Acquiring and importing texts

Max Callaghan

2022-09-15

Objectives

Methods

By the end of this session, you will be able to

- Scrape texts from a website

Methods

By the end of this session, you will be able to

- Scrape texts from a website
- Use an API to retrieve texts

Methods

By the end of this session, you will be able to

- Scrape texts from a website
- Use an API to retrieve texts
- Read texts stored in various formats and process these with R

Fundamentals of the internet

You should also acquire some important fundamental knowledge about how the internet works

- What is a request, how do you make one, and how can this information be specified?

Fundamentals of the internet

You should also acquire some important fundamental knowledge about how the internet works

- What is a request, how do you make one, and how can this information be specified?
- What responses can be generated from a request, and how can we process these?

Fundamentals of the internet

You should also acquire some important fundamental knowledge about how the internet works

- What is a request, how do you make one, and how can this information be specified?
- What responses can be generated from a request, and how can we process these?
- How are html and json files structured, and how can we use them

Text sources

We will explore how to gain access to the following sources of text

- The IEA's Policies and Measures database, by scraping the website

Text sources

We will explore how to gain access to the following sources of text

- The IEA's Policies and Measures database, by scraping the website
- Scientific articles, using the OpenAlex API
- Twitter posts, using their API

Text sources

We will explore how to gain access to the following sources of text

- The IEA's Policies and Measures database, by scraping the website
- Scientific articles, using the OpenAlex API
- Twitter posts, using their API
- Parliamentary data, by parsing XML data published by Hansard

Scraping texts

What does "scrape" mean, and why do we need to do it?

The internet is full of text data, but it is frequently *presented* - **unstructured** - on websites, rather than made available in **structured** data files.

If we want to do more than just **browse** this data, we need to give our computer instructions on how to systematically download the data of interest.

What happens when we browse the internet?

What happens when we browse the internet?

When we write a url into our web browser and press enter, what we are doing is sending a **request** to an **address**.

In our first example, we are going to look at https://www.iea.org/policies.

- https defines the protocol
- www.iea.org defines the hostname
- policies defines the path on the host containing the resources we require

What happens when we browse the internet?

When we write a url into our web browser and press enter, what we are doing is sending a **request** to an **address**.

In our first example, we are going to look at https://www.iea.org/policies.

- https defines the protocol
- www.iea.org defines the hostname
- policies defines the path on the host containing the resources we require

If we click on open the url with chrome or firefox, we can investigate further by opening developer tools (ctrl+shift+i). Today we will look at the **Network** and **Elements** tabs

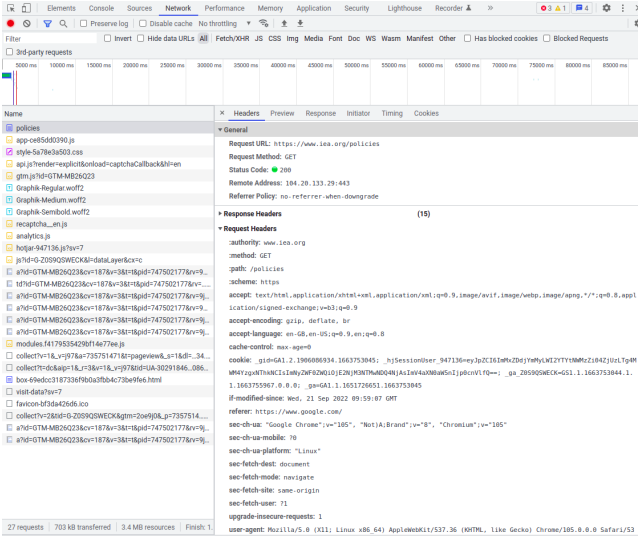
Making Requests

If you click on the **Network** tab and refresh, you can see all the communication that is happening when we visit a page.

Clicking on policies, we can inspect how this starts.

Our browser sends a request to the url, along with **headers**, which explain how the request should be processed.

We then receive a response, which has content, a status code, and it's own set of headers.



