Web Scraping for Data Analytics: A BeautifulSoup Implementation

Conference Paper · March 2023

DOI: 10.1109/WiDS-PSU57071.2023.00025

5 authors, including:

Rabia Latif
Prince Sultan University
12 PUBLICATIONS **125** CITATIONS

Web Scraping for Data Analytics: A BeautifulSoup Implementation

Ayat Abodayeh
College of Computer and Information Sciences
Prince Sultan University
Riyadh, Saudi Arabia
220410035@psu.edu.sa

Reem Hejazi
College of Computer and Information Sciences
Prince Sultan University
Riyadh, Saudi Arabia
219410002@psu.edu.sa

Ward Najjar
College of Computer and Information Sciences
Prince Sultan University
Riyadh, Saudi Arabia
220410019@psu.edu.sa

Leena Shihadeh
College of Computer and Information Sciences
Prince Sultan University
Riyadh, Saudi Arabia
219410254 @psu.edu.sa

Dr. Rabia Latif
College of Computer and Information Sciences
Prince Sultan University
Riyadh, Saudi Arabia
rlatif@psu.edu.sa

***Abstract*— Web scraping is an essential tool for automating the data-gathering process for big data applications. There are many implementations for web scraping, but barely any of them is based on Python's BeautifulSoup library. Therefore, this paper aims at creating a web scraper that gathers data from any website and then analyzes the data accordingly. For results and analysis, the web scraper has been implemented on the Amazon website which collects a product's name, price, number of reviews, rate, and link. We further highlighted the web scraper's capabilities by assimilating the results into an interface that integrates data visualization techniques to analyze the results gathered. The web scraper proved to be efficient upon execution, in which it scraped five pages and analyzed them, and visualized the information in approximately ten seconds. The limitations as a result of this implementation mainly revolved around applying it to specific product names rather than generic ones and extracting specific information that we wanted from the resulting products. Moreover, BeautifulSoup cannot extract all the data available to not compromise on speed. Further studies can be done by researchers who wish to reuse this implementation and modify it according to the data they want to extract, the analysis they wish to perform, and the website they wish to scrape. The implementation can be helpful, thus, to developers who are novices in the web scraping field or to researchers that wish to reuse the code for small data analytics projects.**

Keywords—web scraping, BeautifulSoup, data gathering, data analytics, data visualization

I. INTRODUCTION

There is a potent need to access and analyze a large amount of data. Data harvesting and analytics are always at the heart of any research effort. Most people would directly copy and paste the available information, but this does not apply to immensely large websites or projects that require big data. It can also be a waste of human labor and time.

To facilitate this issue, web scraping has been introduced. Web scraping automates data extraction from web pages, which proves to be fast and effective. Data can be extracted using software services or self-built programs that run on websites with numerous levels of navigation. Web scraping is typically used by individuals and businesses that want to take advantage of the vast amount of freely available web data to make better decisions [1]. This is especially useful when there are websites that contain information that cannot be copied and pasted. With web scraping, data can be retrieved in any form according to the context they are needed in. After retrieving this data, they can be transformed into the preferable format depending on the incentive of the application.

There exists a lack of papers that implement a web scraper using Python's BeautifulSoup4 (the 2022 version) while depicting its capabilities through an interface, which can be troublesome for people who are new to the field. Thus, in this paper, we have developed a web scraper that is based on Python's existing libraries, mainly the BeautifulSoup library while performing data analysis and visualization on the results of the scraper. To implement the web scraper, we targeted the Amazon website in which we extracted certain information that is of interest to the customer (name, price, rate, link, and number of reviews) about an arbitrary product that the user specifies. The proposed method then analyzed the data gathered from the web scraper by using a graphical interface and data visualization techniques based on PySimpleGUI and Matplotlib, respectively, that displayed the top ten products with the cheapest price and highest rate and then graphed price frequencies as well as the number of reviews per rate. The output has indeed proven the efficiency of the proposed web scraping implementation, which can be used in small data analytics projects that do not require a massive amount of extracted data.

