sales	2019	108100
IT	2019	102300
Marketing	2019	88900

La ventaja de este formato es que puede ser usado directamente por un modelo de regresión, por ejemplo.

3.7.2. spread

Realiza una operación inversa a la presentada anteriormente. De un conjunto de datos con dos columnas genera una columna relativa a esta variable con los valores asociados a esta.

spread

```
Sintaxis:
spread(.data,key,value, ...)
```

Si lo que queremos es un conjunto de datos donde los nombres de columna correspondan a los nombres de los departamentos, entonces utilizamos como clave el nombre de esta columna (depart) y el valor que se asocia lo indica value (budget).

```
spread(df2.tidy,key=depart,value=budget)
```

years	IT	Marketing	Sales
2015	95400	131200	88800
2016	102600	134400	101500
2017	74700	86300	109300
2018	124500	107200	91100
2019	102300	88900	108100

3.7.3. separate

Es una operación simple que consiste en separar una columna en dos o tres. Este tipo de operación puede tener sentido si se tiene el nombre de un vehículo y el nombre corresponde a la marca y al modelo. También puede resultar de utilidad si tenemos una columna de fecha y queremos disponer de la información en columnas separadas (una para años, otra para meses y una última para los días).

separate

```
Sintaxis:
separate(.data,into,sep, ...)
```

A continuación, para ver la utilidad (y el funcionamiento) de este verbo, vamos a utilizar el fichero que ya se ha utilizado anteriormente, mtcars, con el fin de, primero, convertir las etiquetas de cada fila en una variable y, posteriormente, dividir esta variable en tres componentes: marca, modelo e información adicional.

```
dfmt <- mtcars %>% rownames_to_column("brandModel")
mtcSep <- dfmt %>% separate(brandModel,into=c("brand","model","adInfo"),sep=" ")
```

Las advertencias que da la sentencia tienen sentido porque estamos interpretando que hay tres piezas de información (de aquí los tres nombres indicados).

brand	model	adInfo	mpg	gear	carb
Mazda	RX4	NA	21.0	4	4
Mazda	RX4	Wag	21.0	4	4
Datsun	710	NA	22.8	4	1
Hornet	4	Drive	21.4	3	1
Hornet	Sportabout	NA	18.7	3	2
Valiant	NA	NA	18.1	3	1
Duster	360	NA	14.3	3	4
Merc	240D	NA	24.4	4	2
Merc	230	NA	22.8	4	2
Merc	280	NA	19.2	4	4
....	....	....	...	...	...
Maserati	Bora	NA	15.0	5	8
Volvo	142E	NA	21.4	4	2

3.7.4. unite

En este caso se realiza una operación inversa a la anterior. Partimos de diferentes columnas y nos queremos quedar en una única columna.

unite

```
Sintaxis:
unite(.data,col,...,sep)
```

Vamos a utilizar el ejercicio anterior con los datos de mtcars para unir las columnas model y adInfo en una única variable.

Observad que la primera sentencia funciona pero genera en la salida una lista de NA que anteriormente se añadieron básicamente en la columna *adInfo*.

```
mtcSep %>% unite(modelName,model,adInfo,sep=" ")
```

```
mtcSep %>% replace_na(list(model = "", adInfo = "")) %>%
  unite(modelName,model,adInfo,sep=" ")
```

El resultado de eliminar los NA (se sustituye por un carácter vacío) permite visualizar el conjunto de datos en el formato deseado. Una columna para indicar la marca del vehículo y otra columna para indicar el modelo. (Observad que no hay ningún dato con el valor NA).

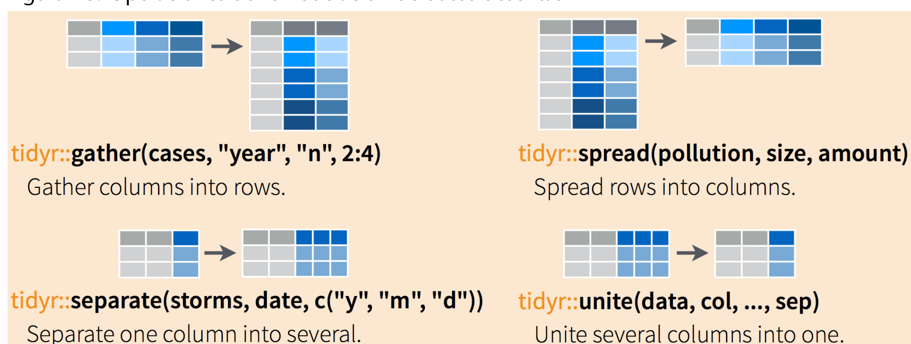
brand	modelName	mpg	cyl	am	gear	carb
Mazda	RX4	21.0	6	1	4	4
Mazda	RX4 Wag	21.0	6	1	4	4
Datsun	710	22.8	4	1	4	1
Hornet	4 Drive	21.4	6	0	3	1
Hornet	Sportabout	18.7	8	0	3	2
Pontiac	Firebird	19.2	8	0	3	2
Fiat	X1-9	27.3	4	1	4	1
Porsche	914-2	26.0	4	1	5	2
Lotus	Europa	30.4	4	1	5	2
Ford	Pantera L	15.8	8	1	5	4
Ferrari	Dino	19.7	6	1	5	6
Maserati	Bora	15.0	8	1	5	8
Volvo	142E	21.4	4	1	4	2