Making requests with R

We can mimic this R using httr

```
library(httr)
r <- GET("https://www.iea.org/policies")
r
```

```
## Response [https://www.iea.org/policies]
##   Date: 2022-09-21 20:08
##   Status: 200
##   Content-Type: text/html; charset=UTF-8
##   Size: 304 kB
## <!DOCTYPE html>
## <html dir="ltr" lang="en-GB"
##       class="no-js page-all-policies ">
## <head>
##     <meta charset="utf-8">
##     <meta name="viewport" content="width=device-width, initial-scale=1.0">
##     <meta http-equiv="X-UA-Compatible" content="IE=Edge">
##     <meta name="csrf-token" content="">
##
##     <link rel="shortcut icon" href="/assets/front/images/favicon-bf3da426d6.i...
## ...
```

Making requests with Python

In Python, a similar no-frills option is requests

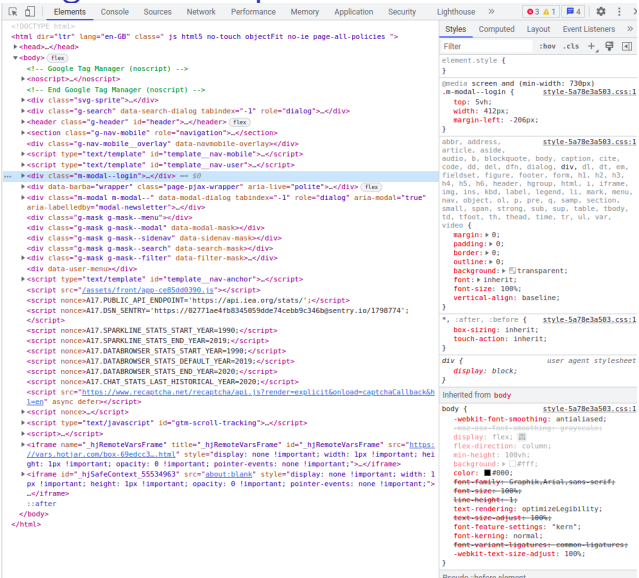
```
import requests
from rich.pretty import pprint
#r = requests.get("https://www.iea.org/policies")
#pprint(r.__dict__, max_string=40)
```

Understanding HTML responses

If you click on the **Elements** tab, you will see the HTML response of the website.

HTML is a hierarchical structure of elements Hello . In this hierarchy we refer to

- the **root**
- parents
- children
- siblings



What is in a web element?

The element **name** is the first word after the opening <, and describes what *type* of element it is.

The element's **attributes** are the key, value pairs either side of the = signs before the >.

Element's should be closed with a / and a >. <a> and <a/> are both closed.

Anything between opening and closing tags (<>) is the element's content, or inner html. It can contain further elements (children)

You can find an element by clicking on the icon with the cursor in the developer tools

```
<a class="m-policy-listing-item__link"
href="/policies/12654-emissions-limit-on-the-capacity-market-regulations">
Emissions limit on the Capacity Market Regulations
</a>
```

Scraping elements

In our example from the IEA, we want to identify each element linking to a policy, and find a common feature of those links. We can select these by passing css selectors to the `html_elements` function from rvest

In this case they all have the class "m-policy-listing-item___link"

```
library(rvest)
html <- read_html("https://www.iea.org/policies")
links <- html %>% html_elements("a.m-policy-listing-item__link")
links
```

```
## {xml_nodeset (30)}
##  [1] <a class="m-policy-listing-item__link" href="/policies/11663-fuel-econom ...
##  [2] <a class="m-policy-listing-item__link" href="/policies/12654-emissions-l ...
##  [3] <a class="m-policy-listing-item__link" href="/policies/8506-gas-boilers- ...
##  [4] <a class="m-policy-listing-item__link" href="/policies/3124-local-govern ...
##  [5] <a class="m-policy-listing-item__link" href="/policies/12046-decommissio ...
##  [6] <a class="m-policy-listing-item__link" href="/policies/8401-enhancements ...
##  [7] <a class="m-policy-listing-item__link" href="/policies/12197-heavy-goods ...
##  [8] <a class="m-policy-listing-item__link" href="/policies/11497-proposals-f ...
##  [9] <a class="m-policy-listing-item__link" href="/policies/13139-resolution- ...
## [10] <a class="m-policy-listing-item__link" href="/policies/11456-updated-mep ...
## [11] <a class="m-policy-listing-item__link" href="/policies/15028-france-2030 ...
## [12] <a class="m-policy-listing-item__link" href="/policies/15026-france-2030 ...
## [13] <a class="m-policy-listing-item__link" href="/policies/15025-france-2030 ...
## [14] <a class="m-policy-listing-item__link" href="/policies/14279-france-2030 ...
```