II. LITERATURE REVIEW

Many existing papers have delved into the field of web scraping through a myriad of methods. One such paper was made by Färholt, who conducted a controlled experiment to develop a web scraping technique using JavaScript's library, Puppeteer, while remaining undetectable to the websites being targeted [2]. Fruitfully, one of the algorithms the study experimented with did achieve this aim on websites adopting semi-security mechanisms (honeypots and activity logging) by evading them. Many factors have distinguished the implementation of Puppeteer in this study. First, it was revealed that it is feasible to control computer performance using its built-in methods. Other than its built-in functions, Puppeteer allows visiting websites within the library itself,

which is useful in the context of the study. Despite Python being an obvious option, the researcher contended that it would be difficult to implement and that future studies can adopt it as the root of their web scraping endeavor. The results could be of importance to those who want to develop more secure websites against malicious users as well as researchers who wish to gather data without facing security breaches.

Another study conducted by Chaulagain, Pandey et al., [3] which focused on developing a web scraping tool that can handle massive data requests *dynamically*, which they then tested against Amazon's web server. Therefore, the researchers have integrated the use of Selenium and WebDriver API to automate web pages per desired state; accordingly, they parsed text files and extracted data from HTML (hypertext markup language) content using the HTMLParser and Requests libraries in Python. As for the architecture, an elastic computer cloud of web services has been chosen for increased flexibility and virtual scalability. The uniqueness of using a cloud-based distributed Selenium scraper lies in the fact that it can run infinite parallel instances in the cloud. Selenium, despite its process delay, stands out among other tools due to its assimilation into major online enterprises, support of multiple browsers and programming languages, emulation of human behavior over the web, and compatibility with dynamic web pages. This makes the proposed solution a viable option for big data applications.

Similarly, a study by Yannikos, Heeger, and Brockmeyer explored illegal products supplied by three of the largest marketplaces over the dark web [4]. The study expounded on the benefits of web scraping in the field of cybersecurity. The researchers have used Selenium as [3] did to cater to dynamic content. They also used the Requests library in Python to dispatch HTML requests. By contrast, they used the SOCKS proxy support to connect with the TOR browser and RabbitMQ to handle URLs (uniform resource locators) of different marketplaces.

To solve the supply chain management problem in Japan and optimize retail distribution, Le and Pishva employed web scraping along with Google's API (application programming interface) service [5]. They scraped pieces of electronic data by using the programming language Ruby, specifically its library Nokogiri. Ruby's benefits can be reaped in web scraping for its high performance and HTTP (hypertext transfer protocol) parser libraries, and Nokogiri is especially helpful for supporting many document encodings and rapid web page analysis. It can search for documents through XPath or CSS3 selectors. This, as a result, allows the web scraping tool to cover many online sources. The website targeted in this research was the Navitime website since it aligns with the paper's aim and is inclusive of a list of convenience stores and gas stations in Japan.

Many other papers have used web scraping in different contexts. Thomas and Mathur [6] utilized a Python-based approach, combining both BeautifulSoup and Scrapy. Scrapy is a web-crawling framework that uses an application programming interface (API) to extract data and allow developers to write crawlers. Here, BeautifulSoup is used to parse HTML responses in Scrapy callbacks. The paper crawls data stored from the social networking site, Reddit, utilizing the XPath method. The results showed effective data extraction that's based on the structure of the website, form submission analysis, and new submission plan. Another study by Bradley and James [7] has provided a tutorial for scraping online data using the R statistical language, which can perform scraping, statistical analysis, and visualization. The software has many packages with functions made by R's open-source community. The tutorial had four steps: downloading the web page, extracting information from the web page through a code that specifies the location and type of the information collected and storing the extracted information. The information was stored in vectors. Overall, the method can help in scraping websites, but websites that display information in unusual formats might not be easily scraped. Finally, Gunawan, Rahmatulloh, Darmawan, and Firdaus [8] compared the various methods of web scraping using Java programming language. They compared performance from the perspective of regular expression (regex), HTML DOM, and XPath using process time, memory usage, and data consumption as parameters. According to the results, regex consumes the least amount of memory. Meanwhile, HTML DOM requires the least amount of time and the smallest data consumption as opposed to the rest.

