Web Scrapping

(Lectures on High-performance Computing for Economists X)

Jesús Fernández-Villaverde,¹ Pablo Guerrón,² and David Zarruk Valencia³
November 18, 2019

¹University of Pennsylvania
²Boston College
³ITAM
Web scraping

• Internet includes thousands of data points that can be used for research.

• Examples:

1. Yelp: David, Dingel, Monras, and Morales: “How segregated is urban consumption” (Accepted JPE).


Web scraping II

- However, data may be split across thousands of URLs (requests):

- And include multiple filters: bedrooms, bathrooms, size, price range, pets:

- Automatize data collection: code that gathers data from websites.

- (Almost) any website can be scraped.
Permissions

- Beware of computational, legal, and ethical issues related with web scrapping. Check with your IT team and read the terms of service of a website.

- Go to *The Robots Exclusion Protocol* of a website, adding “/robots.txt” to the website’s URL: [www.google.com/robots.txt](http://www.google.com/robots.txt).

- E.g.: Spotify’s *robots.txt*’s file:

  ![robots.txt example](https://example.com/robots-txt.png)

  Three components:
  1. User-agent: the type of robots to which the section applies.
  2. Disallow: directories/prefixes of the website not allowed to robots.
  3. Allow: sections of the website allowed to robots.

- *robots.txt* is a *de facto* standard (see [http://www.robotstxt.org](http://www.robotstxt.org)).
How do you scrap?

- You can rely on existing packages:
  1. Scraper for Google Chrome.
  2. Scrapy: https://scrapy.org/

- Or you use your own code:
  1. Custom made.
  2. Python: packages BeautifulSoup, requests, httplib, and urllib.
  3. R: package httr, RCurl, and rvest.
• Nearly all websites are written in standard HTML (Hyper Text Markup Language).

• Due to simple structure of HTML, all data can be extracted from the code written in this language.

• Advantages of web scrapping vs., for example, APIs:
  1. Websites are constantly updated and maintained.
  2. No rate limits (such as limits to daily queries in APIs) – apart from explicit restrictions.
  3. Data is readily available.

• However, there is no bulletproof method:
  1. Data is structured differently on every website (different request methods, HTML labels, etc.).
  2. Unlike APIs, usually no documentation.
  3. Take your time, be patient!
A motivating example in R I

Let us first clear everything:

```r
rm(list=ls())
```

We install and load required packages:

```r
install.packages("rvest")
library(rvest)
library(dplyr)
```

We read a webpage into a parsed HTML document:

```r
my_page <- read_html("relevant_page.html")
```

We extract a table:

```r
my_page %>%
  html_node("table") %>% html_table()
```
A more realistic example of getting financial information:

```r
page <- read_html("https://finance.yahoo.com/quote/MSFT")
```

We get price:

```r
page %>%
  html_node("div#quote-header-info > section > span") %>%
  html_text() %>%
  as.numeric()
```

We get key statistics:

```r
page %>%
  html_node("#key-statistics table") %>%
  html_table()
```
Requests

- Every time you click on a website and data is updated, a request is being made.

Steps to web scraping:

1. Figure out request method of website:
   - Usually data split over different URLs.
   - Tables update with filters.
   - ...
2. Fetch the HTML, JSON, ... data of a website using a request.
3. Parse the data in a structured way.
4. Access/organize the data.

- Avoid 1 if interested only in scraping data from a single URL.
HTTP

• HTTP (Hypertext Transfer Protocol) enables communication between clients and servers.
• Works through a request-response protocol.
• Every time data is updated in browser, a request has been made.
• Most used HTTP request methods are GET and POST (although there are many others, such as HEAD, PATCH, PUT, ...).

Understanding requests is useful to scrape multiple websites/queries:
  • Prices on Craigslist.
  • Government press releases.
  • Flight data.

Before scraping, need to figure out:
  1. What type of request is being made?
  2. What are the parameters of the request/query?
• Most common HTTP request method.

• GET requests sent through URL.

• Look if/how URL changes as you change filters/search terms.

