Introduction to Applied Data Science

Lecture 3: Getting Data, API's and Databases

Bas Machielsen
Utrecht University
2024-04-22

Databases

Lecture 3: Getting Data, API's &

- Overview of this class:
  - Lecture 1: Introduction to Data Science & R
  - Lecture 2: Introduction to Programming
  - *This lecture*: Lecture 3: Getting Data, API & Databases
  - Lecture 4: Getting Data, Web Scraping
  - Lecture 5: Transforming and Cleaning Data
  - Lecture 6: Spatial and Network Data
  - Lecture 7: Text Data & Text Mining
  - Lecture 8: Data Science Project

Where To Get Data?

- Where does data come from?
- Whole **spectrum** of potential data sources, together with some examples
  - R **data packages**
  - Importing data found on **the web**
  - Extracting data from **API servers**
- Next lecture: focus on *one particular way* of data collection, web scraping

Where To Get Data?

- Haggerty (2023) employs the following characterization of datasets:
- **Pre-cleaned datasets** posted on secondary repositories
  - E.g. Harvard Dataverse, Kaggle, Replication Packages, GitHub.
- **Open data libraries**
  - E.g. my overview **here**, IMF, World Bank. R packages containing data.
- **Websites of primary data providers**
  - Government statistical agencies; some private companies and NGOs; scientific researchers.

Where To Get Data?

- Liberate **previously inaccessible data**
  - E.g. Government or private sector; Netherlands: WOB Request
- **Self-compiled data**
  - E.g. Create your own dataset from many disjoint sources; E.g. historical archives, websites, PDF reports.
- Collect your **own primary data**
  - E.g. from surveys or experiments, phones, registration devices, etc.

Starting Points

- Scientific Literature
- Links:
  - Google dataset search
  - Datasets for quantitative research
  - My own overview
- Websites of Governmental Agencies or Institutions
- Google!

Example: the `wbstats` package

Interacting with API's

- Usually, websites of Governmental Agencies or Institutions have a lot of versatile data available
- They usually provide a **User Interface** on their website to enable you to inspect, but also download, the required data
- Recently, however, many have found it useful to **extract and obtain data directly from programming environments** such as R
- That means that we send a request *directly from R* to search for, and acquire, data, without having to resort to a user interface
- In general, in this course, we want to be able to communicate with computers through code, rather than through clicks and user interfaces
    - The following provides an example of this

Example: World Bank Data

- `wbstats` is an R package for **searching and downloading** data from the World Bank API
- For now, you can think of this as a package extracting data from a large on-line database in a convenient way
- There also exist **similar packages** with access to the same data for other programming languages
- We should get used to accessing data by providing instructions in code instead of using a graphical user interface
- In R, you can install this package by `install.packages('wbstats')`
- Or by:

```
library(pacman)
p_load(wbstats)
```

Searching for Data

- A first thing you can do after installing a package is **inspecting its functions**
- In R, packages come with a document, sometimes called a "vignettes", explaining all of their functions
- You can find that through Google if you like: this documentation was found by Googlig "wbstats documentation"
- However, there are also various ways in R to help you navigate the package:
  - Firstly, you can go to the **console**, type the name of the package followed by two colons (::)
  - RStudio will show you a list of all functions contained in this package
- You can also check out the **documentation** by entering `?packagename` in the console
  - If you then click on Index, you will again see a list of all functions, which you can click on to read the documentation
- Finally, you can also access the documentation of a function directly by entering `?functionname` in the console (after loading the corresponding library)

Using the `wbstats` Package

- In the `wbstats` package, the first thing we might want to do is look for data
- This corresponds to what you would do on a website: you'll be looking for some data, deciding what to download
- In R, this is done through functions. In particular, we can look for data in various ways using the `wb_search` function.
    - There is also `wb_cachelist`, which is an overview of the available data:

```
library(wbstats)
overview ← wb_cachelist
```

- Note that this is a `list` containing various objects. Try clicking on it in your memory pane to see what objects it contains.