3.7.5. Síntesis de la remodelación de datos

Para visualizar las operaciones descritas, el siguiente gráfico facilita la comprensión de las cuatro operaciones mostradas:

- *gather*
- *spread*
- *separate*
- *unite*

Figura 18. Operaciones de remodelación de datos descritas



Fuente: https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf

3.8. Unión de datos

En ocasiones toda la información estará contenida en un fichero. En este caso no debemos preocuparnos por este tipo de operaciones que se describen en este subapartado. Si por el contrario una situación compleja obliga a añadir más información a nuestros datos, entonces se debe tener en consideración cómo unir datos.

En situaciones donde el objetivo de estudio implica disponer de datos provenientes de diferentes fuentes debemos decidir de qué forma vamos a unir los datos.

Imaginad un escenario con vehículos (de los que figuran en el conjunto de datos mtcars):

brand	modelName	mpg	cyl	am	gear	carb
Mazda	RX4	21.0	6	1	4	4
Mazda	RX4 Wag	21.0	6	1	4	4
Datsun	710	22.8	4	1	4	1
Hornet	4 Drive	21.4	6	0	3	1
Hornet	Sportabout	18.7	8	0	3	2

brand	modelName	mpg	cyl	am	gear	carb
Ferrari	Dino	19.7	6	1	5	6
Maserati	Bora	15.0	8	1	5	8
Volvo	142E	21.4	4	1	4	2

3.8.1. Enlazar por filas

Esta operación permite enlazar dos *data frames* por filas:

bind_rows

```
Sintaxis:
bind_rows(...)
```

Esta operación no dará problemas mientras los tipos de cada columna coincidan.

```
cars1 %>% bind_rows(cars2)
```

brand	modelName	mpg	cyl	am	gear	carb
Mazda	RX4	21.0	6	1	4	4
Mazda	RX4 Wag	21.0	6	1	4	4
Datsun	710	22.8	4	1	4	1
Hornet	4 Drive	21.4	6	0	3	1
Hornet	Sportabout	18.7	8	0	3	2
Ferrari	Dino	19.7	6	1	5	6
Maserati	Bora	15.0	8	1	5	8
Volvo	142E	21.4	4	1	4	2

3.8.2. Enlazar por columnas

Esta operación permite enlazar dos *data frames* por columnas:

bind_cols

```
Sintaxis:
bind_cols(...)
```

Esta operación no dará problemas mientras las filas de cada *data frame* coincidan.

```
cars1 %>% bind_cols(cars2)
## Atención al error que obtenemos
Error in cbind_all(x) : Argument 2 must be length 5, not 3
```

Debemos asegurar que tienen el mismo número de filas. Una forma de garantizarlo es indicar que del primer fichero únicamente nos quedamos con las tres primeras observaciones:

```
cars1 %>% slice(1:3) %>% bind_cols(cars2)
```

brand	modNam	mpg	cyl	am	gear	carb	brand1	modNam1	mpg1	cyl1	am1	gear1	carb1
Mazda	RX4	21.0	6	1	4	4	Ferrari	Dino	19.7	6	1	5	6
Mazda	RX4 Wag	21.0	6	1	4	4	Maserati	Bora	15.0	8	1	5	8
Datsun	710	22.8	4	1	4	1	Volvo	142E	21.4	4	1	4	2

3.8.3. Unión, intersección y diferencia

La función de unión une la filas que aparecen en los dos *data frames*; la de intersección, únicamente las que aparecen en los dos *data sets*, y la diferencia, las que aparecen en uno pero no en otro.

