

Análisis de Datos

Tema 3 - Data Wrangling

3.1 Importación de Datos

Roi Naveiro

Data Wrangling

Objetivo: dejar los datos listos para su posterior exploración y modelización

Convertir datos crudos en datos procesados

Datos crudos

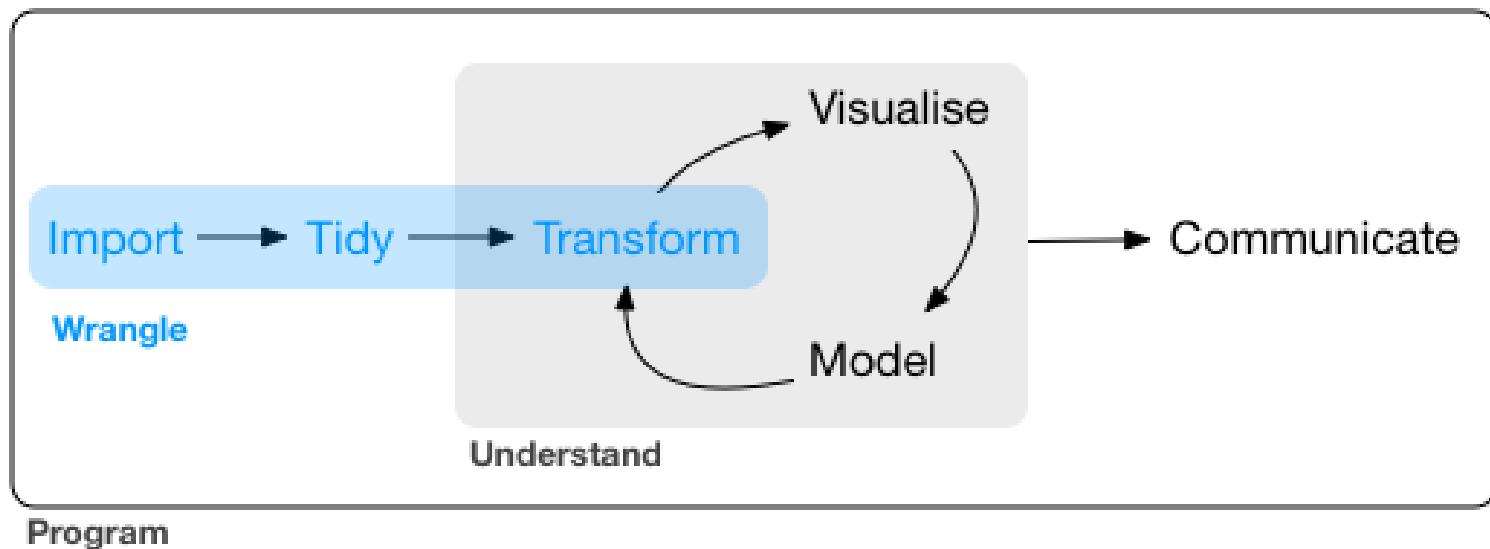
- Los datos tal cual aparecen en la fuente de origen
- No han sufrido ninguna manipulación

Datos procesados

- Cada variable es una columna
- Cada observación una fila
- Cada unidad observacional es una celda
- Datos más complejos, en varias tablas interconectadas

Data Wrangling

- Importación de los datos
- Organización de los datos
- Transformación de los datos



Importación de datos

Importación de datos

- Aprederemos cómo descargar y leer datos en R
- Nos centraremos en **datos rectangulares**
- De nuevo, para esta parte, necesitaremos cargar **tidyverse**

```
library(tidyverse)
```

Seguiremos este esquema

- Descarga de datos
- Lectura de ficheros en R-base
- Lectura de ficheros tidyverse

Importación de datos: Descarga de datos

Directorio de trabajo

- Es esencial conocer el directorio de trabajo
- Los comandos esenciales son **getwd()** y **setwd()**

```
# Devuelve el directorio de trabajo  
getwd()
```

```
# Cambia el directorio de trabajo  
setwd()  
  
# Directorio relativo al directorio raíz  
setwd("./data")  
  
# Directorio relativo al directorio superior al raíz  
setwd("../data")  
  
# Directorio absoluto  
setwd("/home/roi/data")
```

