Comparative Analysis of Dynamic Web Scraping Strategies: Evaluating Techniques for Enhanced Data Acquisition

Kaajal Sharma, Gautam M Borkar

Ramrao Adik Institute of Technology, D Y Patil Deemed to be University, Nerul, Navi Mumbai, India

Corresponding author: Kaajal Sharma, Email: kaajals@gmail.com

Web scraping efficiently extracts large data from websites, often in unstructured HTML, which needs conversion for diverse applications. Web scraping supports multiple languages (e.g., C++, Java, JavaScript, PHP, Python, Ruby). Python stands out due to its efficiency, offering numerous built-in and third-party libraries, superior speed, and tailored selection for precise data extraction. Dynamic web pages, characterized by their ability to update and modify content in real-time, are the integral part of the modern web ecosystem. The dynamism and diversity of these web pages pose a significant challenge when it comes to extracting valuable data from them. Traditional web scraping techniques that rely on static HTML parsing often fall short in the face of JavaScript-driven dynamic content. This technical paper focuses on the critical aspect of web scraping within the context of dynamic web pages. We explore key methods and libraries for handling dynamic content, including BeautifulSoup, LXML, and Selenium. We assess their performance and present statistical significance. The experimental results reveal a notable disparity in the performance of web scraping libraries. Specifically, when compared to the widely used Beautiful Soup and Lxml, Selenium library exhibits superior efficiency, utilizing 90% less data, and reducing processing time by 70%. These results highlight the significance of library selection in web data extraction research and offer useful information for practitioners looking for the best web scraping solutions.

Keywords: Web Scraping, dynamic web page, BeautifulSoup, LXML, XPath, HTML DOM, Selenium, performance evaluation, statistical validation.

in: *Advancements in Communication and Systems*. Ed. by Ashish Kumar Tripathi and Vivek Shrivastava. Computing and Intelligent Systems, SCRS, India., 2023, pp. 241–252. DOI: https://doi.org/10.56155/978-81-955020-7-3-22

1. Introduction

In the ever-evolving landscape of the World Wide Web, the demand for dynamic web content continues to grow, as users seek richer and more interactive online experiences. Dynamic web pages, characterized by their ability to update and modify content in real-time, have become integral to the modern web ecosystem.

This paper highlights the crucial aspect of web scraping, examining fundamental methods and libraries developed to tackle challenges in dynamic content. We explore the time efficiency of web extraction methods, focusing on Python libraries like BeautifulSoup, Selenium, and lxml, specifically for dynamic web pages. We will assess the efficiency of these libraries by examining parameters such as time consumption, memory usage, and data consumption. The internet serves as a rich reservoir of extensive data for various purposes, such as business intelligence, competitive analysis, sentiment assessment, agriculture and food economics, bioinformatics, and understanding human behavior on social media, among other data-centric applications [1]. Approximately 70% of the content available on the web is presented in an unorganized and challenging-to-handle structure.

2. Literature Review

Web scraping is the extraction of data or information from internet websites. Several research studies and surveys [2–5, 14, 15, 17] have been undertaken on Web scraping tools and applications that are available in many various forms and with many distinct characteristics. A Web scraper can be created using tools and frameworks made available by several programming languages, or it can be a more robust desktop application or a website extension.

Abundant online storage holds both structured and unstructured data, creating a challenge of sifting through web data for specific inquiries or research topics. This shift, outlined in [6], emphasizes the need to navigate the vast web data landscape. These methods are indispensable for cost-effective and frequent data collection.

Several web scraping methods have been explored in various research studies, including conventional copy and paste [1], Regular Expression (Regex) [7], Hypertext Markup Language Document Object Model (HTML DOM)[8], and XPath [9].

The methods of Web Scraping evolved together with the World Wide Web. Not all listed methods were available at the beginning [10]. There are two most used techniques, DOM, later on allowed the HTML Parsing technique to evolve to DOM Parsing. Second example is Application Programming interfaces (APIs). This technique is the youngest on the list, the growth of available content APIs is dated from 2005. According to ProgrammableWeb.com the number of APIs has grown within 8 years from 0 to 10302 [11].