Scraping elements

In our example from the IEA, we want to identify each element linking to a policy, and find a common feature of those links. We can select these by passing css selectors to the `select` function of Beautiful Soup

In this case they all have the class “m-policy-listing-item__link”

```
import requests
from bs4 import BeautifulSoup
r = requests.get("https://www.iea.org/policies")
soup = BeautifulSoup(r.content)
links = soup.select("a.m-policy-listing-item__link")
links
```

```
## [<a class="m-policy-listing-item__link" href="/policies/11663-fuel-economy-standards-on-light-duty-vehicles">Fue
##                                                  </a>, <a class="m-policy-listing-item__link" href="/policies
##                                                  </a>, <a class="m-policy-listing-item__link" href="/policies
##                                                  </a>, <a class="m-policy-listing-item__link" href="/policies
##                                                  </a>, <a class="m-policy-listing-item__link" href="/policies
##                                                  </a>, <a class="m-policy-listing-item__link" href="/policies
##                                                  </a>, <a class="m-policy-listing-item__link" href="/policies
##                                                  </a>, <a class="m-policy-listing-item__link" href="/policies
##                                                  </a>, <a class="m-policy-listing-item__link" href="/policies
##                                                  </a>, <a class="m-policy-listing-item__link" href="/policies
##                                                  </a>, <a class="m-policy-listing-item__link" href="/policies
##                                                  </a>, <a class="m-policy-listing-item__link" href="/policies
##                                                  </a>, <a class="m-policy-listing-item__link" href="/policies
```

Following links and extracting information

Now we want to follow each of these links, parse the website, and extract the information we want

```
library(tibble)
df <- tibble(text=character())
for (link in html_attr(links,"href")) {
  link_html <- read_html(paste0("https://iea.org",link))
  text <- link_html %>% html_element("div.m-block p") %>% html_text()
  df <- df %>% add_row(text=text)
  break
}
df
```

```
## # A tibble: 1 x 1
##   text
##   <chr>
## 1 Japan sets and periodically updates fuel economy standards on cars, vans and ~
```

Following links and extracting information

Now we want to follow each of these links, parse the html, and extract the information we want

```
import pandas as pd
data = []
for link in links:
    r = requests.get("https://iea.org" + link["href"])
    link_soup = BeautifulSoup(r.content)
    data.append({"text": link_soup.select("div.m-block p")[0].text})
    break

df = pd.DataFrame.from_dict(data)
df
```

```
##                                                text
## 0  Japan sets and periodically updates fuel econo...
```

Exercise

Now in pairs, build a scraper that returns a dataframe with the columns [Country, Year, Status, Jurisdiction, Text, Link, Topics, Policy types, Sectors, Technologies]

How would you extend this scraper to collect the whole database (not just the first page)?

APIs

What is an API and how do I use it?

An API is a *predefined* set of possible requests, with a given set of possible responses and response formats.

APIs usually return **data** rather than instructions for building a web page.

They are explicitly built for access by machines, and should stay consistent over time.

What is an API and how do I use it?

An API is a *predefined* set of possible requests, with a given set of possible responses and response formats.

APIs usually return **data** rather than instructions for building a web page.

They are explicitly built for access by machines, and should stay consistent over time.

The first API we will look at is for the open catalog of scientific research OpenAlex

For more details on OpenAlex, have a look at this tutorial I gave for a summer school.