Descarga de datos

Vamos a ver cómo crear una carpeta **data** desde R (si esta no existe) y descargar unos datos dentro de la misma

1. Comprobar si existe una carpeta llamada **data** en el directorio actual.
Si no, crearla. Usar funciones **file.exists()** y **dir.create()**.
Inténtalo

Descarga de datos

Vamos a ver cómo crear una carpeta **data** desde R (si esta no existe) y descargar unos datos dentro de la misma

1. Descarga los datos sobre los indicadores de calidad de vino tinto
[https://archive.ics.uci.edu/ml/datasets/Wine+Quality.](https://archive.ics.uci.edu/ml/datasets/Wine+Quality)

```
data_url <- "https://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality.csv"
download.file(data_url, destfile = "data/red_qualities.csv")
list.files("./data")
```

Importación de datos: Lectura de ficheros en R-base

read.table()

```
# Así leemos datos con read.table
# OJO: el separador es ;
red_wine_data <- read.table("data/red_qualities.csv", header = T, sep = ';')
head(red_wine_data)

##   fixed.acidity volatile.acidity citric.acid residual.sugar chlorides
## 1          7.4            0.70      0.00           1.9     0.076
## 2          7.8            0.88      0.00           2.6     0.098
## 3          7.8            0.76      0.04           2.3     0.092
## 4         11.2            0.28      0.56           1.9     0.075
## 5          7.4            0.70      0.00           1.9     0.076
## 6          7.4            0.66      0.00           1.8     0.075
##   free.sulfur.dioxide total.sulfur.dioxide density    pH sulphates alcohol
## 1                 11             0.9978 3.51      0.56     9.4
## 2                 25             0.9968 3.20      0.68     9.8
## 3                 15             0.9970 3.26      0.65     9.8
## 4                 17             0.9980 3.16      0.58     9.8
## 5                 11             0.9978 3.51      0.56     9.4
## 6                 13             0.9978 3.51      0.56     9.4
##   quality
## 1      5
## 2      5
## 3      5
## 4      6
## 5      5
## 6      5
```

read.table()

Algunos parámetros importantes

- *na.strings* - elegir el carácter que representa un valor ausente
- *nrows* - cuántas filas leer (e.g. *nrows=10* lee 10 filas)
- *skip* - número de filas a ignorar hasta empezar lectura

Leer excell

Esto require la librería **xlsx**

```
library(xlsx)
red_wine_data <- read.xlsx("data/red_qualities.xlsx",
                           sheetIndex = 1, header = T)
head(red_wine_data)
```

```
##   NA. fixed.acidity volatile.acidity citric.acid residual.sugar chlorides
## 1   1          7.4            0.70      0.00        1.9       0.076
## 2   2          7.8            0.88      0.00        2.6       0.098
## 3   3          7.8            0.76      0.04        2.3       0.092
## 4   4         11.2            0.28      0.56        1.9       0.075
## 5   5          7.4            0.70      0.00        1.9       0.076
## 6   6          7.4            0.66      0.00        1.8       0.075
##   free.sulfur.dioxide total.sulfur.dioxide density    pH sulphates alcohol
## 1                  11             34  0.9978 3.51      0.56      9.4
## 2                  25             67  0.9968 3.20      0.68      9.8
## 3                  15             54  0.9970 3.26      0.65      9.8
## 4                  17             60  0.9980 3.16      0.58      9.8
## 5                  11             34  0.9978 3.51      0.56      9.4
## 6                  13             40  0.9978 3.51      0.56      9.4
##   quality
## 1      5
## 2      5
## 3      5
## 4      6
## 5      5
```

Leer excell

Se puede hacer *subsetting* al leer

```
small <- read.xlsx("data/red_qualities.xlsx", sheetIndex = 1,  
                     colIndex = 2:3, rowIndex = 1:3)  
head(small)
```

```
##   fixed.acidity volatile.acidity  
## 1          7.4           0.70  
## 2          7.8           0.88
```

Leer JSON

- Javascript Object Notation
- Almacenamiento ligero de datos
- Es un formato común en APIs
- Los datos se guardan objetos con pares atributo-valor y matrices