La sintaxis de las tres operaciones es muy simple:

union

intersect

setdiff

```
Sintaxis:
union(...)

intersect(...)

setdiff(...)
```

Si realizamos un ejemplo con los datos anteriores relativos a los enlaces por columna y fila, obtenemos los siguientes resultados después de ejecutar estos tres ejemplos:

```
cars1 %>% union(cars2)

cars1 %>% intersect(cars2)

cars1 %>% setdiff(cars2)
```

Resultado de la unión de los dos *data frames*:

brand	modelName	mpg	cyl	am	gear	carb
Mazda	RX4 Wag	21.0	6	1	4	4
Mazda	RX4	21.0	6	1	4	4
Datsun	710	22.8	4	1	4	1
Ferrari	Dino	19.7	6	1	5	6
Hornet	Sportabout	18.7	8	0	3	2
Volvo	142E	21.4	4	1	4	2
Hornet	4 Drive	21.4	6	0	3	1
Maserati	Bora	15.0	8	1	5	8

El caso de la intersección es el conjunto vacío, ya que no hay ninguna fila común entre los dos ficheros:

brand	modelName	mpg	cyl	am	gear	carb

El caso de la diferencia de conjuntos da el juego de datos correspondiente a cars1 (son conjuntos disjuntos):

brand	modelName	mpg	cyl	am	gear	carb
Mazda	RX4	21.0	6	1	4	4
Mazda	RX4 Wag	21.0	6	1	4	4
Datsun	710	22.8	4	1	4	1
Hornet	4 Drive	21.4	6	0	3	1
Hornet	Sportabout	18.7	8	0	3	2

3.8.4. Joins

En esta descripción de operaciones que involucran dos tablas suponemos datos relativos a vehículos (de mtcars):

brand	mpg	cyl
Mazda	21.0	6
Datsun	22.8	4
Hornet	18.7	8
Maserati	15.0	8
Volvo	21.4	4

brand	am	gear	carb
Mazda	1	4	4
Hornet	0	3	2
Ferrari	1	5	6
Volvo	1	4	2

3.8.5. Inner Join

Esta operación coloca lo que está en ambas tablas (coincidencia en la clave):

```
inner_join(x, y, by = "brand")
```

El resultado es el siguiente:

brand	mpg	cyl	am	gear	carb
Mazda	21.0	6	1	4	4
Hornet	18.7	8	0	3	2
Volvo	21.4	4	1	4	2

3.8.6. Left Join

```
left_join(x, y, by = "brand")
```

El resultado es el siguiente:

brand	mpg	cyl	am	gear	carb
Mazda	21.0	6	1	4	4
Hornet	18.7	8	0	3	2
Volvo	21.4	4	1	4	2

3.8.7. Right Join

```
right_join(x, y, by = "brand")
```

El resultado es el siguiente:

brand	mpg	cyl	am	gear	carb
Mazda	21.0	6	1	4	4
Hornet	18.7	8	0	3	2
Ferrari	NA	NA	1	5	6
Volvo	21.4	4	1	4	2

3.8.8. Full Joins

```
# Give me everything!
full_join(x, y, by = "brand")
```

El resultado es el siguiente:

brand	mpg	cyl	am	gear	carb
Mazda	21.0	6	1	4	4
Datsun	22.8	4	NA	NA	NA
Hornet	18.7	8	0	3	2
Maserati	15.0	8	NA	NA	NA
Volvo	21.4	4	1	4	2
Ferrari	NA	NA	1	5	6

3.8.9. Otras operaciones de Join

```
# Give me the stuff in X that is also in Y
semi_join(x, y, by = "brand")
```

El resultado corresponde a:

brand	mpg	cyl
Mazda	21.0	6
Hornet	18.7	8
Volvo	21.4	4

```
# Give me the stuff in X that is not in Y
anti_join(x, y, by = "brand")
```

Que nos da el siguiente resultado:

brand	mpg	cyl
Datsun	22.8	4
Maserati	15.0	8

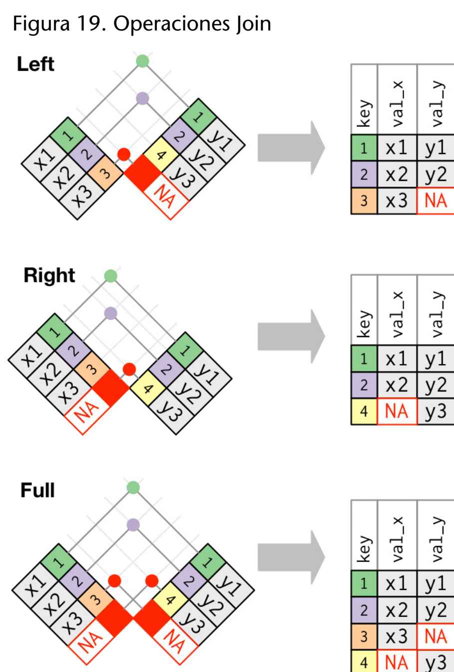
```
# Want everything that doesn't match?
full_join(anti_join(x,y, by = "brand"), anti_join(y,x, by = "brand"), by= "brand")
```