2.1 Manual Scraping

Manual scraping is still an option in specific situations. These situations are: when the amount of data is minimal, when the data being scraped does not require a repetitive task, when setting up automated scraping would take longer than the data collection itself. Possibly security measures or specific characteristics of the website do not allow automated methods [7].

2.2 HTML DOM

The HTML Document Object Model (HTML DOM) serves as a standard for the retrieval, modification, addition, or removal of HTML elements [12]. It achieves this by defining objects and properties for all

HTML elements, along with methods for accessing them. In this context, all HTML elements are treated as objects, and the programming interface comprises the methods and properties associated with each object. For HTML parsing, websites typically lack convenient formats like .csv or .json files. Analyzing HTML structure allows identification of recurring elements, enabling the use of similar-patterned pages as data sources through scripting or web scraping tools[10].DOM parsing represents an advancement in HTML Parsing, JavaScript and Cascading Stylesheets (CSS) both make considerable use of DOM. Targeting particular areas of web pages is now possible incorporating DOM.

2.3 XPath

XPath plays a central role in XSLT (Stylesheet Language Transformation) and serves as a means to navigate elements and attributes within XML documents[13]. XPath is a language for selecting nodes in XML and HTML documents, emphasizing its capability to express location paths. A location path involves at least one step location to pinpoint specific nodes. The simplest path selects the document's root node using a single slash ("/") symbol, akin to the Unix system file hierarchy. XPath, although demanding a more structured webpage than DOM, offers an equivalent ability to target specific segments in HTML format.

2.4 APIs

Application programming interfaces (APIs) require an application to be a communication partner. APIs are so frequently referred to as machine-readable interfaces rather than human-readable ones. API Directories offer their own API, which enables users to look for API Sources in their database. An API Endpoint receives a response from the server in response to a typical HTTP request. Each API has its own options and specifications. The request has an option to specify the format of the response. JSON is the API communication format that is most frequently used [10] [17].

3. Web Scraping Libraries

3.1 BeautifulSoup

BS4 offers basic techniques for interacting with the DOM paradigm and can parse both HTML and XML documents. Two arguments are included in a BeautifulSoup object, web page's source and the parser .The various parsers, such as html parser, lxml, and html5 lib, can be modified to work with the BeautifulSoup.html object. A class named HTMLParser that is included with parser's default Python installation can be used as a simple HTML and XHTML parser [8].

3.2 LXML

The lxml XML toolkit is a Python binding for the C libraries libxml2 and libxslt. It stands out because it combines the speed and thoroughness of these libraries' XML features with the simplicity of a native Python API, which is mostly compatible with but better than the well-known ElementTree API. The most recent version of lxml is compatible with all versions of Python and supports XPath [13] for extracting tree information. XPath can be used to extract the content fragments into a list.

3.3 Selenium

Selenium, a versatile web scraping tool, automates interactions with websites, facilitating data extraction from dynamic, JavaScript-intensive pages. Known for emulating user-like interactions, it offers features like cross-browser compatibility, dynamic page handling, user interaction emulation, and support for multiple languages, parallel execution, headless browsing, and continuous updates. It inte-

grates with webdriver for advanced interactions, ensuring cross-platform compatibility with extensive community and documentation support. These features make Selenium a powerful and flexible choice for web scraping, particularly for websites with dynamic content and complex user interactions [16].

4. Libraries and Combinations Utilised

We compared three systems—BS4 with Selenium, lxml with Selenium, and standalone Selenium—all utilizing Selenium for navigating and interacting with dynamic web pages due to its remarkable versatility.

4.1 BS4 with Selenium

Beautiful Soup (BS4) when used alongside Selenium becomes a potent tool for dynamic web scraping. This combination is highly effective for websites with ever-changing content. It is shows benefit in hhandling dynamic content, automates complex navigation and pagination.

4.2 LXML with Selenium

Lxml sets itself apart from Beautiful Soup (BS4) through its superior parsing speed and structured data processing capabilities, rendering it ideal for demanding tasks. Lxml adheres to a more rigorous tree-based parsing approach. This distinction, complemented by Selenium's interaction capabilities, facilitates high-speed, structured data extraction from dynamically loading web pages.