Overall, it can be concluded that in terms of efficiency and performance, Ruby is the most appropriate programming language for web scraping as the study by [5] has proven. Unfortunately, the language does not bolster machine learning for big data applications, hence making the usage of Selenium along with an HTML parser a better option as demonstrated by [3] and [4]. In terms of extraction, using HTML DOM would be a better option if there's a need for optimal time and data consumption [8]. If minimum memory utility is desired, using regex would be the best option [8]. Finally, if someone wants to perform web scraping dynamically while remaining undetected or to test the security of web applications, the best option would be JavaScript's open-source library Puppeteer [2].

III. Proposed Methodology

There's a limitation within the existing literature in which the web scrapers proposed did not use BeautifulSoup4; this is especially of significant importance since the implementation of this library can help in small projects based on data analytics. Furthermore, a BeautifulSoup4 web scraper can help researchers who are new to the field by deepening their understanding of the web scraping intrinsic functionality. Thus, we have proposed a methodology that explores web scraping using BeautifulSoup4 in Python along with the Requests library for sending HTML/ DOM (document object model) requests; after that, we extended the capabilities of BeautifulSoup by transforming the results gathered into information that can be of significance to the user by adopting data analytics and visualization via Matplotlib graphs and an interface.

The algorithm for our proposed web scraper is shown in Figure 1. As illustrated, the input for the scraper and the

interface generator is the user's desired input. Consequently, the output will be an analysis of the results from the scraper shown in an interface. To carry out this algorithm, we have proposed the following: an extract function that gathers the preferred elements from HTML results and stores them in a list, a scrape function that handles HTTP requests and uses BeautifulSoup4 to search through division documents and return a Pandas data frame of the results, a visualization function that creates graphs for the data needed in the preferred method which is based on Matplotlib, and an interface function that displays the overall analysis as well as the graphs created.

Input: user's search

Output: analysis results and graphs in an interface

```
1.  Function: Extract_data_from_the resulting_division_documents (HTML results)
2.  Return [elements needed]
3.  Function: Scrape_the_pages (user's search)
4.      for pages to maximum_number_of_pages do
5.          if HTTP_status_code != 200 then
6.              Return error
7.          else
8.              res ← Get request responses from the URL
9.              soup ← BeautifulSoup(res.content, html5lib)
10.             results ← Find division documents of soup using BeautifulSoup
11.             for all r in results do
12.                 row ← extract(r)
13.                 dataframe[dataframe.length] ← row
14.             end for
15.         end if
16.     end for
17. Return dataframe
18. Function: Create_data_visualization_graphs_using_matplotlib
19.     Compute: data needed to visualize from dataframe
20. Return figures_plotted
21. Function: Create_interface_with _data_analysis
22.     Compute: analyze the data frame using Pandas
23.     while True do
24.         Start the window and start scraping after user search
25.         Return analysis results and graphs in the interface
26.     end while
```

Figure 1: Our Web Scraper Algorithm

IV. Implementation

The proposed web scraper can be implemented on any website. For this research, we implemented the web scraper on the Amazon website. As aforementioned, the PySimpleGUI's library has been used for the interface and Matplotlib's library to visualize the data gathered using a bar chart and a histogram. Finally, the interface will display the results of the ten lowest-priced products as well as the ten products that have received the highest ratings which could typically accommodate the user's needs. Each row in the table must contain a hyperlink to the product they want to check out.