Constructing an API call

Let's start by searching the institutions endpoint for the Hertie School

https://api.openalex.org/institutions?filter=display_name.search:hertie

We can plug the ID we find here into a query of the works enpoint, where we search works where an author is affiliated with Hertie

https://api.openalex.org/works?filter=authorships.institutions.id:I24830596

Parsing Json

Now we just need to parse the json, which is very easy in python

```
from dotenv import load_dotenv
import os
load_dotenv()
```

```
## True
```

```
headers = {"email": os.getenv("bearer_token")}
r = requests.get(
  "https://api.openalex.org/works?filter=authorships.institutions.id:I24830596",
  headers=headers
)
res = r.json()
pprint(res, max_string=21, max_length=5)
```

```
## {
##     'meta': {
##         'count': 1275,
##         'db_response_time_ms': 17,
##         'page': 1,
##         'per_page': 25
##     },
##     'results': [
##         {
##             'id': 'https://openalex.org/'+11,
##             'doi': 'https://doi.org/10.10'+15,
```

Parsing Json

Now we just need to parse the json, which is very easy in python, and a bit of a pain in R. For now we'll just let create a dataframe with dataframes inside it

```
library(jsonlite)
library(dplyr)
library(dotenv)
load_dot_env(".env")
r <- GET(
  "https://api.openalex.org/works?filter=authorships.institutions.id:I24830596",
  add_headers(email=Sys.getenv("email"))
  )
data <- fromJSON(content(r, "text"))
```

```
## No encoding supplied: defaulting to UTF-8.
```

```
df <- cbind(
  select(data$results, where(is.character)),
  select(data$results, where(is.numeric))
)
head(df)
```

```
##                                 id                                  doi
## 1 https://openalex.org/W2195453830         https://doi.org/10.1038/nclimate2870
## 2   https://openalex.org/W18536190 https://doi.org/10.1007/978-3-658-22261-1_12
## 3 https://openalex.org/W2041842081      https://doi.org/10.1111/1468-0386.00031
## 4 https://openalex.org/W2092902022   https://doi.org/10.1016/j.riob.2014.09.001
## 5 https://openalex.org/W2003457148    https://doi.org/10.1007/s10551-012-1414-3
```

Paginated results

Where datasets are large, APIs will often not give us the whole dataset at once, but deliver it in chunks. They will have their own way of letting us navigate through these, but often this will involve cursors.

With open Alex, we simply add &cursor=* to our url the first time we make a request, and keep using the new cursor it returns until it is Null

Paginated results

```
cursor <- "*"
base_url <- "https://api.openalex.org/works?filter=authorships.institutions.id:I24830596"
df <- NULL
while (!is.null(cursor)) {
  r <- GET(paste0(
    base_url, "&per-page=200",
    "&cursor=",cursor
  ), add_headers(email=Sys.getenv("email")))
  data <- fromJSON(content(r, "text", encoding="utf-8"), simplifyDataFrame = TRUE)
  if (length(data$results) == 0) { break }
  page_df <- cbind(
    select(data$results, where(is.character)),
    select(data$results, where(is.numeric))
  )
  df <- rbind(df, page_df)
  cursor <- data$meta$next_cursor
}
nrow(df)
```

```
## [1] 1275
```

Paginated results

```
cursor = "*"
base_url = "https://api.openalex.org/works?filter=authorships.institutions.id:I24830596"
works = []
while cursor is not None:
    r = requests.get(f"{base_url}&per-page=200&cursor={cursor}", headers=headers)
    res = r.json()
    if len(res["results"])==0:
        break
    for work in res["results"]:
        w = {}
        for k, v in work.items():
            if type(v) not in [dict, list] and v is not None:
                w[k] = v
        works.append(w)
    cursor = res["meta"]["next_cursor"]

df = pd.DataFrame.from_dict(works)
print(df.shape)
```

```
## (1275, 14)
```

```
df.head()
```

```
##                                 id  ... created_date
## 0  https://openalex.org/W2195453830  ...   2016-06-24
## 1    https://openalex.org/W18536190  ...   2016-06-24
## 2  https://openalex.org/W2041842081  ...   2016-06-24
## 3  https://openalex.org/W2092902022  ...   2016-06-24
```

Using a Library to speak to an API

Often, someone will already have built a scraper or an API for the dataset you are looking for. These might be called Client libraries.

Always search this first, but these libraries often do very simple things.