4.3 Standalone Selenium

In dynamic web scraping, Selenium automates user actions such as clicks, form submissions, and scrolling, synchronizing with dynamically loaded elements. This interaction-driven approach ensures comprehensive data extraction from dynamic content websites. Selenium is memory-efficient, allowing automatic script execution without excessive memory usage for lengthy or complex scraping tasks. Additionally, it streamlines data transfer and automates extraction by facilitating data exchange between the web browser and the scraping script.

5. Methodology

In this section we will discuss the executed method for comparing the three libraries.
as shown in Figure 1.

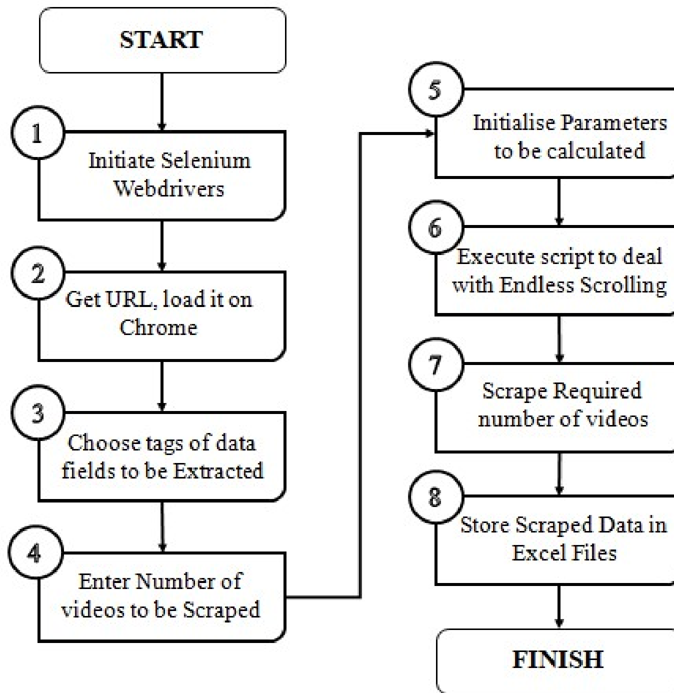

Figure 1. Steps for web scraping process

1. Initialise Selenium webdrivers

Selenium webdrivers are browser-specific components that enable it to automate interactions with web browsers, which enable to simulate user interactions with different browsers, automating tasks like clicking, form-filling, and data extraction.
Necessary Python Libraries were used to import

from selenium import webdriver

Appropriate WebDriver is downloaded, ensuring compatibility with chosen browser and its version
WebDriver Path is set, Instantiate the WebDriver Object

driver = webdriver.Chrome(executable_path="/path/to/chromedriver")

2. Get URL load it in on Chrome

Following the initialization of the webdrivers, the designated website link intended for scraping was obtained, and Selenium utilized the Chrome webdriver to automate the process of opening the specified webpage URL.
Specify the Target URL (Set the URL of the webpage to be loaded within a variable.)

target_url = "https://www.example.com"

Invoke the get() method on the Chrome webdriver instance, passing the target URL as an argument. This triggers Selenium to launch a new Chrome browser instance and navigate to the specified URL.

driver.get(target_url)

3. Choose tags of data fields to be extracted

Upon the webpage's complete loading, an examination and analysis of the necessary video elements for scraping were conducted. This investigation revealed that each video was distinctly segmented into separate video blocks, each bearing unique index values, while sharing identical HTML tag references including class name, id name, and title tag.

Page Structure Examination is done to understand the structure of the web page using browser developer tools (e.g., Chrome DevTools) to inspect the webpage's Document Object Model (DOM). Analyze the HTML structure to pinpoint the specific tags encapsulating the desired video data fields.