Leer JSON

```
{  
    "firstName": "John",  
    "lastName": "Smith",  
    "isAlive": true,  
    "age": 27,  
    "address": {  
        "streetAddress": "21 2nd Street",  
        "city": "New York",  
        "state": "NY",  
        "postalCode": "10021-3100"  
    },  
    "phoneNumbers": [  
        {  
            "type": "home",  
            "number": "212 555-1234"  
        },  
        {  
            "type": "office",  
            "number": "646 555-4567"  
        }  
    ],  
    "children": [  
        "Catherine",  
        "Thomas",  
        "Trevor"  
    ],  
    "spouse": null  
}
```

Leer JSON

```
library(jsonlite)

# Leer JSON
iris_JSON <- read_json("data/iris.json", simplifyVector = TRUE)

# Convertir en dataframe
iris_df <- fromJSON(iris_JSON)

head(iris_df)
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
## 1	5.1	3.5	1.4	0.2	setosa
## 2	4.9	3.0	1.4	0.2	setosa
## 3	4.7	3.2	1.3	0.2	setosa
## 4	4.6	3.1	1.5	0.2	setosa
## 5	5.0	3.6	1.4	0.2	setosa
## 6	5.4	3.9	1.7	0.4	setosa

Leer JSON

```
{  
    "firstName": "John",  
    "lastName": "Smith",  
    "isAlive": true,  
    "age": 27,  
    "address": {  
        "streetAddress": "21 2nd Street",  
        "city": "New York",  
        "state": "NY",  
        "postalCode": "10021-3100"  
    },  
    "phoneNumbers": [  
        {  
            "type": "home",  
            "number": "212 555-1234"  
        },  
        {  
            "type": "office",  
            "number": "646 555-4567"  
        }  
    ],  
    "children": [  
        "Catherine",  
        "Thomas",  
        "Trevor"  
    ],  
    "spouse": null  
}
```

Leer JSON

```
library(jsonlite)

# La función prettyf nos imprime el JSON de forma limpia
prettyf(iris_JSON)
```

```
## [
##   {
##     "Sepal.Length": 5.1,
##     "Sepal.Width": 3.5,
##     "Petal.Length": 1.4,
##     "Petal.Width": 0.2,
##     "Species": "setosa"
##   },
##   {
##     "Sepal.Length": 4.9,
##     "Sepal.Width": 3,
##     "Petal.Length": 1.4,
##     "Petal.Width": 0.2,
##     "Species": "setosa"
##   },
##   {
##     "Sepal.Length": 4.7,
##     "Sepal.Width": 3.2,
##     "Petal.Length": 1.3,
##     "Petal.Width": 0.2,
##     "Species": "setosa"
##   },
##   {
##     "Sepal.Length": 4.6,
```

Importación de datos: Lectura de ficheros con paquetes de tidyverse

El paquete **readr**

Ahora veremos cómo leer datos con el paquete **readr** que es parte de **tidyverse**

```
library(tidyverse)
```

Las funciones de **readr** se encargan de transformar ficheros planos en *tibbles*

readr

- **read_csv()** - ficheros delimitados por ,
- **read_csv2()** - ficheros delimitados por :
- **read_tsv()** - ficheros delimitados por tabulador
- **read_delim()** - ficheros con cualquier delimitación
- ...

readxl

- **read_excel()** - ficheros xls o xlsx
- ...

Lectura de datos

```
red_wine_data <- read_csv2("data/red_qualities.csv")  
red_wine_data
```

```
## # A tibble: 1,599 × 12  
##   `fixed acidity` `volatile acidity` `citric acid` `residual sugar` chlorides  
##   <dbl> <chr>           <chr>           <dbl> <chr>  
## 1 74 0.7             0               19 0.076  
## 2 78 0.88            0               26 0.098  
## 3 78 0.76            0.04            23 0.092  
## 4 112 0.28           0.56            19 0.075  
## 5 74 0.7             0               19 0.076  
## 6 74 0.66            0               18 0.075  
## 7 79 0.6              0.06            16 0.069  
## 8 73 0.65            0               12 0.065  
## 9 78 0.58            0.02            2 0.073  
## 10 75 0.5             0.36            61 0.071  
## # ... with 1,589 more rows, and 7 more variables: `free sulfur dioxide` <dbl>,  
## #   `total sulfur dioxide` <dbl>, density <chr>, pH <dbl>, sulphates <chr>,  
## #   alcohol <dbl>, quality <dbl>
```

Lectura de datos

También podemos pasarle un csv escrito a mano