Inspecting Data

- For example, we can look at the countries contained in the World Bank Database:

```
overview$countries ▷ select(1:9) ▷ head(4)
```

```
## # A tibble: 4 × 9
##   iso3c iso2c country     capital_city longitude latitude region_iso3c region_iso2c
##   <chr> <chr> <chr>       <chr>            <dbl>    <dbl> <chr>        <chr>
## 1 ABW   AW    Aruba       Oranjestad       -70.0    12.5  LCN          ZJ
## 2 AFG   AF    Afghanistan Kabul             69.2    34.5  SAS          8S
## 3 AFR   A9    Africa      <NA>              NA      NA    <NA>         <NA>
## 4 AGO   AO    Angola      Luanda            13.2    -8.81 SSF          ZG
```

Other Useful Overviews

- A couple of other useful overviews are:

```
overview$topics ▷ head(7)
```

```
## # A tibble: 7 × 3
##   topic_id topic                          topic_desc
##      <dbl> <chr>                          <chr>
## 1        1 Agriculture & Rural Development "For the 70 percent of the world's poor w
## 2        2 Aid Effectiveness               "Aid effectiveness is the impact that aid
## 3        3 Economy & Growth                "Economic growth is central to economic d
## 4        4 Education                       "Education is one of the most powerful in
## 5        5 Energy & Mining                 "The world economy needs ever-increasing
## 6        6 Environment                     "Natural and man-made environmental resou
## 7        7 Financial Sector                "An economy's financial markets are criti
```

Example: World Bank Data

- Or:

```
overview$sources ▷ head(5)
```

```
## # A tibble: 5 × 9
##   source_id last_updated source                  source_code source_desc source_url
##       <dbl> <date>       <chr>                   <chr>       <lgl>       <lgl>
## 1         1 2019-10-23   Doing Business          DBS         NA          NA
## 2         2 2020-10-15   World Development Indic… WDI        NA          NA
## 3         3 2020-09-28   Worldwide Governance In… WGI        NA          NA
## 4         5 2016-03-21   Subnational Malnutritio… SNM        NA          NA
## 5         6 2020-10-16   International Debt Stat… IDS        NA          NA
```

- Usually, many similar packages contain such "overview" functions so you can search for the data you want.

Searching for Data of Interest

- You can search within any category you desire with `wb_search`
- Most obviously, with the name of a variable:

```
wb_search("GDP per capita") ▷ head(5)
```

```
## # A tibble: 5 × 3
##   indicator_id       indicator                                          indicator_d
##   <chr>              <chr>                                              <chr>
## 1 5.51.01.10.gdp     Per capita GDP growth                              GDP per cap
## 2 6.0.GDPpc_constant GDP per capita, PPP (constant 2011 international $) GDP per cap
## 3 NV.AGR.PCAP.KD.ZG  Real agricultural GDP per capita growth rate (%)    The growth
## 4 NY.GDP.PCAP.CD     GDP per capita (current US$)                       GDP per cap
## 5 NY.GDP.PCAP.CN     GDP per capita (current LCU)                       GDP per cap
```

Searching for Data of Interest

- It might also make sense to see what data there is available in a more detailed way:

```
wb_search("Gender", fields = "topics") ▷ head(5)
```

```
## # A tibble: 5 × 3
##   indicator_id         indicator
##   <chr>                <chr>
## 1 IC.FRM.FEMM.ZS       Firms with female top manager (% of firms)
## 2 IC.FRM.FEMO.ZS       Firms with female participation in ownership (% of firms)
## 3 SE.ADT.1524.LT.FE.ZS Literacy rate, youth female (% of females ages 15-24)
## 4 SE.ADT.1524.LT.FM.ZS Literacy rate, youth (ages 15-24), gender parity index (GPI)
## 5 SE.ADT.1524.LT.MA.ZS Literacy rate, youth male (% of males ages 15-24)
```

Combining Data

- After you've found the variables you are interested in, you need to find a way to extract the data.
- This is also done using a function, in this case `wb_data`. Check out the documentation with `?wb_data`
- In order to instruct the function to extract data, we need to work with indicators
- We can use R's memory to save the `indicator_id`'s somewhere, e.g.:

```
variables ← c("NY.GDP.PCAP.KD.ZG", "SE.ADT.LITR.ZS")
data ← wbstats::wb_data(variables,
                start_date = 2015,
                end_date = 2022)
```

Inspecting the Result

- We can inspect the result:

```
data ← data ▷
  filter(if_all(everything(), ~ !is.na(.x)))
```

```
data ▷ head(8)
```

```
## # A tibble: 8 × 6
##   iso2c iso3c country              date NY.GDP.PCAP.KD.ZG SE.ADT.LITR.ZS
##   <chr> <chr> <chr>               <dbl>             <dbl>          <dbl>
## 1 AW    ABW   Aruba                2020          -24.1              98.0
## 2 AF    AFG   Afghanistan          2021          -23.0              37.3
## 3 AO    AGO   Angola               2022           -0.0968           72.4
## 4 AL    ALB   Albania              2022            6.14             98.5
## 5 AE    ARE   United Arab Emirates 2019            0.324            97.8
## 6 AE    ARE   United Arab Emirates 2021            3.49             98.1
## 7 AE    ARE   United Arab Emirates 2022            6.98             98.3
## 8 AM    ARM   Armenia              2016            0.646            99.7
```

- This is a `data.frame` like the one we've seen in the previous lecture!

