Revista Romana de Interactiune Om-Calculator 11 (1) 2018, 63-75

Modern techniques of web scraping for data scientists

Mihai Gheorghe, Florin-Cristian Mihai, Marian Dârdală

The Bucharest University of Economic Studies
6 Piata Romana, 1st district, Bucharest, 010374 Romania
E-mail: mihai.gheorghe@gdm.ro, fcmihai@gmail.com, dardala@ase.ro

Abstract. Since the emergence of the World Wide Web an outstanding amount of information has become easily available with the use of a web browser. Harvesting this data for scientific purposes isn't feasible to be done manually and has evolved into a distinct new field, Web Scraping. Although at the beginning automatically collecting data in a structured format was at hand with any programming language able to process a text block, which was the HTML response of a HTTP request, with the latest evolution of web pages, complex techniques to achieve this goal are needed. This article identifies problems a data scientist may encounter when trying to harvest web data, describes modern procedures and tools for web scraping and presents a case study on collecting data from the Bucharest's Public Transportation Authority's website in order to use it in a geo-processing analysis. The paper is addressed to data scientists with little or no prior experience in automatically collecting data from the web in a way that doesn't require extensive knowledge of Internet protocols and programming technologies therefore achieving rapid results for a wide variety of web data sources.

Keywords: human-computer interaction, web scraping, data harvesting, content mining.

1. Introduction

A web page is basically the interpretation a web browser gives to an HTTP success response which includes an HTML document. The HTTP response, which is plain text, might drive the browser to initiate other HTTP requests for resources such as images, style sheets or client-side scripts. The information is visually rendered by the browser for the human user to assess. Therefore, in order to collect relevant data from a web page, we can either manually identify and store it in a structured data format or we can programmatically process the response the browser receives, extract the information and push it to the desired data container for storage and analysis.

For Big Data scenarios, manually collecting the information is both unfeasible and prone to copy/paste errors. *Web scraping* usually refers to the

automated process of data transfer between a web page and a structured data format implemented using a toolkit called a web robot, scraper or crawler. Despite sharing some of the same processes and challenges, *web archiving* and *web scraping* are two distinct fields. There is a growing number of non-profit organizations, university-backed initiatives, and government data services currently working to archive web sources for various research purposes such as The Internet Archive, PANDORA, and the Internet Memory Foundation. Unlike archiving, web scraping selectively stores data, ignoring, for instance, images, multimedia content and style sheets when dealing with a text-driven project.

Apart from the technical challenges, an impediment to easy web harvesting is the legit effort of website owners to limit or completely block automated access to their data. Confidentiality, security or business related concerns may lead to this decision. For instance, without being limited to this, web scrapers can be used for real-time monitoring of competitor prices, which can generate business decisions with significant impact for the target of automated data extraction. They can as well generate the loss of advertisement revenue or increase costs with server infrastructure.

Given their ability to initiate a large number of requests over a short period of time, web scrapers may add significant server load to the detriment of the usability degree for regular human users. This study identifies methods used to prevent web scraping and describes techniques to completely or partially avoid these methods.

Non-malicious data retrieval can be separated into two categories: *formally supported* and *informally permitted* (Black, 2016). Formally supported data retrieval is usually performed through Application Programming Interfaces (API) or other frameworks designed explicitly to handle and respond to automated data requests, while the second is based on techniques which involve downloading the same resources used to render web pages on-screen for human users and searching through their HTML code for specific types of information.

Although in most cases, the information rendered by a browser seems visually organized, thus making it easy for a human operator to identify certain logical structures, HTML is an unstructured data format and extracting relevant and consistent data generates a couple of challenges. In order to overcome these, various strategies have been developed. These include *text pattern matching*, HTML and DOM (Document Object Model) *parsing* for textual content and *Computer Vision* and *Machine Learning* for

images and multimedia content. Although web crawling and indexing is a large-scale phenomenon, which includes core features for Search Engine giants like Google Search or Microsoft Bing, the current study is focused only on the textual content extraction challenges and techniques for Data Scientists.