• Remove/add parameters in URL to see changes in data displayed.

• On every request there’s usually a “?” at the beginning of request, and a “&” between each key/value.
In JSTOR, search for “sargent” with publication dates starting in 1960 and ending in 1980:


Try to remove unnecessary filters/parameters until left with only necessary ones to load data.

Usually there’s limit on number of results displayed – multiple pages.

Go to “next” page and see how URL changes:


OR try to change “Display 10 results per page”
GET requests III

• Anatomy of GET request:

GET /library.html?Query=sargent HTTP/1.0 (optional headers)

URL Query string HTTP version

• Response (HTML):

HTTP/1.1 404 Not Found
Date: Mon, 15 Nov 2018 12:15:08 GMT
Server: Apache/2.2.14 (Win32)
Content-Length: 204
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "...">
<html><head>
title>404 Not Found</title>
</head><body>... </body></html>

• HTML code ready to use
POST requests not sent through URL ⇒ data displayed changes without URL changing.

- Sent through an HTML form with headers.
- Response usually in nicely-structured format (e.g. JSON).
- To inspect headers and response of request, go to Chrome's: DevTools >> Network >> XHR.
- Look through XHR requests for the ones that are pulling data:
• Anatomy of POST request:

```plaintext
POST /library.html HTTP/1.0
Content-Type: mime-type
Content-Length: number-of-bytes
(Queue string)
```

• Response is usually nicely-formatted data.
### GET vs. POST requests

<table>
<thead>
<tr>
<th></th>
<th>GET</th>
<th>POST</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>History</strong></td>
<td>Parameters saved in browser history</td>
<td>Parameters not saved in browser history</td>
</tr>
<tr>
<td><strong>Bookmark</strong></td>
<td>Can be bookmarked</td>
<td>Cannot be bookmarked</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
<td>Length restrictions (characters in URL)</td>
<td>No restrictions on data/parameter length</td>
</tr>
<tr>
<td><strong>Cache</strong></td>
<td>Can be cached</td>
<td>Cannot be cached</td>
</tr>
<tr>
<td><strong>Security</strong></td>
<td>Low – sent through URL</td>
<td>Higher – data not exposed in URL</td>
</tr>
</tbody>
</table>
GET vs. POST requests II

- GET: data has to be gathered directly from HTML:

```
• GET: data has to be gathered directly from HTML:
  <a href="/stable/2335694975search=yes&resultItemClick=true&searchText=sargent&searchText=sargent&3Bacc%3Dcn%26amp%3Bfc%3Doff%26amp%3Bgroup%3Dnone%26amp%3Bquery%3Dsargent" data-itemtype="Article" data-access="Yes" data-mboxname="search-journal-click,search-content-access">
      Agents as Empirical Macroeconomists: Thomas J. Sargent's Contribution to Economics
  </a>
```

- POST: data usually comes in structured way. E.g. JSON:

```
{ 
  "name":"John", 
  "age":30, 
  "cars":[ "Ford", "BMW", "Fiat" ]
}
```
Fetching the data 1: Python

- Libraries: requests, httplib, urllib

```python
import requests
URL = "http://maps.googleapis.com/maps/..."
location = "Philadelphia"
PARAMS = {'address': location}

r = requests.get(url = URL, params = PARAMS)
```

```python
import requests
API_ENDP = "http://pastebin.com/api/...
API_KEY = "123456"
data = {'api_key': API_KEY, 'api_opt': 'paste'}

r = requests.post(url = API_ENDP, data = data)
```
Fetching the data II: R

- Packages: httr, RCurl, rvest

```r
library(httr)
r <- GET("http://maps.googleapis.com/maps/...", request = list(address = "Mexico"))
```

```r
library(httr)
API_KEY = "123456"
r <- POST("http://httpbin.org/post", body = list(api_key = "123456", api_opt = 'paste'))
```

- Or if interested on a single URL:

```r
library(rvest)
mypage <- read_html("https://finance.yahoo.com/quote/MSFT")
```
• Recall that HTML/XML code comes in nested structure of tags:

```html
<!DOCTYPE html>
<html>
<head>
  <title>Your web page</title>
</head>
<body>
  <h1>Heading 1</h1>
  <p>Paragraph 1.</p>
</body>
</html>
```

• Many of those websites employ CSS (Cascading Style Sheets).