One thing that can be especially annoying is **authentication**. We're going to use rtweet

Rtweet

To use rtweet, you will need to authenticate interactively (by leaving the argument blank) using the "Bearer Token" you generated on the twitter website, or provide the token directly to 'rtweet_app'. NEVER expose your secret keys in a Github repository!

Once you have done this you can pass the result as the **token** argument to your API call. For now, we will explore the `search_tweets` endpoint

```
library(rtweet)
library(dotenv)
load_dot_env(".env")
auth <- rtweet_app(Sys.getenv("bearer_token"))
rt <- search_tweets("hertie", n = 1000, include_rts = FALSE, token=auth)
rt
```

```
## # A tibble: 105 x 43
##    created_at                id id_str       full_~1 trunc~2 displ~3 entities
##    <dttm>                 <dbl> <chr>        <chr>   <lgl>     <dbl> <list>
##  1 2022-09-21 10:26:04 1.57e18 15725024087~ "Say h~ FALSE       274 <named list>
##  2 2022-09-21 20:00:34 1.57e18 15726469878~ "\"Coo~ FALSE       275 <named list>
##  3 2022-09-21 19:02:12 1.57e18 15726322963~ "@Deba~ FALSE        66 <named list>
##  4 2022-09-21 17:38:08 1.57e18 15726111413~ "@Shub~ FALSE       125 <named list>
##  5 2022-09-21 17:22:54 1.57e18 15726073093~ "@Shub~ FALSE        97 <named list>
##  6 2022-09-21 16:45:41 1.57e18 15725979419~ "@Shub~ FALSE       110 <named list>
##  7 2022-09-21 16:01:02 1.57e18 15725867068~ "From ~ FALSE       284 <named list>
##  8 2022-09-21 14:18:25 1.57e18 15725608826~ "@Hert~ FALSE        59 <named list>
```

Tweepy

Tweepy is a similar library for python, which works in a similar way

```
import tweepy
from dotenv import load_dotenv
import os
load_dotenv()
```

```
## True
```

```
auth = tweepy.OAuth2BearerHandler(os.getenv("bearer_token"))
api = tweepy.API(auth)
results = api.search_tweets("hertie")
json_data = [r._json for r in results]
df = pd.json_normalize(json_data)
df
```

```
##                         created_at  ...  possibly_sensitive
## 0   Wed Sep 21 20:27:13 +0000 2022  ...                 NaN
## 1   Wed Sep 21 19:59:54 +0000 2022  ...                 NaN
## 2   Wed Sep 21 18:20:33 +0000 2022  ...                 NaN
## 3   Wed Sep 21 18:00:34 +0000 2022  ...               False
## 4   Wed Sep 21 17:02:12 +0000 2022  ...                 NaN
## 5   Wed Sep 21 15:38:08 +0000 2022  ...                 NaN
## 6   Wed Sep 21 15:22:54 +0000 2022  ...                 NaN
## 7   Wed Sep 21 14:51:17 +0000 2022  ...                 NaN
## 8   Wed Sep 21 14:45:41 +0000 2022  ...                 NaN
## 9   Wed Sep 21 14:25:35 +0000 2022  ...                 NaN
## 10  Wed Sep 21 14:01:02 +0000 2022  ...               False
```

Some limitations of twitter

Without the academic API, twitter search offers only a non-random sample of recent tweets

Even with the academic API, many limitations remain

- Geotag availability
- Non-representative populations
- Country/language overlaps
- Bots

Still, it is a very interesting data source, and valuable for understanding social behaviour *on twitter*

Other data sources

Parliamentary data from Hansard

Hansard keeps a record of all the debates made in the UK parliament. These have have been parsed as XML files (which are quite like html) and are available in a time series going back more than a century here.

