Methods for unstructured data

Lecture 2: Web scraping

Helge Liebert

Roadmap and goals

- Use processing of web data to introduce various ideas along the way.
- Accessing data on the web: APIs.
- Gathering (semi-structured) web data and transforming it into structured data ("*web scraping*").

- Give you a general understanding how web scrapers work.
- Foster an understanding of the development process.
- Point you towards the necessary tools so you can write your own.

Contents

Guiding problem

- How to turn unstructured into structured data?

- Consider a situation where
  - You want to get data from the internet.
  - The data is in unstructured/semi-structured form.
  - Possibly embedded in a website.
  - You want to transform it into a differently structured format for further use.
 - You need to filter the available information.

From this …

…or this

…to this.

Goal: Automation

- Digitalization offers exciting data for research. But: *Data is messy.*
- Gathering or processing data often involves repetitive manual tasks.
- Disadvantages:
    - Manual tasks are often not well documented or reproducible ex post.
    - Manual work is frustrating and a huge time-sink.
    - Manual work may not be feasible with large data.

➥ Automation helps!
    - Frees you to engage in other work.
    - You learn new things.
    - Should you encounter the same class of problem in the future, you already have a solution at hand.
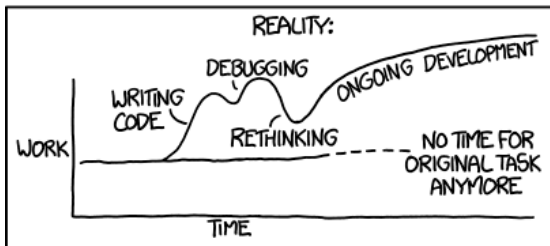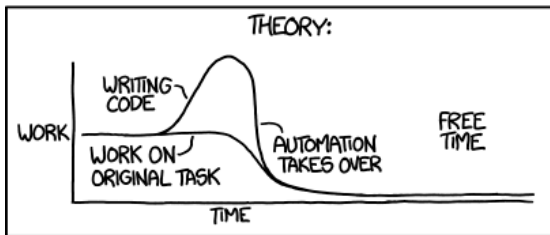
Getting started—things to consider before you begin

- Pick up the phone and try to get the data directly.
- Search if somebody has already faced the same or a similar problem.
- Does the site or service provide an API that you can access directly?
- Is there a wrapper for it?

- Is the website only online for a limited time? Do you want an original snapshot as a backup? Is it more convenient to filter your data offline?

Static and dynamic websites

Static Website

Scheme A

HTTP request

HTTP response

Dynamic Website

Scheme B

HTTP request

HTTP response

Server-side processing
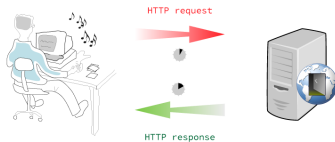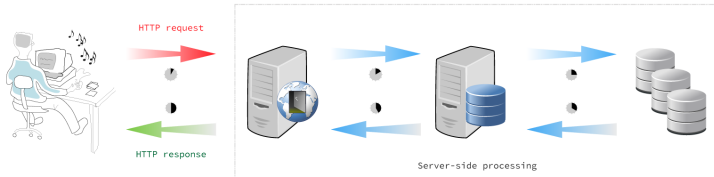
Save an offline copy

- Use the shell utilities `wget` or `curl` to download the complete site.
- Also useful if you just want a set of files (e.g. pdf documents) from the same site directory.
- Convenient for static sites of limited size.
- Infeasible for large sites or sites that create content dynamically.