2. Modern web applications and implications on scraping

Present-day websites are modeled around higher user expectations and greater performance metrics than ever before. Modern web apps are desired to have 100% uptime, be available from anywhere in the world, and usable from virtually any device or display size. Web applications must be secure, flexible and scalable to accommodate traffic spikes in demand. More than this, complex scenarios should be taken care of by rich user interactions, such as single page apps, built on the client using JavaScript, and communicating effectively through web APIs (Hernanez et al., 2018; Zheng et al., 2007). These developments have influenced the methods scraping can be implemented.

2.1 Advanced user interfaces of web applications

In the first years of the Internet, web pages were simple static HTML documents or generated HTML code blocks using server-side programming languages. For publicly available pages, the main effort in the web scraping process was held by initiating the HTTP request and the interpretation of the response containing the HTML. However, in the recent years, due to a variety of concerns, including security, scalability and software maintainability, many web applications have been built around an evolved architecture which separates data and presentation layers (Angular, 2018; Paterno, 1999; React, 2018).

The HTML in the response is usually created for the visual template of the page, while relevant data, subject to data extraction, is dynamically filled in using advanced front-end techniques among some of the most popular are Angular (developed and maintained by Google) and React (developed and maintained by Facebook).

These are JavaScript libraries designed to help create dynamic user interfaces which pull data from API architectures. The following example,

written using React, depicts the explicit code the browser receives and what the HTML code looks after the browser has rendered it.

```
<div id="myReactNewsApp"></div>
<script type="text/babel">
  class Headliner extends React.Component {
    render() {
      return <h1>{this.props.title}</h1>
    }
  }
  ReactDOM.render(<Headliner title="News Headline" />,
document.getElementById('myReactNewsApp'));
</script>
```

Code example 1. Explicit source code for retrieving the Title of an article using React

```
<div id="myReactNewsApp">
  <h1>News Headline</h1>
</div>
```

Code example 2. HTML code after the page has been rendered by the browser

It is easily understood that by interpreting the HTTP response alone through a web scraping program, it would be counter-intuitive to identify the title in our example, which otherwise would have been extracted using a simple JavaScript regular expression: regex = /<h1>(.*?)<\/h1>/g;

For scenarios where data is pulled from an API response after the initial HTML was sent to the browser, this would be impossible. Also, in order to build a modern web page, a browser makes on average 75 HTTP requests [10]. Building a scraper to deal with this amount of complexity would be highly inefficient.

Also, there are circumstances where social media posts, blog posts, products or other abstract data types are continuously revealed on preset user interactions, such as *scroll-down* on the bottom of the page *(infinite scrolling)* eliminating the need for pagination. Therefore, *parsing the DOM* (Document Object Model), which may be rebuilt after each user action, rather than the initial HTML, would resolve these shortcomings. However, the DOM is a structure generated by the browser after it processes all the server responses and also interprets all the client side scripts associated with a web page. This means that, apart from the core scraping solution, a web browser needs to be

included to accommodate with modern websites.

Compared to conventional web scraping, having a browser to create a DOM for each page requires additional computing resources, thus processing times. For instance, loading Facebook's first page, without being authenticated, measured with Gtmetrix (a public web performance analysis service which was set to use the Google Chrome browser) resulted in 0.8 seconds since the initial request until the DOM was first constructed, while the final DOM was rendered in 2.1 seconds (GTMetrix, 2018).

2.2 Methods for preventing automated data extraction from websites

As previously mentioned, in some cases, web scraping is not encouraged by owners of websites with public access for human users. Therefore, a handful of techniques have been developed in order to minimize the impact of such practices.