Con el siguiente resultado:

brand	mpg	cyl	am	gear	carb
Datsun	22.8	4	NA	NA	NA
Maserati	15.0	8	NA	NA	NA
Ferrari	NA	NA	1	5	6

Estas operaciones se pueden visualizar con este gráfico:

Figura 19. Operaciones Join



Fuente: https://r4ds.had.co.nz/relational-data.html

Figura 20. Dplyr Join

a

x1	x2
A	1
B	2
C	3

b

x1	x3
A	T
B	F
D	T

Mutating Joins

x1	x2	x3
A	1	T
B	2	F
C	3	NA

dplyr::**left_join(a, b, by = "x1")**
Join matching rows from b to a.

x1	x3	x2
A	T	1
B	F	2
D	T	NA

dplyr::**right_join(a, b, by = "x1")**
Join matching rows from a to b.

x1	x2	x3
A	1	T
B	2	F

dplyr::**inner_join(a, b, by = "x1")**
Join data. Retain only rows in both sets.

x1	x2	x3
A	1	T
B	2	F
C	3	NA
D	NA	T

dplyr::**full_join(a, b, by = "x1")**
Join data. Retain all values, all rows.

Fuente: https://bit.ly/1LaYWBd

4. Identificación de *outliers* (anomalías)

4.1. Introducción

El primer punto que conviene definir antes de empezar a tratar el tema de los *outliers* es qué entendemos por *outlier* (o valor atípico).

> Una posible definición suficientemente general establece que un *outlier* es una observación que difiere sustancialmente del resto de las observaciones y que induce dudas razonables sobre si procede de otra población.

Esta definición tiene como punto de partida el análisis de una única variable. ¿Qué sucede si tenemos más de una variable?

Otra posible definición que fundamentalmente proviene de un contexto estadístico sería la siguiente:

> El mecanismo subyacente generador de los datos es puesto en cuestión por alguna observación (univariante o multivariante) atípica/*outlier* que parece proceder de otro proceso generador y, por tanto, suscita dudas.

Siempre hemos de presuponer que cualquier variable observada puede contener *outliers*. No únicamente esto, sino que dos variables tratadas independientemente pueden indicar que no hay error, pero al tratarlas de forma conjunta sí que pueden indicar observación atípica.

4.2. ¿Por qué es importante la detección de *outliers*?

La detección de *outliers* es en muchas ocasiones el primer paso que se debe realizar antes de aplicar una técnica o método determinado. Si este paso no se realiza, podemos obtener unos resultados no coherentes ni realistas.

A continuación indicamos una lista de aplicaciones donde la detección de *outliers* es básica:

- **Detección de fraude**: El comportamiento de compra de una tarjeta de crédito cambia de forma brusca porque ha sido sustraída. La identificación de patrones anormales/atípicos de compra puede ser sinónimo de fraude.

- **Control de epidemias**: Al monitorizar una determinada enfermedad y su incidencia, puede ser vital detectar situaciones anómalas que pueden descubrirse por la identificación de *outliers*. Obviamente, la naturaleza de estos datos es multidimensional y se pueden tener en cuenta aspectos relativos a la zona geográfica, temporalidad (fecha) o frecuencia.

- ***Scouting***: En el seguimiento de deportistas la identificación de un *outlier* (por la parte superior) es vital para que se pueda identificar un rendimiento excepcional (fuera del comportamiento promedio visualizado).

- **IoT**: En contextos donde hay gran cantidad de sensores que hacen fluir cantidad de datos es vital comprender si los datos que el sensor genera contienen medidas de error. La identificación de valores extremos puede ser una pista para corregir o evitar estas anomalías.

- **Control de calidad**: En los procesos de control de calidad la identificación de errores queda definida por los estándares de calidad que el fabricante quiere imponer a su proceso de fabricación.

4.3. Metodos tradicionales

Los *outliers* o valores atípicos pueden afectar a las predicciones y distorsionarlas y, en consecuencia, modificar significativamente la precisión o el rendimiento del modelo que tengamos en consideración. Esta problemática es general pero especialmente delicada en los modelos de regresión (o en su generalización con los modelos GLM).

La detección, manipulación y/o transformación de *outliers* o valores extremos en nuestros datos no es un procedimiento estándar y debe ser realizado con precaución. Dependerá de los objetivos de nuestro estudio y evidentemente de la naturaleza de los *outliers*.

Existen diferentes métodos para la detección automática de *outliers* que aquí citamos:

- A partir de un número de desviaciones estándar (más o menos tres veces la desviación).
- A partir del *boxplot* (Tukey's method) y que en R se puede aplicar de forma directa.
- Valor desconocido o inexistente (R los tipifica como *NA*).