```
df <- read_csv("a,b,c  
               1,2,3  
               4,5,6")  
df
```

```
## # A tibble: 2 × 3  
##       a     b     c  
##   <dbl> <dbl> <dbl>  
## 1     1     2     3  
## 2     4     5     6
```

Lectura de datos

También podemos pasarle un csv escrito a mano

```
# Así prevenimos la lectura de una línea
df <- read_csv("Línea de metadatos
                a,b,c
                1,2,3
                4,5,6", skip=1)
df
```

```
## # A tibble: 2 × 3
##       a     b     c
##   <dbl> <dbl> <dbl>
## 1     1     2     3
## 2     4     5     6
```

Lectura de datos

También podemos pasarle un csv escrito a mano

```
# Así prevenimos la lectura de comentarios
df <- read_csv("# Línea de metadatos
                a,b,c
                # Un comentario
                1,2,3
                4,5,6", comment = '#')
df
```

```
## # A tibble: 3 × 3
##       a     b     c
##   <dbl> <dbl> <dbl>
## 1     NA     NA     NA
## 2     1     2     3
## 3     4     5     6
```

Lectura de datos

Si las columnas no tienen nombre

```
df <- read_csv("a,b,c
                1,2,3
                4,5,6", col_names=FALSE)
df
```

```
## # A tibble: 3 × 3
##   X1     X2     X3
##   <chr> <chr> <chr>
## 1 a      b      c
## 2 1      2      3
## 3 4      5      6
```

Lectura de datos

Si queremos nombrar las columnas

```
df <- read_csv("a,b,c
                1,2,3
                4,5,6", col_names=c("p", "q", "r"))
df
```

```
## # A tibble: 3 × 3
##   p     q     r
##   <chr> <chr> <chr>
## 1 a     b     c
## 2 1    2     3
## 3 4    5     6
```

Lectura de datos

Dar nombres en formato apropiado

```
df <- read_csv("Nombre Apellido, Es Fumador, EDAD_PACIENTE
                Juan Montero, SI, 35
                Maria Chacón, NO, 64")
df %>% janitor::clean_names()
```

```
## # A tibble: 2 × 3
##   nombre_apellido es_fumador edad_paciente
##   <chr>          <chr>        <dbl>
## 1 Juan Montero   SI            35
## 2 Maria Chacón   NO            64
```

Lectura de datos

Podemos indicar que caracteres se refieren a un valor ausente

```
df <- read_csv("a,b,c  
                1,na,3  
                4,5,na", na = "na")  
df
```

```
## # A tibble: 2 × 3  
##       a     b     c  
##   <dbl> <dbl> <dbl>  
## 1     1     NA     3  
## 2     4     5     NA
```

Importación de datos: Parsear vectores y ficheros

Parsear vectores

La función `read_csv` parsea un fichero de texto. Es decir, convierte un fichero de texto en una estructura de datos organizada.

Para entender mejor cómo hace esto debemos entender las funciones `parse*`()

Estas convierten un vector de caracteres en un vector de datos especializado

```
str(parse_logical(c("T", "F", "F")))
```

```
##  logi [1:3] TRUE FALSE FALSE
```

```
str(parse_integer(c("1", "4")))
```

```
##  int [1:2] 1 4
```

```
str(parse_date(c("2022-01-04")))
```

```
##  Date[1:1], format: "2022-01-04"
```

Parsear vectores

Como todas las funciones de tidyverse, las funciones **parse***() son uniformes: el primer argumento es el vector a parsear y **na** permite indicar que caracteres serán tratados como valores ausentes

```
parse_integer(c("1", "4", "."), na=".")
```

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

Parsear vectores

Si hay problemas, sale un mensaje warning

```
parse_integer(c("1", "4", "c"))