The implementation of the proposed method is slightly similar to that made by [3] and [4] in which we have used the Requests library in Python to dispatch HTML requests. However, our web scraping project is solely reliant on the BeautifulSoup library while those studies rather incorporated Selenium for web browser automation. Finally, the proposed implementation in terms of parsing made use of the finding made by [8] since extracting data using HTML DOM is more efficient in terms of processing time and data consumption.

The flowchart of the proposed Amazon web scraper implementation is shown in Figure 2. First, the user has to enter the specific name of the product. This will induce the scraper to send a request to the Amazon website via the Mozilla Firefox web browser. If the HTTP status code is equal to 200, it means that the product has been found. Otherwise, it will terminate and give an error. The scraper then finds all the resulting divs (division documents) and searches for the information we specified in the results. Consequently, the results will be displayed via the graphical interface and Matplotlib graphs.

There are two program files in the proposed implementation. We have made them accessible on GitHub[1]. The AmazonScraper.py file includes the main class for scraping as well as the functions for visualizing the data throughout graphs. The Gui.py file includes the main class for the graphical user interface which is contingent on AmazonScraper. Through AmazonScraper's scrape function, we send a request to the Amazon server with the product's name and the page number, then we start parsing using Python's BeautifulSoup4 and html5lib libraries. The resulting divs have the data component type as *s-search-result* [2] , so we filtered the elements based on that. Then the program loops over the results. In the loop body, we call the extract function which finds the product's name, link, rate, number of reviews, and price and stores them in a Pandas data frame. For this function, the proposed method referred to Amazon's DOM and gathered HTML element types and class names for the information we need. The last function is the visualize function which plots two graphs: one for the price frequencies and another for the number of reviews per each rate.

In the gui.py file, the proposed method used the PySimpleGUI library to create a simple graphical user interface that calls the AmazonScraper class and outlines the layout for the interface. It also contains a while loop to control the flow of the program, keep track of events, and call their proper functions.

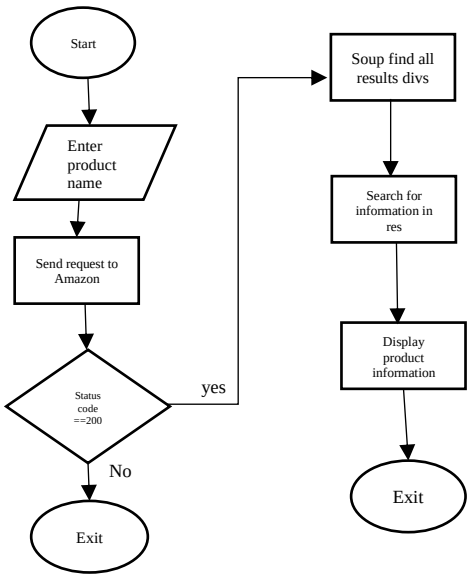

Figure 2: Implementation Flowchart

[1] https://github.com/rem2718/AmazonScraper

[2] Based on Amazon's current html

V. Results

In order to test the proposed web scraper, the code was run twice. The first time was to check the results for the New Apple iPhone 14 pro max as shown in Figure 3. The second was to check the results for the Raspberry Pi 4 as shown in Figure 4. The implementation successfully displayed the data we wanted to analyze on the interface, which is shown in the histogram and bar charts in figure 3 and 4. The total run time for scraping five pages, analyzing the data, visualizing the information via Matplotlib, and displaying the results was approximately 10.9 seconds. The web scraper rummaged through five different pages of results on Amazon for both products.