Examples

- Simple http GET request.
  ```
  wget http://www.google.com
  ```
- Recursively download a website.
  ```
  wget -r http://www.some-site.com/some-subdir/
  ```
- Download all pdfs from a site.
  ```
  wget -r -A.pdf http://url-to-webpage-with-pdfs/
  ```
- Mirror a site offline and convert links for local browsing.
  ```
  wget --mirror -p --convert-links -p ./local-dir
        http://target-website.com
  ```

Application Programming Interfaces

- Data providers often offer Web APIs (*Application Programming Interface*) to access data.
- Allow programmable access to data via a defined set of HTTP messages. Similar to visiting a website: you specify a URL and information is sent to your machine.
- With a website, you receive code interpreted by your browser (HTML, CSS, JavaScript). With an API, you receive data.
- Usually in JSON (*JavaScript Object Notation*) or XML (*Extensible Markup Language*) format.

Web APIs

- Often just two steps:
    1. Construct the URL query that serves as the API request.
    2. Process the response message the API sends back.
- Examples:
    - https://api.kivaws.org/v1/loans/newest.html
    - https://api.kivaws.org/v1/loans/newest.json
    - https://api.kivaws.org/v1/loans/search.json?sector=Agriculture&
      country=VN
    - https://www.theyworkforyou.com/api/getMPs?&key=someapikeyhere&
      output=js
- Libraries may offer wrappers for APIs: WDI, wbstats, twfy, pvsR,
  Google Maps, OpenStreetMap/OSRM, …
- Sometimes it is possible to reverse engineer a site's internal API rather
  than scraping the HTML.

The Document Object Model

HTML and the Document Object Model

- Extracting information from the web requires a basic understanding of HTML and the associated Document Object Model (DOM).
- HTML elements provide the structure and content of web pages.
- Consist of `<start>` and `</end>` tags, with content in between.
    `<tagname>`Content here`</tagname>`
- A page consists of nested elements.
- The html element is the outer-most element, nesting the head and body elements, which in turn have nested elements.
- Nesting structure of elements can be represented by a tree (DOM).

Document Object Model

- The DOM is a programming interface for HTML and XML documents.
- Provides a structured representation of the document.
- A document as a group of nodes, each node representing a part of the document.
- Allows programmatic access to the tree to change the structure, style and content of the document.
- Connects web pages to scripts or programming languages.

A simple HTML page:

```html
<html>
  <head>
    <title>My Web Page</title>
  </head>
  <body>
    <h1>Welcome To My Web Page</h1>
  </body>
</html>
```

How a browser renders this page:

Welcome To My Web Page

A simple HTML page:

```
<html>
  <head>
    <title>My Web Page
    </title>
  </head>
  <body>
    <h1>Welcome To My Web Page
    </h1>
  </body>
</html>
```
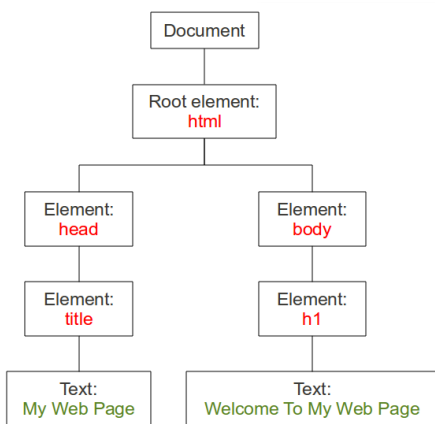
Corresponding node tree:

DOM node trees

- HTML DOM views a document as a tree structure called node tree.
- Everything in an HTML document is a node.
  - The entire document is a document node
  - Every HTML element is an element node
  - Every HTML attribute is an attribute node
  - Text content in the HTML elements is a text node
- Nodes can be accessed through the tree.
- Nodes may be assigned unique id attributes.

Example: An HTML table element
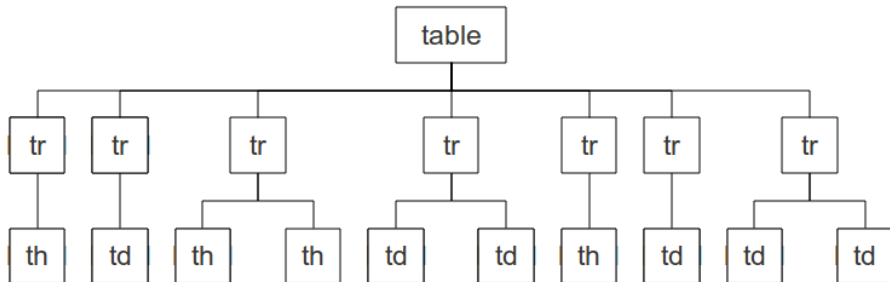
- Tables are represented by a top-level `table` element.
- The `table` element nests `tr` (*table row*) elements.
- These nest `th` (*table header*) and `td` (*table data*) element cells.