## Warning: 1 parsing failure.
## row col      expected actual
##   3 -- no trailing characters     c

## [1]  1  4 NA
## attr(,"problems")
## # A tibble: 1 × 4
##       row   col expected           actual
##     <int> <int> <chr>            <chr>
## 1     3     NA no trailing characters c
```

Tipos de parseadores

- **parse_logical()** y **parse_integer()** para lógicos y enteros. Los más sencillos, dada su uniformidad
- **parse_double()** para números y **parse_number()** (más flexible). Son más complejos
- **parse_factor()** para variables tipo factor (categóricas con valores fijos y conocidos)
- **parse_datetime()**, **parse_date()** y **parse_time()** para tiempos y fechas. Son los más complejos dada la variabilidad en la escritura de fechas y tiempos.

Tipos de parseadores - Números

Tres problemas

1. Los números se escriben de manera diferente en distintas partes del mundo (1.45 vs 1,45)
2. A veces los números vienen con símbolos (100%)
3. Caracteres de agrupamiento (1.000.000)

Tipos de parseadores - Números

1. Los números se escriben de manera diferente en distintas partes del mundo (1.45 vs 1,45)

```
parse_double("1,45", locale = locale(decimal_mark = ",",))
```

```
## [1] 1.45
```

Tipos de parseadores - Números

1. A veces los números vienen con símbolos (100%)

```
parse_number("100$")
```

```
## [1] 100
```

```
parse_number("30%")
```

```
## [1] 30
```

```
parse_number("Me costó 100 €")
```

```
## [1] 100
```

Tipos de parseadores - Números

1. Caracteres de agrupamiento (1.000.000)

```
parse_number("%100,330")
```

```
## [1] 100330
```

```
parse_number("100.330€$", locale = locale(grouping_mark = "."))
```

```
## [1] 100330
```

Tipos de parseadores - Strings

Los ordenadores representan el texto usando números hexadecimales

```
charToRaw("Hola")
```

```
## [1] 48 6f 6c 61
```

El mapeo entre hexadecimales y caracteres se denomina **encoding** **readr** utiliza el encoding UTF-8, bastante general (muy bueno por defecto)

A veces falla con datos producidos por sistemas viejos

```
x1 <- "Nos vemos ma\xflana"  
parse_character(x1, locale = locale(encoding = "ISO-8859-1"))
```

```
## [1] "Nos vemos mañana"
```

Tipos de parseadores - Strings

Podemos determinar el encoding usando

```
x1 <- "Nos vemos ma\xflana"  
guess_encoding(charToRaw(x1))
```

```
## # A tibble: 3 × 2  
##   encoding    confidence  
##   <chr>          <dbl>  
## 1 ISO-8859-1      0.52  
## 2 ISO-8859-2      0.35  
## 3 ISO-8859-9      0.35
```

Tipos de parseadores - Factores

Hay que dar a **parse_factor** un vector de posibles niveles

```
parse_factor(c("hombre", "mujer", "mujer", "unicornio"), levels=c("hombre", "  
  
## [1] hombre mujer mujer <NA>  
## attr(,"problems")  
## # A tibble: 1 × 4  
##       row   col expected           actual  
##     <int> <int> <chr>           <chr>  
## 1     4     NA value in level set unicornio  
## Levels: hombre mujer
```

Tipos de parseadores - Fechas y horas

parse_datetime() para fecha con hora. Espera, en este orden, año, mes, dia, hora, minuto y segundo

```
parse_datetime("2022-01-01T1015")
```

```
## [1] "2022-01-01 10:15:00 UTC"
```

parse_date() para fecha. Espera, en este orden, año, mes, dia, separados por - o /

```
parse_date("2022/01/01")
```

```
## [1] "2022-01-01"
```

Tipos de parseadores - Fechas y horas

`parse_time()` para hora. Espera, en este orden, hora, minuto y segundo separados por : y am pm opcionales

```
parse_time("10:15 am")
```

```
## 10:15:00
```

También se puede especificar el formato

```
parse_date("01/04/22", "%d/%m/%y")
```

```
## [1] "2022-04-01"
```

[Más información sobre los formatos](#)

Tipos de parseadores - Fechas y horas

- Año

: **%Y** (4 dígitos).

: **%y** (2 dígitos); 00-69 -> 2000-2069, 70-99 -> 1970-1999.

- Mes

: **%m** (2 dígitos).

: **%b** (nombre abreviado, como "Jan").

: **%B** (nombre completo, "January").

- Día

: **%d** (2 dígitos).

: **%e** (espacio opcional).

Tipos de parseadores - Fechas y horas

- Tiempo

: **%H** 0-23 horas

: **%I** 0-12, usar con **%p**.

: **%p** AM/PM

: **%M** minutos

: **%S** segundos, número entero.

: **%OS** segundos reales.

: **%Z** Franja horaria (e.g. **America/Chicago**).

: **%z** (como offset de UTC, e.g. **+0800**).

Parsear un fichero

Intentemos entender

1. Cómo **readr** acierta automáticamente el tipo de columna para parsearla
2. Cómo cambiar la especificación por defecto

Parsear un fichero

1. Cómo **readr** acierta automáticamente el tipo de columna para parsearla

Para esto, **readr** lee las primeras 1000 files e cada columna y usa una heurística para adivinar el tipo.