Manually Importing Data in R

Importing Data in R

- If we have found a dataset somewhere on the web, we can also use the function `download.file` to download a file from a URL:
- Here, I'm downloading the Cross-country Historical Adoption of Technology (CHAT) dataset. CHAT is an unbalanced panel dataset with information on the adoption of over 100 technologies in more than 150 countries since 1800.

```
data_url ← "https://raw.githubusercontent.com/datasets/historical-adoption-of-tec
```

```
# Data from https://www.nber.org/research/data/cross-country-historical-adoption-t
download.file(data_url,
              destfile = "chat.csv")
```

- The function `download.file()` needs two arguments, an URL which represents an endpoint for a downloaded file, and a destination file. (To which directory is this file now downloaded?)

Importing Data in R

- In this case, I have found this dataset on this repository
  - You should make sure to find the right link
  - The button you click that actually downloads the file is usually the correct link
  - On that button, right click the file > copy download link
- The advantage of this approach is that you engage in reproducible downloading
- The disadvantage is that it downloads anew every time you run the script
  - Which can be prevented by:

```
if(is.na(list.files(path=".", pattern='chat.csv')))
  { download.file(data_url, destfile = "chat.csv") }
```

Importing Data in R - .csv

- Now we have downloaded the file, we should import it to R
- This file is an `.csv` file. We can use the `readr` library to read the data into R:

```
library(readr)
data ← read_csv('chat.csv')

data ▷ head(5)
```

```
## # A tibble: 5 × 113
##   country_name  year ag_harvester ag_milkingmachine ag_tractor   atm aviationpkm avi
##   <chr>        <dbl>        <dbl>              <dbl>      <dbl> <dbl>       <dbl>
## 1 Afghanistan   1750           NA                 NA         NA    NA          NA
## 2 Afghanistan   1751           NA                 NA         NA    NA          NA
## 3 Afghanistan   1752           NA                 NA         NA    NA          NA
## 4 Afghanistan   1753           NA                 NA         NA    NA          NA
## 5 Afghanistan   1754           NA                 NA         NA    NA          NA
## # i 102 more variables: cabletv <dbl>, cellphone <dbl>, cheque <dbl>, computer <dbl>
## #   elecprod <dbl>, fert_total <dbl>, internetuser <dbl>, irrigatedarea <dbl>, kidne
## #   loom_auto <dbl>, loom_total <dbl>, mail <dbl>, med_catscanner <dbl>, med_lithotr
## #   med_mriunit <dbl>, med_radiationequip <dbl>, newspaper <dbl>, pctdaysurg_catarac
## #   pctdaysurg_hernia <dbl>, pctdaysurg_lapcholecyst <dbl>, pctdaysurg_tonsil <dbl>,
## #   pcthomedialysis <dbl>, pctimmunizdpt <dbl>, pctimmunizmeas <dbl>, pctirrigated <
## #   pos <dbl>, radio <dbl>, railline <dbl>, railp <dbl>, railpkm <dbl>, railt <dbl>,
```

Importing Data in R - Other Formats

- There are also other formats you can import in R. Usually, you can still use the `read_csv` function (or its alternative, `read.csv`), but you have to specify more arguments. For example:

- `.tab`:

```
url1 ← "https://dataverse.harvard.edu/api/access/datafile/3205064?format=tab&gbre
download.file(url1, destfile="data.tab")
```

Importing Data in R - Other Formats

- This can be imported using `read_delim` from `readr`:

```
data ← read_delim('data.tab', skip = 1)

data ▷
  head(5)
```

```
## # A tibble: 5 × 6
##       A     B `Concatenating A and B` bdate penroll penroll0
##   <dbl> <dbl>                   <dbl> <dbl>   <dbl>    <dbl>
## 1     1    10                     110 35339    21.5     24.5
## 2     2    10                     210 35340    21.4     24.5
## 3     3    10                     310 35341    21.4     24.4
## 4     4    10                     410 35342    21.4     24.4
## 5     5    10                     510 35343    21.3     24.4
```

- In this case, mind the `skip` argument. What does it do?

