Practical Web Scraping for Economists

Javad Shamsi

London School of Economics

June, 2024

Web Scraping Applications in Economics

- **Collect Pricing Data:** Analyze price fluctuations, market competition, or inflation trends. (Baye and Morgan, 2009; Boivin, Clark, and Vincent, 2012; Cavallo and Rigobon, 2016)
- **Monitor Job Listings:** Study labor market dynamics, unemployment trends, or wage changes. (Deming and Kahn, 2018; Kuhn and Shen, 2013; Kureková, Beblavỳ, and Thum-Thysen, 2015)
- **Gather Public Discourse Data:** Analyze public sentiment or policy content using Media/Twitter/etc. (Shamsi, 2024)
- **Compile Real Estate Information:** Study housing markets, pricing trends, and geographic economic disparities. (Bricongne, Meunier, and Pouget, 2023; Halket and Custoza, 2015)

Web Scraping vs. APIs

- **Web Scraping:** Extracts content from web pages when no direct data access is available.
- **APIs:** Directly request data from servers; more reliable and efficient than scraping when available.
- **SERP APIs:** Retrieve search engine results (e.g., Google) without scraping, avoiding layout changes and CAPTCHAs.

Ethical and Legal Considerations

- **Respect Terms of Service:** Many websites have specific guidelines for data usage. Scraping may violate these terms, so it's important to review them before starting.
- **Server Load:** Implement rate limiting and respect `robots.txt` files to avoid overwhelming servers. Use polite scraping practices by spacing requests.
- **Intellectual Property:** Some website content is protected by copyright, and scraping such data without permission can lead to legal issues.

Disclaimer: Please consult your department's IRB or ethics office before conducting web scraping. These materials reflect personal methods from past projects. There are certainly other, more technical resources and textbooks available.

Types of Web Scraping: Static vs. Dynamic Pages

- **Static Pages:**
  - Content is directly available in the page's source code.
  - Easier since all data is loaded once the page is accessed.
  - Tools like `requests` and `BeautifulSoup` are effective.
- **Dynamic Pages:**
  - Data is loaded asynchronously using JavaScript after the initial page load.
  - Requires simulating user interactions (e.g., clicks) or waiting for JavaScript execution.
  - Tools like `Selenium` or `Scrapy+Splash` are needed to handle these pages.

Key Challenge: Dynamic pages require handling delays and JavaScript-rendered content, making scraping more complex.

Static Web Scraping: Overview

Static Web Scraping: Extracting data from web pages where content is fixed once the page is loaded.

- All data is delivered in the initial HTML response.
- Simpler than dynamic scraping, but still requires planning.

Steps in Static Web Scraping: Sending HTTP Requests

Step 1: Sending an HTTP Request

- Send a request to the server using methods like `GET` or `POST`.
- **Tool:** Use `requests` library in Python.
- **Challenge:** Ensure valid response, avoid bot detection, handle CAPTCHAs.

Example:

```
import requests
response = requests.get('http://example.com')
if response.status_code == 200:
    html_content = response.content
```

Steps in Static Web Scraping: Parsing the HTML

Step 2: Parsing the HTML

- Identify and extract relevant data from the HTML.
- **Tool:** Use `BeautifulSoup` to navigate HTML.
- **Challenge:** Understanding complex or inconsistent HTML structures.

Example:

```python
from bs4 import BeautifulSoup
soup = BeautifulSoup(html_content, 'html.parser')
titles = soup.find_all('h1')
```

Challenges in Static Scraping: Identifying HTML Elements

Key Challenge: Identifying the Right HTML Elements

- Websites have extraneous information like ads or sidebars.
- **Approach:** Use browser dev tools to inspect HTML structure.
- Use **CSS selectors** or **XPath** for precise targeting.

Example:

```python
products = soup.select('div.product-name')
for product in products:
    print(product.text)
```

Challenges in Static Scraping: Paginated Content

Key Challenge: Navigating Paginated Content

- Many websites split content across multiple pages.
- Identify URL patterns and loop through pages to scrape all data.

Example:

```python
base_url = 'http://example.com/page/'
for page_number in range(1, 10):
    url = base_url + str(page_number)
    response = requests.get(url)
    soup = BeautifulSoup(response.content, 'html.
        parser')
```

Challenges in Static Scraping: Rate Limiting and Captchas

Key Challenge: Rate Limiting and Captchas

- Websites may throttle requests or use CAPTCHAs to block bots.
- **Approach:**
    - Use delays between requests to avoid overwhelming servers.
    - Set user-agent headers to mimic real browser behavior and reduce detection.
    - Employ proxies or rotate IPs to avoid being blocked.

Example:

```python
import time
headers = {'User-Agent': 'Mozilla/5.0'}
base_url = 'http://example.com/page/'
for page_number in range(1, 10):
    url = base_url + str(page_number)
    response = requests.get(url, headers=headers)
    ...
    time.sleep(2)  # Wait 2 seconds between
        requests
```

Challenges in Static Scraping: Extracting Tables or Lists

Key Challenge: Extracting Data from Tables or Lists

- Tables and lists are commonly used for structured data.
- Use `BeautifulSoup` to extract rows and cells from tables.
- For well-structured tables, use `pandas.read_html()` for direct extraction.