We can use RVest to parse these

```
library(rvest)
data <- read_html("https://www.theyworkforyou.com/pwdata/scrapedxml/debates/debates2022-09-10a.xml")
speeches <- data %>% html_elements("speech")
df <- as_tibble(do.call(rbind, html_attrs(speeches)))
df$text <- speeches %>% html_text()
df
```

```
## # A tibble: 155 x 8
##    id                              speak~1 type  perso~2 colnum time  url   text
##    <chr>                           <chr>   <chr> <chr>   <chr>  <chr> <chr> <chr>
##  1 uk.org.publicwhip/debate/2022~ Lindsa~ uk.o~ 653     ""     ""    "uk.~ "Fol~
##  2 uk.org.publicwhip/debate/2022~ Lindsa~ uk.o~ 654     ""     ""    "uk.~ "I w~
##  3 uk.org.publicwhip/debate/2022~ Lindsa~ uk.o~ 654     ""     ""    "uk.~ "We ~
##  4 uk.org.publicwhip/debate/2022~ Lindsa~ uk.o~ 655     "13:1~ ""    "uk.~ "I n~
##  5 uk.org.publicwhip/debate/2022~ Theres~ Star~ uk.org~ "655"  "13:~ ""    "Tha~
##  6 uk.org.publicwhip/debate/2022~ Yvette~ Star~ uk.org~ "656"  "13:~ ""    "Thi~
##  7 uk.org.publicwhip/debate/2022~ Amanda~ Star~ uk.org~ "657"  "13:~ ""    "My ~
##  8 uk.org.publicwhip/debate/2022~ George~ Star~ uk.org~ "658"  "13:~ ""    "Her~
##  9 uk.org.publicwhip/debate/2022~ Grant ~ Star~ uk.org~ "658"  "13:~ ""    "Lik~
## 10 uk.org.publicwhip/debate/2022~ Lindsa~ uk.o~ 659     "13:3~ ""    "uk.~ "I c~
```

Parliamentary data from Hansard

Hansard keeps a record of all the debates made in the UK parliament. These have have been parsed as XML files (which are quite like html) and are available in a time series going back more than a century here.

We can use RVest to parse these, or BeautifulSoup in Python

```
r = requests.get("https://www.theyworkforyou.com/pwdata/scrapedxml/debates/debates2022-09-10a.xml")
soup = BeautifulSoup(r.content)
speeches = soup.select("speech")
rows = []
for s in speeches:
    row = s.attrs
    row["text"] = s.text
    rows.append(row)
df = pd.DataFrame.from_dict(rows)
print(df.shape)
```

```
## (155, 9)
```

```
df.head()
```

```
##                                          id  ... nospeaker
## 0  uk.org.publicwhip/debate/2022-09-10a.653.1  ...        NaN
## 1  uk.org.publicwhip/debate/2022-09-10a.654.1  ...        NaN
## 2  uk.org.publicwhip/debate/2022-09-10a.654.2  ...        NaN
## 3  uk.org.publicwhip/debate/2022-09-10a.655.1  ...        NaN
```

Wrapup

Exercise

We've had a look at 4 different data sources. Pick one, alter the query parameters (if applicable) and try to process it as we did last week. Report on commonly used words in the data.

Extensions

Sometimes, neither RVest / BeautifulSoup nor APIs will get you the data you want. You may need to sign in, or click on certain buttons to make pages load, especially if they use a lot of Javascript to generate the pages.

In these cases, check out Selenium, which allows you to automate a browser and interact fully with websites.

Ethics

We are talking about collecting data that is publicly available, but it still matters

- What data you scrape or access

Ethics

We are talking about collecting data that is publicly available, but it still matters

- What data you scrape or access
- Who you are

Ethics

We are talking about collecting data that is publicly available, but it still matters

- What data you scrape or access
- Who you are
- Who created the data and what their expectations were about its use

Ethics

We are talking about collecting data that is publicly available, but it still matters

- What data you scrape or access
- Who you are
- Who created the data and what their expectations were about its use
- How you intend to use the data, and what potential consequences that entails

Ethics

We are talking about collecting data that is publicly available, but it still matters

- What data you scrape or access
- Who you are
- Who created the data and what their expectations were about its use
- How you intend to use the data, and what potential consequences that entails

As a general rule, when working with twitter data, we only publish individual tweets when the user is a public person or has expressly approved the use

We should also be considerate not to overload sites with requests, and to follow their instructions for scraping when these are reasonable (check robots.txt)

Wrapup and outlook

In the next session, we'll cover **regex** expressions, and how we can use stringr to clean, manage, manipulate, and extract useful data from unstructured texts.