Desde una perspectiva académica los métodos se agrupan en modelos estadísticos, modelos basados en distancias o modelos basados en situaciones multidimensionales.

4.3.1. Métodos avanzados

Los métodos tradicionales no tienen en cuenta la realidad multidimensional de los datos que actualmente tratamos.

Nos encontramos con tres escenarios posibles y que se basan en técnicas de *clustering* y/o en técnicas de clasificación, pero en todos ellos se aborda esta realidad de una forma multidimensional.

- **Caso supervisado**: en ocasiones se dispone de un conjunto de datos que sabemos que es correcto y de otro conjunto de datos que es incorrecto. Esto se puede generalizar a más grupos de datos correctos e incorrectos. Por lo general, esta situación no es deseable al ser problemas muy poco balanceados. En función del entorno podemos tener una proporción de errores de 1 a 1 millón. En otras, esta proporción puede ser todavía peor o un poco mejor.

- **Caso semisupervisado**: en los escenarios semisupervisados suele estar disponible la información de una de las clases, por ejemplo, se tiene información de lo que son datos correctos, pero no de los incorrectos. También se puede dar la situación al revés.

- **Caso no supervisado**: es la situación más compleja en el sentido de que no disponemos de datos para decir que es correcto o incorrecto.

En la mayor parte de las situaciones los métodos de *clustering* están pensados para identificar grupos, no *outliers*.

Otras aproximaciones se basan en un modelo preestablecido. Por ejemplo, pensar que los datos provienen de una distribución normal permite identificar de una manera más fácil la identificación de *outliers*. En la figura 21 podemos apreciar dos modelos con comportamientos en los extremos completamente distintos. En la parte izquierda un modelo normal (o gausiano) y en la izquierda un modelo log normal (observad la asimétrica tipo derecha).

Habrá errores que únicamente tienen sentido desde el método de estudio que se analiza. En una regresión simple valores no identificados como errores pueden cambiar radicalmente el modelo y hacerlo absolutamente inválido para el propósito perseguido. Un ejemplo de esta situación lo encontramos en el caso de la relación entre mpg y hp (consumo y potencia). Si artificialmente añadimos observaciones erróneas (en el ejemplo, cinco) como se muestra en la figura 22, se observa cómo el modelo es radicalmente distinto al inicial.

Figura 21. Formas de dos modelos de distribución parecidos pero con un comportamiento distinto en los extremos. La distribución lognormal es asimétrica.

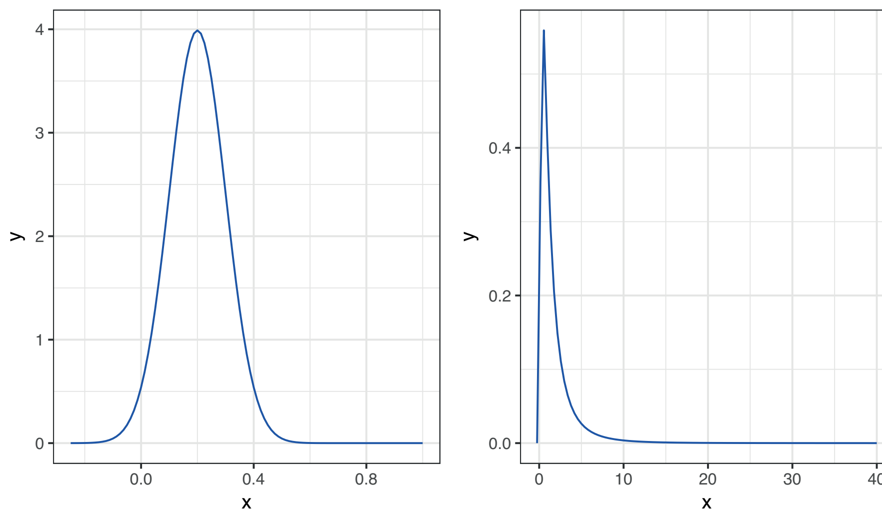
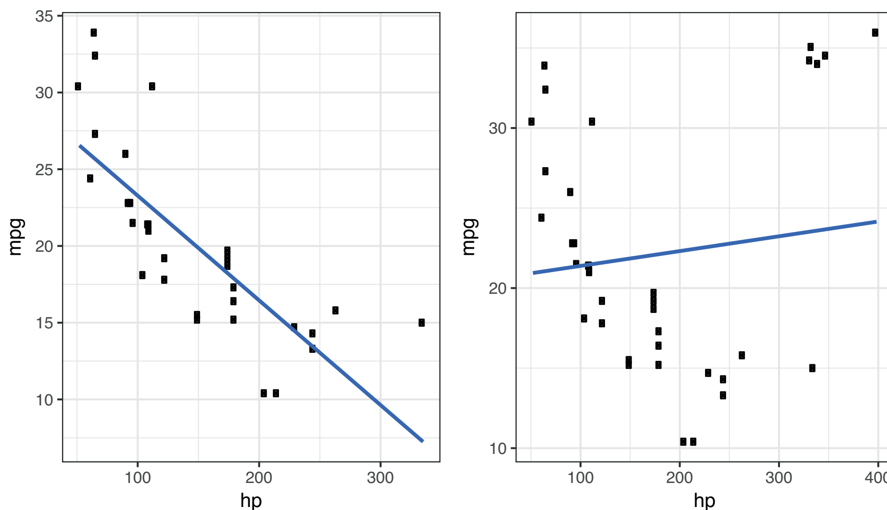