Voluntary compliance is usually implemented through the *robots exclusion protocol*, which is a standard meta-document with simple syntax (*robots.txt*), placed in the top-level directory which website owners use to instruct web robots (typically search engine crawlers) if and how often they prefer to have their pages indexed. The following example indicates Google Bot to crawl the pages at intervals of 120 seconds, allows indexing of the /ads/public/ directory while forbidding indexing of /ads/. It also forbids all crawlers to index the /archive/ folder.

```
User-agent: googlebot
Crawl-delay: 120
Allow: /ads/public/
Disallow: /ads/

User-agent: *
Disallow: /archives/
```

Code example 3. A robots.txt file

Human vs. Machine tests have been introduced to block access for automated scripts to certain web pages. Out of these, *CAPTCHA* (which stands for Completely Automated Public Turing Test To Tell Computers and Humans Apart) is the most acknowledged. Initially, CAPTCHA tests

displayed distorted bitmap images of words or simple expressions, prompting the user to submit the text representation in order to gain access to information. The images were generated in a way that regular OCR (Optical Character Recognition) algorithms were not able to solve the test. However, along with advancements in Machine Learning, the efficiency of these tests started to drop, culminating in 2012 with automated scripts that could solve the tests at rates higher than 99% (Burszstein et al., 2011; Stillwalker, 2018). Also, using cheap human labor to solve batches of tests as a service started to be used.

Increasing the complexity of the tests led to criticism regarding accessibility (edwards & Bagozzi, 2000), and although alternatives for visually impaired users were introduced, in 2013 reCAPTCHA (a free service developed and maintained by Google) began implementing behavioral analysis. The current version, called NoCAPTCHA, prompts the user to move the mouse pointer to a certain checkbox, which results in less interaction friction and performs a back-end advanced risk analysis based on various criteria such as speed and trajectory of the movement to determine if the user is a human or a computer program. For high-risk scores, conventional CAPTCHA tests might follow before granting access (Google, 2014).

Obfuscation of sensitive data, such as contact details, by converting it to images is a practice which indeed pushes web scraping solutions to have advanced architectures to include OCR components and to approach more educated strategies on the target website, but also comes with notable drawbacks. Real-time generating images is a greedy computing task. Responding with images instead of plain text has a significant impact on network bandwidth and also adds more HTTP requests for retrieving a web page. Layout concerns in regard to the web page's responsiveness to a wide variety of displays also become apparent.

Monitoring the server access logs is a method for identifying excess traffic, unlikely access patterns for a human user or if the user-agent header in the HTTP request does not have the correct format for human-operated web browsers (HTTP, 2018). The monitoring process can be performed either manually or automated through various firewall applications. IPs initiating requests falling under these circumstances can be temporary or permanently blocked. However, there are means to bypass these scenarios by implementing *crawl politeness* features, make requests with *fake user-agents* and imply *proxy services* for distributed IPs.

Granting access to certain web pages after mandatory *password*

authentication increases the required complexity of the scraping solution, even if credentials are known. Authentication involves HTTP redirects, sessions, and cookies which are difficult to manage without a web browser. Consequently, this makes the information virtually unavailable for broadly used scrapers and crawlers like the search engines bots. Adding an extra layer of security, such as multiple factor authentication (SMS confirmation codes for example), or CAPTCHA tests makes completely automated scrapers not worth considering for usual data science projects. A multi-step process where a basic human input is needed is thus more feasible in these scenarios.

3. Proposed Scraping Architecture and Techniques

Prior to harvesting data from a web page or a set of similar web pages, it is mandatory to manually analyze its structural layout in order to build the logic of data extraction. Following this analysis, key components such as HTML tags, CSS and DOM selectors, meta-tags specific to semantic web approach or API requests that return data in a structured format should be identified. Modern web browsers feature tools which facilitate this step. With Google Chrome, to identify a CSS selector for a particular text element, the easiest way is by right-clicking on it and choosing *Inspect*. The selector will be listed in the *DevTools* console.

3.1 Choosing the programming language, libraries, and proper environment

Almost every major programming language provides means for performing web scraping tasks. Notable examples are libraries such as *BeautifulSoup* for Python, *rvest* for R, *cURL* for PHP or *jsoup* for Java. However, although fetching and parsing regular HTML pages is easily achievable, and processing performance is remarkable, these solutions were not designed to accommodate with websites featuring modern user interfaces as depicted in Section 2.