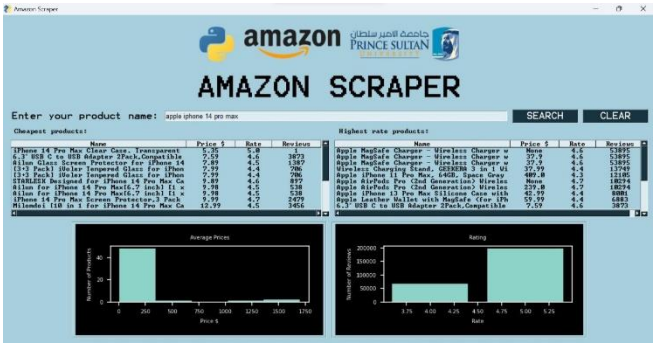

Figure 3: Graphical Interface Output for Apple iPhone 14 Pro Max

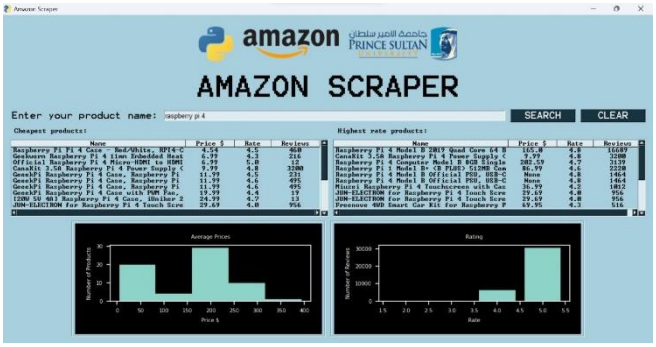

Figure 4: Graphical Interface Output for Raspberry Pi 4

VI. Discussion

There are multiple reasons behind the selection of Python as a tool to implement the proposed web scraper and, its BeautifulSoup4 library. The proposed static web scraper's incentive is to gather a small scale of data from a web page, eliminating the need for Selenium. Ruby can prove to be a practical option for implementation, but Python's variety of libraries, by contrast, would enable us to enhance the visualization of the results according to our preference. Python is also a faster, more popular, and more efficient option to program with as opposed to JavaScript, which [2] has recommended as well. It is crucial to note that the web scraping method to use is heavily dependent on the type of data we wish to extract from an arbitrary web page as well as the scale we are targeting.

The BeautifulSoup library has many strengths associated with its usage. It constructs a parse tree from the HTML and has straightforward methods for flexibly "traversing, searching, and modifying a parse tree" [9]. Furthermore, it relieves the programmer from the burden of encoding since it handles it, a similar trait to that of the Nokogiri gem in Ruby [5].

BeautifulSoup contains backends that provide HTML parsing algorithms which are the following: HTML.parser, lxml, and html5lib. Html.parser is a built-in Python algorithm slower than its counterparts while lxml is C-based and difficult to install [10]. Thus, for our project, we have used the html5lib parser, which is efficient and written in Python.

The results of the proposed method have successfully depicted the strengths of BeautifulSoup, which efficiently traversed the HTML parse tree based on the information we specified in the proposed algorithm. To further highlight them to interested users, we displayed the results in an interface, outlining the flexibility of incorporating such results in different ways for different projects. Our research aimed at analyzing the data extracted and presenting valuable information regarding the price and rate frequencies that can be of interest to the customer; this has proven to be successful due to Matplotlib library's numerous functionalities.

VII. Limitations

The proposed implementation has some limitations associated with it. First, we have made it such that it only works for writing the specific name of the product as listed on Amazon. Also, it scrapes only some information about the products; no details or seller names are extracted. Furthermore, the library that is used for the interface is a simple demonstration, so we encourage implementing the proposed methodology using a different library for the interface. Lastly, if Amazon changes anything in their DOM, this code might not work. It needs to be maintained. Therefore, we recommend reusing this code for small projects in which the researchers can make some alterations to the functions to suit their incentive.

We recommend that further studies reuse this code on other websites to test its efficiency or to extract information based on generic names. To reuse the code, the programmer must make the changes according to the chosen website's HTML. Furthermore, they must alter the elements based on what they wish to extract from the website and maintain those changes in the visualization and interface functions. Graphs different from the ones we have incorporated can be also used since Matplotlib supports many visualization techniques. Seaborn can alternatively be used if the programmer does not wish to use Matplotlib. We also recommend integrating other ways information to highlight the strengths of BeautifulSoup. Finally, a future study that compares the implementation to analyze the data scraped and to display the resulting of a BeautifulSoup4 web scraper with a JavaScript implementation would be an interesting contribution.