• Useful to find data within the code.
### Data on website:

#### HTML code:

```html
<table class="wikitable sortable jquery-tablesorter" style="text-align:right">
  <thead>
    <tr>
      <th>Rank</th>
      <th>Country (or dependent territory)</th>
      <th>Population</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>1</td>
      <td><a href="#">China<sup>Note 2</sup></a></td>
      <td>1,395,430,000</td>
    </tr>
    <tr>
      <td>2</td>
      <td><a href="#">India<sup>Note 3</sup></a></td>
      <td>1,340,140,000</td>
    </tr>
    <tr>
      <td>3</td>
      <td><a href="#">United States<sup>Note 4</sup></a></td>
      <td>328,252,000</td>
    </tr>
    <tr>
      <td>4</td>
      <td>Indonesia</td>
      <td>265,015,300</td>
    </tr>
  </tbody>
</table>
```

- Idea: extract the "1,395,430,000" from HTML
“A parser is a software component that takes input data (frequently text) and builds a data structure – often some kind of parse tree, abstract syntax tree or other hierarchical structure...”

- Use DOM (Document Object Model) to parse HTML.
- Take as input XML/HTML code and generate a tree.
- Functions used to access nodes in tree:
  - Root: returns root node.
  - Name: returns name of node.
  - Attributes: returns node attributes.
  - Parent: parent of a node.
  - Siblings: siblings of a node.
  - Value: value of node.
- Use XPath language (described later) to query nodes, extract data.
In Python, library BeautifulSoup:

```python
import requests
from bs4 import BeautifulSoup

URL = "https://www.wikipedia.org/

r = requests.get(url = URL)
soup = BeautifulSoup(r.text)
```

In R, library XML:

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

tree = htmlTreeParse(html)
```

Data stored as an XML object
Accessing the data: XPath I

- Once we have parsed HTML into an XML object, we need to locate specific nodes with data.
- **XPath (XML Path Language):** language to query and access XML elements.
- Path-like syntax to navigate through nodes.
- Expressions that return nodes:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>node</code></td>
<td>Selects nodes with name “node”</td>
</tr>
<tr>
<td><code>/node</code></td>
<td>Selects root element “node”</td>
</tr>
<tr>
<td><code>//node</code></td>
<td>Selects all elements of type “node”</td>
</tr>
<tr>
<td><code>node[@attrname]</code></td>
<td>Selects node with attribute named “attrname”</td>
</tr>
<tr>
<td><code>node[@attrname='name']</code></td>
<td>Node with “attrname” and value ‘name’</td>
</tr>
</tbody>
</table>
Accessing the data: XPath II

```
/movies/movie[@lang='spa']/title
```

Diagram:

- `movies`
- `movie mina='126' lang='en'`
  - `title`
  - `director`
  - `year` 1998
  - `genre` drama
- `movie mina='106' lang='spa'`
  - `title`
  - `director`
  - `year` 2001
  - `genre` drama
- `Good Will Hunting`
- `Van Sant`
- `Y tu mama tambien`
- `Alfonso`
- `Cuaron`
XPath in Python

- Many functions, depending on parsing package.

- Using lxml:

```python
from lxml import html
import requests

page = requests.get('http://econpy.pythonanywhere.com/...')

import requests

from lxml import html

tree = html.fromstring(page.content)

buyers = tree.xpath('//div[@title="buyer-name"]/text()')
prices = tree.xpath('//span[@class="item-price"]/text()')
```
Main function to access nodes of XML tree using XPath: `getNodeSet(tree, path)`

- `tree` is the XML tree stored.
- `path` is the XPath path of the node of interest.

In R:
```
getNodeSet(movies_xml, "//movies/movie")
getNodeSet(movies_xml, "//title")
getNodeSet(movies_xml, "//movie[@lang='eng']")
```