Locator Identification to select appropriate locator strategies to uniquely identify target elements
ID: Use find_element_by_id() for elements with unique IDs.
Class Name: Use find_elements_by_class_name() for elements sharing a class.
Tag Name: Use find_elements_by_tag_name() for broad element types.
XPath: Use find_elements_by_xpath() for complex navigation and conditional searching.
CSS Selector: Use find_elements_by_css_selector() for flexible and concise targeting.
Technical Observations in Context:
Unique Index Values: Each video block possesses a distinct index, potentially employable for identification via XPath or CSS selectors.
Shared Tag References: HTML class name, ID name, and title tag are common among videos, necessitating careful locator strategies to isolate individual blocks.
Targeting by Class Name: video_blocks = driver.find_elements_by_class_name("video-block")
Extracting Title Attribute: video_title = video_block.get_attribute("title")
Dynamic Content: If content loads asynchronously, employ explicit waits or visibility checks.
Element Hierarchies: Understand parent-child relationships within the DOM for accurate targeting.
Attribute Extraction: Utilize get_attribute() to retrieve specific attribute values.
Text Content Extraction: Use text property to extract text content within elements.

4. Enter number of videos to be scraped

After determining the data to be scraped, the specific count of videos to be extracted was also determined in order to restrict the number of iterations required for the loop. In our research, we gathered data from more than 10,000 video blocks during the scraping process.
We have used Python's input() function to solicit the desired number of videos from the user,

```
num_videos_to_scrape = int(input("Enter the number of videos to scrape: "))
for i in range(num_videos_to_scrape):
  # Perform video scraping actions within the loop
```

5. Initialize parameters to be calculated

Parametric Values which were to be calculated to compare the three library combinations were initialized. These parameters encompass metrics like the total time taken for the scraping process, the process time involving CPU and Kernel resources for computing the output, system memory usage during the process, and the volume of data bytes exchanged throughout the entire operation.

6. Execute script to deal with endless scrolling

A fully loaded YouTube page provides a finite number of videos or video blocks for data extraction. To avoid the manual effort of scrolling to the page's end, we deployed a Selenium script to automate this process once the initial page had been thoroughly scraped for all available data.

7. Scrape required number of videos

The fully loaded page is downloaded in HTML format using Selenium. Subsequently, it is processed within the relevant parsers. A loop iterates until all the videos scheduled for scraping have been processed, consistently updating the scraped data in a Data Frame.

8. Store scraped data in excel files
The Data Frames containing the scraped data were exported to Excel for in-depth data analysis, with a primary focus on assessing data integrity and identifying any redundancy.
We integrated the pandas library for DataFrame manipulation and file export using

```
import pandas as pd
df = pd.DataFrame(video_data)
df.to_excel("scraped_data.xlsx", index=False)
df.to_excel("existing_file.xlsx", index=False, mode="a", startrow=10)
df.to_excel("scraped_data.xlsx", sheet_name="Videos")
```

Additionally for data validation we can employ pandas' validation techniques to ensure data quality and consistency. For file compression formats like ZIP for storage optimization can be used. And alternative data storage methods databases (e.g., SQLite, PostgreSQL) for enhanced data management and analysis capabilities can also be used.

6. Results Analysis

In the course of our comparative study, we conducted comprehensive research and analysis of four key parameters, as discussed in this section.

6.1 Overall Time

Overall time refers to the total time required to perform the entire web scraping procedure which includes multiple stages, such as sending requests to web servers, receiving and parsing web content, extracting specific data, processing and organizing the collected information, and finally storing it in the desired format.

$$TTT = T_{req} + T_{resp} + T_{scrap} \quad (1)$$

where,
TTT = Total Time Taken
T_{req} = Time Taken send HTTP Request
T_{resp}= Time Taken to send HTTP Response
T_{scrap}= Time Taken to scrap through received Response

Table 1. shows the overall time taken by the libraries. The last row shows the average of all values, with LXML being the faster one.