Example:

```
import pandas as pd
tables = pd.read_html('http://example.com')
df = tables[0]  # First table on the page
```

Data Cleaning and Preprocessing

Cleaning and Preprocessing the Data

- Extracted data is often messy, with extra characters or tags.
- Use string methods or regular expressions to clean the data.
- **Challenge:** Inconsistent formatting across pages.

Example:

```python
import re
clean_text = re.sub(r'\s+', '-', text)  # Replace
    multiple spaces
```

Best Practices for Static Web Scraping

Best Practices:

- Respect `robots.txt` and website terms of service.
- Store data in structured formats (CSV, JSON, SQL).
- Make code modular and scalable for reuse across websites.
- Regularly test and debug scraping scripts to handle site changes.

Dynamic Web Scraping: Overview

Dynamic Web Scraping: Involves extracting data from websites where content is loaded dynamically via JavaScript after the initial page load.

- Essential for websites using JavaScript frameworks (React, Angular, Vue).
- Required for websites using infinite scrolling or needing user interaction.

Steps in Dynamic Web Scraping: Simulating a Browser Environment

Step 1: Simulating a Browser Environment

- Dynamic content requires executing JavaScript, so fetching the HTML using `requests` won't suffice.
- **Tool:** Use `Selenium` for browser automation to open a real browser and interact with the page like a user.

Example:

```
from selenium import webdriver
driver = webdriver.Chrome()
driver.get('http://example.com')
```

Steps in Dynamic Web Scraping: Waiting for JavaScript to Load

Step 2: Waiting for JavaScript to Load

- Dynamic pages take time to load certain elements.
- **Tool:** Use Selenium's `WebDriverWait` to pause until specific elements are loaded.

Example:

```python
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

element = WebDriverWait(driver, 10).until(
    EC.presence_of_element_located((By.ID, "content"))
)
```

Steps in Dynamic Web Scraping: Interacting with the Webpage

Step 3: Interacting with the Webpage

- Simulate user interactions like clicking buttons or scrolling.
- **Tool:** Selenium can simulate these actions to load more data or reveal hidden content.

Example:

```
    # Clicking a button
    button = driver.find_element(By.ID, 'load-more')
    button.click()

    # Scrolling to the bottom of the page
    driver.execute_script("window.scrollTo(0, document.
        body.scrollHeight);")
```

Steps in Dynamic Web Scraping: Extracting Data from Rendered Content

Step 4: Extracting Data from Rendered Content

- After interactions, extract data by accessing the fully rendered page.
- **Tool:** Use Selenium with `BeautifulSoup` to parse the content and extract data.

Example:

```
from bs4 import BeautifulSoup
soup = BeautifulSoup(driver.page_source, 'html.
    parser')
data = soup.find_all('div', class_='item')
```

Steps in Dynamic Web Scraping: Dealing with Infinite Scrolling

Step 5: Dealing with Infinite Scrolling

- Some sites load more content dynamically as you scroll.
- **Approach:** Use `Selenium.execute_script()` to scroll to the bottom and capture newly loaded content.

Example:

```
while True:
    driver.execute_script("window.scrollTo(0, document.body.scrollHeight);")
    time.sleep(3)  # Wait for new content to load
```

Challenges in Dynamic Web Scraping: Handling JavaScript-Rendered Content

Key Challenge: Handling JavaScript-Rendered Content

- JavaScript renders content after the initial page load, so traditional scraping methods don't work.
- **Approach:** Use Selenium to render the page fully.
- **Challenges:**
  - Frameworks like React or Angular may have delayed element loading.
  - Selenium can be resource-intensive and slow for large-scale scraping.

Challenges in Dynamic Web Scraping: Timing and Delays

Key Challenge: Timing and Delays

- Dynamic elements load at different times, making proper timing critical.
- **Approach:**
  - Use explicit waits with `WebDriverWait` for specific elements.
  - Set general wait times using implicit waits to balance speed and completeness.

Example:

```
driver.implicitly_wait(10)  # Sets a 10-second max wait for all elements
```

Challenges in Dynamic Web Scraping: Simulating User Behavior

Key Challenge: Simulating User Behavior

- Websites may require interaction, such as clicking buttons or handling pop-ups, to reveal data.
- **Approach:** Use Selenium to simulate user interactions like clicks and form submissions.
- **Example:**

```
# Handle a pop-up modal
modal = driver.find_element(By.CLASS_NAME,
    'close-modal')
modal.click()
```

Challenges in Dynamic Web Scraping: Handling CAPTCHAs and Anti-Scraping Measures

Key Challenge: CAPTCHAs and Anti-Scraping Measures

- Many dynamic websites use CAPTCHAs and rate-limiting to prevent scraping.
- **Approach:**
  - Use third-party services for CAPTCHA solving or avoid triggering CAPTCHAs.
  - Rotate proxies and use User-Agent spoofing to simulate a real user.