Figura 22. Plot de las variable mpg (consumo) en función de hp (la potencia del vehículo) y dos modelos de regresión lineal simple estimados. Izquierda con datos originales; derecha con el error añadido.



Podríamos también añadir que el cambio en el modelo tampoco mejora la situación. Es decir, si identificamos una relación no lineal, aunque el ajuste parece bueno, la similitud entre las dos soluciones es muy distinta. Los modelos visualizados en la figura 23 advierten del efecto de no tratar los valores atípicos.

Es evidente que hay una serie de observaciones que tienen influencia en el modelo. Si calculamos la distancia de Cook y la representamos por cada observación, podremos apreciar el efecto de los valores que más efecto tienen sobre el modelo y que corresponden a los añadidos sobre el modelo (ver la figura 24).

Figura 23. Plot de las variable mpg (consumo) en función de hp (la potencia del vehículo) y dos modelos de regresión con ajuste polinómico de grado 3. Izquierda con datos originales; derecha con el error añadido.

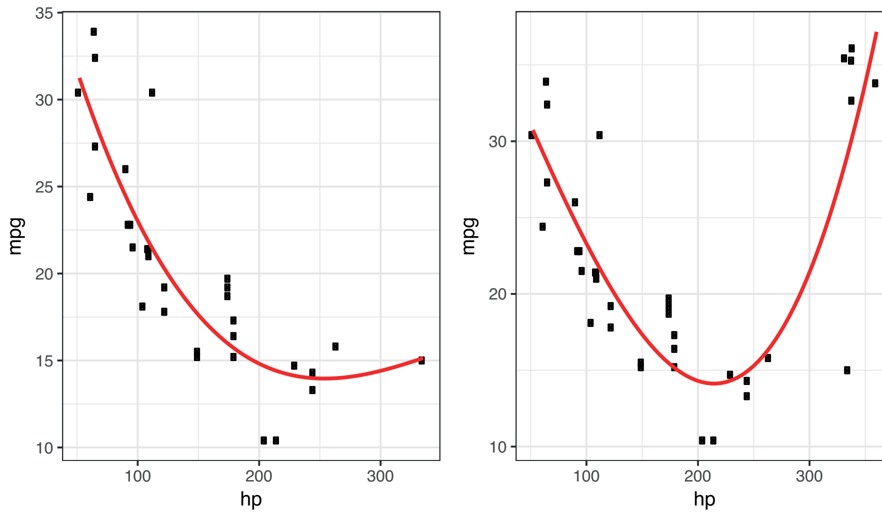
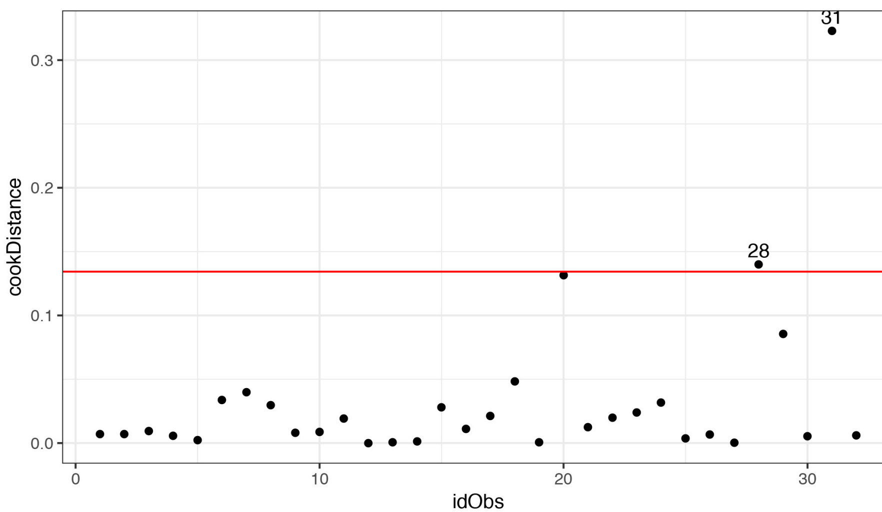


Figura 24. Distancia de Cook e identificación de las observaciones causantes de posibles problemas en el modelo anterior. Con los datos originales también se identifica una observación con etiqueta de influyente. Para el caso con los errores se identifican cuatro de ellos.