Automating a web browser is a more viable approach through which concerns like HTTP redirects, cookies or dynamic changes to the page content and structure are not the task of data scientists. Selenium is an acknowledged web automation tool which was primarily designed for automated web testing. Selenium is an open-source software compatible with

Windows, Linux, and MacOS and through its WebDriver component is able to automate all major web browsers by starting an instance, sending commands and retrieving the result. Selenium is scriptable through a wide variety of programming languages: Java, Python, PHP, Ruby, C# (Selenium, 2018). For the web harvesting area, a data scientist can thus create a scraping program in a convenient language with the scope of generating Selenium commands which automates a browser to fetch arrays of web pages, evaluate, process and export the results. *Figure 1* depicts a proposed architecture to accomplish the objective.

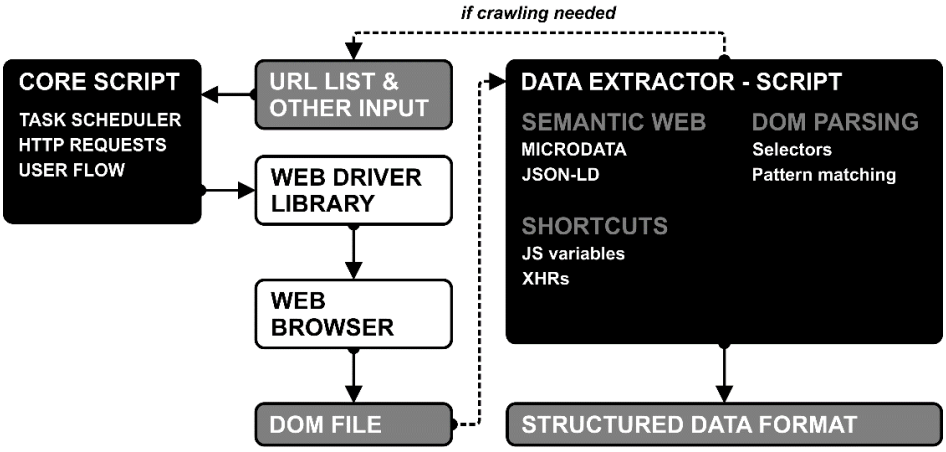

Figure 1. The proposed architecture for web scraping textual content

3.2 Scraper strategy

Constructing HTTP requests via the scraper isn't required in a configuration that uses a browser, especially a regular one. Nevertheless, if the script is using a headless browser and the target webpage should render as if were a human operator, changing the user-agent is an option. *Code example 4* depicts how to get the DOM output for www.google.com, through the command line, having the user-agent set as a regular Google Chrome, although the headless version interprets the page.

```
C:\Program Files (x86)\Google\Chrome\Application>chrome
--headless --disable-gpu --enable-logging --user-
agent="Mozilla/5.0 (X11; Linux x86_64)
AppleWebKit/537.36 (KHTML, like Gecko)
```

```
Chrome/60.0.3112.50 Safari/537.36" --dump-dom
https://www.google.com/
```

Code example 4. Via command line, the --user-agent overwrites the default user agent the request is sent with, while --dump-dom flag prints document.body.innerHTML to stdout

Simulating user flow is facilitated by straightforward methods produced by the automation frameworks. A script can mimic mouse movements, scrolls and clicks, keyboard typing actions, or simply add time delays specific to human behavior. *Code example 5* showcases the use of Puppeteer's *keyboard.type* and *click* methods to go through a basic login form. Prior knowledge about the page's structure is obviously needed in order to assign the correct DOM selectors such as #login > form > div.auth-form-body.mt-3 > input.btn.btn-primary.btn-block for the submit button in our example.

```
const USERNAME_SELECTOR = '#login_field';
const PASSWORD_SELECTOR = '#password';
const BUTTON_SELECTOR = '#login > form > div.auth-form-
body.mt-3 > input.btn.btn-primary.btn-block';
await page.click(USERNAME_SELECTOR);
await page.keyboard.type(CREDS.username);
await page.click(PASSWORD_SELECTOR);
await page.keyboard.type(CREDS.password);
await page.click(BUTTON_SELECTOR);
await page.waitForNavigation();
```