Example:

```
from selenium.webdriver.chrome.options import
    Options
options = Options()
options.headless = True
options.add_argument('user-agent=Mozilla/5.0')
driver = webdriver.Chrome(options=options)
```

Closing the Browser in Selenium

Use of `driver.quit()` or `driver.close()`:

- It is important to close the browser after completing the task to free up system resources.
- Without closing, the browser will continue running in the background, consuming memory and CPU.
- `driver.quit()` ends the entire browser session, while `driver.close()` closes the current window.

Example:

```
driver.quit()  # Close the browser and end the session
```

Best Practices for Dynamic Web Scraping

Best Practices:

- **Use APIs:** If available, APIs provide structured data directly.
- **Optimize Scraping Speed:** Use headless browsers and disable unnecessary features like images.
- **Use Proxies and Rotate User Agents:** Avoid being blocked by rotating IPs and user agents.
- **Error Handling:** Implement error handling and logging to prevent script crashes.
- **Respect Website Policies:** Follow the website's `robots.txt` and terms of service.

Scheduled Web Scraping

Scheduled scraping is crucial for researchers to:

- **Price Monitoring:** Track price changes and market trends on e-commerce platforms.
- **Labor Market Studies:** Analyze wage dynamics and employment trends from job boards.
- **Housing Market Analysis:** Monitor property prices and rent trends from real estate sites.
- **Consumer Sentiment:** Scrape social media and reviews to study behavior and firm reputation.
- **Macro-Economic Indicators:** Collect real-time economic data like GDP and unemployment rates.

Scheduling and Automating Web Scraping Tasks

There are several methods to schedule and automate web scraping efficiently:

- **Cron Jobs (Linux/macOS) or Task Scheduler (Windows):** Run scripts at specific intervals (daily, weekly) using system tools like Cron or Task Scheduler.
- **Python Libraries (e.g., schedule, APScheduler):** Schedule tasks directly in Python code. `schedule` runs tasks at intervals, while `APScheduler` provides advanced time-based scheduling.
- **Cloud Services (e.g., AWS Lambda, Google Cloud Functions):** Run scraping scripts in the cloud without managing servers. Trigger tasks based on time or events.
- **Others:** Docker containers, CI/CD pipelines, or third-party automation platforms like Zapier or Integromat.

Questions?

Thanks for your attention!
If you have any questions, concerns, or just want to chat about web scraping (or coffee), feel free to reach out!

m.shamsi@lse.ac.uk

(No bots, please!)

Further Reading on Web Scraping

- Boegershausen et al., 2022
- Śpiewanowski, Talavera, and Vi, 2022
- Jarmin, 2019
- Edelman, 2012

References I

Baye, Michael R and John Morgan (2009). “Brand and price advertising in online markets”. In: *Management Science* 55.7, pp. 1139–1151.

Boegershausen, Johannes et al. (2022). “Fields of gold: Scraping web data for marketing insights”. In: *Journal of Marketing* 86.5, pp. 1–20.

Boivin, Jean, Robert Clark, and Nicolas Vincent (2012). “Virtual borders”. In: *Journal of International Economics* 86.2, pp. 327–335.

Bricongne, Jean-Charles, Baptiste Meunier, and Sylvain Pouget (2023). “Web-scraping housing prices in real-time: The Covid-19 crisis in the UK”. In: *Journal of Housing Economics* 59, p. 101906.

References II

Cavallo, Alberto and Roberto Rigobon (2016). "The billion prices project: Using online prices for measurement and research". In: *Journal of Economic Perspectives* 30.2, pp. 151–178.

Deming, David and Lisa B Kahn (2018). "Skill requirements across firms and labor markets: Evidence from job postings for professionals". In: *Journal of Labor Economics* 36.S1, S337–S369.

Edelman, Benjamin (2012). "Using internet data for economic research". In: *Journal of Economic Perspectives* 26.2, pp. 189–206.

Halket, Jonathan and Matteo Pignatti Morano di Custoza (2015). "Homeownership and the scarcity of rentals". In: *Journal of Monetary Economics* 76, pp. 107–123.

Jarmin, Ron S (2019). "Evolving measurement for an evolving economy: thoughts on 21st century US economic statistics". In: *Journal of Economic Perspectives* 33.1, pp. 165–184.

References III

Kuhn, Peter and Kailing Shen (2013). "Gender discrimination in job ads: Evidence from china". In: *The Quarterly Journal of Economics* 128.1, pp. 287–336.

Kureková, Lucia Mỳtna, Miroslav Beblavỳ, and Anna Thum-Thysen (2015). "Using online vacancies and web surveys to analyse the labour market: A methodological inquiry". In: *IZA Journal of Labor Economics* 4, pp. 1–20.

Shamsi, Javad (2024). "Immigration and political realignment". In: *Available at SSRN 4748438*.

Śpiewanowski, Piotr, Oleksandr Talavera, and Linh Vi (2022). "Applications of Web Scraping in Economics and Finance". In: *Oxford Research Encyclopedia of Economics and Finance*.