Parsing

- By default, `read.` or `read_` type functions assume that the fields in the file are **separated by commas** and that the first row contains column names
- It also automatically converts character columns containing only numbers into numeric type, making it convenient for numeric data processing
- However, CSV files may use **delimiters other than commas**, such as tabs or semicolons. If not specified correctly, `read.csv` may not parse the file accurately

```
read.csv('chat.csv', header=F) ▷
  as_tibble() ▷
  head(3)
```

```
## # A tibble: 3 × 113
##   V1    V2    V3    V4    V5    V6    V7    V8    V9    V10   V11   V12   V13   V14
##   <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr
## 1 coun… year  "ag_… "ag_… "ag_… "atm" "avi… "avi… "bed… "bed… "bed… "cab… "cel… "che
## 2 Afgh… 1750  ""    ""    ""    ""    ""    ""    ""    ""    ""    ""    ""    ""
## 3 Afgh… 1751  ""    ""    ""    ""    ""    ""    ""    ""    ""    ""    ""    ""
## # ℹ 92 more variables: V22 <chr>, V23 <chr>, V24 <chr>, V25 <chr>, V26 <chr>, V27 <ch
## #   V31 <chr>, V32 <chr>, V33 <chr>, V34 <chr>, V35 <chr>, V36 <chr>, V37 <chr>, V38
## #   V42 <chr>, V43 <chr>, V44 <chr>, V45 <chr>, V46 <chr>, V47 <chr>, V48 <chr>, V49
## #   V53 <chr>, V54 <chr>, V55 <chr>, V56 <chr>, V57 <chr>, V58 <chr>, V59 <chr>, V60
```

Writing Files

- Similarly, if you have constructed a dataset in R, you can also write it to a file.
- You can choose the extension by picking the appropriate function.
- For example, we can use `write_csv2` in the `readr` package:
- `write_csv2` writes without row names, whereas `write_csv` does

```
readr::write_csv2(data, 'data1.csv')
```

Deleting Files

```
data ▷ head(5)
```

```
## # A tibble: 5 × 6
##       A     B `Concatenating A and B` bdate penroll penroll0
##   <dbl> <dbl>                   <dbl> <dbl>   <dbl>    <dbl>
## 1     1    10                     110 35339    21.5     24.5
## 2     2    10                     210 35340    21.4     24.5
## 3     3    10                     310 35341    21.4     24.4
## 4     4    10                     410 35342    21.4     24.4
## 5     5    10                     510 35343    21.3     24.4
```

- Sometimes, a good practice might be deleting files
- Be careful with this because it may depend on the situation
- If you want to delete files, use:

```
# Delete the file
unlink('data.tab')
unlink('data.csv')
unlink('data1.csv')
```

SQL Servers

SQL Servers

- Sometimes data is stored on so-called **SQL servers**.
- For example, because the data is **too large to fit into memory at once**
  - Or you don't actually need all of it, only parts or a summary.
  - Or your data is continuously updated by other people/processes.
- To get a part of the data, you submit a query to the SQL server

SQL Servers

- SQL is a separate programming language and a little bit less flexible and accessible than R
- We can use the `dbplyr` package to "translate" R code into SQL code
    - Thereby using the familiar R syntax to make SQL queries
- In this running example, we will make use of a **toy SQL database** to which we will submit queries
    - Usually, you will first connect with **online SQL servers** and then submit queries
    - In order to do so, you might have to get an **API key**, or submit a username and password
    - In this case you won't, but we'll touch on how to do that when we talk about API servers

Connect to SQL Server

- We need the packages `RSQLite`, `DBI` and `dbplyr` packages to interact with the SQL server:

```
if (!require("pacman")) install.packages("pacman")
pacman::p_load(tidyverse, DBI, dbplyr, RSQLite)
```

- Usually SQL servers are part of a paid data collection service
- By means of demonstration, we can connect to an empty SQL server and we'll transfer a `data.frame` to be the data content
- Then we'll submit queries to retrieve the data
    - This last operation is usually what you do when dealing with *real* SQL servers

Connect to SQL server

- Let us create a connection to a newly-created database:

```
# Create a connection to our new database, penguins.db
# You can check that the .db file has been created in your working directory
conn ← dbConnect(RSQLite::SQLite(), "penguins.db")
```