La distancia de Cook para la observación i se calcula con la expresión:

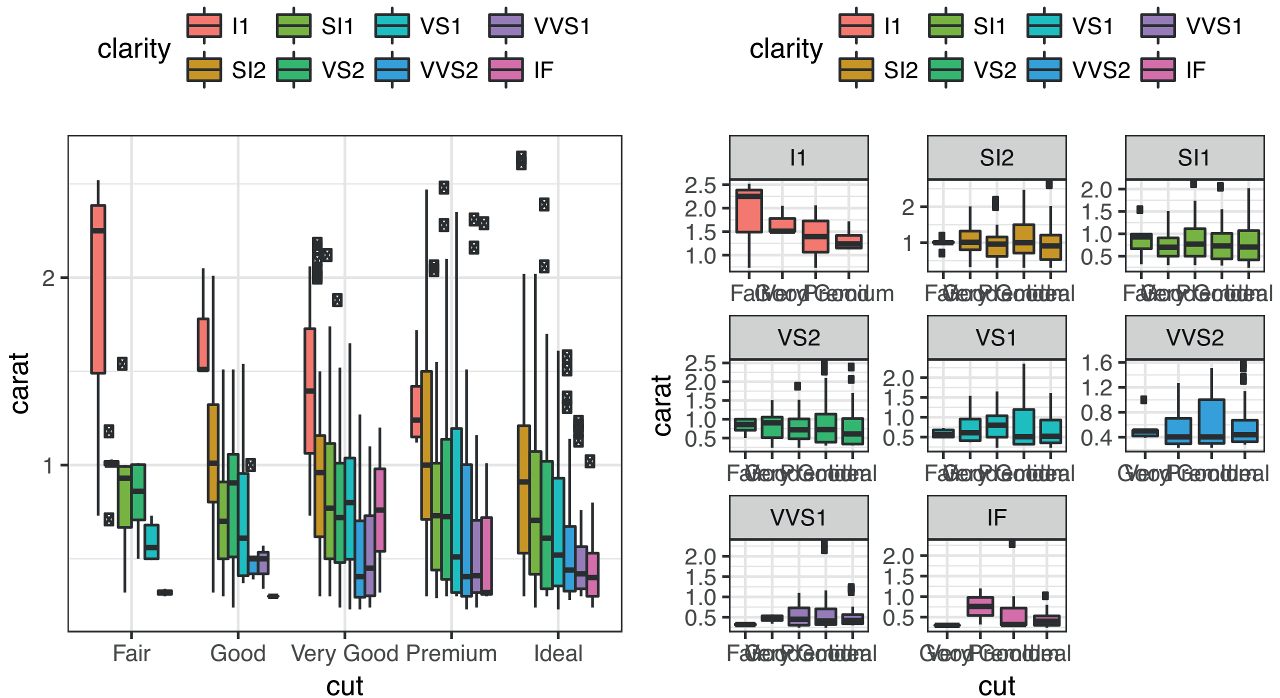
$$D_i = \frac{\sum_{j=1}^{n} \left(\hat{Y}_j - \hat{Y}_{j(i)}\right)^2}{p \times MSE} \quad (1)$$

donde p corresponde al número total de parámetros del modelo. En una regresión lineal el valor será $p = 2$.

En cualquier caso, el uso de herramientas de visualización permite clarificar y observar de una forma directa si existen o pueden existir valores atípicos. En la figura 25 el uso del *boxplot* junto con otras dos variables de tipo categórico ayuda a una rápida identificación de los posibles valores extremos. Será

el responsable o el experto quien a partir de su conocimiento juzgará como razonables estos valores (por el motivo que sea) o intentará tratarlos.

Figura 25. Utilización del diagrama de *boxplot* para visualizar posibles valores atípicos



Concluimos este subapartado destacando que el tema de la identificación de *outliers* es un tema de interés y que en función de nuestros conocimientos y/o habilidades del entorno de análisis disponible optaremos por una solución u otra.