Table 1. Overall Time Measurement Results

No. of iterations	No. Video Blocks Scraped	BS4 with Selenium (sec)	LXML with Selenium (sec)	Standalone Selenium(sec)
1-5	600	8.86	5.08	9.12
6-10	1350	17.19	7.73	17.72
11-15	2100	18.92	11.57	26.90
16-20	2850	24.30	14.95	35.83
21-25	2820	29.68	20.39	45.55
	9720	19.79	**11.94**	27.02

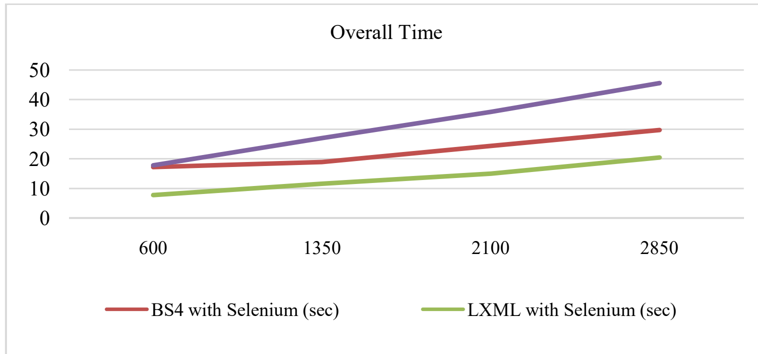

Figure 2. Average Results of Overall Time Measurement

As can be seen in Figure 2. upon recording the total time required for scraping over 10,000 video blocks and graphically representing the data, it was determined that utilizing lxml with Selenium provides the most efficient approach for dynamic web scraping, wherein time constraints are considered.

6.2 Process Time Utilized

Process time is the duration consumed by the Central Processing Unit (CPU) and Kernel to execute the web scraping procedure. It encompasses the time spent on computational tasks, data parsing, and information extraction.. It plays a crucial role in resource management and determining the overall effectiveness of the web scraping process.

$$T_{process} = T_{user} + T_{system} \tag{2}$$

where:
$T_{process}$= Total Process Time
T_{user} = Total time spent executing the actual code in user space
T_{system} = Total Time spent on the operating system kernel on behalf of the application

Table 2. below shows the process times taken by the libraries during the parsing through over 10000 videos. The last row shows the average of all the values, with Selenium coming out as the fastest comparatively.

Table 2. Process Time Measurement Result

No. of iterations	No. Video Blocks Scraped	BS4 with Selenium (sec)	LXML with Selenium (sec)	Standalone Selenium(sec)
1-5	600	6.44	2.98	1.453
6-10	1350	14.25	5.60	1.53
11-15	2100	15.90	9.00	2.2
16-20	2850	20.94	11.94	3.05
21-25	2820	26.15	16.58	5.132
	9720	16.73	9.22	**2.675**

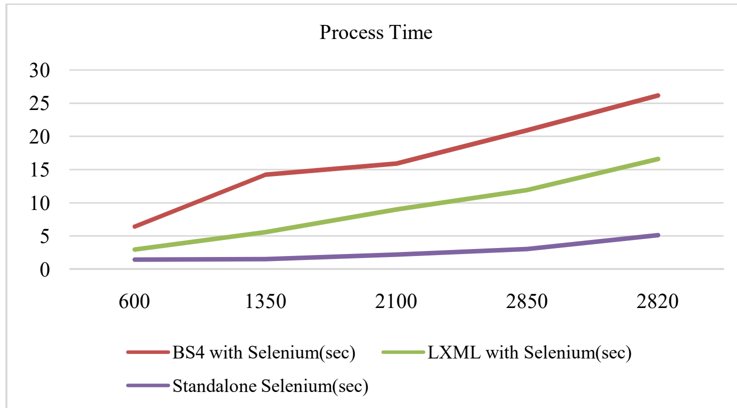

Figure 3. Average Results of Process Time

As can be seen in Figure 3. Standalone Selenium exhibited the most optimal process time.

6.3 Memory Utilized

This pertains to the amount of RAM (Random Access Memory) consumed during the execution of web scraping operations. It reflects the memory allocation and utilization by the script, libraries, and the web browser automation tool (e.g., Selenium)

$$MemoryUsed = M_{Pk1} - M_{pk0} \quad (3)$$

where,
M_{Pk1} = Peak Memory Recorded before execution of code lines
M_{Pk2} = Peak Memory Recorded after execution of code lines

Table 3. shows memory used to store the data during each method's execution for web scraping.

Table 3. Memory Usage Measurement Results.