VIII. Conclusion and Future Work

In conclusion, a web scraper is a tool used for crawling databases and extracting data. In this project, we developed a web scraper that uses the BeautifulSoup4 library in Python because it is fast and efficient in gathering the required data. This is essential in the field of data analytics since there exists a lack of papers that explore the strengths of this library

through an application of data analytics and visualization. Our proposed web scraper can work on any website by incorporating the changes accordingly. For our implementation, we tested our web scraper on the Amazon website. It sends a request to Amazon servers to search for a product based on its name. Then it displays the ten cheapest products and the ten products with the highest ratings and graphs the price frequencies as well as the number of reviews per rate. The results have proven to be successful as demonstrated by the interface, and the scraper managed to go over five pages and analyze the data according to our preference in approximately ten seconds. This can be of use in projects that want to gather data from websites in an efficient manner. We have some recommendations based on the limitations. The first recommendation is to make the tool display products using generic names if the scraper is reused on Amazon. We also recommend reusing the scraper on other websites and analyzing the data in methods different from the ones we have proposed in this paper. Finally, we recommend conducting research that compares the efficiency of a BeautifulSoup4 web scraper against a JavaScript implementation since it would be useful in the context of data-gathering techniques.

REFERENCES

[1] M. Perez, "What is Web Scraping and What is it Used For?," *ParseHub*, Aug. 06, 2019. https://www.parsehub.com

[2] F. Färholt, "Less Detectable Web Scraping Techniques," Bachelor Thesis, Linnaeus University, Faculty of Technology, Department of computer science and media technology (CM), 2021.

[3] R. S. Chaulagain, S. Pandey, S. R. Basnet, and S. Shakya, "Cloud Based Web Scraping for Big Data Applications," *2017 IEEE International Conference on Smart Cloud (SmartCloud)*, pp. 138–143, Nov. 2017, doi: 10.1109/smartcloud.2017.28.

[4] Y. Yannikos, J. Heeger, and M. Brockmeyer, "An Analysis Framework for Product Prices and Supplies in Darknet Marketplaces," *Proceedings of the 14th International Conference on Availability, Reliability and Security*, Aug. 2019, doi: 10.1145/3339252.3341485.

[5] Q. T. Le and D. Pishva, "Application of web scraping and Google API service to optimize convenience stores' distribution," *2015 17th International Conference on Advanced Communication Technology (ICACT)*, pp. 478–482, Aug. 2015, doi: https://doi.org/10.1109/ICACT.2015.7224841.

[6] D. M. Thomas and S. Mathur, "Data Analysis by Web Scraping using Python," 2019 3rd International conference on Electronics, Communication and Aerospace Technology (ICECA), Coimbatore, India, 2019, pp. 450-454, doi: 10.1109/ICECA.2019.8822022.

[7] A. Bradley and R. J. James, "Web scraping using R," Advances in Methods and Practices in Psychological Science, vol. 2, no. 3, pp. 264–270, 2019.

[8] R. Gunawan, A. Rahmatulloh, I. Darmawan, and F. Firdaus, "Comparison of web scraping techniques : Regular expression, HTML dom and xpath," Proceedings of the 2018 International Conference on Industrial Enterprise and System Engineering (IcoIESE 2018), 2019.

[9] L. Richardson, "Beautiful Soup," *Crummy*, 2020. https://www.crummy.com/software/BeautifulSoup/

[10] Scrapfly, "Web Scraping with Python and BeautifulSoup," *ScrapFly*, Jan. 03, 2022. https://scrapfly.io/blog/web-scraping-with-python-beautifulsoup/