HTML and the DOM

- HTML tags can have attributes and text content.

  ```
  <tag attribute="value" attribute2="value">Text content.</tag>
  ```

- Example page:

  ```
  <html>
    <head>
      <title>My Web Page</title>
    </head>
    <body>
      <h1>Welcome To My Web Page</h1>
      <img src="picture.png" alt="example photo" width="100">
      <a href="pagelink.html" id="pageid">Check this other page.</a>
    </body>
  </html>
  ```

Data from the web

Wikipedia on infant mortality

Epidemiology [edit]

See also: List of countries by infant mortality rate

For the world, and for both less developed countries (LDCs) and more developed countries (MDCs), IMR decli

However, IMR was, and remains, higher in LDCs. In 2001, the IMR for LDCs (91) was about 10 times as large
countries are, on average, much less than those among the more developed countries.[*clarification needed*]

A factor of about 67 separate countries with the highest and lowest reported infant mortality rates. The top and

Rank	Country	Infant mortality rate (deaths/1,000 live births)
1	Afghanistan	121.63
2	Niger	109.98
3	Mali	109.08
4	Somalia	103.72
5	Central African Republic	97.17
218	Sweden	2.74
219	Singapore	2.65
220	Bermuda	2.47
221	Japan	2.21
222	Monaco	1.80

According to Guillot, Gerland, Pelletier and Saabneh "birth histories, however, are subject to a number of error

Fetching a table from Wikipedia

```
library("rvest")

# 1) fetch and parse the website
page <- read_html("https://en.wikipedia.org/wiki/Infant_mortality")
# 2) extract the html node containing the table
table <- html_node(page,
                    xpath = "//*[@id='mw-content-text']/div/table[2]")
# 3) extract the table as a data frame
mrates <- html_table(table)
```

Inspecting the HTML source

- Convenient with modern browsers: Use the developer tools (right-click *Inspect*).
- Look at the HTML source to grasp the structure.
- Find out how to navigate the site.
- Find the element(s) you want to extract.
- Get the Xpath expression or CSS selector to extract elements.

HTML elements visualized

Infant mortality rates from Wikipedia

```html
<table class="wikitable" style="text-align:left">
  <tbody>
    <tr>
      <th>Rank</th>
      <th>Country</th>
      <th>Infant mortality rate <br> (deaths/1,000 live births)</th>
    </tr>
    <tr>
      <td>1</td>
      <td style="text-align:left;"><a href="/wiki/Afghanistan" title="Afghanistan">Afghanistan</a></td>
      <td>121.63</td>
    </tr>
    <tr>
      <td>2</td>
      <td style="text-align:left;"><a href="/wiki/Niger" title="Niger">Niger</a></td>
      <td>109.98</td>
    </tr>
    <tr>
      <td>3</td>
      <td style="text-align:left;"><a href="/wiki/Mali" title="Mali">Mali</a></td>
      <td>109.08</td>
    </tr>
    <tr>
      <td>4</td>
      <td style="text-align:left;"><a href="/wiki/Somalia" title="Somalia">Somalia</a></td>
      <td>103.72</td>
    </tr>
    ...
  </tbody>
</table>
```

CSS selectors and XPath expressions

```
# fetch and parse the website
page <- read_html("https://en.wikipedia.org/wiki/Infant_mortality")
# list the table nodes
html_nodes(page, "table")
# using xpath expressions or css selectors is equivalent
table <- html_node(page,
                   xpath = "//*[@id='mw-content-text']/div/table[2]")
table <- html_node(page,
                   css = "#mw-content-text > div > table:nth-child(121)")
```

Examples

The general structure

- There is no universal recipe. But most programs follow a certain structure.
    1. Open a website mimicking a browser and navigate it (optional).
    2. Get the page source HTML and feed it to a parser.
    3. Extract the elements you need.
    4. Filter and arrange them as needed and save them.
    5. Repeat 1.–4. as needed.