No. of iterations	No. Video Blocks Scraped	BS4 with Selenium (bytes)	LXML with Selenium (bytes)	Standalone Selenium (bytes)
1-5	600	233578	344862.4	347760.6
6-10	1350	197150.2	226848.4	359055.8
11-15	2100	183749.8	537070.4	192669
16-20	2850	225882.8	251270.2	300123
21-25	2820	200195.75	457740	416771.5
	9720	514.7	887.98	788.59

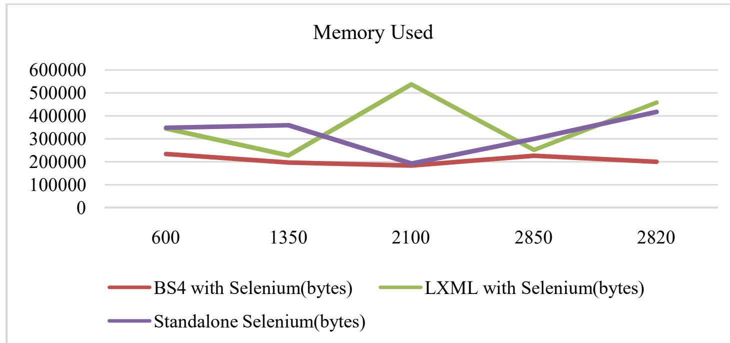

Figure 4. Average Results of Memory Usage Measurement

As can be seen in Figure 4, BS4 utilizes lesser memory.

6.4 Data Bytes Used

This measures the data exchanged between the web scraping script and targeted websites, encompassing requests and responses. Accurate measurement and optimization of data bytes usage assess overall efficiency and cost-effectiveness in web scraping operations.

$$DataBytesUsed = (BytesRecv_1 + BytesSent_1) - (BytesRecv_0 + BytesSent_0) \quad (4)$$

where,
$BytesRecv_0$ = Number of Bytes Received before execution of code lines
$BytesSent_1$= Number of Bytes Sent before execution of code lines
$BytesRecv_1$= Number of Bytes Received after execution of code lines
$BytesRecv_0$ = Number of Bytes Sent after execution of code lines

Table 4. shows the data byte exchanged by the request library and the URLs, while accessing the HTML for each webpage. The last row gives us the average values.

Table 4. Data Usage Measurement Results

No. of iterations	No. Video Blocks Scraped	BS4 with Selenium (bytes)	LXML with Selenium (bytes)	Standalone Selenium (bytes)
1-5	600	39083619	38737464	391270
6-10	1350	73428297	68549221	526857
11-15	2100	106894048	98539870	822412
16-20	2850	138107954	128780070	1247569
21-25	2820	166133655	156229809	1779807
	9720	252274	236414	**2269**

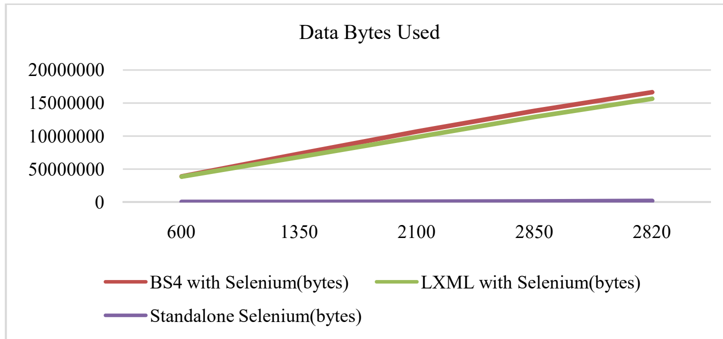

Figure 5. Average Results of Data Usage Measurement

As can be seen in Figure 5. Standalone Selenium gave very brilliant results as compared to other combinations, after aggregating data of over 10,000 scraped videos.

6.5 Average Comparisons

The table below displays the average of process time measurement, overall time measurement, data usage measurement and memory usage measurement results of bs4, Selenium and LXML.