```
guess_parser("2022-01-01")  
  
## [1] "date"  
  
str(parse_guess("2022-01-01"))  
  
## Date[1:1], format: "2022-01-01"
```

Parsear un fichero

A veces esta estrategia falla

- Puede ser que las 1000 primeras filas sean casos especiales (1000 enteros en variable double)
- La columna puede contener valores ausentes

Es recomendable, cuando se pueda, especificar el tipo de columnas

Ejemplo

```
read_csv("data/df-na.csv")
```

```
## # A tibble: 9 × 3
##   x     y         z
##   <chr> <chr>    <chr>
## 1 1     a        hi
## 2 <NA>  b        hello
## 3 3     Not applicable 9999
## 4 4     d        ola
## 5 5     e        hola
## 6 .     f        whatup
## 7 7     g        wassup
## 8 8     h        sup
## 9 9     i        <NA>
```

Ejemplo

```
read_csv("data/df-na.csv", col_types = list(col_double(), col_character(),  
                                         col_character()))
```

```
## # A tibble: 9 × 3  
##       x     y         z  
##   <dbl> <chr>    <chr>  
## 1     1     a        hi  
## 2     NA    b        hello  
## 3     3 Not applicable 9999  
## 4     4     d        ola  
## 5     5     e        hola  
## 6     NA    f        whatup  
## 7     7     g        wassup  
## 8     8     h        sup  
## 9     9     i        <NA>
```

Importación de datos: escritura de datos

Escritura de datos

Tres funciones principales

- **write_csv()**: escribe en csv
- **write_tsv()**: escribe en tsv
- **write_excel_csv()**: escribe en excell

El uso

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

Escritura de datos

Con estas funciones se pierde información (por ejemplo el tipo de columnas).

Recomendable usar **write_rds()** y **read_rds()** (análogas a **readRDS()** y **writeRDS()** de R Base)

Estas guardan ficheros en formato binario RDS

Escritura de datos

```
write_rds(iris, "data/iris.RDS")
iris_tibble = read_rds("data/iris.RDS")
iris_tibble
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
## 1	5.1	3.5	1.4	0.2	setosa
## 2	4.9	3.0	1.4	0.2	setosa
## 3	4.7	3.2	1.3	0.2	setosa
## 4	4.6	3.1	1.5	0.2	setosa
## 5	5.0	3.6	1.4	0.2	setosa
## 6	5.4	3.9	1.7	0.4	setosa
## 7	4.6	3.4	1.4	0.3	setosa
## 8	5.0	3.4	1.5	0.2	setosa
## 9	4.4	2.9	1.4	0.2	setosa
## 10	4.9	3.1	1.5	0.1	setosa
## 11	5.4	3.7	1.5	0.2	setosa
## 12	4.8	3.4	1.6	0.2	setosa
## 13	4.8	3.0	1.4	0.1	setosa
## 14	4.3	3.0	1.1	0.1	setosa
## 15	5.8	4.0	1.2	0.2	setosa
## 16	5.7	4.4	1.5	0.4	setosa
## 17	5.4	3.9	1.3	0.4	setosa
## 18	5.1	3.5	1.4	0.3	setosa
## 19	5.7	3.8	1.7	0.3	setosa
## 20	5.1	3.8	1.5	0.3	setosa

Otros datos

Para el resto de tipos de datos que no hemos estudiado, existen paquetes de tidyverse

- **haven**: SPSS, Stata, SAS
- **readxl** para excel
- **DBI** junto con **RMySQL**, **RSQlite**, etc. para lanzar queries de SQL contra bases de datos y devolver un data frame

Bibliografía

Este tema está fundamentalmente basado en [R for Data Science](#), Wickham and Grolemund (2016)