Navigating to another page

```r
# Open infant mortality page
session <- session("https://en.wikipedia.org/wiki/Infant_mortality")
# Goto page on Somalia
session <- session_follow_link(session, "Somalia")
# Read the source
page <- read_html(session)
# Extract html
table <- html_node(page,
                   xpath = "//*[@id='mw-content-text']/div/table[4]")
regions <- html_table(table)
```

Filtering links

```
# read wiki page
page <- read_html("https://en.wikipedia.org/wiki/Infant_mortality")
# get the links
wikilinks <- html_attr(html_nodes(page, "a"), "href")
# use regex to filter internal links:
#   select only articles, no files or category pages,
#   matching with mortality or somalia
links <- grep("^(?!.*:)(/wiki/.*Mortality)|(/wiki/.*Somalia)", wikilinks,
              ignore.case = TRUE, value = TRUE, perl = TRUE)
links <- unique(links)
# go to first selected article page and process it
session <- jump_to(session, links[1])
page <- read_html(session)
html_nodes(page, "title")
```

- Phillippine Statistics Authority *Good Governance Index*.
- ~~Available at http://nap.psa.gov.ph/ggi/default.asp~~.
- Available at `https://web.archive.org/web/20190915135458/http://nap.psa.gov.ph/ggi/default.asp`.
- Get all GGI data tables for all municipalities.
- Save them in a local data file for further analysis.
- How would you go about this?

Another example

- WHO venomous snakes distribution and species risk categories
- Database form link:
  https://apps.who.int/bloodproducts/snakeantivenoms/database/SearchFrm.aspx
- Collect snake data for all countries.
- Getting dropdown options and initial form submission straightforward.
- So is table extraction.
- Navigating further links is tricky.

If simple navigation fails

- Some web pages cannot be navigated easily with simpler requests.
- Often due to hidden Javascript or other server-side processing.
- In this case, resort to Selenium (`library(rselenium)` in R).
- Under the hood, Selenium relies on a complete browser running in a container.
- Slower and comes with substantial overhead costs.
- Only use when necessary.

General remarks

- Start simple and expand your program incrementally.
- Keep it simple. Do not overengineer the problem. Do not repeat yourself.
- Limit the number of iterations for test runs. Use print statements to inspect objects.
- Write tests to verify things work as intended.
- If your program requires complex monitoring/validation procedures or threading for performance, use Python.

Final remarks

- Sometimes small programs can go a long way.
- Do not lose sight of your ultimate goal. Time is valuable.
- Do not engage in perfectionism, focus on GTD.
- Identify everyday tasks that you can optimize.

Assignment

Assignment, part I

Choose *one* of the following.

Open Data Tanzania
https://tanzania.opendataforafrica.org/

➡ For each region in Tanzania, get the basic facts (top of the page: capital, languages, area, …), population and mortality statistics (tables under links 'Population projections, total' and 'Deaths').

Books to sell
https://books.toscrape.com/index.html

➡ For every category, collect all books with title, price, availability, stock and product description.

Swiss Tax Calculator
https://swisstaxcalculator.estv.admin.ch/#/taxburden/income-wealth-tax

➡ Collect municipal income tax burden statistics for the following parameters: All payers, atheist, geographical comparison (all municipalities), all years, income 50'000 CHF.

Assignment, part I

- Submission deadline is November 7, 2021.
- Submit code only, no data.
- Comment your code or submit a short description alongside.
- A proof-of-concept restricted to the first couple of regions/... is fine.
- Accounts for 20% of the final grade.

Next lecture: Text as data.