Code example 5. Using Puppeteer and Headless Chrome for authenticating through a login form, simulating human user interaction

A key component of a scraping system is the *task scheduler*. In our proposed architecture, the task scheduler is the part of the script that initiates page loading methods after fetching URLs from the input. In order to simulate human behavior or just to take into account a *crawling politeness policy*, which is a practice meant to save server overload, time delays are usually inserted. If the input URL list is dynamically updated as a consequence of the data retrieved through the scraping process, the system becomes a *web crawler*.

3.3 Extracting Data

Once a web page is fetched and interpreted by the browser, the DOM

representation becomes available. In a web scraping architecture which includes a browser, data can be thus, extracted immediately, or the document can be stored as XML for later processing. *DOM parsing* is the technique used to navigate through the tree structure of the model in search of pre-identified *nodes* in order to extract target data *elements*. Selenium and Puppeteer are bundled with methods to perform this kind of tasks. *Code example 6* produces the following effects: it identifies the selector for a target name attribute, it stores the data in a string variable, it identifies the "footer" section and commands the browser to scroll to it and click on the "next page" element, with a 3 seconds delay between.

```
browser = webdriver.Edge()
browser.get('https://www.example.com')
Name =
browser.find_element_by_css_selector('span[itemprop
=
"name"]').get_attribute('innerHTML').replace('\n','
')
browser.find_element_by_class_name('footer').locati
on_once_scrolled_into_view
time.sleep(3)
Next =
browser.find_element_by_class_name('pagination-
next').find_element_by_tag_name('a')
Next.click()
```

Code example 6. Selenium and Microsoft Edge of retrieving the name attribute from a paginated web source

Semantic web approaches are usually implemented in websites which informally permit web scraping. *Schema.org* is an initiative of Google, Microsoft, Yahoo and Yandex, yet maintained by an open community, which proposes a meta-vocabulary meant to increase the level of structured data over the Internet. It can be included in the body web pages or with data transfer processes such as JSON Linked Data. With over 10 million websites using the schema.org vocabulary (Schema, 2018), *unsupervised crawling* processes can be performed. One veritable application is the Google Knowledge Graph which aggregates data from different web sources which share the same syntax.

Nevertheless, a good analysis of the target page can reveal *timesaving methods* to harvest data. Websites with features for continuous loading items,

such as e-commerce platforms or social networks often use XMLHttpRequests (XHRs) to retrieve information from APIs. Isolating XHRs significantly reduces fetching and processing times while saving network bandwidth which would have been otherwise consumed for resources unused in the process.

Once fetched and interpreted, data can be pushed to a structured data format using conventional means supported by the chosen programming language.

Case Study on Bucharest's Transportation Authority's website

The surface public transportation means in Bucharest count 26 light rails, 16 trolleybuses and 71 bus routes with approximately 2568 geographically dispersed stations. A geo-processing analysis was required for a research on identifying areas with public infrastructure issues.

The information about the stations, including 6 digit precision longitude and latitude coordinates, is publicly available on the Bucharest's Transportation Authority's website – www.ratb.ro on three distinct sections, with sub-sections for each route. However, the coordinates are placed as meta-data and are not displayed to the human user, which is required to inspect each of the 113 pages for the information.

Manually collecting the needed data would have been time consuming, prone to data errors and wouldn't have coped with future changes to the public transportation network in case subsequent simulations were needed. Therefore, based on the proposed architecture, a web scraper was built using Python, Selenium and Firefox browser. When executed, the process of harvesting data consumed less than 5 minutes and had a .csv with the retrieved coordinates and type for each station, output which was then imported in ArcGIS, which is a GIS (Geographical Information System) software suite, and symbolized over the map of Bucharest for the required analysis.