4.4. *Missing values*

En ocasiones nos encontraremos con valores que no tenemos. R los representa con la indicación NA, que se puede interpretar como no disponible (*not available*). ¿Cuál es el problema de estos *missing values*? Que cualquier operación que implique este valor dará como resultado otro NA.

La detección de valores *missing* en R es muy fácil con el uso de la función:

```
is.na(objeto)
```

A partir de esta detección, está en las manos del investigador decidir o bien si elimina la fila involucrada o bien si tiene opciones de recuperar este valor (por vía directa o realizando una inferencia o predicción).

Obviamente, la solución que se deba aplicar debe tener en cuenta la dimensión de los datos. Observad que si tenemos cien millones de observaciones y un centenar de NA que suponemos distribuidos, normalmente se pueden eliminar sin problemas. Esto no afectará al problema al representar un porcentaje muy pequeño. Si por el contrario nuestro conjunto de datos es muy reducido y con pocas observaciones, sí que debemos considerar asignar valores al conjunto de las observaciones con NA. Esta solución puede permitir aplicar correctamente la técnica de ML que tengamos en mente.

Vamos a aplicar estos conceptos con nuestro fichero mtcars. ¿Cómo podríamos resolver las siguientes preguntas?

- ¿Cuántos NA tenemos?
- ¿Cuántas observaciones están fuera de 3 veces la desviación estándar?
- ¿Cuántas observaciones están fuera de 1.5 × IQR?

Estas operaciones son muy fáciles si consideramos que nuestro *target* corresponde a la variable kml.

Enlace de interés

Para complementar la información podéis consultar el siguiente enlace: http://uc-r.github.io/missing_values

Utilizando la notación de *pipes*, las tres cuestiones se resuelven:

```
mtcars %>% filter(is.na(kml))

mtcars %>% filter(kml<(mean(kml)-3*sd(kml))|(kml>mean(kml)+3*sd(kml)))

mtcars %>% filter(kml < quantile(kml, 0.25) - 1.5*IQR(kml) |
 kml > quantile(kml, 0.75) + 1.5*IQR(kml) )
```

mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb	pWrat	kml
33.9	4	71.1	65	4.22	1.835	19.9	1	1	4	1	35.42234	14.41238

Resumen

Debemos resaltar que en la mayoría de los proyectos donde existe un gran volumen de datos, la mayor parte del tiempo se dedicará a cuestiones relativas a la preparación y limpieza de los datos. Algunas consultoras cifran el porcentaje de tiempo alrededor del 60 %, con probabilidades de encontrarnos fácilmente con escenarios donde el esfuerzo entregado a la correcta manipulación de los datos sea de hasta el 80 %. Estos datos no quieren sino resaltar la importancia de la etapa de preparación.

Si bien no es fácil elegir una herramienta para la realización de este tipo de operaciones, es recomendable elegirla de acuerdo con nuestros intereses y preferencias. De hecho, hay diferentes fabricantes de software que ofrecen herramientas semiautomatizadas para la preparación y eliminación de errores. Incluso si optamos por entornos de R o Python, veremos que existen diferentes librerías para este tipo de operaciones. La opción presentada en este módulo es la que ofrece una coherencia y simplicidad al tener comandos que se asocian a una acción determinada y clara. Para buscar alternativas:

- Acceso con la sintaxis de R.
- Acceso con la librería de data.table.

Si se pretende primar el rendimiento, la alternativa data.table es realmente eficiente. No obstante, la librería dplyr cuenta con mejoras para optimizar la eficiencia y hacer uso de paralelismo. Si disponemos de máquina con diferentes *cores* (un i7 dispone de ocho *cores*) y además disponemos de una tarjeta gráfica (Nvidia GTX 1080 ofrece más de dos mil *cores*), es una buena alternativa.

La gracia de estas alternativas es que no necesitamos intermediarios para "hablar" con los datos. Si alguien dispone de conocimientos de SQL, existe una función que traduce automáticamente las operaciones realizadas a la sintaxis de SQL usando la misma sintaxis presentada:

```
con %>%
  tbl("mtcars") %>%
  filter(cyl > 2) %>%
  select(mpg:hp) %>%
  head(10) %>%
  show_query()
```

En realidad, se traslada a la base de datos como:

```
<SQL>
SELECT `mpg`, `cyl`, `disp`, `hp`
FROM `mtcars`
WHERE (`cyl` > 2.0)
LIMIT 10
```

En resumen, es imprescindible poder obtener un formato adecuado de los datos que queremos analizar y también es imprescindible la capacidad para identificar y corregir errores en los datos suministrados.

Algunas referencias de interés:

https://www.jstatsoft.org/article/view/v059i10
https://www.listendata.com/2016/08/dplyr-tutorial.html