Table 5. A comparison of each parameter's average value

Parameters	BS4 with Selenium	LXML with Selenium	Standalone Selenium
Overall Time (in sec)	19.79	**11.95**	27.03
Process Time (in sec)	16.74	9.23	**2.68**
Memory Usage (in Bytes)	**514.67**	887.9	788.556
Data Usage (in Bytes)	252274.	236414.9	**2269.52**

7. Conclusion

In this experiment, using Python libraries, data from a dynamic website is extracted and compared to determine how long each takes to complete. Following key conclusions were reached based on the findings of the trials.

Using Selenium with LXML yields superior performance in terms of process time and overall time, whereas Selenium combined with BS4 demonstrates better performance in overall memory usage. Selenium alone provides moderately balanced results compared to BS4 and LXML.

For beginners and simpler HTML, opt for Beautiful Soup and Selenium. Experienced developers dealing with complex XML or seeking high performance should consider Lxml and Selenium.

To improve the preceding approach, more parameters may be included while testing, mending, or combining procedures to address their shortcomings.

References

[1] Nigam, H., Biswas, P.:From Web Scraping to Web Crawling. In:Choudhary, A., Agrawal, A.P., Logeswaran, R., Unhelkar, B.(eds) Applications of Artificial Intelligence and Machine Learning. Lecture Notes in Electrical Engineering, vol. 778, Springer, Singapore(2021).

[2] Laender, A.H.F., Ribeiro-Neto, B.A, da Silva, A.S, Teixeira, J.S.: A brief survey of web data extraction tools. ACM SIGMOD Rec 31(2):84, (2002).

[3] Singrodia, V., Mitra, A., Paul, S.:A review on web scraping and its applications. In: 2019 International Conference on Ccomputer Communication and Informatics (ICCCI). Coimbatore, Tamil Nadu, India, pp 1–6, (2019).

[4] Vanden Broucke, S., Baesens, B.: Practical Web scraping for data science. 1st edn. Apress, Berkeley, CA, (2018).

[5] Castrillo-Fernández Q.:Web scraping: applications and tools. European Public Sector Information Plat form Topic Report No.2015/10, (2015).

[6] Boehmke, B. C. (2016). Data wrangling with R(Use R!). Dayton, OH: Springer International, (2106).

[7] Sirisuriya, D.: A Comparative Study on Web Scraping. Proceedings of 8th International Research Conference, KDU, pp. 135–140,(November 2015).

[8] Uzun, E.Yerlİkaya.T, Kirat,O.: Comparison Of Python Libraries Used For Web Data Extraction. Journal of the Technical University - SofiaPlovdiv branch, Bulgaria, Fundamental Sciences and Applications, Vol. 24, (May2018).

[9] Grasso.G., Furche.T., Schallhart, C.:Effective Web Scraping with OXPath. WWW '13 Companion, Pages 23–26,(2013).

[10] Draxl,V.: Web Scraping Data Extraction from websites. BS Thesis, University of Applied Sciences, (2018).

[11] Berlind,D.:APIs Are Like User Interfaces--Just With Different Users in Mind. at: https://www. Programmableweb . com, (2015).

[12] W3C:What is the Document Object Model?, Available: https://www.w3.org/TR/WD-DOM/introduction.html,(2016).

[13] XML and XPath., https: // www.w3schools.com /xml/ xml_xpath.asp, last accessed 2023/10/24

[14] Saurkar, A.V., Pathare, K. G., Gode, S.:An overview on web scraping techniques and tools. IJFRCSCE, 4(4), 363-367, (2018).

[15] Lotfi,C., Srinivasan,S., Ertz,M. and Latrous, I.: Web Scraping Techniques and Applications: A Literature Review. SCRS Conference Proceedings on Intelligent Systems, SCRS, India, pp. 381-394, (2022).

[16] Bale,A.S.,Ghorpade,N.,Rohith,S.,Kamalesh,S.,Rohith,R.,Rohan,B.S.:Web Scraping Approaches and their Performance on Modern Websites ,ICESC IEEE Xplore Part Number: CFP22V66-ART; ISBN: 978-1-6654-7971-4, (2022).

[17] Glez-Peñna,D.,Lourenco,A.,López-Fernández,H.,Reboiro-Jato,M,and Fdez-Riverola, F.: Web scraping technologies in an API world Briefings in Bioinformatics, 15(5), 788–797, 2013