- And let us make a `data.frame` in R which we subsequently export to the SQL server:
- Let us use the `palmerpinguins` package as a standard data package:

```
pacman::p_load('palmerpenguins')
penguins_df ← palmerpenguins::penguins
```

Create Table in Database

Once you have the database created and your data in proper shape, you can go ahead and create a table inside the database using the `dbWriteTable()` function. This function can take multiple arguments, but, for now, let's focus on the following:

- `conn`: The connection to your SQL database
- `name`: The name for the table
- `value`: The table itself

```
dbWriteTable(conn, "penguins", penguins_df, overwrite=TRUE)
```

Retrieve Data from Table

- Now, we have a connection to an SQL server with data!
- Let us try to request some data from it using the SQL language:

```
dbGetQuery(conn, "SELECT species, island,
           bill_length_mm, body_mass_g, sex
           FROM penguins
           WHERE year == 2007 LIMIT 10")
```

```
##    species    island bill_length_mm body_mass_g    sex
## 1   Adelie Torgersen           39.1        3750   male
## 2   Adelie Torgersen           39.5        3800 female
## 3   Adelie Torgersen           40.3        3250 female
## 4   Adelie Torgersen             NA          NA   <NA>
## 5   Adelie Torgersen           36.7        3450 female
## 6   Adelie Torgersen           39.3        3650   male
## 7   Adelie Torgersen           38.9        3625 female
## 8   Adelie Torgersen           39.2        4675   male
## 9   Adelie Torgersen           34.1        3475   <NA>
## 10  Adelie Torgersen           42.0        4250   <NA>
```

Translate from R to SQL

- SQL is an entirely separate language from R
    - Its exclusive goal is communicating with SQL servers
- Fortunately, we don't have to learn SQL: if (when) you know R, you can use the `dbplyr` package to translate R code to SQL:
    - To do so, we use the `tbl` function to retrieve the data.frame *as if it were* in memory:

```
penguins_sql ← tbl(conn, 'penguins')
```

Translate from R to SQL

- And then we can use R to "translate" our query from R to SQL, so that an SQL server will understand us!

```
query ← penguins_sql ▷
  filter(bill_length_mm < 39, flipper_length_mm > 190) ▷
  arrange(desc(bill_depth_mm))

query ▷ show_query()
```

```
## <SQL>
## SELECT *
## FROM `penguins`
## WHERE (`bill_length_mm` < 39.0) AND (`flipper_length_mm` > 190.0)
## ORDER BY `bill_depth_mm` DESC
```

Retrieving Data

- Finally, you can execute the query and retrieve the data by:

```
query ▷
  collect()
```

```
## # A tibble: 24 × 8
##    species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g sex
##    <chr>   <chr>              <dbl>         <dbl>             <int>       <int> <chr
##  1 Adelie  Torgersen           38.6          21.2               191        3800 male
##  2 Adelie  Torgersen           34.6          21.1               198        4400 male
##  3 Adelie  Torgersen           37.3          20.5               199        3775 male
##  4 Adelie  Torgersen           37.7          19.8               198        3500 male
##  5 Adelie  Torgersen           35.1          19.4               193        4200 male
##  6 Adelie  Torgersen           36.7          19.3               193        3450 fema
##  7 Adelie  Biscoe              37.6          19.1               194        3750 male
##  8 Adelie  Torgersen           38.7          19                 195        3450 fema
##  9 Adelie  Biscoe              37.9          18.6               193        2925 fema
## 10 Adelie  Dream               36.8          18.5               193        3500 fema
## # ℹ 14 more rows
```

API Servers

Interacting with API Servers

(from Haggerty, 2023)

- An API (Application Programming Interface) is a set of rules that allow different pieces of software to communicate with each other. Working with an API is like sending a letter with a request to somebody. That somebody is now a server.

- The "roles" in this exchange are as follows:

- Server: A powerful computer that runs an API.

- Client: A program that exchanges data with a server through an API. (e.g. our R session)
- Protocol: The "etiquette" underlying how computers talk to each other (e.g. HTTP).
- Methods: The "verbs" that clients use to talk with a server.
    - The main one that we'll be using is GET (i.e. ask a server to retrieve information that is stored on there)
- Request: Something specifying what data we're looking for
- Response: The server's response. This includes:
    - Status Code (e.g. "404" if not found, or "200" if successful).
    - Header (i.e. meta-information about the response).
    - Body (i.e. the actual content that we're interested in).