Figure 2 represents the transportation stations (the left side) on Bucharest's map using the data which was automatically collected and a preliminary geo-processing analysis result – a heatmap (on the right), both created using ArcGIS. The heatmap reveals high or low capacity public transportation areas which are easy to visualize given the color ramp, an analysis which otherwise would had been difficult to achieve using other means.

Figure 2. Public Transportation stations in Bucharest and a heat-map representation

Conclusions

Compared to conventional parsing of HTTP responses, the proposed scraping architecture and techniques have the advantage of dealing with modern websites in a more practical manner, particularly for users who are not proficient in server administration and Internet protocols. Nevertheless, this comes at the cost of *increased processing times*, *low portability* (a scraper built for a particular dynamic website is unlikely to be operational for a different one) and *misspending of networking resources* (in order to mimic a normal human behavior, the browser loads elements such as images, multimedia content and styling scripts which are unusable for achieving the desired goal).

In this study, we have identified, both through reviewing the academic literature and assessing industry products and techniques, common barriers a data scientist may encounter when performing web scraping and suggested methods to overcome them. Code examples were provided in order to stress out the facile manner the architecture can be used by a data scientist.

A case study to reveal the practical use of the proposed architecture has been included as well.

Future research is focused on identifying means to improve the overall model, especially information retrieval speed and its ability to cope with very large data sets.

References

Angular - One framework. Mobile & desktop, July 2018, https://angular.io/

Black, M.L., 2016. The World Wide Web as Complex Data Set: Expanding the Digital Humanities into the Twentieth Century and Beyond through Internet Research. International Journal of Humanities and Arts Computing, 10(1), pp.95-109.

Bursztein, E., Beauxis, R., Perito, H., Paskov, D., Fabry, C., Mitchell, J.C., 2011. "The failure of noise-based non-continuous audio captchas". IEEE Symposium on Security and Privacy (S&P).

Edwards, J., Bagozzi, R., 2000, On the nature and direction of relationship between constructs and measures. Psychological Methods 5(2), 155-174.

Getting Started with Headless Chrome: https://developers.google.com/web/updates/2017/04/headless-chrome

/Google Security Blog, December 2014, https://security.googleblog.com/2014/12/are-you-robot-introducing-no-captcha.html

GTmetrix – Performance report for facebook.com, July 2018, https://gtmetrix.com/reports/facebook.com/PdLhObgQ

Document Object Model (DOM), July 2018, https://www.w3.org/DOM/#what

Hernandez-Suarez, A., Sanchez-Perez, G., Toscano-Medina, K., Martinez-Hernandez, V., Sanchez, V. and Perez-Meana, H., 2018. A Web Scraping Methodology for Bypassing Twitter API Restrictions. arXiv preprint arXiv:1803.09875.

HTTP Archive, July 2018, https://httparchive.org/reports/state-of-the-web

Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content, IETF, The Internet Society, June 2014, https://tools.ietf.org/html/rfc7231#section-5.5.3

Paternò, F. Model based evaluation of interactive applications. Springer Verlag, 1999

PhantomJS - Scriptable Headless Browser, July 2018, http://phantomjs.org/

React - A JavaScript library for building user interfaces, July 2018, https://reactjs.org/

Schema.org, July 2018, https://schema.org/

Selenium - Web browser automation, July 2018, https://www.seleniumhq.org/

Stiltwalker Project, July 2018, http://www.dc949.org/projects/stiltwalker/

Wind, J., Riege, K., Bogen M., 2007. Spinnstube®: A Seated Augmented Reality Display System, Virtual Environments: Proceedings of IPT-EGVE – EG/ACM Symposium, 17-23.

Zheng, S., Song, R., Wen, J., & Wu, D., 2007. Joint optimization of wrapper generation and template detection. KDD