API Servers

- API Servers usually have different **endpoints**, corresponding to *different pieces of information* you might be looking for.
    - E.g. for Twitter API: Search Tweets, Timelines, Retweets, Likes, etc.
- API Servers don't give out information to just anyone
    - Often you have to **register and get access** through an *API Key*
    - This key should then be **paired with your request** so that the server knows it's you

Some Examples

- Some interesting / frequently used API servers are linked to below:
- Twitter
- Spotify
- Rechtspraak (and here)
- Youtube
- KNMI (Dutch Metereological Institute)

- We will continue with an example from the Dutch judicial system, *Rechtspraak*, but you might want to find it interesting to try out one of the others as well.

Example: Retrieving Data from the Dutch Judicial System

API Keys

- Fortunately, in this case, the API is free and open to the public.
- In other cases, you may need to register.

- Usually, in this case, after registering, you have to provide servers with an *API key*, which is like a password. This key is user-specific and should be kept secret.

- You can use this key to **contact the server**. The server then knows an authenticated user is submitting a particular request to obtain data.

- You need to read the **documentation** to find out how to send requests for the specific data that you want.

Extracting Data

- You have to read the documentation to find out the precise logic of the request you want to make
- Usually, they way you have to deal with API's is described on the website or in other documentation
    - Especially for R and Python users
- The logic generally revolves around *sending requests* in a certain way
- Then, after we submit a request, we get a reponse in the form of a `.xml` datafile.
    - For which we load the `XML` package, which converts `XML` to R-objects
    - We also load the `httr` package, which allows us to make requests from R to an API server

```
library(httr)
library(XML)
```

Extracting Data

- Define the parameters for the search:

```
params ← list(max=1000,
              creator="http://standaarden.overheid.nl/owms/terms/Rechtbank_Utrech
              type="uitspraak",
              c(date = "1995-01-01T12:00:00",
                date = "2000-01-01T12:00:00")
              )
```

- Send the request to the API server and receive a response:

```
url_handle ← 'https://data.rechtspraak.nl/uitspraken/zoeken?'

response ← httr::GET(url=url_handle,
                     query= params)
```

Inspecting Response

- We can now see what's inside after extracting the data, and converting the `XML` format to a `data.frame`
- In the next lecture(s), we continue with this dataset and see how we can process the text proceedings of the court cases

```
obj ← httr::content(response, as="text")
text ← XML::xmlParse(obj)
df ← XML::xmlToDataFrame(text)
```

Inspecting Response

```
df ▷
  as_tibble() ▷
  head(10)
```

```
## # A tibble: 10 × 6
##    text                                                  id                      title
##    <chr>                                                 <chr>                   <chr>
##  1 Rechtspraak Open Data (Uitspraken)                    <NA>                    <NA>
##  2 Aantal gevonden ECLI's: 59858                         <NA>                    <NA>
##  3 uuid:51e960a8-c9f2-4085-88f5-7baf6d78c342;id=512      <NA>                    <NA>
##  4 Copyright 2024 Rechtspraak.                           <NA>                    <NA>
##  5 2024-02-21T11:26:30+01:00                             <NA>                    <NA>
##  6 <NA>                                                  ECLI:NL:RBUTR:1999:AA3478 ECLI:N
##  7 <NA>                                                  ECLI:NL:RBUTR:1999:AA3625 ECLI:N
##  8 <NA>                                                  ECLI:NL:RBUTR:1999:AA3725 ECLI:N
##  9 <NA>                                                  ECLI:NL:RBUTR:1999:AA3730 ECLI:N
## 10 <NA>                                                  ECLI:NL:RBUTR:1999:AA3732 ECLI:N
```

Recapitulation

Recapitulation

- We have seen how to get data in a number of ways
- Using special **data packages**, using the web and search engines, using SQL servers.
- We have also seen an example of how to retrieve Tweets and meta information using an API server
    - This information could potentially be used to answer research questions
    - But keep in mind that sometimes, some of the options are paid

Recapitulation

- In general, there are many types of requests you can make using **API Servers**
- You should always read the documentation: each API server can work differently
- In particular, you want to know:
  - In what file format the API returns information
  - The **correct syntax** for making a request
  - The parameters that you can provide to make a refined request
- There are also other ways in which we can extract data.
  - For example, some of the things we haven't done are **extracting data from pictures** or from pdf documents